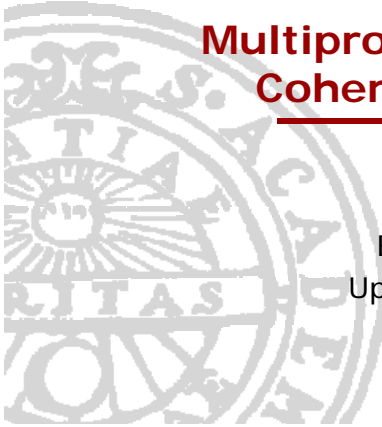


# Multiprocessors and Coherent Memory

Erik Hagersten  
Uppsala University



## DARK2 in a nutshell

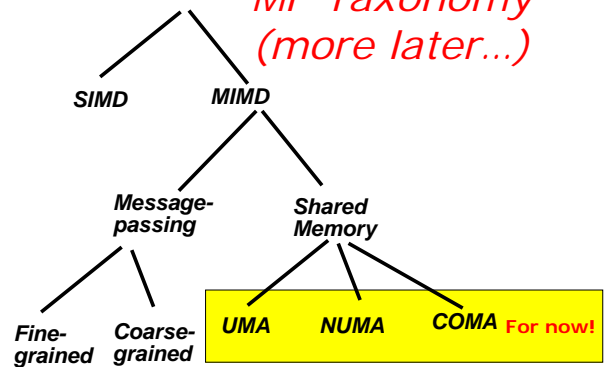
1. Memory Systems (caches, VM, DRAM, microbenchmarks, ...)
2. Multiprocessors (TLP, coherence, memory models, interconnects, scalability, ...)
3. CPUs (pipelines, ILP, scheduling, Superscalars, VLIWs, embedded, ...)
4. Future: (physical limitations, TLP+ILP in the CPU,...)

## The era of the "Rocket Science Supercomputers" 1980-1995

- The one with the most blinking lights wins
- The one with the niftiest language wins
- The more different the better!



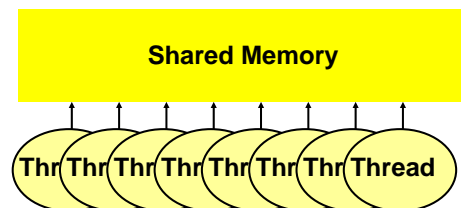
## MP Taxonomy (more later...)



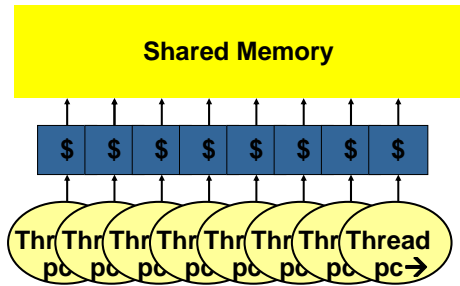
## Models of parallelism

- Processes (**fork** or **&** in UNIX)
  - A parallel execution, where each process has its own process state, e.g., memory mapping
- Threads (**thread\_create** in POSIX)
  - Parallel threads of control inside a process
  - There are some thread-shared state, e.g., memory mappings.
- Sverker will tell you more...

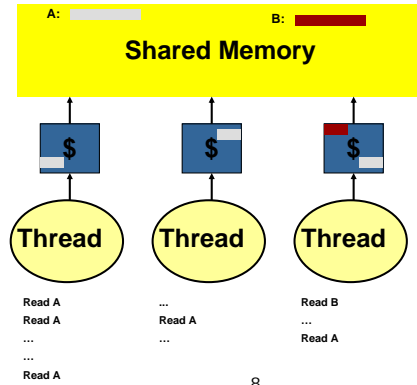
## Programming Model:



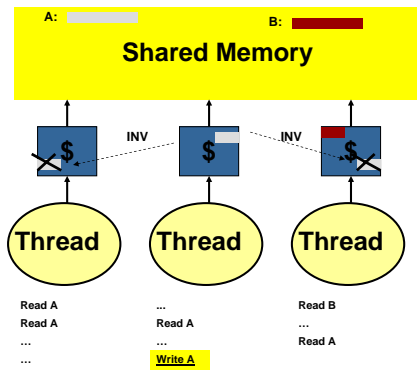
## Adding Caches: More Concurrency



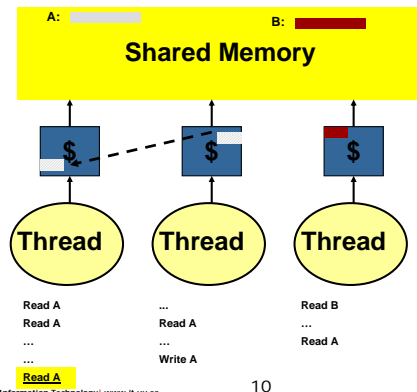
## Caches: Automatic Replication of Data



## The Cache Coherent Memory System



## The Cache Coherent \$2\$



## Summing up Coherence

*There are multiple copies of a datum, but only one value*

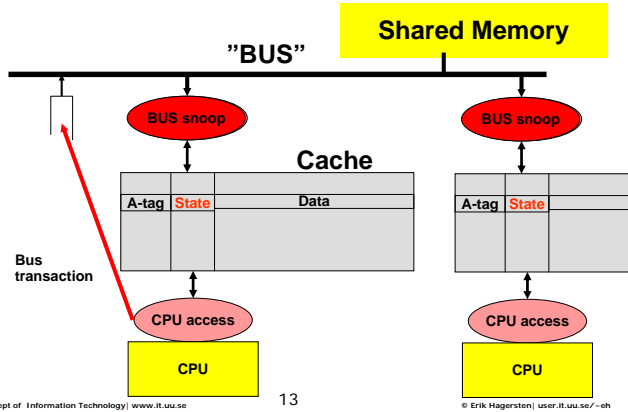
**Too strong definition!**

*There is a single global order of value changes to each datum*

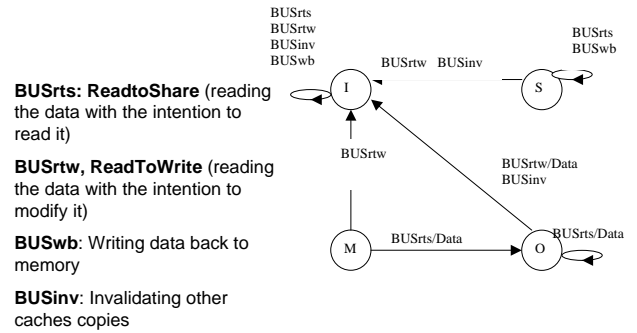
## Implementation options for memory coherence

- Two coherence options
  - Snoop-based ("broadcast")
  - Directory-based ("point to point")
- Different memory models
- Varying scalability

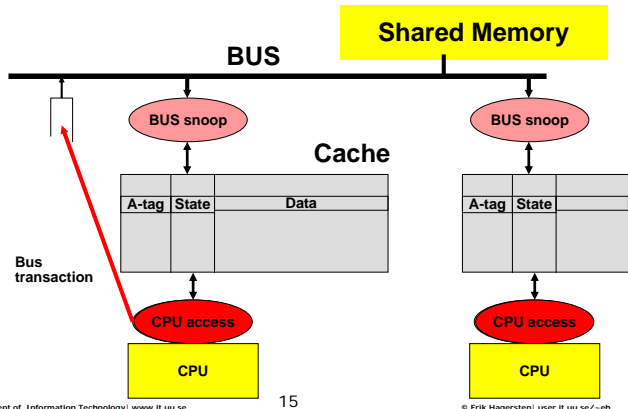
### Snoop-based Protocol Implementation



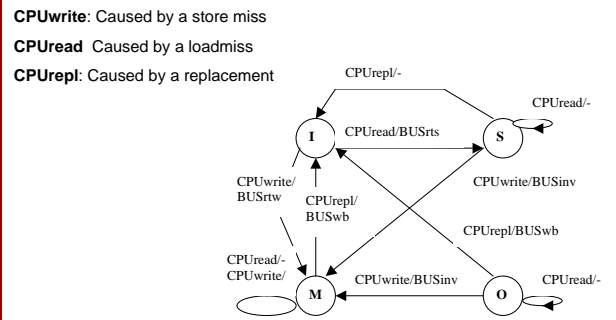
### Example: Bus Snoop MOSI



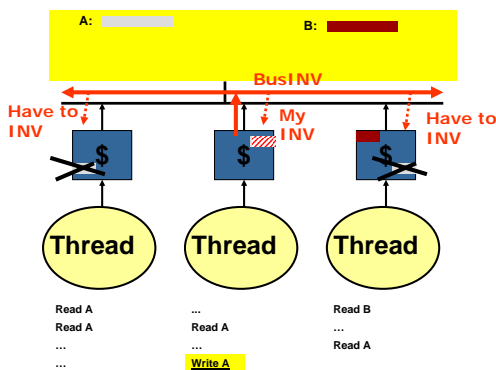
### Snoop-based Protocol Implementation



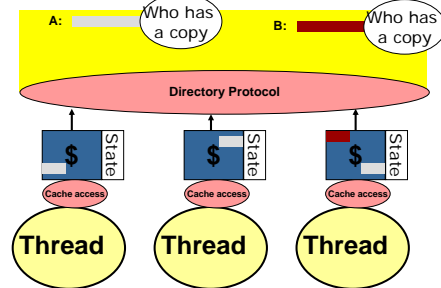
### Example: CPU access MOSI



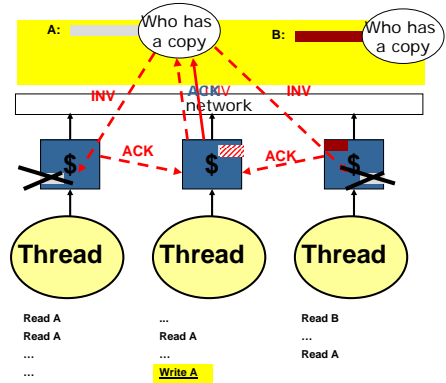
### "Upgrade" in snoop-based



### Directory-based coherence: per-cacheline info in the memory



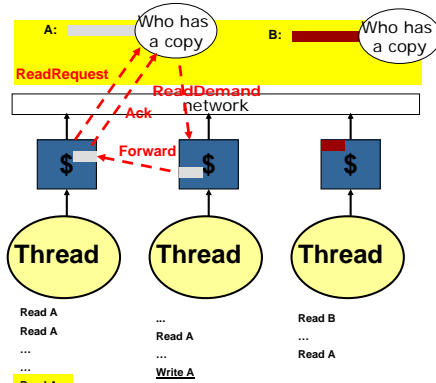
## "Upgrade" in dir-based



19

© Erik Hagersten | user.it.uu.se/~eh

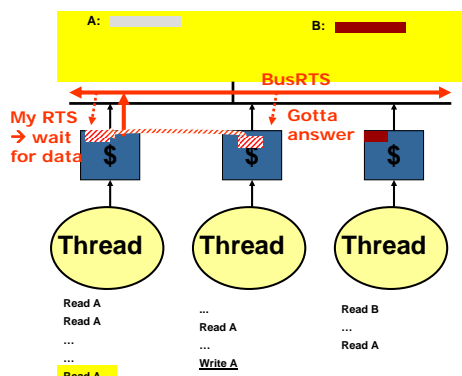
## Cache-to-cache in dir-based



20

© Erik Hagersten | user.it.uu.se/~eh

## Cache-to-cache in snoop-based



21

© Erik Hagersten | user.it.uu.se/~eh

## A New Kind of Cache Miss

- Capacity – too small cache
- Conflict – limited associativity
- Compulsory – accessing data the first time
- Communication (or "Coherence") [Jouppi]
  - Caused by downgrade (modified → shared)
    - "A store to data I had in state M, but now it's in state S" ☹
  - Caused my invalidation (shared → invalid)
    - "A load to data I had in state S, but now it's been invalidated" ☹

22

© Erik Hagersten | user.it.uu.se/~eh

## Why snoop?

- A "bus": a serialization point helps coherence and memory ordering
- Upgrade is faster [producer/ consumer and migratory sharing]
- Cache-to-cache is **much** faster [i.e., communication...]
- Synchronization, a combination of both
- ...but it is hard to scale the bandwidth ☹

23

© Erik Hagersten | user.it.uu.se/~eh

## Why directory-based

- P2P messages → **high** bandwidth
- Suits out-of-the-box coherence
- Note:
  - Dir-based can be used to build a uniform-memory architecture (UMA)
  - Bandwidth will be great!!
  - Memory latency will be OK
  - Cache-to-cache latency will not!

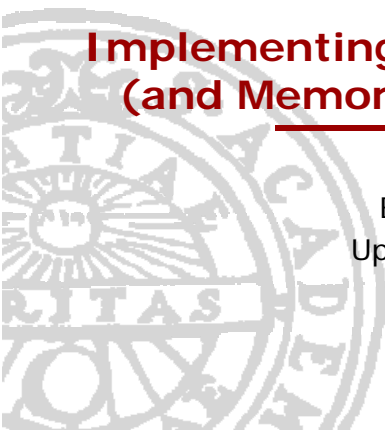
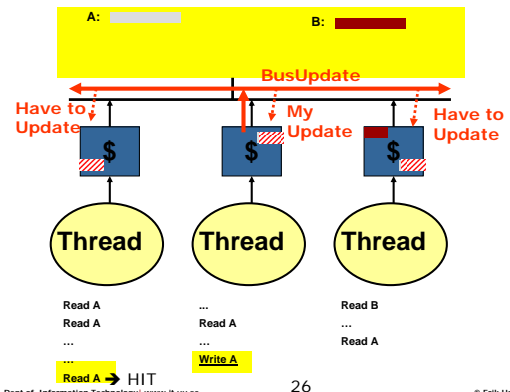
24

© Erik Hagersten | user.it.uu.se/~eh

## Update Instead of Invalidate?

- Write the new value to the other caches holding a shared copy (instead of invalidating...)
- Will avoid coherence misses
- Consumes a large amount of bandwidth
- Hard to implement strong coherence
- Few implementations: SPARCCenter2000, Xerox Dragon

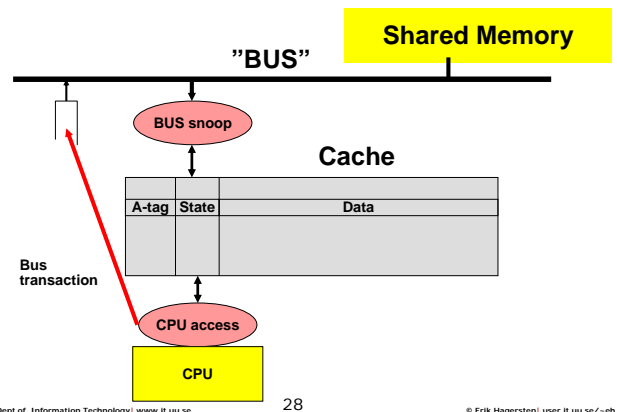
## Update in MOSI snoop-based



## Implementing Coherence (and Memory Models...)

Erik Hagersten  
Uppsala University  
Sweden

## Snoop-based Protocol Implementation



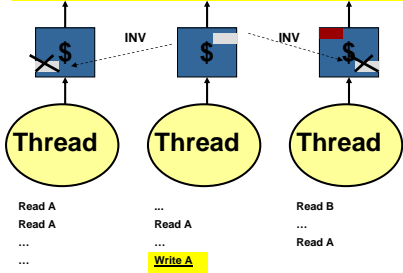
## Common Cache States

- M – Modified**  
My dirty copy is the only cached copy
- E – Exclusive**  
My clean copy is the only cached copy
- O – Owner**  
I have a dirty copy, others may also have a copy
- S – Shared**  
I have a clean copy, others may also have a copy
- I – Invalid**  
I have no valid copy in my cache

## Some Coherence Alternative

- MSI**
  - Writeback to memory on a cache2cache.
- MOSI**
  - Leave one dirty copy in a cache on a cache2cache
- MOESI**
  - The first reader will go to E and can later write cheaply

# The Cache Coherent Memory System

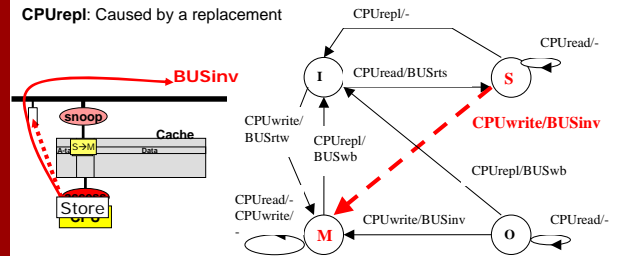


# Upgrade – the requesting CPU

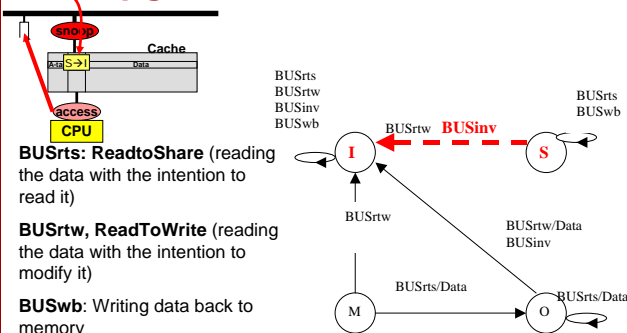
**CPUwrite:** Caused by a store miss

**CPUread:** Caused by a loadmiss

**CPUrepl:** Caused by a replacement



# Upgrade – the other CPUs



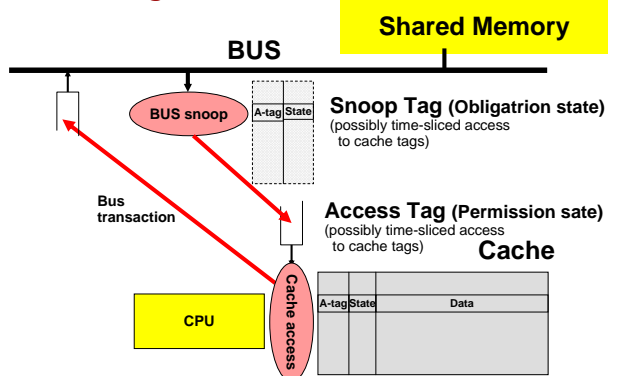
**BUSrts:** ReadtoShare (reading the data with the intention to read it)

**BUSrtw, ReadToWrite** (reading the data with the intention to modify it)

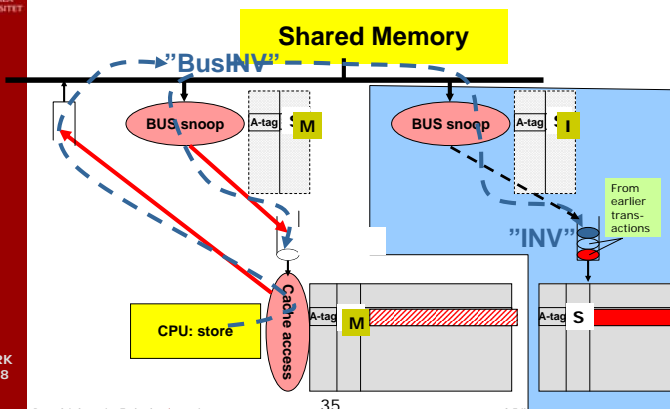
**BUSwb:** Writing data back to memory

**BUSinv:** Invalidating other caches copies

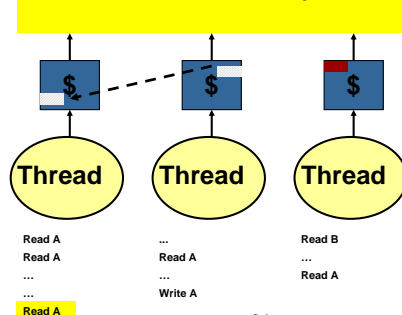
# Modern snoop-based architecture -- dual tags



# "Upgrade" in snooped-based

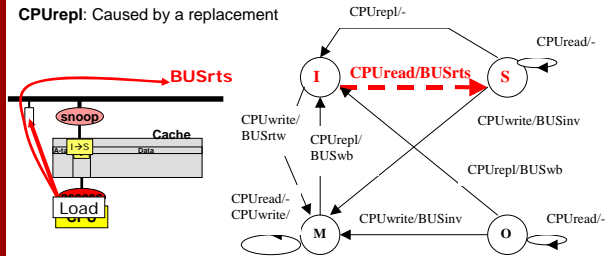


# The Cache Coherent Cache-to-cache



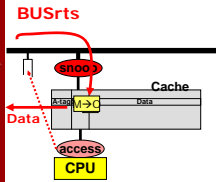
## Cache2cache – the requesting CPU

- CPUwrite:** Caused by a store miss
- CPUread** Caused by a loadmiss
- CPUrepl:** Caused by a replacement

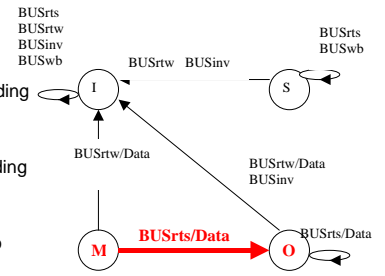


37

## Cache-to-cache – the other CPU

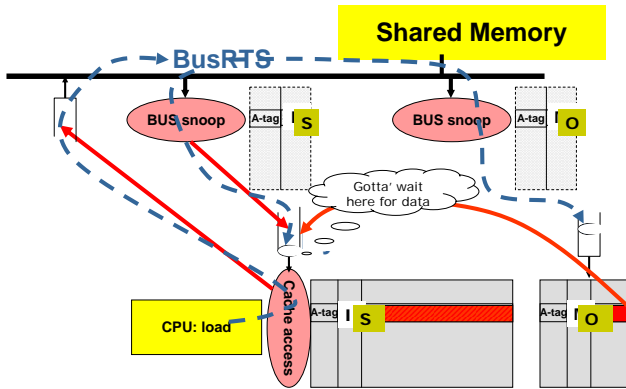


- BUSRts: ReadToShare** (reading the data with the intention to read it)
- BUSrtw, ReadToWrite** (reading the data with the intention to modify it)
- BUSwb:** Writing data back to memory
- BUSinv:** Invalidating other caches copies



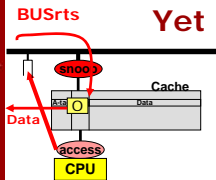
38

## Cache-to-cache in snoop-based

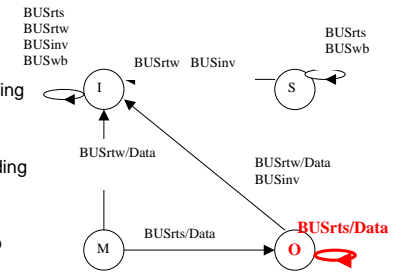


39

## Yet Another Cache-to-cache



- BUSRts: ReadToShare** (reading the data with the intention to read it)
- BUSrtw, ReadToWrite** (reading the data with the intention to modify it)
- BUSwb:** Writing data back to memory
- BUSinv:** Invalidating other caches copies



40

All the three RISC CPUs in a MOSI shared-memory sequentially consistent multiprocessor executes the following code almost at the same time:

```

while(A != my_id){}; /* this is a primitive kind of lock */
B := B + A * 2;
A := A + 1; /* this is a primitive kind of unlock */
while (A != 4) {}; /* this is a primitive kind of barrier*/
<after a long time>
<some other execution replaces A and B from the caches, if still present>
    
```

Initially, CPU1 has its local variable my\_id=1, CPU2 has my\_id=2 and CPU3 has my\_id=3 and the globally shared variables A is equal to 1 and B is equal to 0. CPU2 and 3 are starting slightly ahead of CPU1 and will execute the first while statement before CPU1. Initially, both A and B only reside in memory.

The following four bus transaction types can be seen on the snooping bus connecting the CPUs:

- **RTS:** ReadtoShare (reading the data with the intention to read it)
- **RTW,** ReadToWrite (reading the data with the intention to modify it)
- **WB:** Writing data back to memory
- **INV:** Invalidating other caches copies

Show every state change and/or value change of A and B in each CPU's cache according to one possible interleaving of the memory accesses. After the parallel execution is done for all of the CPUs, the cache lines still in the caches will be replaced. These actions should also be shown. For each line, also state what bus transaction occurs on the bus (if any) as well as which device is providing the corresponding data (if any).

41

## Example of a state transition sheet:

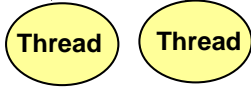
CPU action	Bus Transaction (if any)	State/value after the CPU action						Data is provided by (CPU 1,2,3 or Mem) (if any)
		CPU1		CPU2		CPU3		
		A	B	A	B	A	B	
Initially		1	1	1	1	1	1	
CPU1: LD A	RTS(A)	S/1						Mem
CPU2: LD B	RTS(B)			S/0				Mem
...some time elapses...								
CPU1: replace A	-	I						-
CPU2: replace B	-			I				-

42

## False sharing



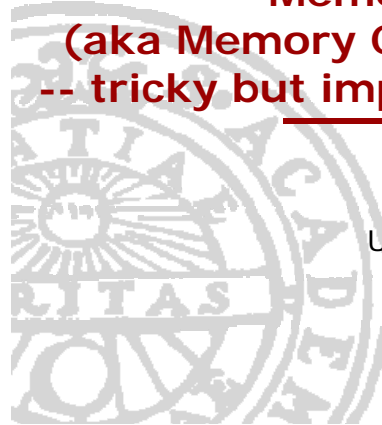
Communication misses even though the threads do not share data "the cache line is too large"



Read A  
Write A  
...  
Read A

Read E  
...  
Write E

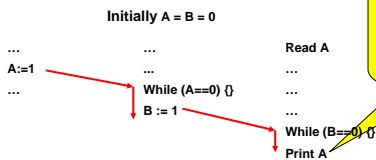
## Memory Ordering (aka Memory Consistency) -- tricky but important stuff



Erik Hagersten  
Uppsala University  
Sweden

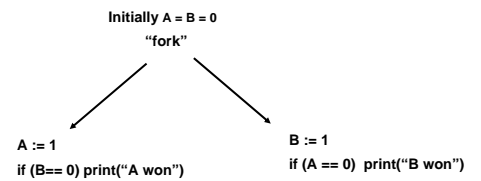
## Memory Ordering

- Coherence defines a per-datum valuechange order
- Memory model defines the valuechange order for all the data.



Q: What value will get printed?

## Dekker's Algorithm



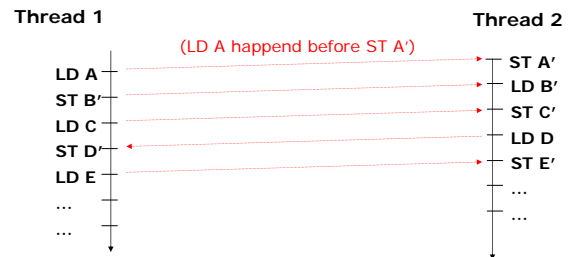
Q: Is it possible that both A and B win?

## Memory Ordering

- Defines the guaranteed memory ordering
- Is a "contract" between the HW and SW guys
- Without it, you can not say much about the result of a parallel execution

## In which order were these threads executed?

(A' denotes a modified value to the data at addr A)





**One possible observed order**      **Another possible observed order**

Thread 1 Thread 2      Thread 1 Thread 2

DARK 2008      49      © Erik Hagersten | user.it.uu.se/~eh

**“The intuitive memory order”**  
**Sequential Consistency (Lampert)**

Shared Memory  
loads, stores

- Global order achieved by *interleaving all* memory accesses from different threads
- “Programmer’s intuition is maintained”
  - Store causality? Yes
  - Does Dekker work? Yes
- Unnecessarily restrictive ==> performance penalty

DARK 2008      50      © Erik Hagersten | user.it.uu.se/~eh

**Dekker’s Algorithm**

Initially A = B = 0  
“fork”

A := 1  
if (B == 0) print(“A won”)

B := 1  
if (A == 0) print(“B won”)

**Q: Is it possible that both A and B win?**

DARK 2008      51      © Erik Hagersten | user.it.uu.se/~eh

**Sequential Consistency (SC) Violation**  
**→ Dekker: both wins**

Access graph

A := B := 0

A := 1  
If (B == 0) print “Left wins”

B := 1  
If (A == 0) print “Right wins”

Both Left and Right wins → SC violation

VO: Value order: c < d (i.e., c happened before d in the global order)

PO: Program order: a < b (the order specified by the program)

**Cyclic access graph → Not SC**  
**(there is no global order)**

DARK 2008      52      © Erik Hagersten | user.it.uu.se/~eh

**SC is OK if one thread wins**

Only Right wins → SC is OK

A := B := 0

A := 1  
If (B == 0) print “Left wins”

B := 1  
If (A == 0) print “Right wins”

**Not cyclic graph → SC**

**One global order:**  
STB < LDA < STA < LDB

DARK 2008      53      © Erik Hagersten | user.it.uu.se/~eh

**SC is OK if no thread wins**

No thread wins → SC is OK

A := B := 0

A := 1  
If (B == 0) print “Left wins”

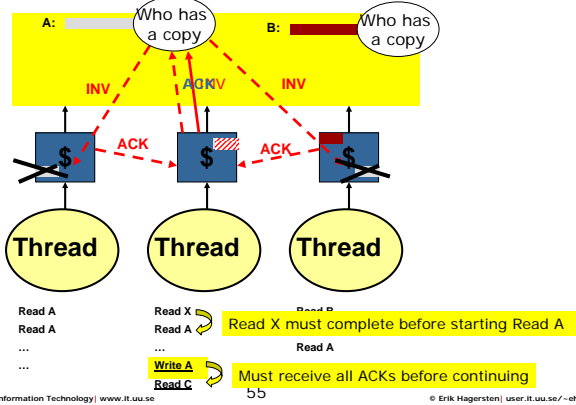
B := 1  
If (A == 0) print “Right wins”

**Not cyclic graph → SC**

**Four Partial Orders, still SC**  
STB < LDA; STA < LDA; STB < LDB; STA < LDA

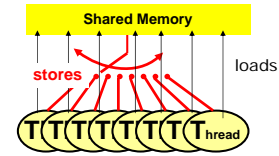
DARK 2008      54      © Erik Hagersten | user.it.uu.se/~eh

## One implementation of SC in dir-based (...without speculation)



DARK 2008  
 Dept of Information Technology| www.it.uu.se  
 © Erik Hagersten| user.it.uu.se/~eh

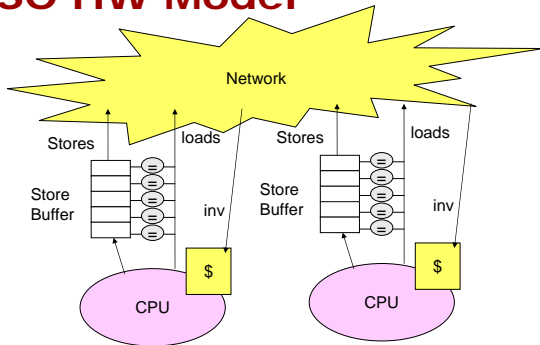
## "Almost intuitive memory model" Total Store Ordering (P. Sindhu)



- Global order achieved by interleaving all store accesses from different threads
- "Programmer's intuition is maintained"
  - Store causality? Yes
  - Does Dekker work? No
- Unnecessarily restrictive ==> performance penalty

DARK 2008  
 Dept of Information Technology| www.it.uu.se  
 © Erik Hagersten| user.it.uu.se/~eh

## TSO HW Model



→ Stores are moved off the critical path  
 Coherence implementation can be the same as for SC

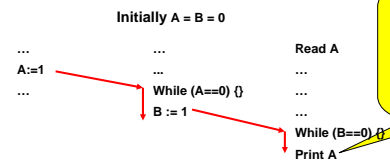
DARK 2008  
 Dept of Information Technology| www.it.uu.se  
 © Erik Hagersten| user.it.uu.se/~eh

## TSO

- Flag synchronization works

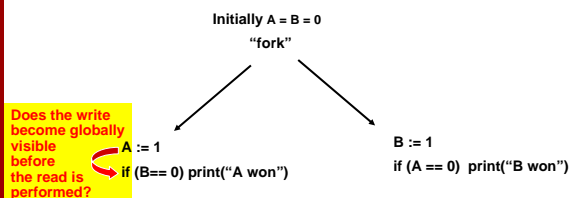
```
A := data      while (flag != 1) {};  
flag := 1      X := A
```

- Provides causal correctness



DARK 2008  
 Dept of Information Technology| www.it.uu.se  
 © Erik Hagersten| user.it.uu.se/~eh

## Dekker's Algorithm

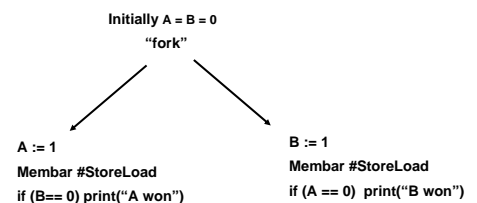


Q: Is it possible that both A and B wins?

- Left: The read (i.e., test if B==0) can bypass the store (A:=1)
- Right: The read (i.e., test if A==0) can bypass the store (B:=1)
- both loads can be performed before any of the stores
- yes, it is possible that both wins
- Dekker's algorithm breaks

DARK 2008  
 Dept of Information Technology| www.it.uu.se  
 © Erik Hagersten| user.it.uu.se/~eh

## Dekker's Algorithm for TSO

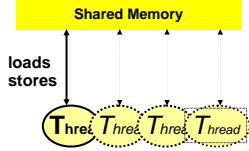


Q: Is it possible that both A and B win?

- Membar: The read is stored after all previous stores have been "globally ordered"
- behaves like SC
- Dekker's algorithm works!

DARK 2008  
 Dept of Information Technology| www.it.uu.se  
 © Erik Hagersten| user.it.uu.se/~eh

## Weak/release Consistency (M. Dubois, K. Gharachorloo)



- Most accesses are unordered
  - "Programmer's intuition is not maintained"
    - Store causality? No
    - Does Dekker work? No
  - Global order only established when the programmer explicitly inserts memory barrier instructions
- ++ Better performance!!  
--- Interesting bugs!!

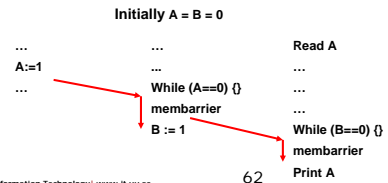
61

## Weak/Release consistency

- New flag synchronization needed

```
A := data;           while (flag != 1) {};
membarrier;         membarrier;
flag := 1;           X := A;
```

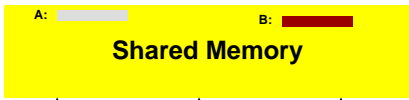
- Dekker's: same as TSO
- Causal correctness provided for this code



**Q:** What value will get printed?  
**Answer:** 1

62

## Example1: Causal Correctness Issues



```
Thread 1: Read A
           A := 1
           ...

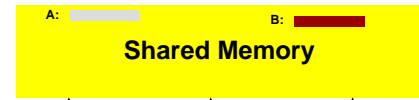
Thread 2: Write B
           ...
           While (A==0) {}
           B := 1

Thread 3: Read A
           ...
           While (B==0) {}
           Print A
```

What is the value of A?

63

## Example1: Causal Correctness Issues



```
Thread 1: Read A
           A := 1
           ...

Thread 2: Write B
           ...
           While (A==0) {}
           B := 1

Thread 3: Read A
           ...
           While (B==0) {}
           Print A
```

64

## Example1: Causal Correctness Issues



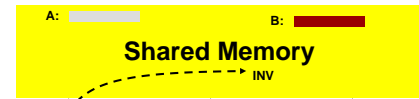
```
Thread 1: Read A
           A := 1
           ...

Thread 2: Write B
           ...
           While (A==0) {}
           B := 1

Thread 3: Read A
           ...
           While (B==0) {}
           Print A
```

65

## Example1: Causal Correctness Issues



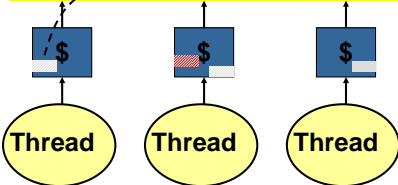
```
Thread 1: Read A
           A := 1
           ...

Thread 2: Write B
           ...
           While (A==0) {}
           B := 1

Thread 3: Read A
           ...
           While (B==0) {}
           Print A
```

66

### Example1: Causal Correctness Issues



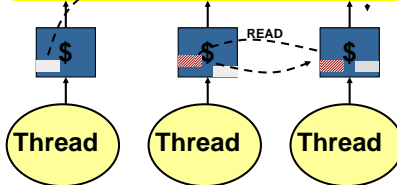
```

Thread 1: Read A
           A:=1
           ...

Thread 2: Write B
           ...
           While (A==0) {}
           B := 1

Thread 3: Read A
           ...
           While (B==0) {}
           Print A
    
```

### Example1: Causal Correctness Issues



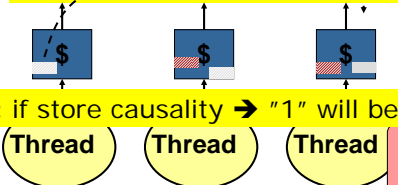
```

Thread 1: Read A
           A:=1
           ...

Thread 2: Write B
           ...
           While (A==0) {}
           B := 1

Thread 3: Read A
           ...
           While (B==0) {}
           Print A
    
```

### Example1: Causal Correctness Issues



A: if store causality → "1" will be printed

What is the value of A?  
It depends...

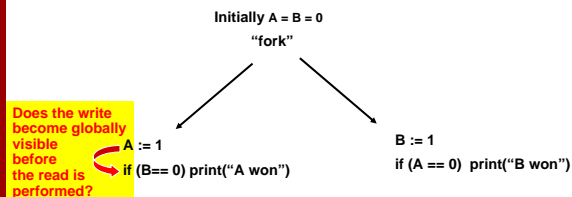
```

Thread 1: Read A
           A:=1
           ...

Thread 2: Write B
           ...
           While (A==0) {}
           B := 1

Thread 3: Read A
           ...
           While (B==0) {}
           Print A
    
```

### Dekker's Algorithm



Q: Is it possible that both A and B win?

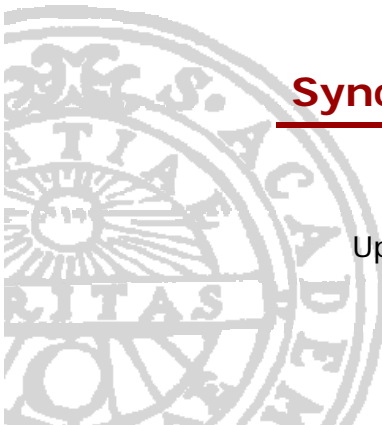
A: Only known if you know the memory model

### Learning more about memory models

Shared Memory Consistency Models: A Tutorial by Sarita Adve, Kourosh Gharachorloo in IEEE Computer 1996 (in the "Papers" directory)

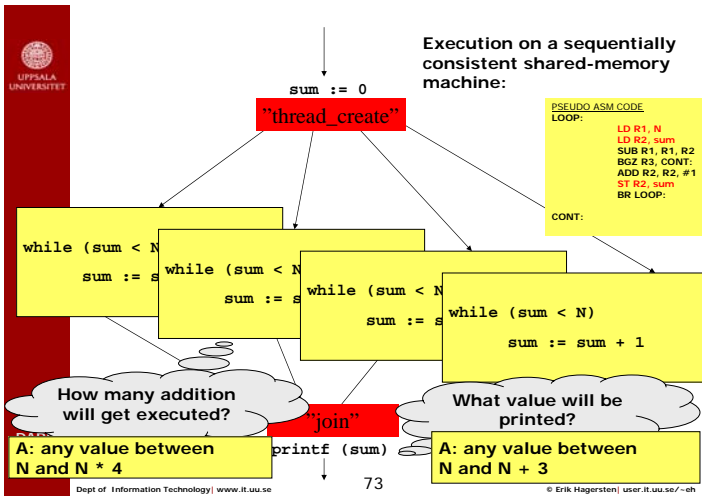
RFM: Read the F\*\*\*\*\*n Manual of the system you are working on!  
(Different microprocessors and systems supports different memory models.)

**Issue to think about:** What code reordering may compilers really do?  
Have to use "volatile" declarations in C.



### Synchronization

Erik Hagersten  
Uppsala University  
Sweden



UPPSALA UNIVERSITET

## Need to introduce synchronization

- Locking primitives are needed to ensure that only one process can be in the critical section:

```

LOCK(lock_variable) /* wait for your turn */
if (sum > threshold) {
    sum := my_sum + sum
}
UNLOCK(lock_variable) /* release the lock*/

```

Critical Section

```

if (sum > threshold) {
    LOCK(lock_variable) /* wait for your turn */
    sum := my_sum + sum
    UNLOCK(lock_variable) /* release the lock*/
}

```

Critical Section

DARK 2008

Dept of Information Technology| www.it.uu.se

74

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

## Components of a Synchronization Event

- Acquire method
  - Acquire right to the synch (enter critical section, go past event)
- Waiting algorithm
  - Wait for synch to become available when it isn't
- Release method
  - Enable other processors to acquire right to the synch

DARK 2008

Dept of Information Technology| www.it.uu.se

75

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

## Atomic Instruction to Acquire

**Atomic example: test&set "TAS" (SPARC: LDSTB)**

- The value at Mem(lock\_addr) loaded into the specified register
- Constant "1" atomically stored into Mem(lock\_addr) (SPARC: "FF")
- Software can determine if won (i.e., set changed the value from 0 to 1)
- Other constants could be used instead of 1 and 0

**Looks like a store instruction to the caches/memory system**

**Implementation:**

- Get an exclusive copy of the cache line
- Make the atomic modification to the cached copy

**Other read-modify-write primitives can be used too**

- Swap (SWAP): atomically swap the value of REG with Mem(lock\_addr)
- Compare&swap (CAS): SWAP if Mem(lock\_addr) == REG2

DARK 2008

Dept of Information Technology| www.it.uu.se

76

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

## Waiting Algorithms

**Blocking**

- Waiting processes/threads are de-scheduled
- High overhead
- Allows processor to do other things

**Busy-waiting**

- Waiting processes repeatedly test a lock\_variable until it changes value
- Releasing process sets the lock\_variable
- Lower overhead, but consumes processor resources
- Can cause network traffic

**Hybrid methods:** busy-wait a while, then block

DARK 2008

Dept of Information Technology| www.it.uu.se

77

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

## Release Algorithm

- Typically just a store "0"
- More complicated locks may require a conditional store or a "wake-up".

DARK 2008

Dept of Information Technology| www.it.uu.se

78

© Erik Hagersten| user.it.uu.se/~eh

## A Bad Example: "POUNDING"

```
proc lock(lock_variable) {
  while (TAS[lock_variable]==1) {} /* bang on the lock until free */
}
```

```
proc unlock(lock_variable) {
  lock_variable := 0
}
```

Assume: The function TAS (test and set)  
-- returns the current memory value and **atomically**  
writes the busy pattern "1" to the memory

**Generates too much traffic!!**  
-- spinning threads produce traffic!

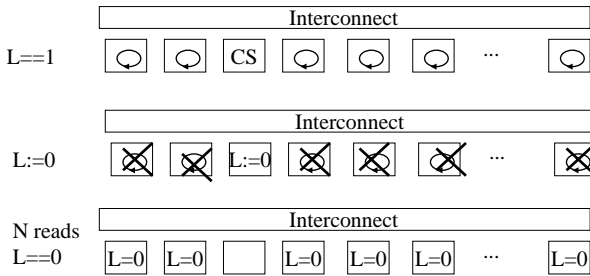
## Optimistic Test&Set Lock "spinlock"

```
proc lock(lock_variable) {
  while true {
    if (TAS[lock_variable] == 0) break; /* bang on the lock once, done if TAS==0 */
    while(lock_variable != 0) {} /* spin locally in your cache until "0" observed*/
  }
}
```

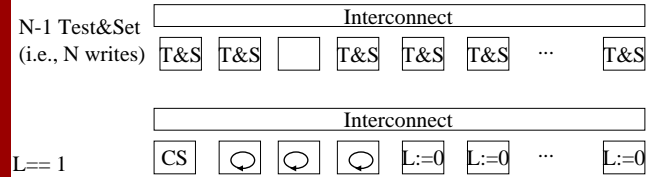
```
proc unlock(lock_variable) {
  lock_variable := 0
}
```

**Much less coherence traffic!!**  
-- still lots of traffic at lock handover!

## It could still get messy!



## ...messy (part 2)



potentially:  $\sim N^2/2$  reads :-)

Problem1: Contention on the interconnect slows down the CS proc

Problem2: The lock hand-over time is  $N \times \text{read\_throughput}$

Fix1: Some back-off strategy, bad news for hand-over latency

Fix1: Queue-based locks

## Could Get Even Worse on a NUMA

- Poor communication latency
- Serialization of accesses to the same cache line
- WF: added hardware optimization:
  - TAS can bypass loads in the coherence protocol
  - ==>N-2 loads queue up in the protocol
  - ==> the winner's atomic TAS will bypass the loads
  - ==>the loads will return "busy"

## Ticket-based queue locks: "ticket"

```
proc lock(lstruct) {
  int my_num;
  my_num := INC(lstruct.ticket) /* get your unique number*/
  while(my_num != lstruct.nowserving) {} /* wait here for your turn */
}
```

```
proc unlock(lstruct) {
  lstruct.nowserving++ /* next in line please */
}
```

**Less traffic at lock handover!**

## Ticket-based back-off "TBO"

```

proc lock(lstruct) {
    int my_num;
    my_num := INC(lstruct.ticket) /* get your number*/
    while(my_num != lstruct.now_serving) { /* my turn ?*/
        idle_wait(lstruct.now_serving - my_num) /* do other shopping */
    }
}

proc unlock(lock_struct) {
    lock_struct.now_serving++ /* next in line please */
}
    
```

Even less traffic at lock handover!

## Queue-based lock: CLH-lock -- a variation of the MCS lock

"Initially, each process owns one global flag, pointed to by private \*I and \*P  
The global lock is pointed to by global \*L  
1) Initialize the \*I flag to busy (= "1")  
2) Atomically, make \*L point our "flag" and make \*P point where \*L pointed  
3) Wait until \*P points to a "0"

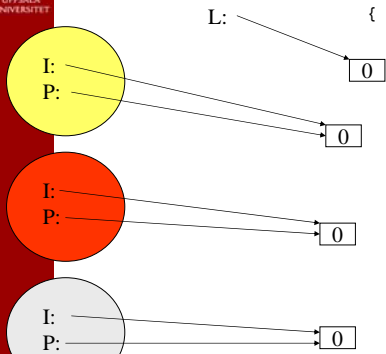
```

proc lock(int **L, **I, **P)
{
    **I = 1; /* initialized as "busy"*/
    atomic_swap (*P = *L; *L = *P)
    /* P now stores a pointer to the flag L pointed to*/
    /* L now stores a pointer to our flag */
    while (**P != 0){} /* keep spinning until prev owner releases lock*/
}
    
```

```

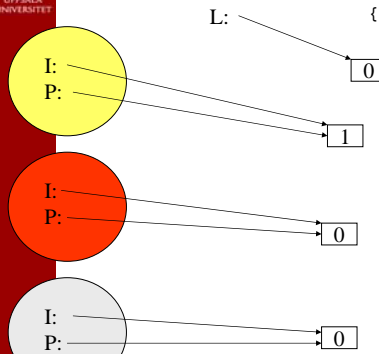
proc unlock(int **I, **P)
{
    **I = 0; /* release the lock */
    *I = *P; /* next time *I to reuse the previous guys flag*/
}
    
```

## CLH lock



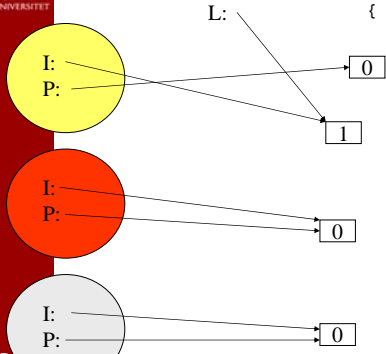
```

proc lock(int **L, **I, **P)
{
    **I = 1 /* init to "busy"*/
    atomic_swap (*P = *L; *L = *P)
    /* *L now points to our I* */
    while (**P != 0){} }
    /* spin until prev is done */
}
    
```



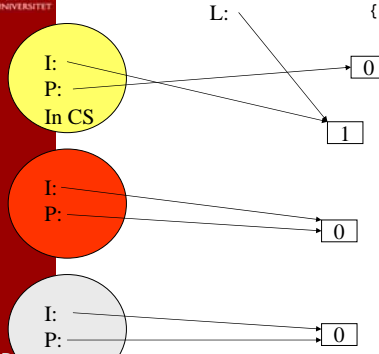
```

proc lock(int **L, **I, **P)
{
    **I = 1 /* init to "busy"*/
    atomic_swap (*P = *L; *L = *P)
    /* *L now point to our I* */
    while (**P != 0){} }
    /* spin until prev is done */
}
    
```



```

proc lock(int **L, **I, **P)
{
    **I = 1 /* init to "busy"*/
    atomic_swap (*P = *L; *L = *P)
    /* *L now point to our I* */
    while (**P != 0){} };
    /* spin until prev is done */
}
    
```



```

proc lock(int **L, **I, **P)
{
    **I = 1 /* init to "busy"*/
    atomic_swap (*P = *L; *L = *P)
    /* *L now point to our I* */
    while (**P != 0){} ;
    /* spin until prev is done */
}
    
```

UPPSALA UNIVERSITET

```

proc lock(int **L, **I, **P)
{
  **I = 1 /* init to "busy"*/
  atomic_swap { *P = *L; *L = *P }
  /* *L now point to our I* */
  while (**P != 0){};
  /* spin unit prev is done */
}

```

Dept of Information Technology| www.it.uu.se

91

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

```

proc lock(int **L, **I, **P)
{
  **I = 1 /* init to "busy"*/
  atomic_swap { *P = *L; *L = *P; }
  /* *L now point to our I* */
  while (**P != 0){};
  /* spin unit prev is done */
}

```

Dept of Information Technology| www.it.uu.se

92

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

```

proc lock(int **L, **I, **P)
{
  **I = 1 /* init to "busy"*/
  atomic_swap { *P = *L; *L = *P; }
  /* *L now point to our I* */
  while (**P != 0){};
  /* spin unit prev is done */
}

```

Dept of Information Technology| www.it.uu.se

93

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

```

proc unlock(int **I, **P)
{
  **I = 0;
  /* release the lock */
  *I = *P; }
  /* reuse the previous guy's *P*/
}

```

Dept of Information Technology| www.it.uu.se

94

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

```

proc unlock(int **I, **P)
{
  **I = 0;
  /* release the lock */
  *I = *P; }
  /* reuse the previous guy's *P*/
}

```

Dept of Information Technology| www.it.uu.se

95

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

```

proc unlock(int **I, **P)
{
  **I = 0;
  *I = *P; }
}

```

Dept of Information Technology| www.it.uu.se

96

© Erik Hagersten| user.it.uu.se/~eh



UPPSALA UNIVERSITET

**Minimizes traffic at lock handover!  
May be too fair for NUMAs ....**

```

proc unlock(int **I, **P)
{
  **I = 0;
  **P = *P;
}

```

DARK 2008

Dept of Information Technology| www.it.uu.se

97

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

**E6800 locks 12 CPUs**

DARK 2008

Dept of Information Technology| www.it.uu.se

98

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

**E6800 locks (exluding POUND)**

DARK 2008

Dept of Information Technology| www.it.uu.se

99

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

**NUMA: Non-uniform Comm Arch.**

WF

**Directory-latency = 6x snoop  
i.e., roughly CMP NUCA-ness**

DARK 2008

Dept of Information Technology| www.it.uu.se

100

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

**Trad. chart over lock performance on a hierarchical NUMA (round robin scheduling)**

**Benchmark:**  
for i = 1 to 10000 {  
lock(AL)  
A:= A + 1;  
unlock(AL)}

DARK 2008

Dept of Information Technology| www.it.uu.se

101

© Erik Hagersten| user.it.uu.se/~eh

UPPSALA UNIVERSITET

**Introducing RH locks**

**Benchmark:**  
for i = 1 to 10000 {  
lock(AL)  
A:= A + 1;  
unlock(AL)}

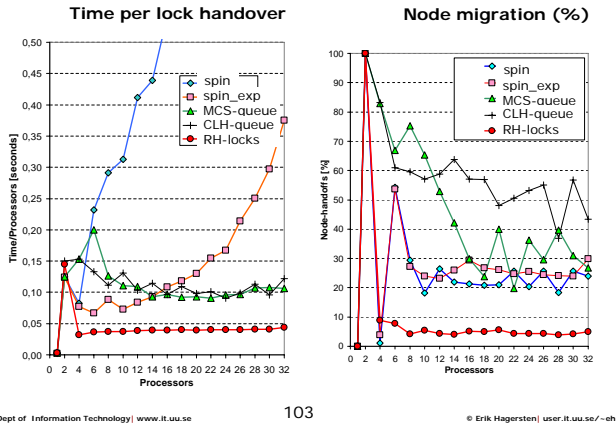
DARK 2008

Dept of Information Technology| www.it.uu.se

102

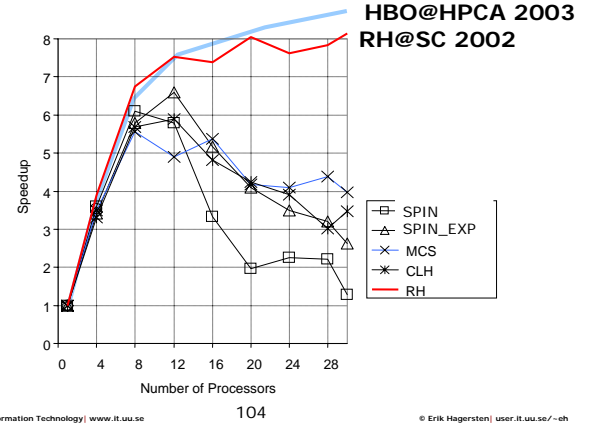
© Erik Hagersten| user.it.uu.se/~eh

## RH locks: encourages unfairness



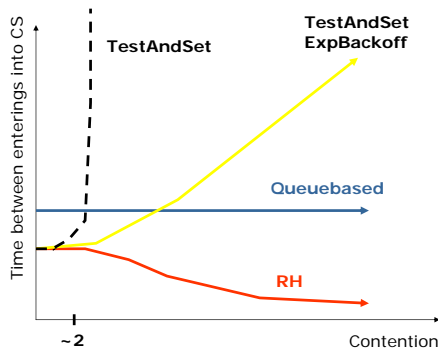
103

## Ex: Splash Raytrace Application Speedup



104

## Performance under contention



105

## Barriers: Make the first threads wait for the last thread to reach a point in the program

- Software algorithms implemented using locks, flags, counters
- Hardware barriers
  - Wired-AND line separate from address/data bus
  - Set input high when arrive, wait for output to be high to leave
  - (In practice, multiple wires to allow reuse)
  - Difficult to support arbitrary subset of processors
    - even harder with multiple processes per processor
  - Difficult to dynamically change number and identity of participants
    - e.g. latter due to process migration

106

## A Centralized Barrier

```

BARRIER (bar_name, p) {
int loops;
loops = 0;

local_sense = !(local_sense); /* toggle private sense variable
                               each time the barrier is used */

LOCK(bar_name.lock);
bar_name.counter++; /* globally increment the barrier count */
if (bar_name.counter == p) { /* everybody here yet? */
bar_name.flag = local_sense; /* release waiters */
UNLOCK(bar_name.lock);
}
else {
UNLOCK(bar_name.lock);
while (bar_name.flag != local_sense) { /* wait for the last guy */
if (loops++ > UNREASONABLE) report_warning(pid)
}
}
}
    
```

107

## Centralized Barrier Performance

- Latency
    - Want short critical path in barrier
    - Centralized has critical path length at least proportional to  $p$
  - Traffic
    - Barriers likely to be highly contended, so want traffic to scale well
    - About  $3p$  bus transactions in centralized
  - Storage Cost
    - Very low: centralized counter and flag
  - Key problems for centralized barrier are latency and traffic
    - Especially with distributed memory, traffic goes to same node
- Hierarchical barriers

108

## New kind of synchronization: Transactional Memory

- OLD: lock(ID); unlock(ID) around critical sections
- NEW: start\_transaction; end\_transaction around "critical sections" (note: no ID!!)
  - Underlying mechanism to guarantee atomic behavior
  - This is not the same as guaranteeing that only one thread is in the critical action!!
  - Supported in HW (soon) or in SW (normally very inefficient)
- Suggested by Maurice Herlihy in 1993
- HW support announced by Sun and Intel (??)

## Support for TM

- Start\_transaction:
  - Save original state to allow for rollback (i.e., save register values)
- In critical section
  - Do not make any global state change
  - Detect "atomic violations" (others writing data you've read in CS or reading data you have written)
  - At atomic violation: roll-back to original state
  - Forward progress must be guaranteed
- End\_transaction
  - Atomically commit all changes performed in the critical section.

## Advantage of TM

- Do not have to "name" CS
- Less risk for deadlocks
- Performance:
  - Several thread can be in "the same" CS as long as they do not mess with each other
  - CS can often be large with a small performance penalty



## Introduction to Multiprocessors

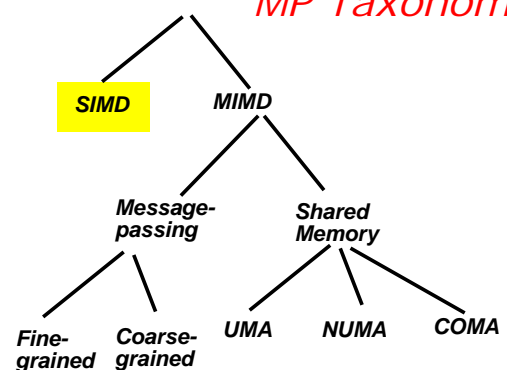
Erik Hagersten  
Uppsala University

## Flynn's Taxonomy

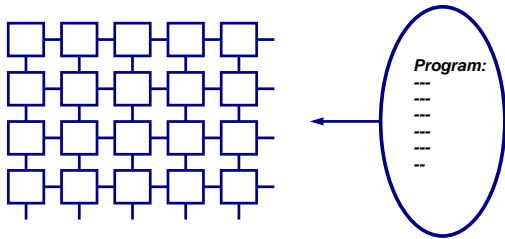
{Single, Multiple} Instruction +  
{Single, Multiple} Data

- SISD - Our good old "simple" CPUs
- SIMD - Vectors, "MMX", DSPs, CM-2, ...
- MIMD - TLP, cluster, shared-mem MP, ...
- MISD - Can't think of any...

## MP Taxonomy



## SIMD = "Dataparallelism"



## SIMD: Thinking Machine

- Connection Machine: CM1, CM2, CM200 (at KTH ~1990: CM200 "Bellman")
- One-bit ALU and a small local memory
- FP accelerator available
- Programmed in "ASM", \*C and \*Lisp
- Hard to program (in my opinion...)

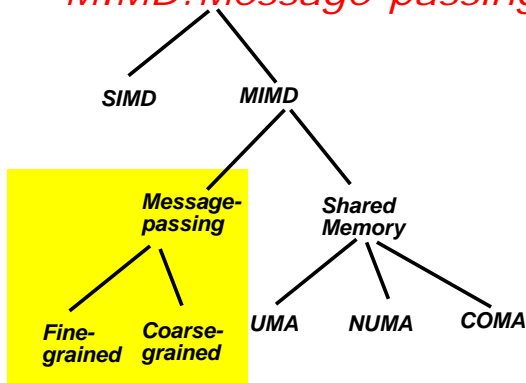
## Other Flavors of SIMD

- MMX/Altivec/VIS instructions/SSE...
  - Divide register content into smaller items (e.g., bytes)
  - Special instructions operate on all items in parallel, e.g., BYTE-COMPARE...
- Some DSPs (Digital Signal Processors)
- Some Image Processors

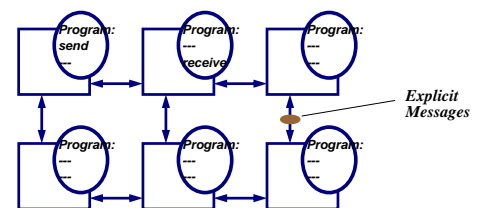
## Vector architectures CRAY, NEC, Fujitsu, ...

- Vector Processors
  - LD/ST operate on vectors of data
  - ALU Ops operate on vectors of data
- Example:
  - 8 "vector register" contains 64 vector entries each
  - A single LD/ST instr loads/stores entire vectors
  - A single ALU instr  $V1 \leftarrow V2 \text{ op } V3$
  - 64 bit mask vectors make execution conditional
  - Overlaps Mem and ALU ops
  - One form of "SIMD" -- Single Instruction Multiple Data

## MIMD: Message-passing



## Message-passing Arch MIMD



## Message-Passing HW

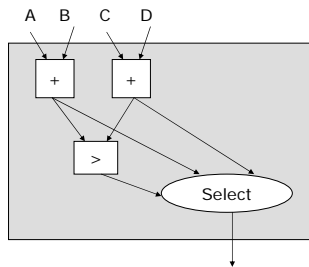
- Programmed in MPI or PVM (or HPFortran...)
  - Thinking Machines: CM5
  - Intel: Paragon
  - IBM: SP2
  - Meiko (Bristol, UK!!): CS2
  - Today: Clusters with high-speed interconnect (Important today, but not covered in this course)
- Clusters can be used as message-passing HW, but is most often used as capacity computing (i.e., throughput computing)

## Dataflow

- Often programmed in functional languages (e.g., ID)
- Compile program to Dataflow graph
- Operands + graph = executable
- Operation ready when the source operands are available

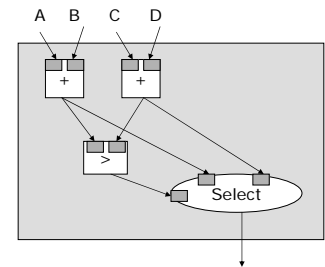
## Dataflow Example:

```
X := A + B
Y := C + D
If (X > Y)
  output X
else
  output Y
```



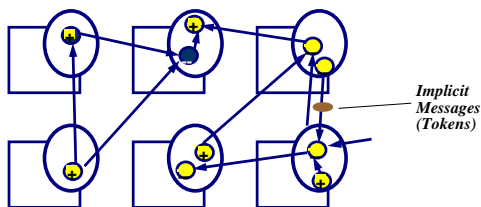
## Static Dataflow (Dennis)

```
X := A + B
Y := C + D
If (X > Y)
  output X
else
  output Y
```



Each operand executed exactly once per program  
Location assigned for each input data

## Fine-grained Message-passing Dataflow ==> Multithreading



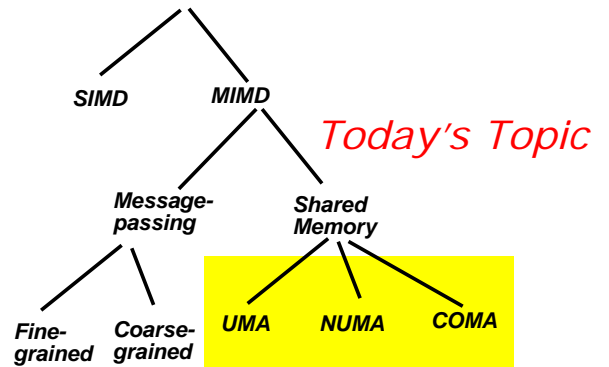
## Dynamic Dataflow (Arvind)

- Allows for recursion and loops
- Each invocation is assigned a "color"
- Pairs of operands are matched dynamically
  - Based on {Color, Operation}
  - In the Waiting-Matching Section (I.e., a cache)
- One problem: too much parallelism in the wrong place

## Carlstedts Elektornik

Gunnar Carlstedt, Staffan Truve' et al

- Processor "8601"
  - Gothenburg 1990-1997
  - Functional language "H"
  - Execution performed by a reduction a CAM memory
  - ALU rarely used
  - Many parallel processors on a wafer (Wafer-scale integration)
- CRT (Carlstedt Research Technology)



## The server market 1995

Server Size	High-Perf. Computing	Commercial Computing
<\$10k	1%	19%
<\$50k	5%	24%
<\$250k	5%	27%
<\$1M	2%	20%
>\$1M	3%	8%

UNIX shared-mem servers

The target of the rocket science supercomputers