

Multiprocessors and Coherent Memory

Erik Hagersten
Uppsala University



Goal for this course

- Understand **how and why** modern computer systems are designed the way they are:
 - pipelines
 - ✓ memory organization
 - ✓ virtual/physical memory ...
- Understand **how and why** multiprocessors are built
 - Cache coherence
 - Memory models
 - Synchronization...
- Understand **how and why** parallelism is created and
 - Instruction-level parallelism
 - Memory-level parallelism
 - Thread-level parallelism...
- Understand **how and why** multiprocessors of combined SIMD/MIMD type are built
 - GPU
 - Vector processing...
- Understand **how** computer systems are adopted to different usage areas
 - General-purpose processors
 - Embedded/network processors...
- Understand the physical limitation of modern computers
 - Bandwidth
 - Energy
 - Cooling...

AVDARK
2010

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

2

© Erik Hagersten | user.it.uu.se/~eh



Schedule in a nutshell

1. **Memory Systems** (~Appendix C in 4th Ed)
Caches, VM, DRAM, microbenchmarks, optimizing SW
2. **Multiprocessors**
TLP: coherence, memory models, synchronization
3. **Scalable Multiprocessors**
Scalability, implementations, programming, ...
4. **CPUs**
ILP: pipelines, scheduling, superscalars, VLIWs, SIMD instructions...
5. **Widening + Future** (~Chapter 1 in 4th Ed)
Technology impact, GPUs, Network processors, **Multicores (!!)**

AVDARK
2010

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

3

© Erik Hagersten | user.it.uu.se/~eh



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!



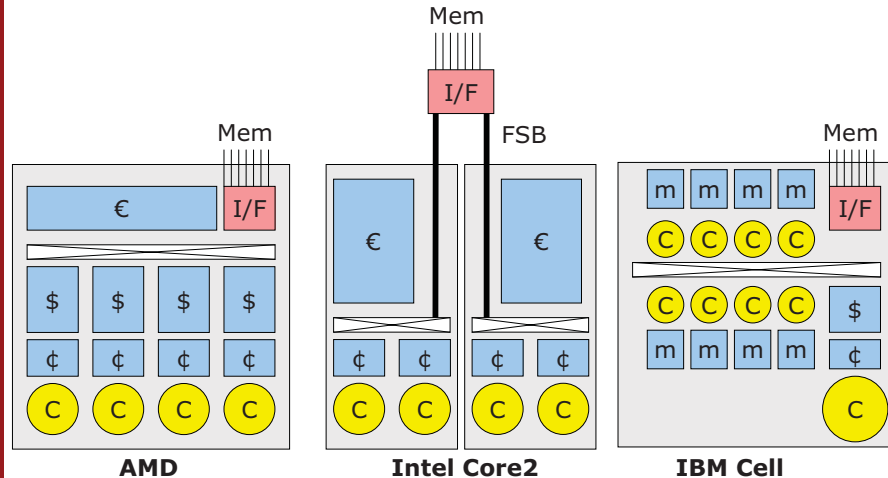
AVDARK
2010

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

4

© Erik Hagersten | user.it.uu.se/~eh

Multicore: Who has not got one?



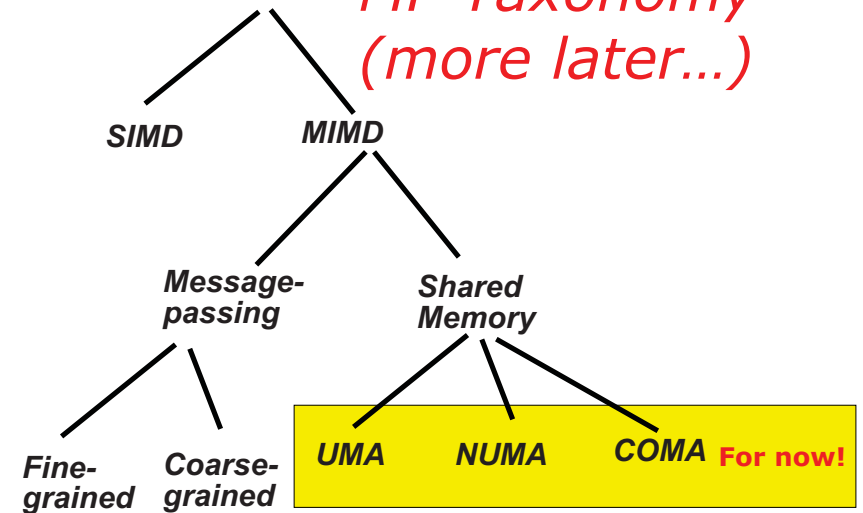
AVDARK
2010

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

5

© Erik Hagersten | user.it.uu.se/~eh

MP Taxonomy (more later...)



AVDARK
2010

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

6

© Erik Hagersten | user.it.uu.se/~eh

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...**

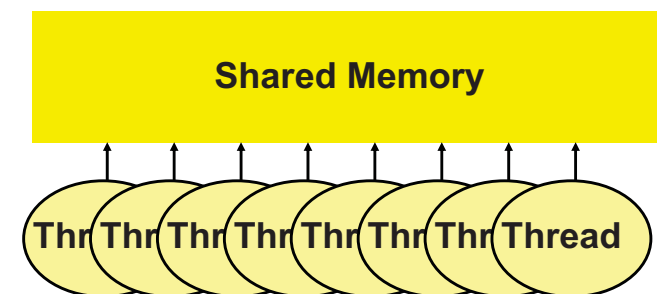
AVDARK
2010

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

7

© Erik Hagersten | user.it.uu.se/~eh

Programming Model:



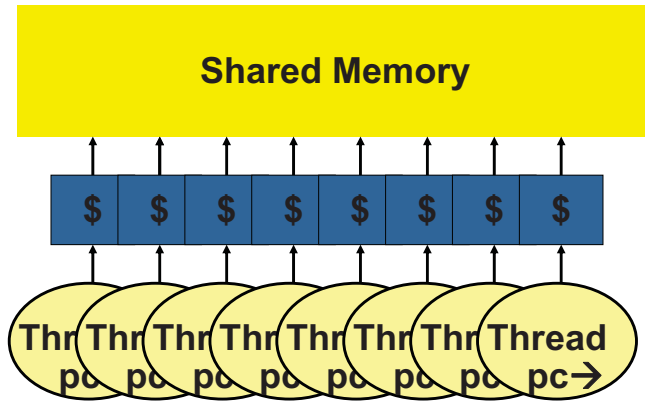
AVDARK
2010

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

8

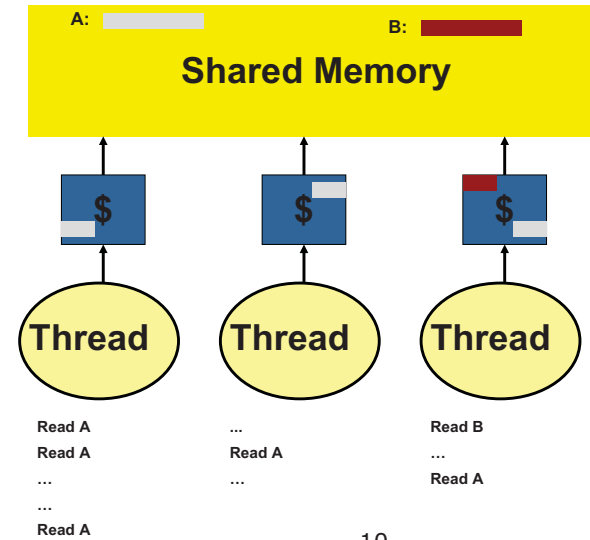
© Erik Hagersten | user.it.uu.se/~eh

Adding Caches: More Concurrency



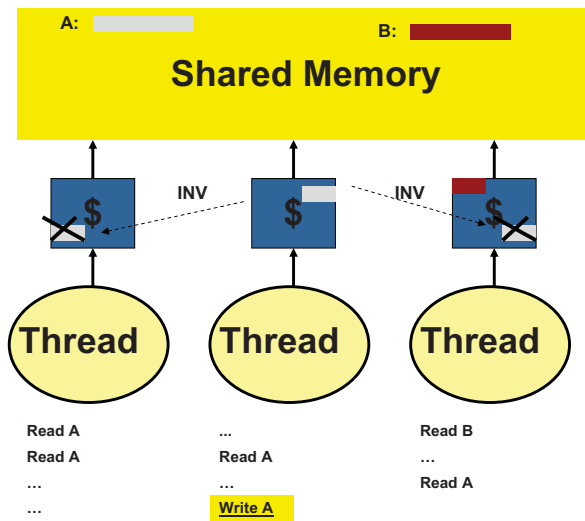
AVDARK
2010

Caches: Automatic Replication of Data



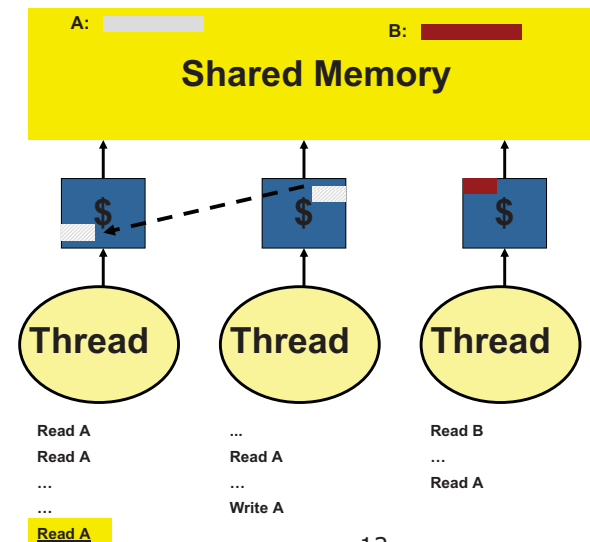
AVDARK
2010

The Cache Coherent Memory System



AVDARK
2010

The Cache Coherent \$2\$



AVDARK
2010

Summing up Coherence

There can be many copies of a datum, but only one value

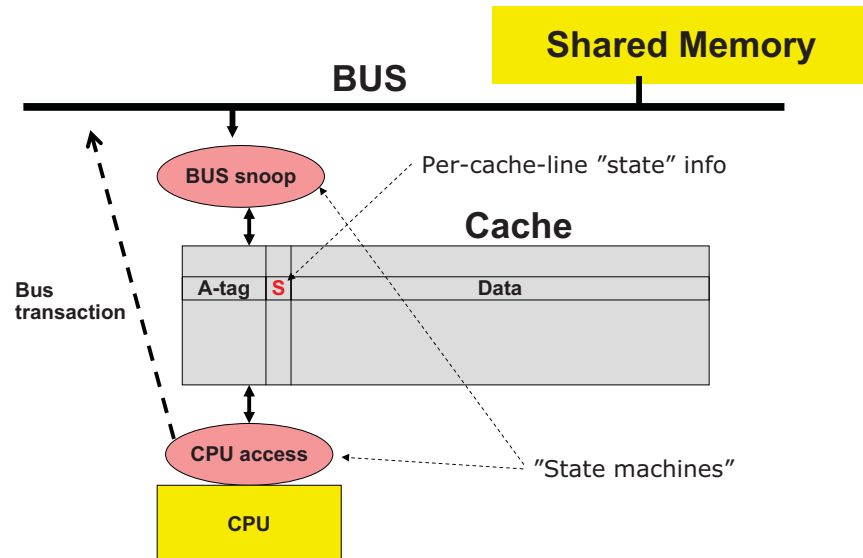
Too strong definition!

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

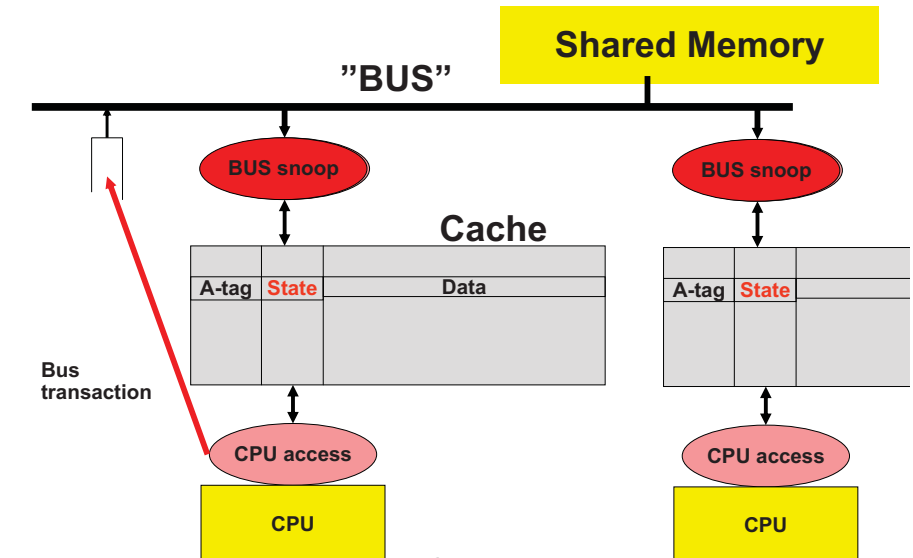
Implementation options for memory coherence

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

Snoop-based Protocol Implementation



Snoop-based Protocol Implementation



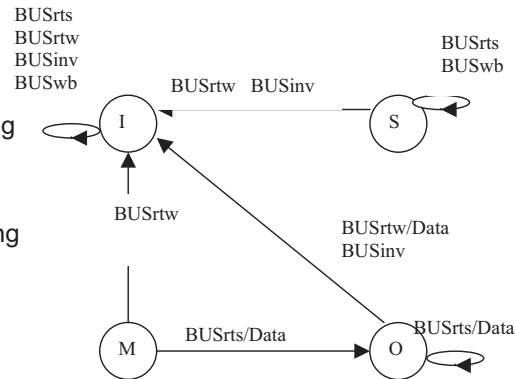
Example: Bus Snoop MOSI

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



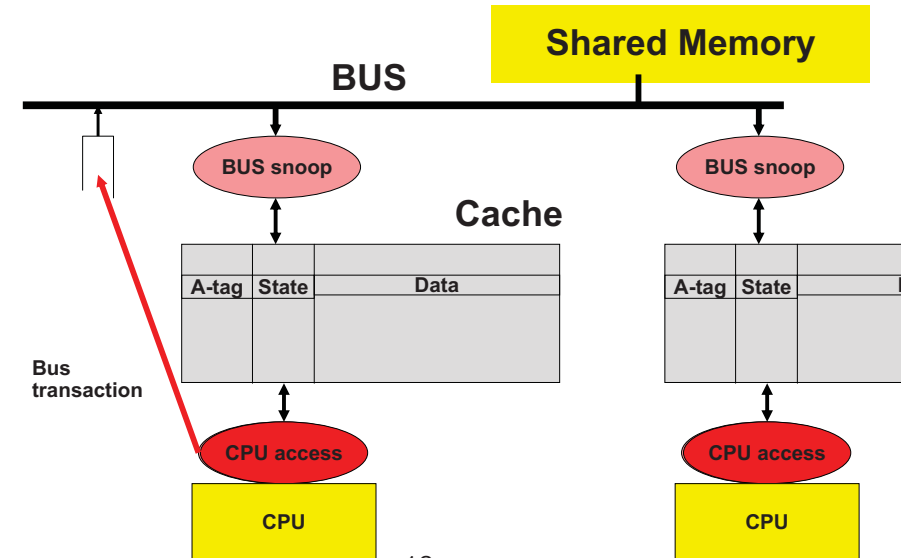
17

© Erik Hagersten | user.it.uu.se/~eh

AVDARK
2010

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

Snoop-based Protocol Implementation



18

© Erik Hagersten | user.it.uu.se/~eh

AVDARK
2010

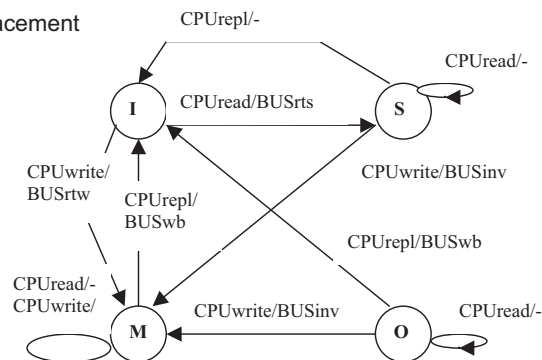
Dept of Information Technology | www.it.uu.se

Example: CPU access MOSI

CPUwrite: Caused by a store miss

CPUread Caused by a loadmiss

CPUrepl: Caused by a replacement



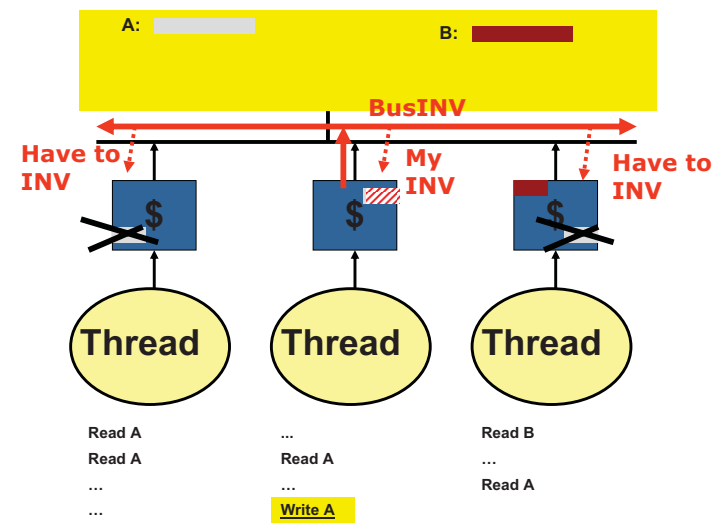
19

© Erik Hagersten | user.it.uu.se/~eh

AVDARK
2010

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

"Upgrade" in snoop-based



20

© Erik Hagersten | user.it.uu.se/~eh

AVDARK
2010

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

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" ☹

AVDARK
2010

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 ☹

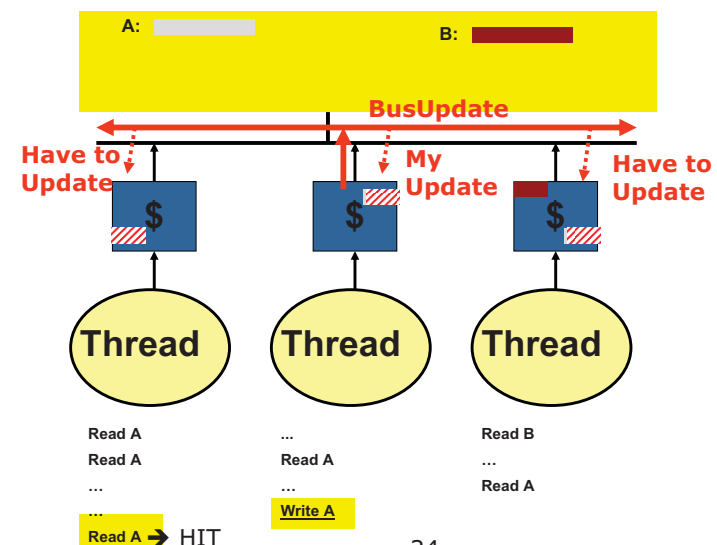
AVDARK
2010

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

AVDARK
2010

Update in MOSI snoop-based

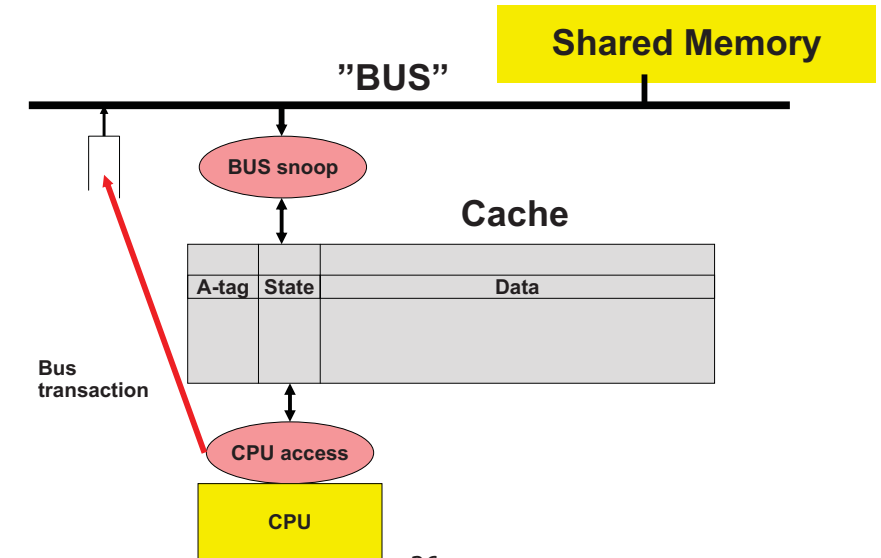


AVDARK
2010

Implementing Coherence (and Memory Models...)

Erik Hagersten
Uppsala University
Sweden

Snoop-based Protocol Implementation



AVDARK
2010

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

26

© Erik Hagersten | user.it.uu.se/~eh

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

AVDARK
2010

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

27

© Erik Hagersten | user.it.uu.se/~eh

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

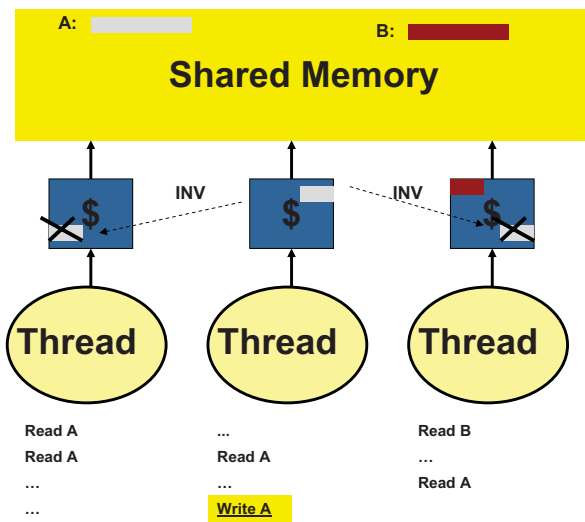
AVDARK
2010

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

28

© Erik Hagersten | user.it.uu.se/~eh

The Cache Coherent Memory System



29

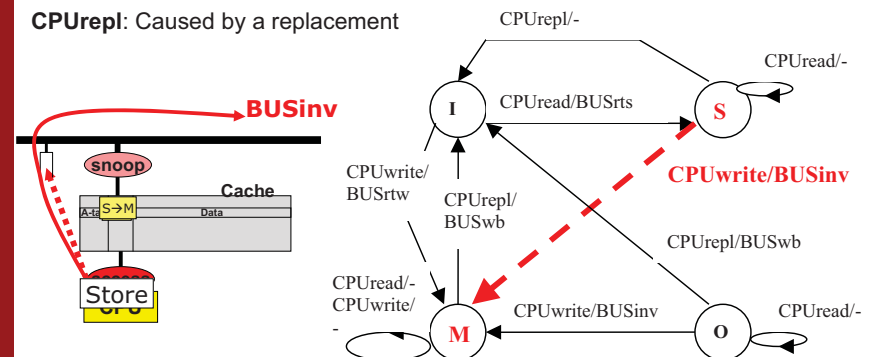
© Erik Hagersten | user.it.uu.se/~eh

Upgrade – the requesting CPU

CPUwrite: Caused by a store miss

CPUread: Caused by a loadmiss

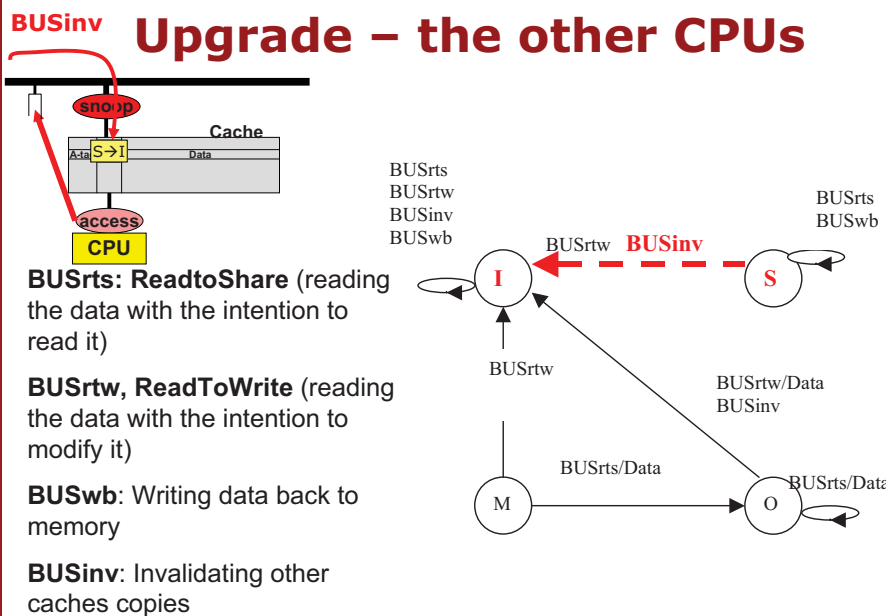
CPUrepl: Caused by a replacement



30

© Erik Hagersten | user.it.uu.se/~eh

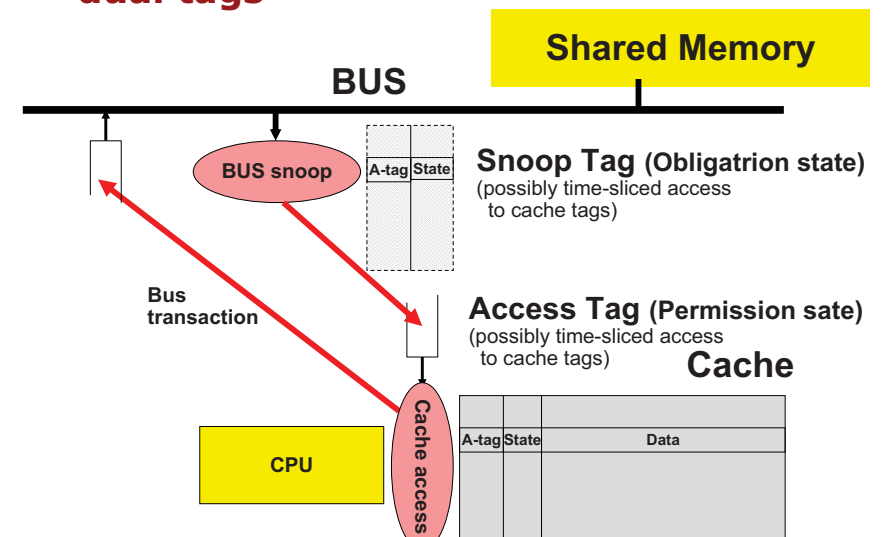
Upgrade – the other CPUs



31

© Erik Hagersten | user.it.uu.se/~eh

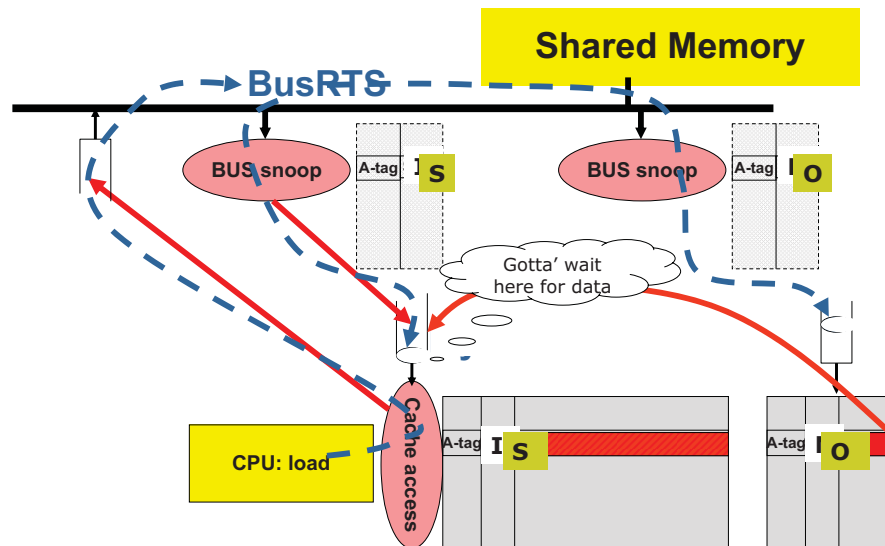
Modern snoop-based architecture -- dual tags



32

© Erik Hagersten | user.it.uu.se/~eh

Cache-to-cache in snoope-based



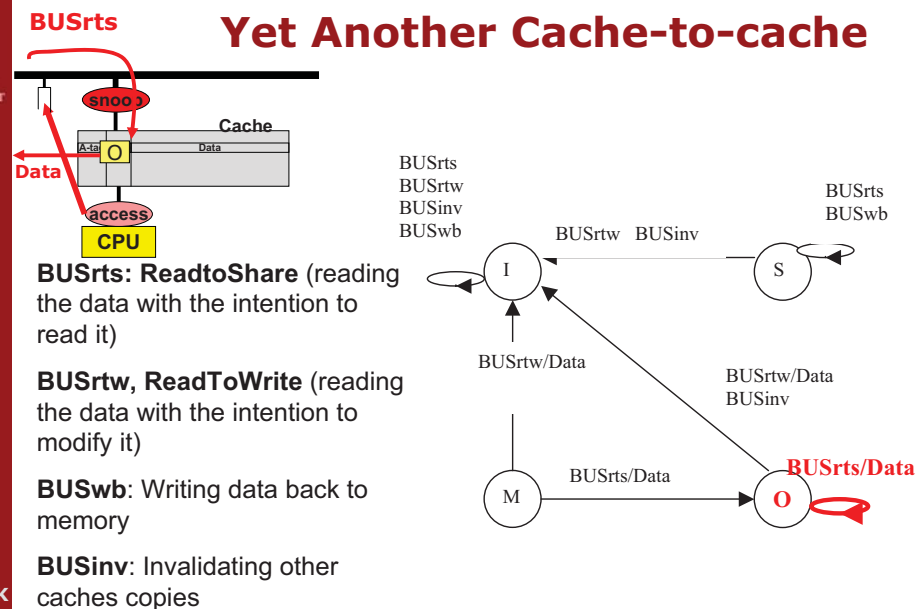
37

© Erik Hagersten | user.it.uu.se/~eh

AVDARK
2010

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

Yet Another Cache-to-cache



38

© Erik Hagersten | user.it.uu.se/~eh

AVDARK
2010

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

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`, CPU 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).

39

© Erik Hagersten | user.it.uu.se/~eh

AVDARK
2010

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

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 A B		CPU2 A B		CPU3 A B		
Initially		I	I	I	I	I	I	
CPU1: LD A	RTS(A)	S/I						Mem
CPU2: LD B	RTS(B)				S/O			Mem
...some time elapses .								
CPU1: replace A	-	I						-
CPU2: replace B	-				I			-

40

© Erik Hagersten | user.it.uu.se/~eh

AVDARK
2010

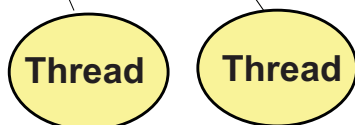
Dept of Information Technology | www.it.uu.se

False sharing



Cache Line

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

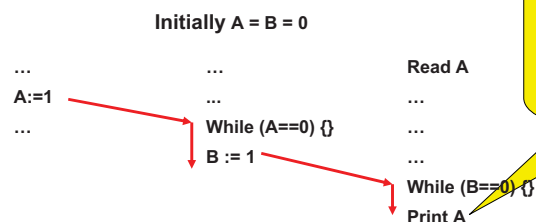
41

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

Erik Hagersten
Uppsala University
Sweden

Memory Ordering

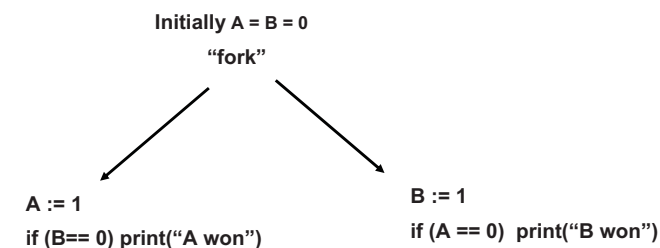
- Coherence defines a per-datum valuechange order
- Memory model defines the valuechange order for all the data.



Q: What
value will
get printed?

43

Dekker's Algorithm



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

44

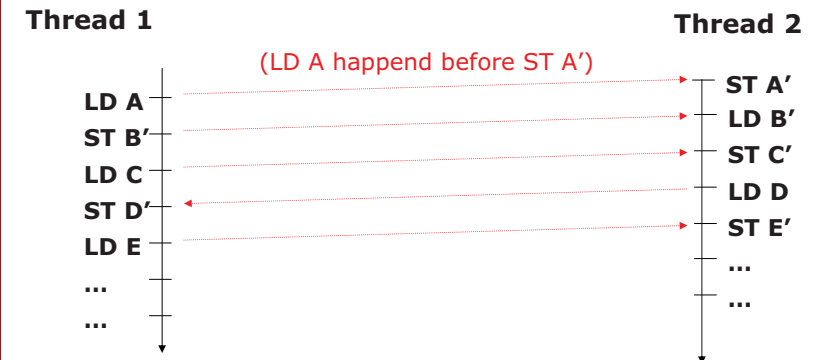
Memory Ordering

- 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

AVDARK
2010

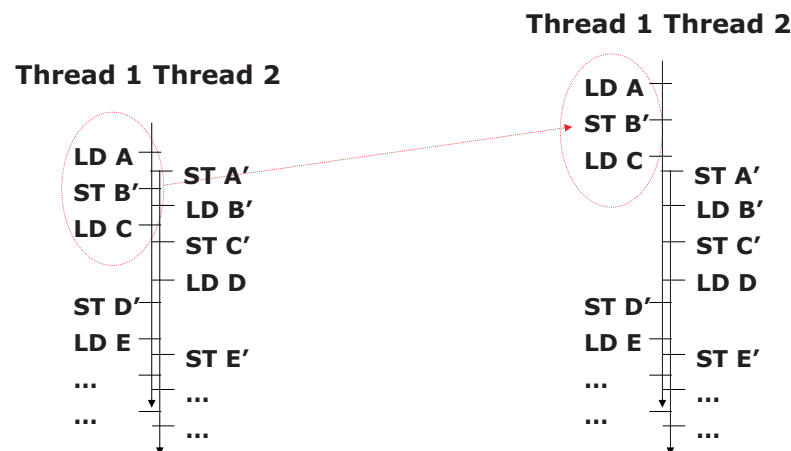
In which order were these threads executed?

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



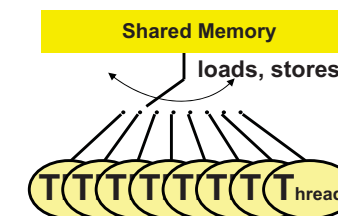
AVDARK
2010

One possible observed order Another possible observed order



AVDARK
2010

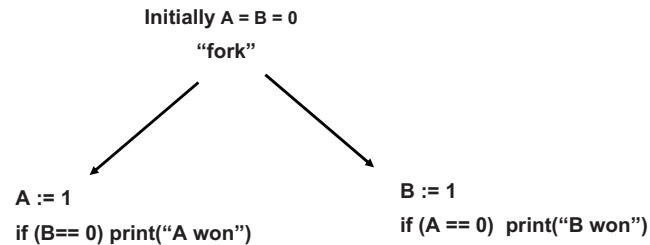
"The intuitive memory order" Sequential Consistency (Lamport)



- 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

AVDARK
2010

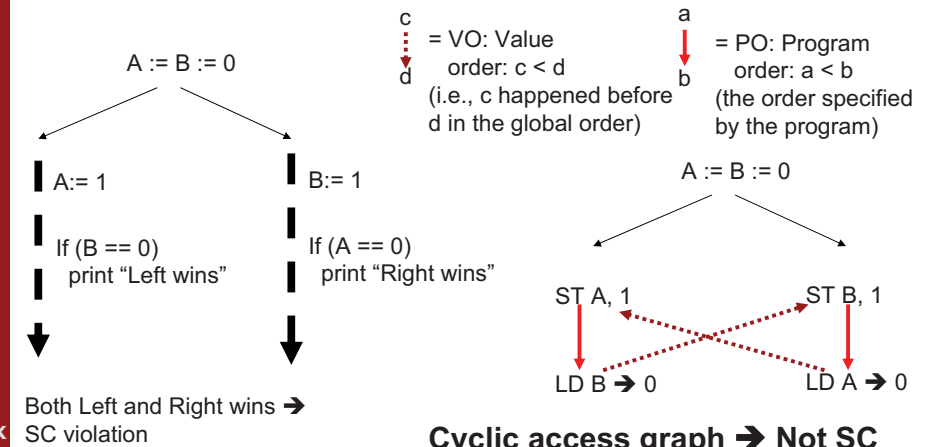
Dekker's Algorithm



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

Sequential Consistency (SC) Violation → Dekker: both wins

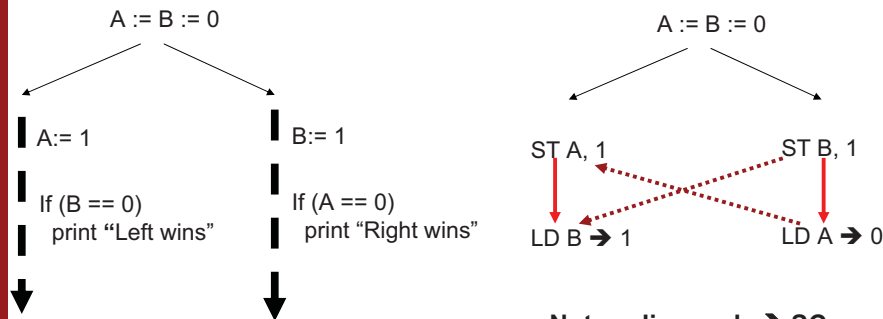
Access graph



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

SC is OK if one thread wins

Only Right wins → SC is OK

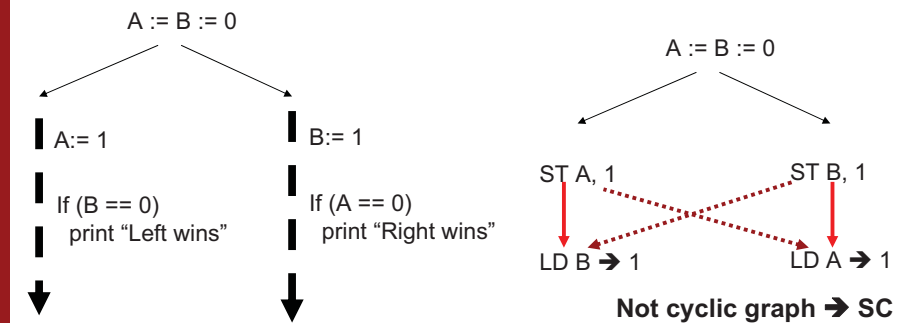


Not cyclic graph → SC

One global order:
STB < LDA < STA < LDB

SC is OK if no thread wins

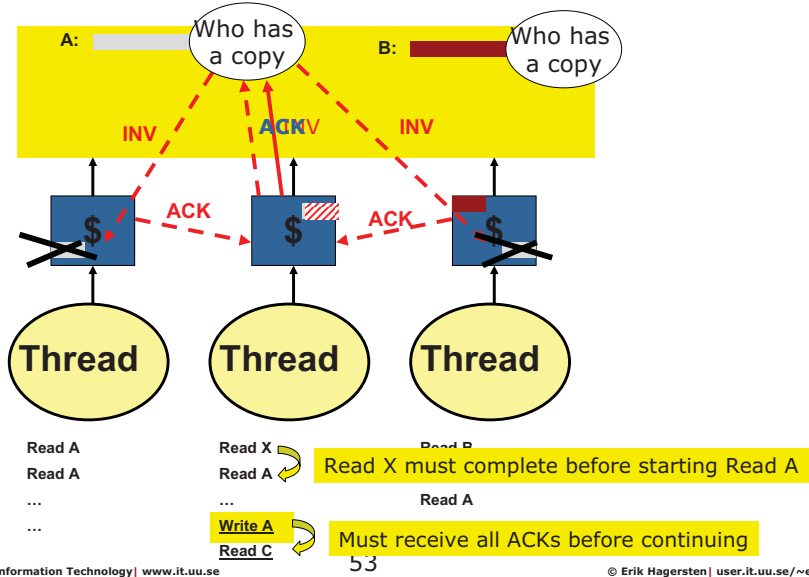
No thread wins → SC is OK



Not cyclic graph → SC

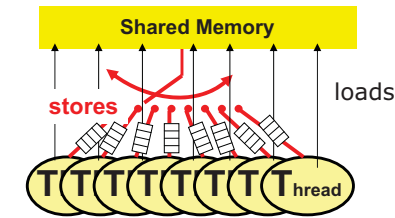
Four Partial Orders, still SC
STB < LDA ; STA < LDA ; STB < LDB ; STA < LDA

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



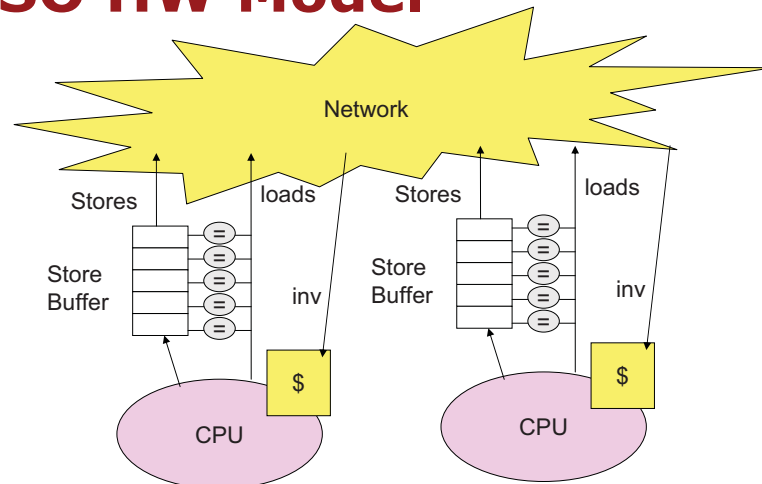
“Almost intuitive memory model”

Total Store Ordering [TSO] (P. Sindhu)



- Global *interleaving* [order] for all stores from different threads (own stores excepted)
- “Programmer’s intuition is maintained”
 - Store causality? Yes
 - Does Dekker work? No
- Unnecessarily restrictive ==> performance penalty

TSO HW Model



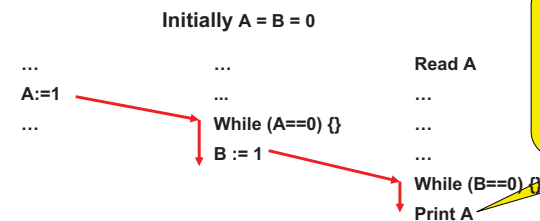
→ Stores are moved off the critical path
Coherence implementation can be the same as for SC

TSO

- Flag synchronization works

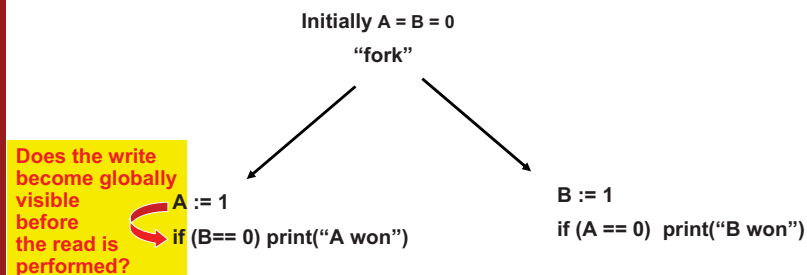
```
A := data      while (flag != 1) {};  
flag := 1      X := A
```

- Provides causal correctness



Q: What value will get printed?
Answer: 1

Dekker's Algorithm, TSO

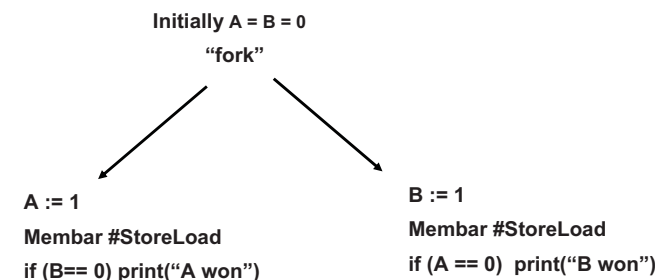


Q: Is it possible that both A and B wins?

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

57

Dekker's Algorithm for TSO

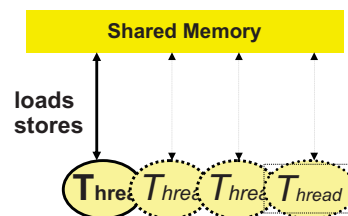


Q: Is it possible that both A and B win?

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

58

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!!

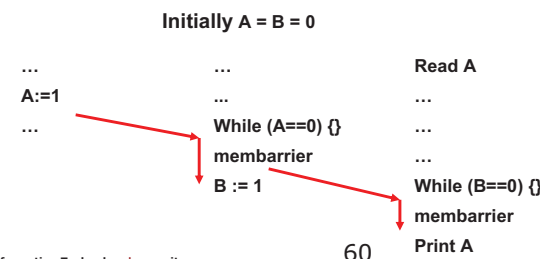
59

Weak/Release consistency

- New flag synchronization needed
- ```

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

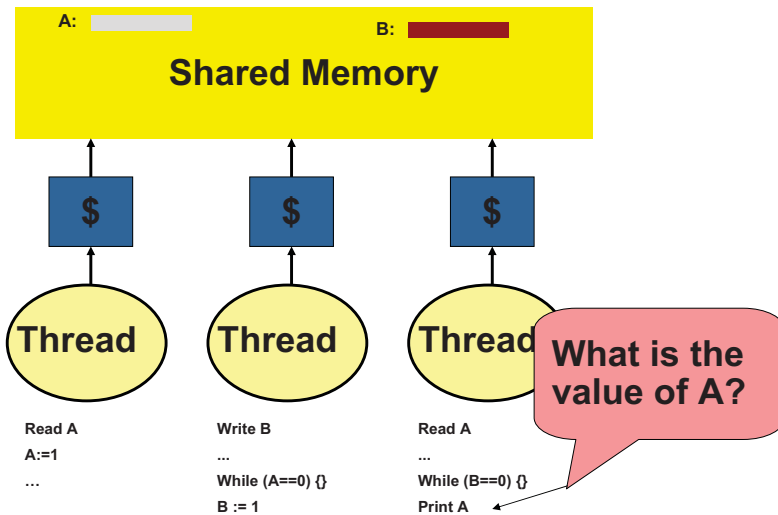
```
- Dekker's: same as TSO
  - Causal correctness provided for this code



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

60

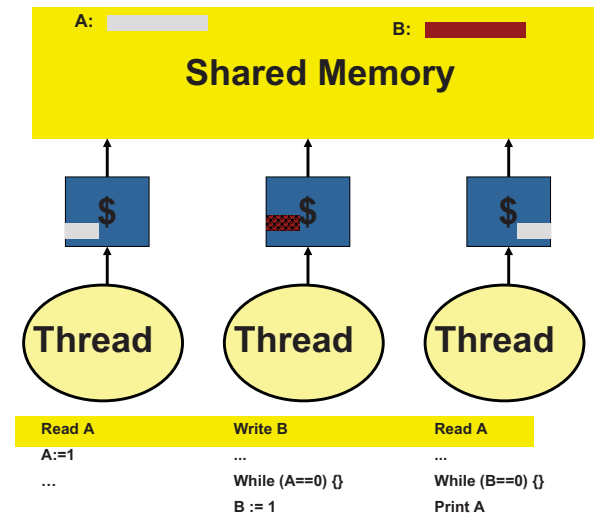
## Example1: Causal Correctness Issues



61

© Erik Hagersten | user.it.uu.se/~eh

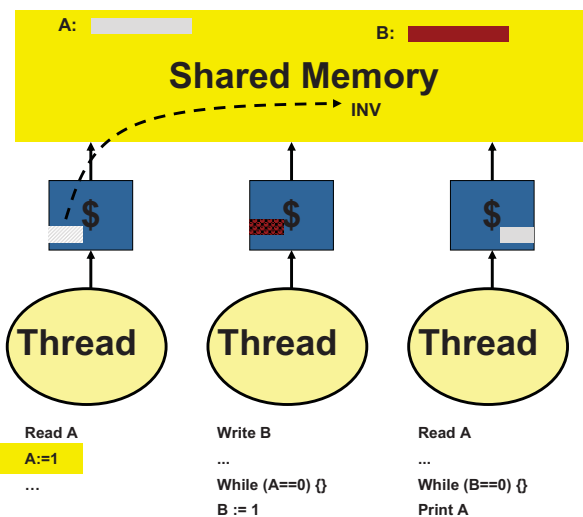
## Example1: Causal Correctness Issues



62

© Erik Hagersten | user.it.uu.se/~eh

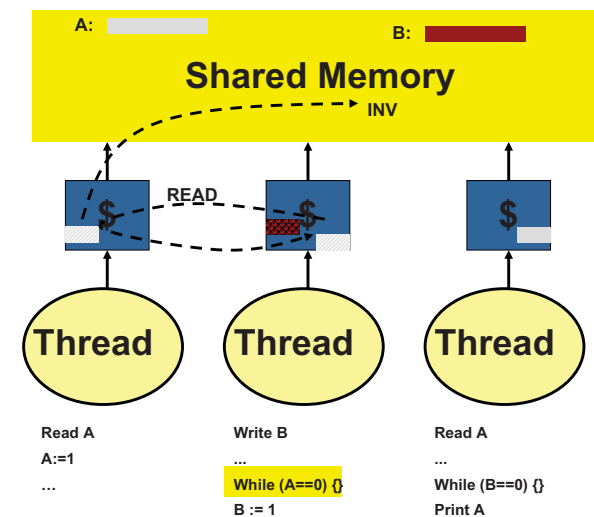
## Example1: Causal Correctness Issues



63

© Erik Hagersten | user.it.uu.se/~eh

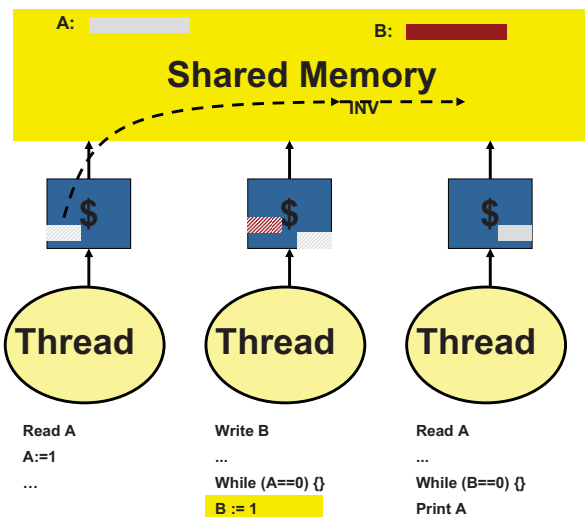
## Example1: Causal Correctness Issues



64

© Erik Hagersten | user.it.uu.se/~eh

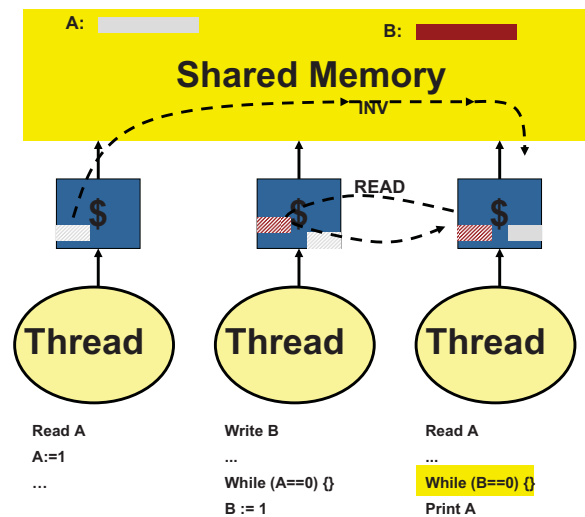
## Example1: Causal Correctness Issues



65

© Erik Hagersten | user.it.uu.se/~eh

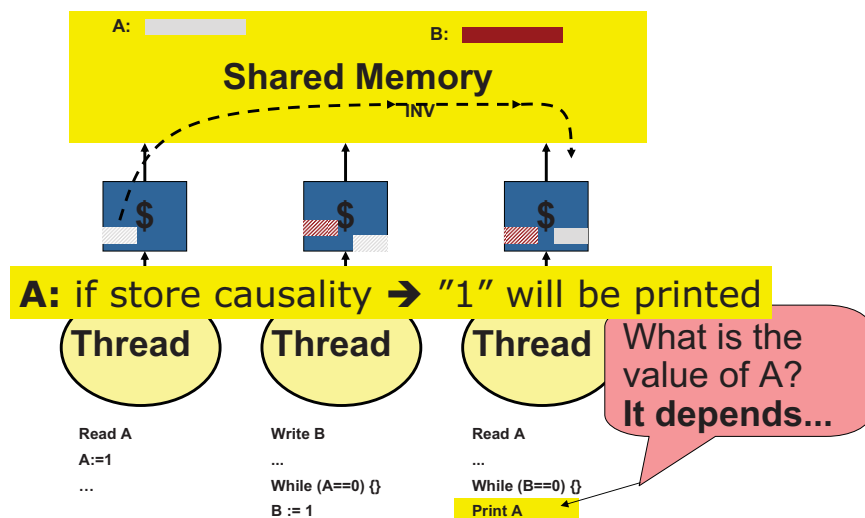
## Example1: Causal Correctness Issues



66

© Erik Hagersten | user.it.uu.se/~eh

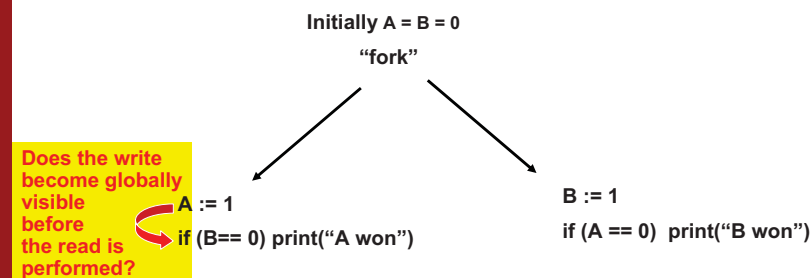
## Example1: Causal Correctness Issues



67

© Erik Hagersten | user.it.uu.se/~eh

## Dekker's Algorithm



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

A: Only known if you know the memory model

68

© Erik Hagersten | user.it.uu.se/~eh

# Learning more about memory models

*Shared Memory Consistency Models: A Tutorial*  
by Sarita Adve, Kouroush 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.

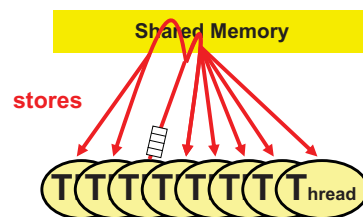
AVDARK  
2010

# X86's new memory model

- Processor consistency with causal correctness for non-atomic memory ops
- TSO for atomic memory ops
- Video presentation:  
<http://www.youtube.com/watch?v=WUfvvFD5tAA&hl=sv>
- See section 8.2 in this manual:  
<http://developer.intel.com/Assets/PDF/manual/253668.pdf>

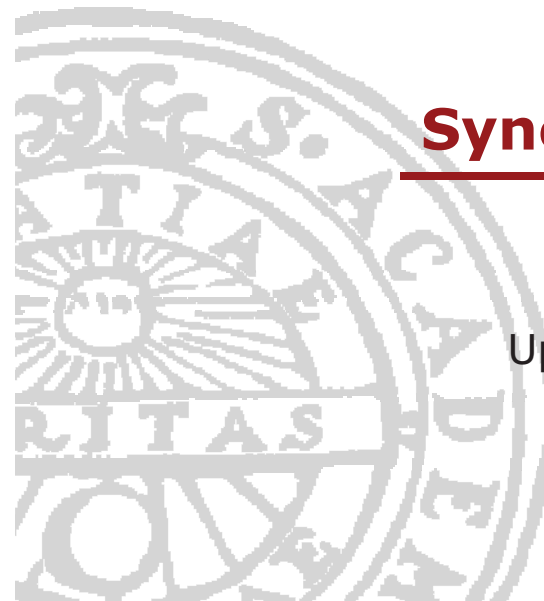
AVDARK  
2010

# Processor Consistency [PC] (J. Goodman)



- PC: The stores from a processor appears to others in program order
- Causal correctness (often added to PC): if a processor observes a store before performing a new store, the observed store must be observed before the new store by all processors
- Flag synchronization works.
- No causal correctness issues

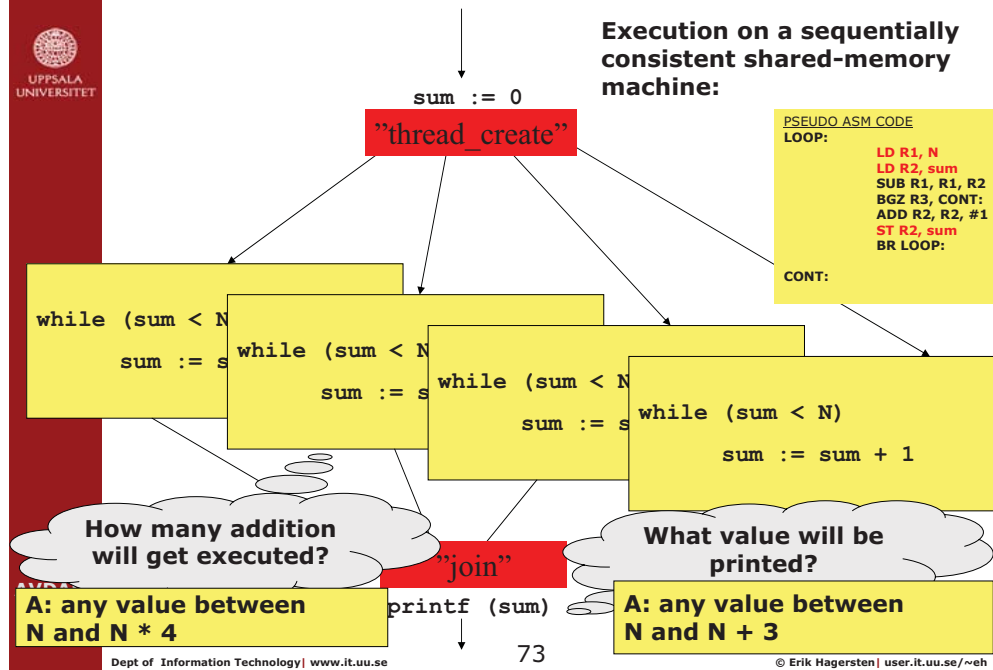
AVDARK  
2010



# Synchronization

Erik Hagersten  
Uppsala University  
Sweden

Execution on a sequentially  
consistent shared-memory  
machine:



## 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**

AVDARK  
2010

## 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

AVDARK  
2010

## 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

AVDARK  
2010

# 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

 AVDARK  
2010

# Release Algorithm

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

 AVDARK  
2010

# 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!**

 AVDARK  
2010

# Optimistic Test&Set Lock "spinlock"

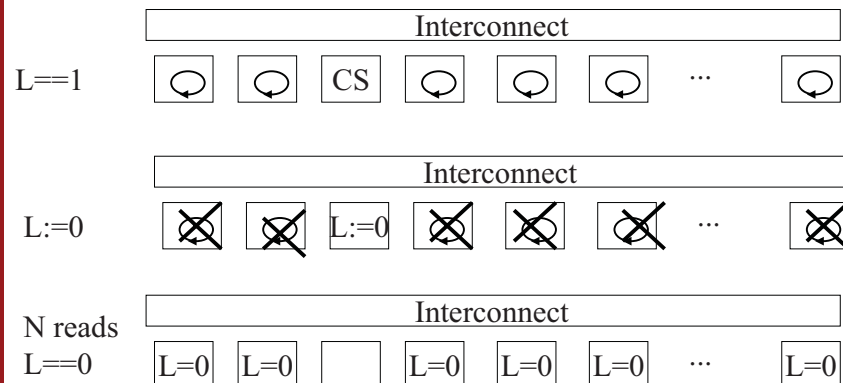
```
proc lock(lock_variable) {
 while true {
 if (TAS[lock_variable] == 0) break; /* bang on the lock once, done if TAS==0 */
 while(lock_variable != 0) {} /* spin locally in your cache until "0" observed */
 }
}
```

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

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

 AVDARK  
2010

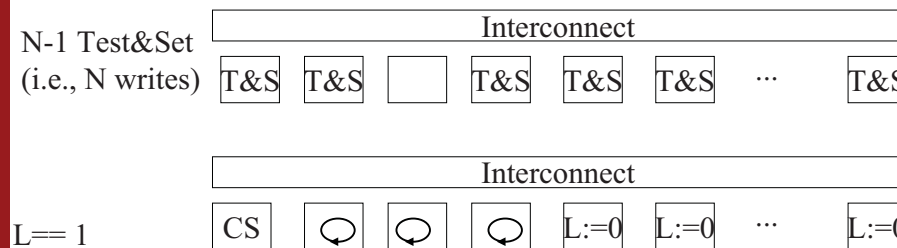
## It could still get messy!



81

© Erik Hagersten | user.it.uu.se/~eh

## ...messy (part 2)



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

Problem1: Contention on the interconnect slows down the CS proc

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

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

Fix1: Queue-based locks

82

© Erik Hagersten | user.it.uu.se/~eh

## 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"

83

© Erik Hagersten | user.it.uu.se/~eh

## Ticket-based queue locks: "ticket"

```

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

proc unlock(lstruct) {
 lstruct.now-serving++ /* next in line please */
}

```

**Less traffic at lock handover!**

84

© Erik Hagersten | user.it.uu.se/~eh

# 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 */
}

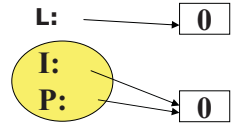
```

Even less traffic at lock handover!

85

© Erik Hagersten | user.it.uu.se/~eh

# Queue-based lock: CLH-lock



"Initially, each process owns one global cell, pointed to by private \*I and \*P  
Another global cell is pointed to by global \*L "lock variable"

- 1) Initialize the \*I flag to busy (= "1")
- 2) Atomically, make \*L point to "our" cell and make "our" \*P point where \*L's cell
- 3) Wait until \*P points to a "0"

```

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

```

```

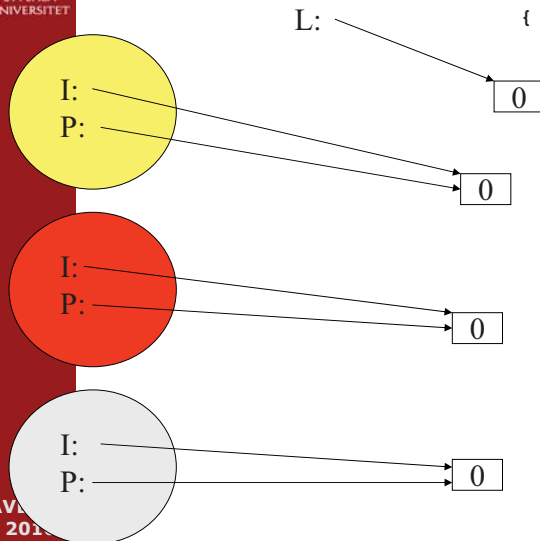
proc unlock(int **I, **P)
{
 **I = 0; /* release the lock */
 *I = *P; /* next time *I to reuse the previous guy's cell */
}

```

86

© Erik Hagersten | user.it.uu.se/~eh

# CLH lock



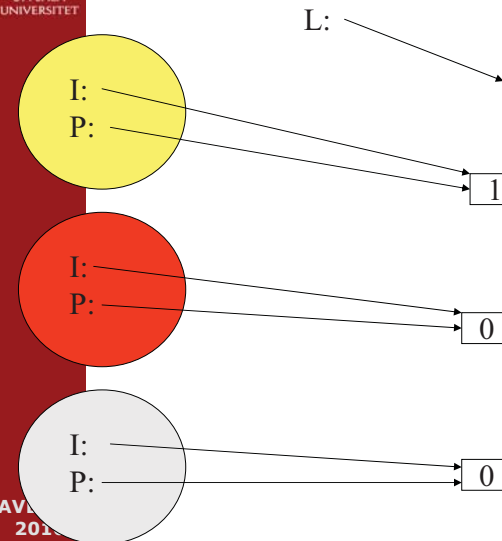
```

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 */
}

```

87

© Erik Hagersten | user.it.uu.se/~eh




```

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 */
}

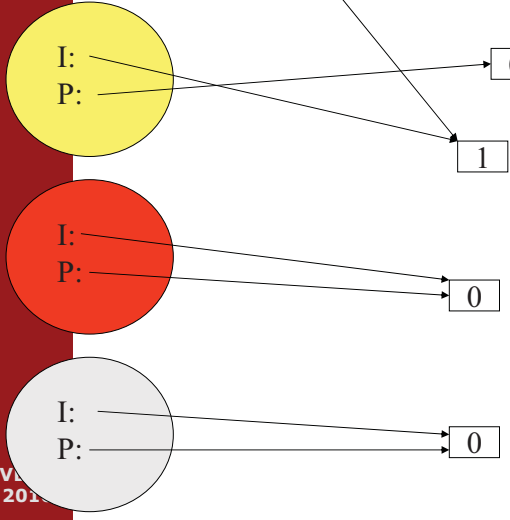
```

88

© Erik Hagersten | user.it.uu.se/~eh



Uppsala University



```


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 */
}

```

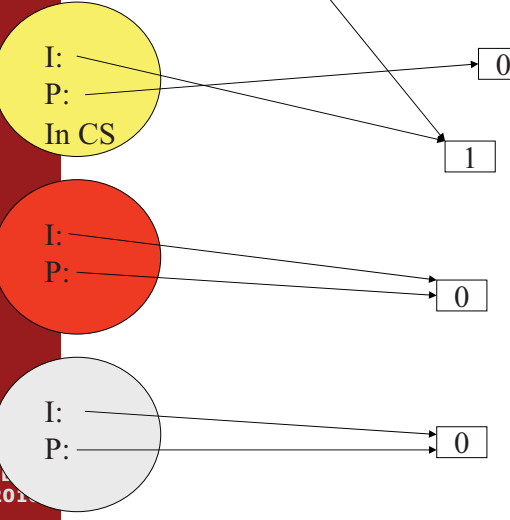
Dept of Information Technology | www.it.uu.se

89

© Erik Hagersten | user.it.uu.se/~eh



Uppsala University



```


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 */
}

```

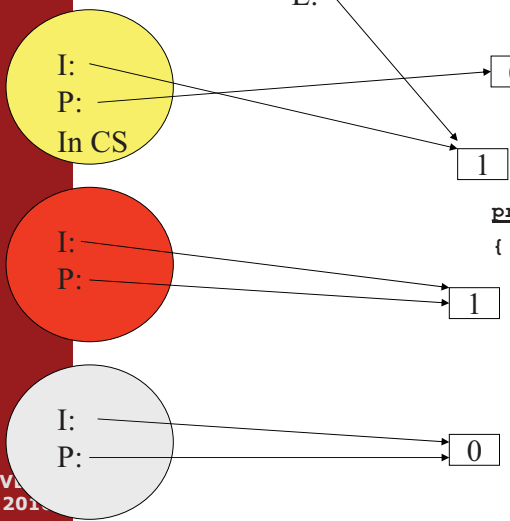
Dept of Information Technology | www.it.uu.se

90

© Erik Hagersten | user.it.uu.se/~eh



Uppsala University



```


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 */
}

```

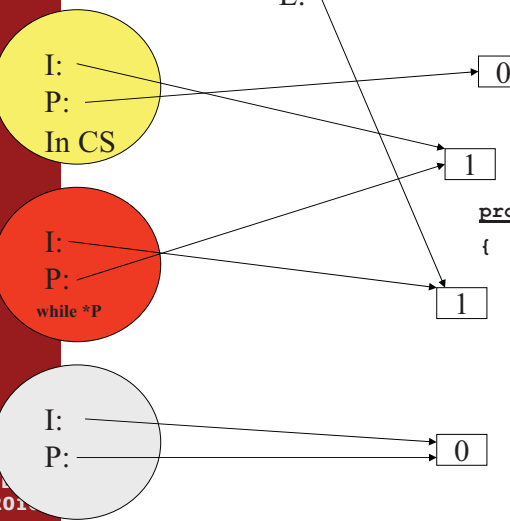
Dept of Information Technology | www.it.uu.se

91

© Erik Hagersten | user.it.uu.se/~eh



Uppsala University



```

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

AV 201

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

93

© Erik Hagersten | user.it.uu.se/~eh

**L:**

**I:**  
**P:**  
In CS

**I:**  
**P:**  
while \*\*P...

**I:**  
**P:**  
while \*\*P...

**0**

**1**

**1**

**1**

```

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 */
}

```

UPPSALA UNIVERSITET

AV 201

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

94

© Erik Hagersten | user.it.uu.se/~eh

**L:**

**I:**  
**P:**  
In CS

**I:**  
**P:**  
while \*\*P...

**I:**  
**P:**  
while \*\*P...

**0**

**1**

**1**

**1**

```

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

```

UPPSALA UNIVERSITET

AV 201

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

95

© Erik Hagersten | user.it.uu.se/~eh

**L:**

**I:**  
**P:**

**I:**  
**P:**  
while \*\*P...

**I:**  
**P:**  
while \*\*P...

**0**

**0**

**1**

**1**

```

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

```

UPPSALA UNIVERSITET

AV 201

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

96

© Erik Hagersten | user.it.uu.se/~eh

**L:**

**I:**  
**P:**

**I:**  
**P:**  
In CS

**I:**  
**P:**  
while \*\*P...

**0**

**0**

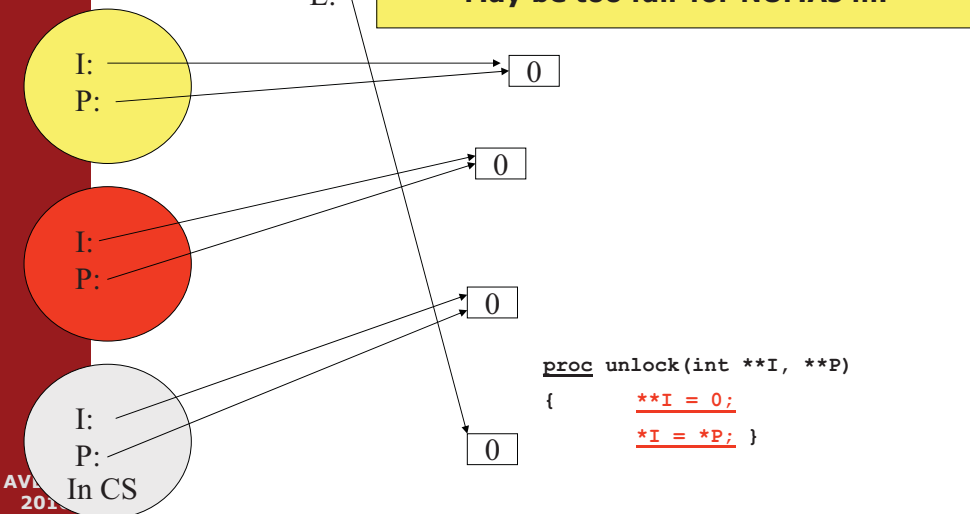
**0**

**1**

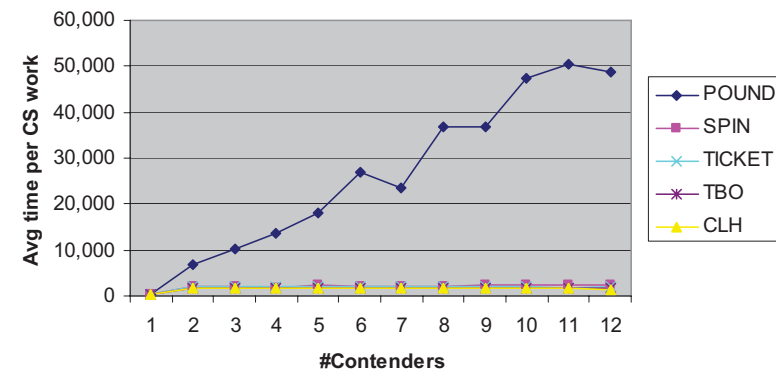
```

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

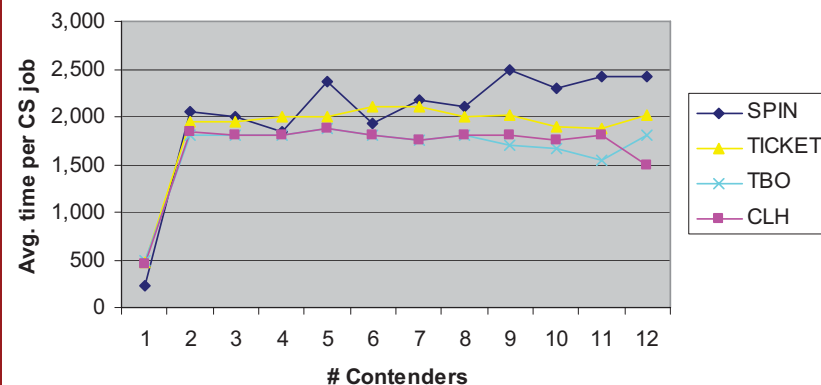
```



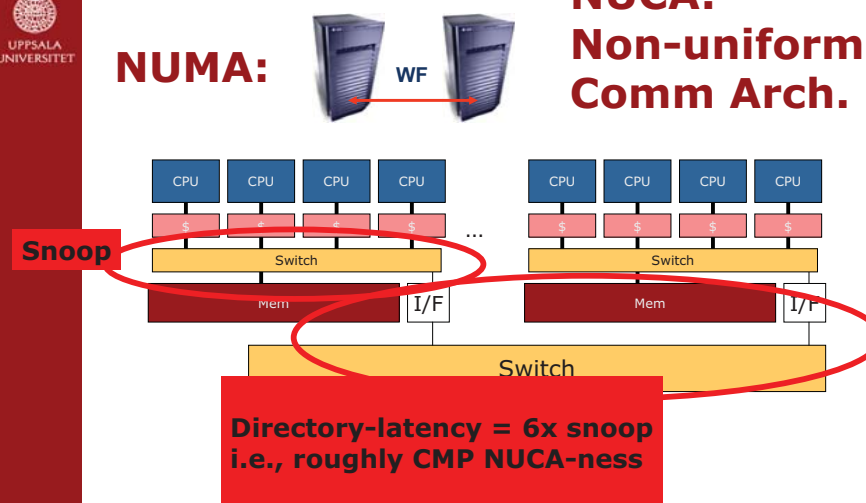
## E6800 locks 12 CPUs



## E6800 locks (exluding POUND)



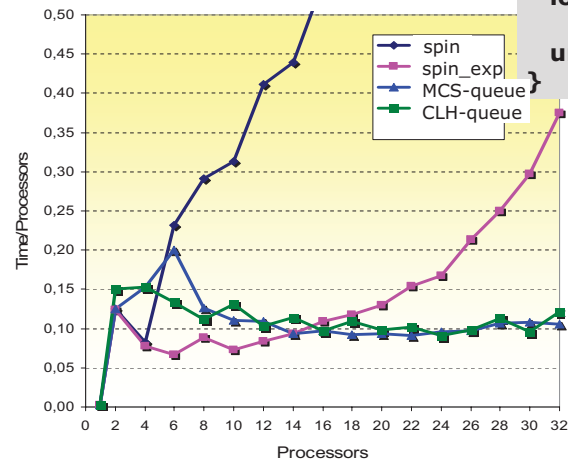
## NUMA: Non-uniform Comm Arch.



## 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)
}
```



101

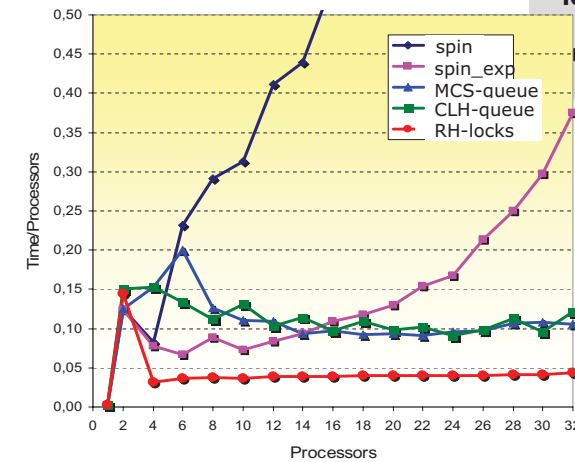
Dept of Information Technology | www.it.uu.se

© Erik Hagersten | user.it.uu.se/~eh

## Introducing RH locks

### Benchmark:

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



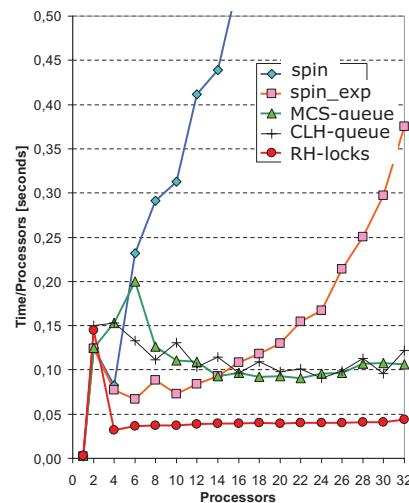
102

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

© Erik Hagersten | user.it.uu.se/~eh

## RH locks: encourages unfairness

### Time per lock handover

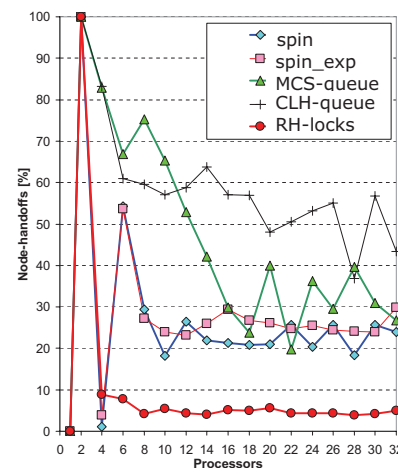


103

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

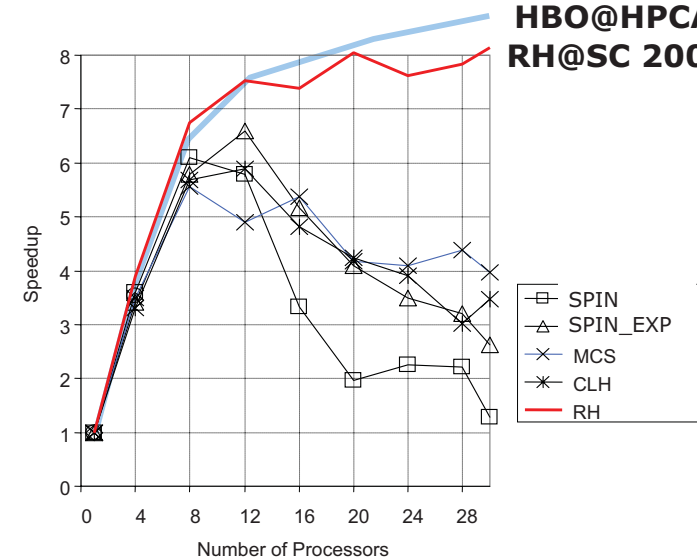
© Erik Hagersten | user.it.uu.se/~eh

### Node migration (%)



## Ex: Splash Raytrace Application Speedup

HBO@HPCA 2003  
RH@SC 2002

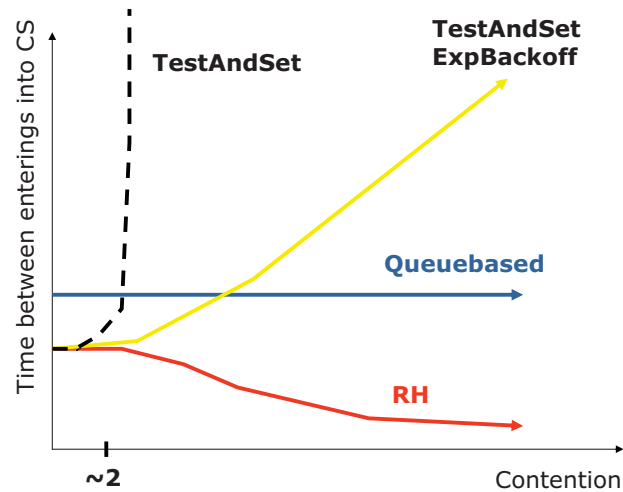


104

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

© Erik Hagersten | user.it.uu.se/~eh

## Performance under contention



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

1. Software algorithms implemented using locks, flags, counters
2. 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

## 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);
 }
 }
}

```

## 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

## New kind of synchronization: Transactional Memory (TM)

- Traditional critical section: lock(ID); unlock(ID) around critical sections
- TM: start\_transaction; end\_transaction around "critical sections" (note: no ID!!)
  - Underlying mechanism to guarantee atomic behavior often by rollback mechanisms
  - This is not the same as guaranteeing that only one thread is in the critical action!!
  - Supported in HW or in SW (normally very inefficient)
- Suggested by Maurice Herlihy in 1993
- HW support announced for Sun's Rock CPU (RIP)

AVDARK  
2010

## 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.

AVDARK  
2010

## 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

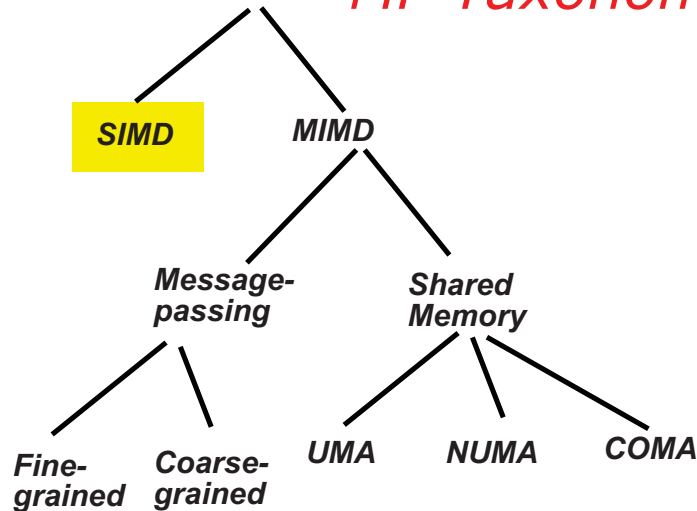
AVDARK  
2010



## Introduction to Multiprocessors

Erik Hagersten  
Uppsala University

## MP Taxonomy



AVDARK  
2010

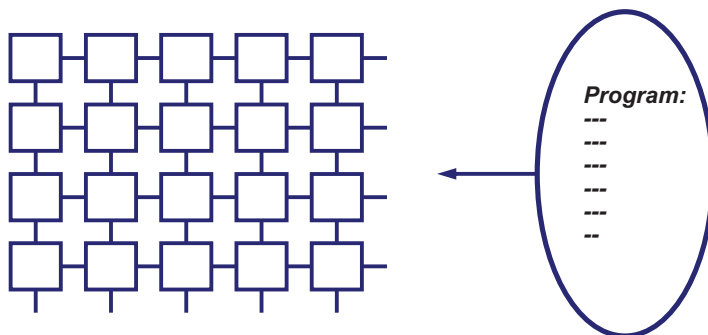
## Flynn's Taxonomy

{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...

AVDARK  
2010

## SIMD = "Dataparallelism"



AVDARK  
2010

## 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...)

AVDARK  
2010

## 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

AVDARK  
2010

## Vector architectures

**CRAY, NEC, Fujitsu,**

**Also x86 extensions: e.g., SSE instruction**

### ■ Vectory Processors

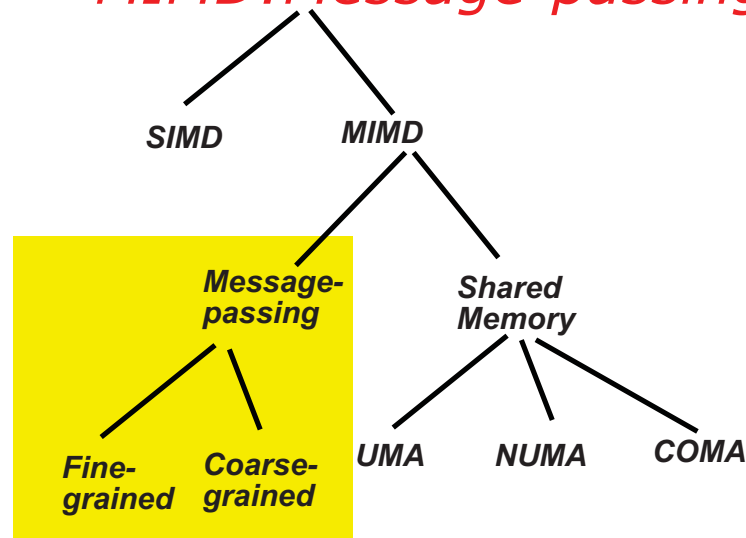
- ✱ LD/ST operate on vectors of data
- ✱ ALU Ops operate on vectors of data

### ■ Example:

- 8 "vector register" contain 64 vector "words" 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

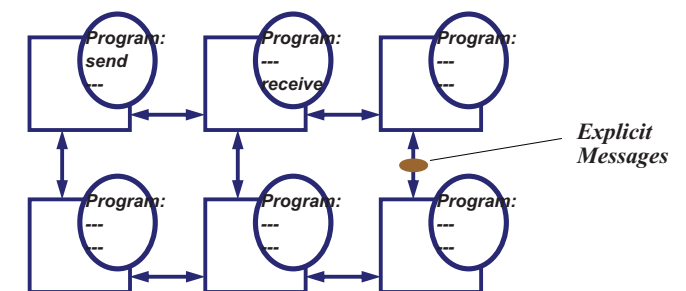
AVDARK  
2010

## MIMD: Message-passing



AVDARK  
2010

## Message-passing Arch MIMD



AVDARK  
2010

# 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)

AVDARK  
2010

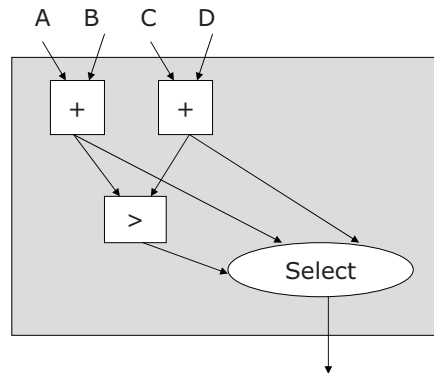
# Dataflow

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

AVDARK  
2010

# Dataflow Example:

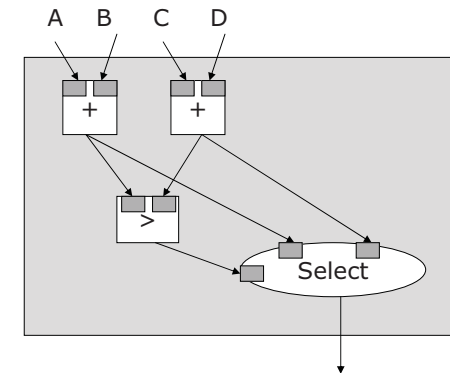
```
X := A + B
Y := C + D
If (X > Y)
 output X
else
 output Y
```



AVDARK  
2010

# Static Dataflow (Dennis)

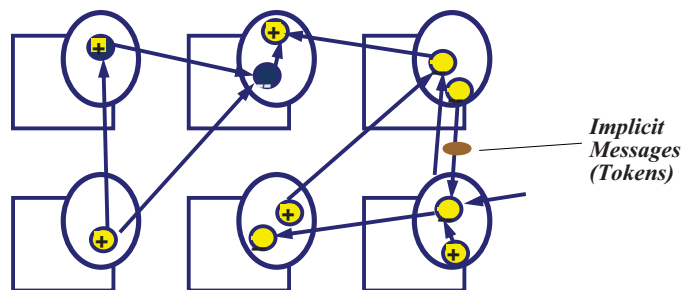
```
X := A + B
Y := C + D
If (X > Y)
 output X
else
 output Y
```



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

AVDARK  
2010

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

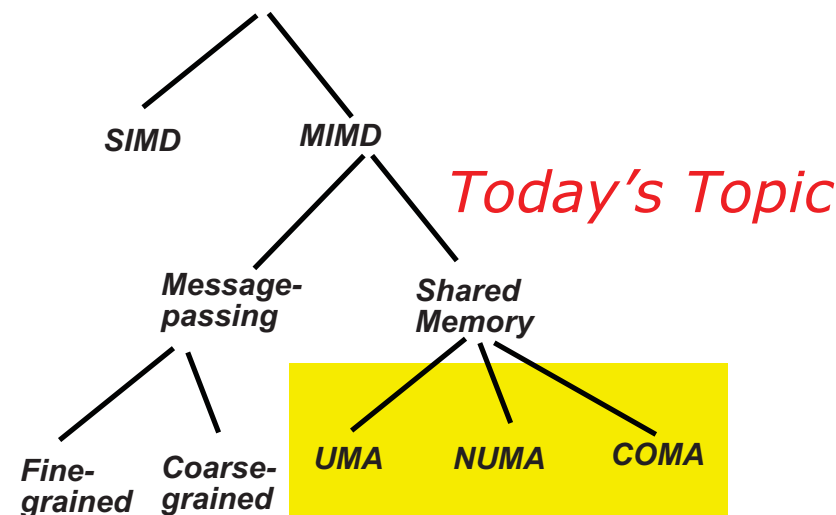


## Dynamic Dataflow (Arvind)

- 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%                   | 24%                  |
| <\$1M       | 2%                   | 9%                   |
| >\$1M       | 3%                   | 8%                   |

UNIX  
shared-  
mem  
servers

The target of the rocket science supercomputers