Multiprocessors and Coherent Memory

Erik Hagersten
Uppsala University

UPPSALA UNIVERSITET

AVDARK 2013

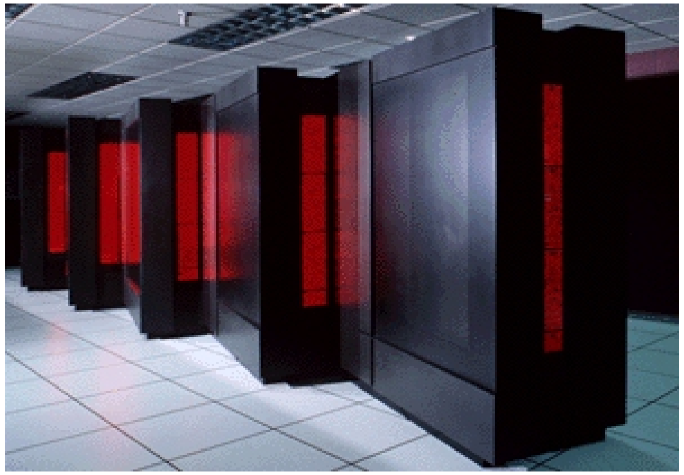
Goal for this course

- Understand **how and why** modern computer systems are designed the way the 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
  - 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...

This batch of lectures

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!



UPPSALA UNIVERSITET

AVDARK 2013

The server market 1995

Server Size	High-Perf. Computing	Commercial Computing
<$10k	1%	19%
<$50k	5%	25%
<$250k	5%	24%
<$1M	2%	9%
>$1M	3%	8%

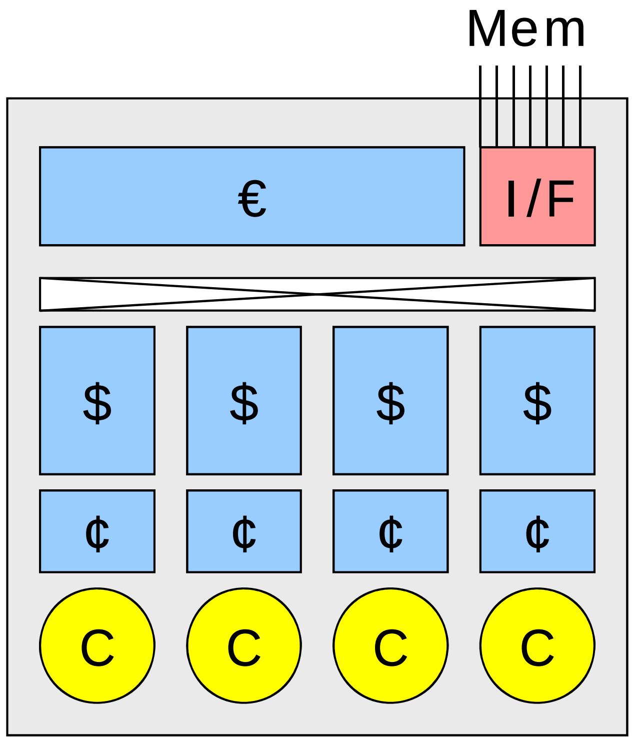
UNIX shared-mem servers

The target of the rocket science supercomputers

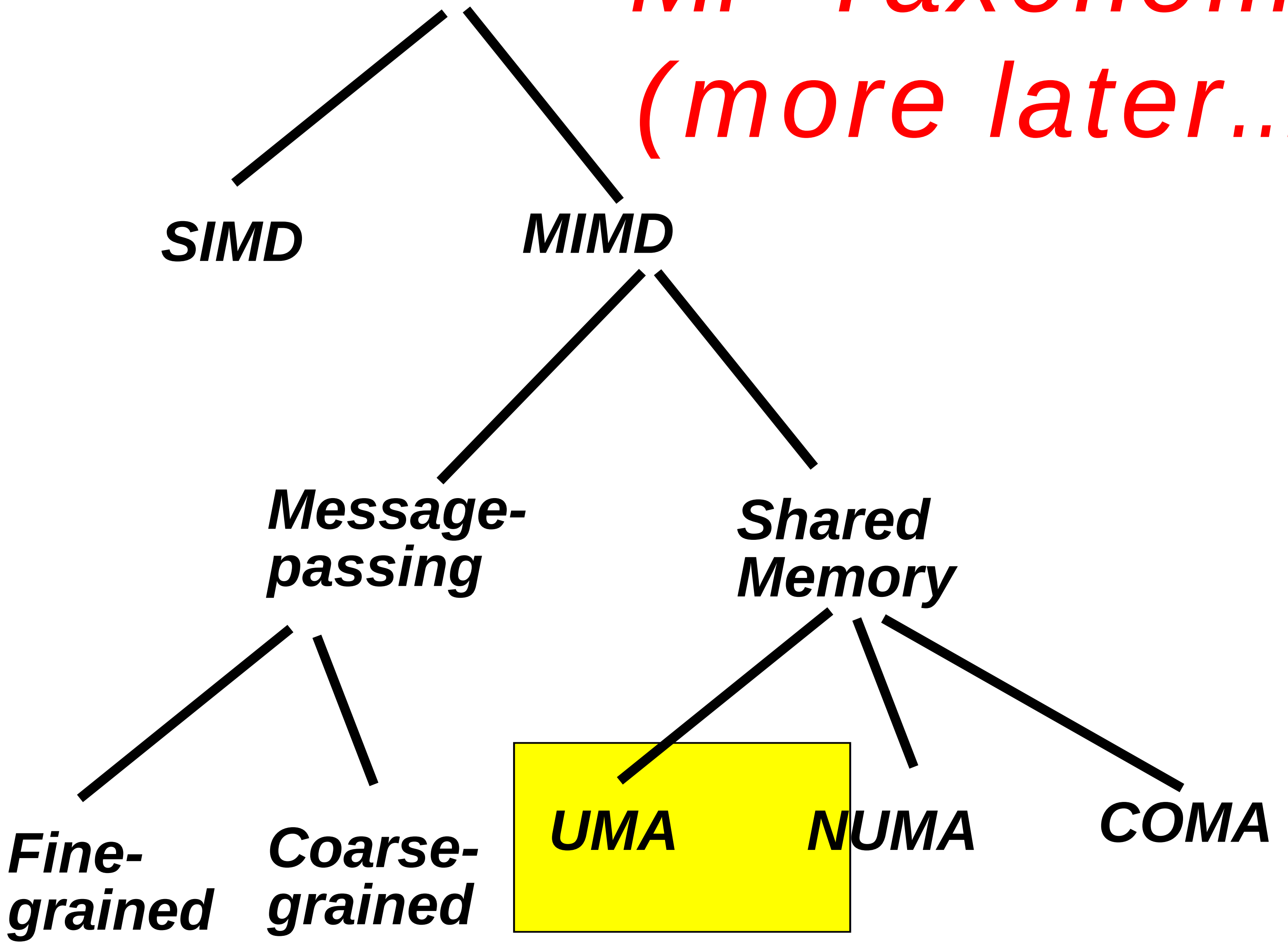
UPPSALA UNIVERSITET

AVDARK 2013

Multicore: Who has <u>not</u> got one?



MP Taxonomy (more later...)



UPPSALA UNIVERSITET

AVDARK 2013

Models of parallelism

- Processes (**`fork`** `or` **`&`** `in UNIX`)
  - A parallel execution, where each process has its own process state, e.g., its own VA→PA mapping
- Threads (**`thread_create`** `in POSIX`)
  - Parallel threads of control inside a process
  - There are some thread-shared state, e.g., VA→PA mapping.
- More: OpenMP, OpenACC, OpenCL, SILC, ...
- Common property: Each thread has its own PC (i.e. executes its own code independently)
- More during lab lecture...

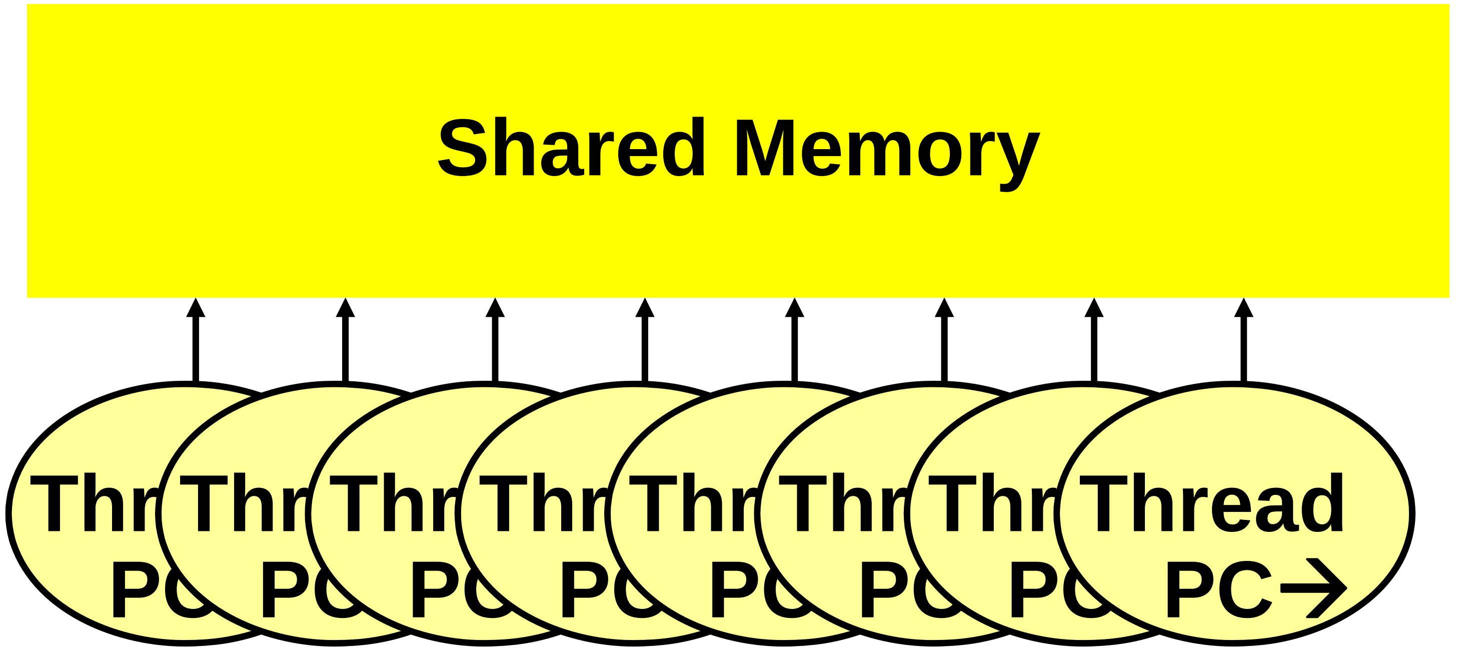
What is: Coherent Shared Memory?

Erik Hagersten
Uppsala University

UPPSALA UNIVERSITET

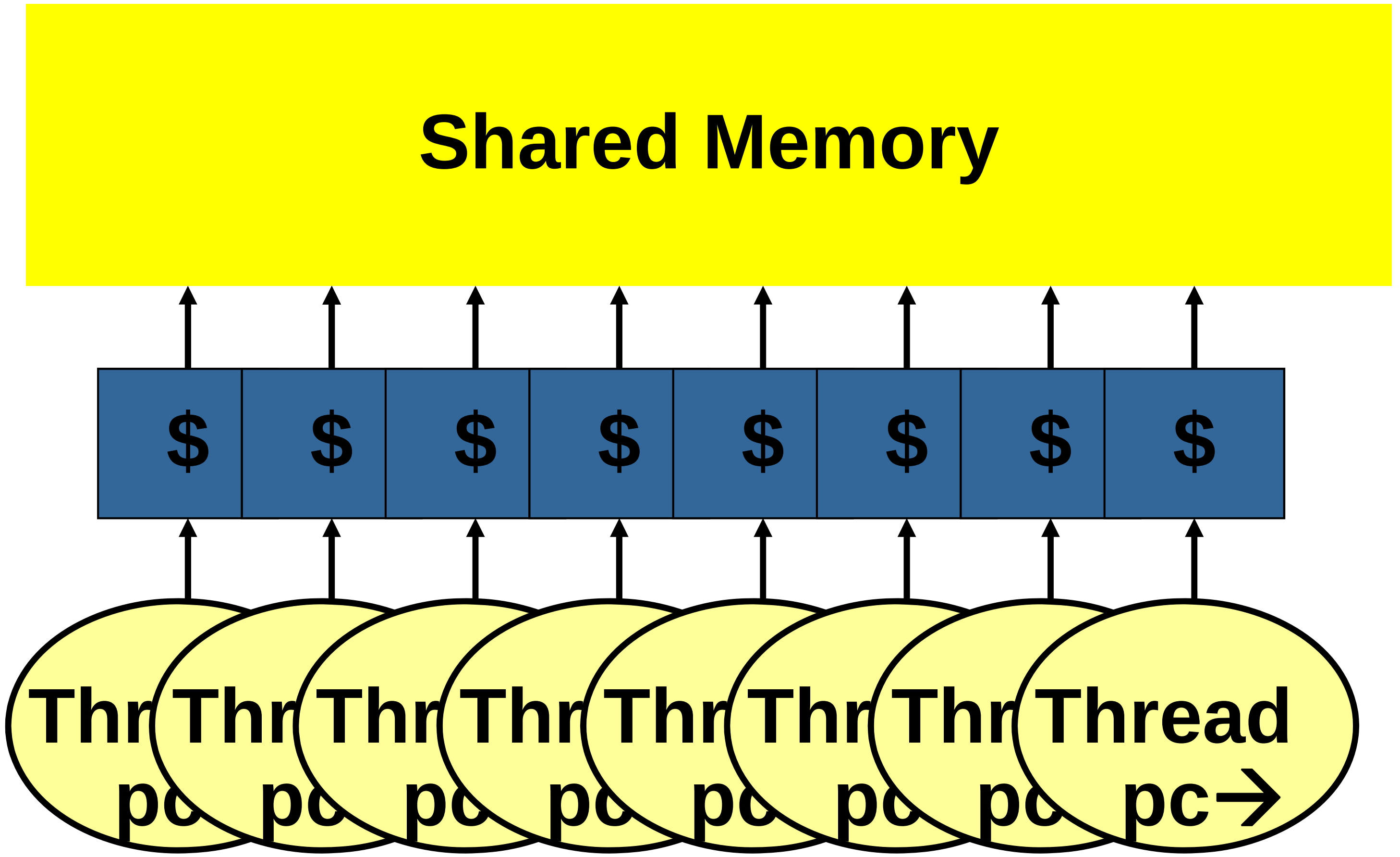
AVDARK 2013

Programming Model: Coherent shared memory



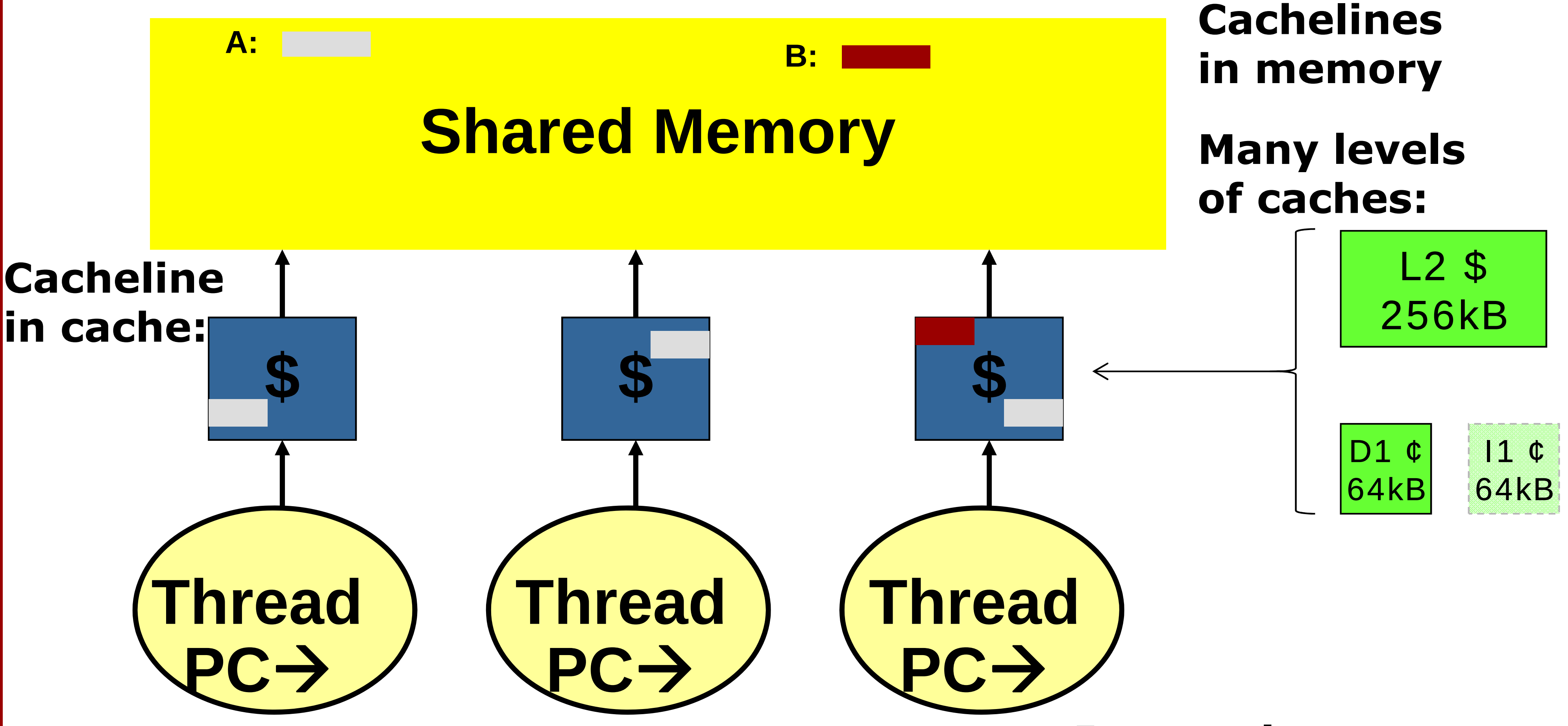
Adding Caches. Gives the illusion of:

- Shorter Memory Latency
- Higher Memory Bandwidth



UPPSALA UNIVERSITET

Our Generic Shared Memory Arch.



Cachelines in memory

Many levels of caches:

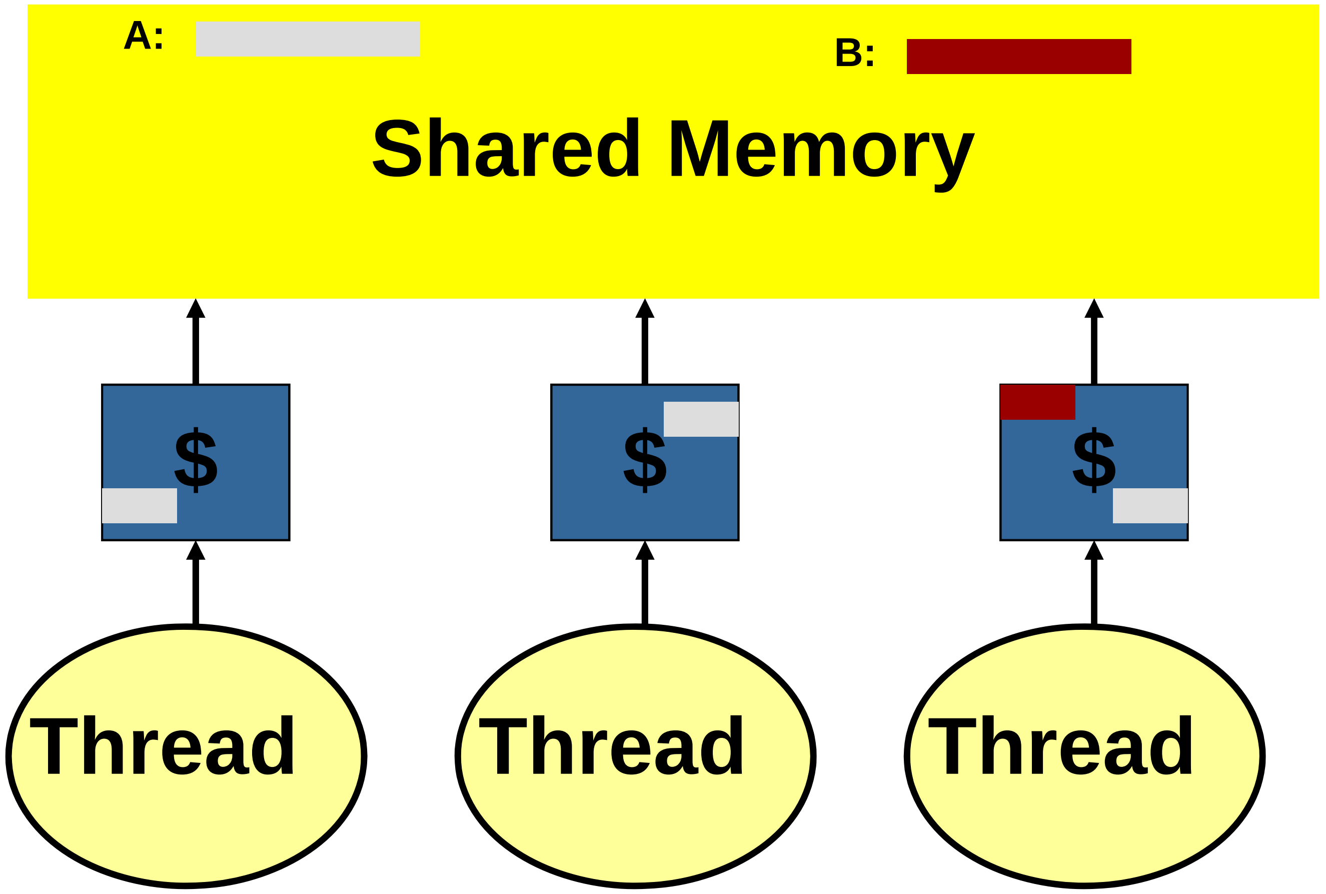
Read A	...	Read B
Read A	Read A	...
...	...	Read A
...		
Read A		

Instructions executed in some global order by the threads

UPPSALA UNIVERSITET

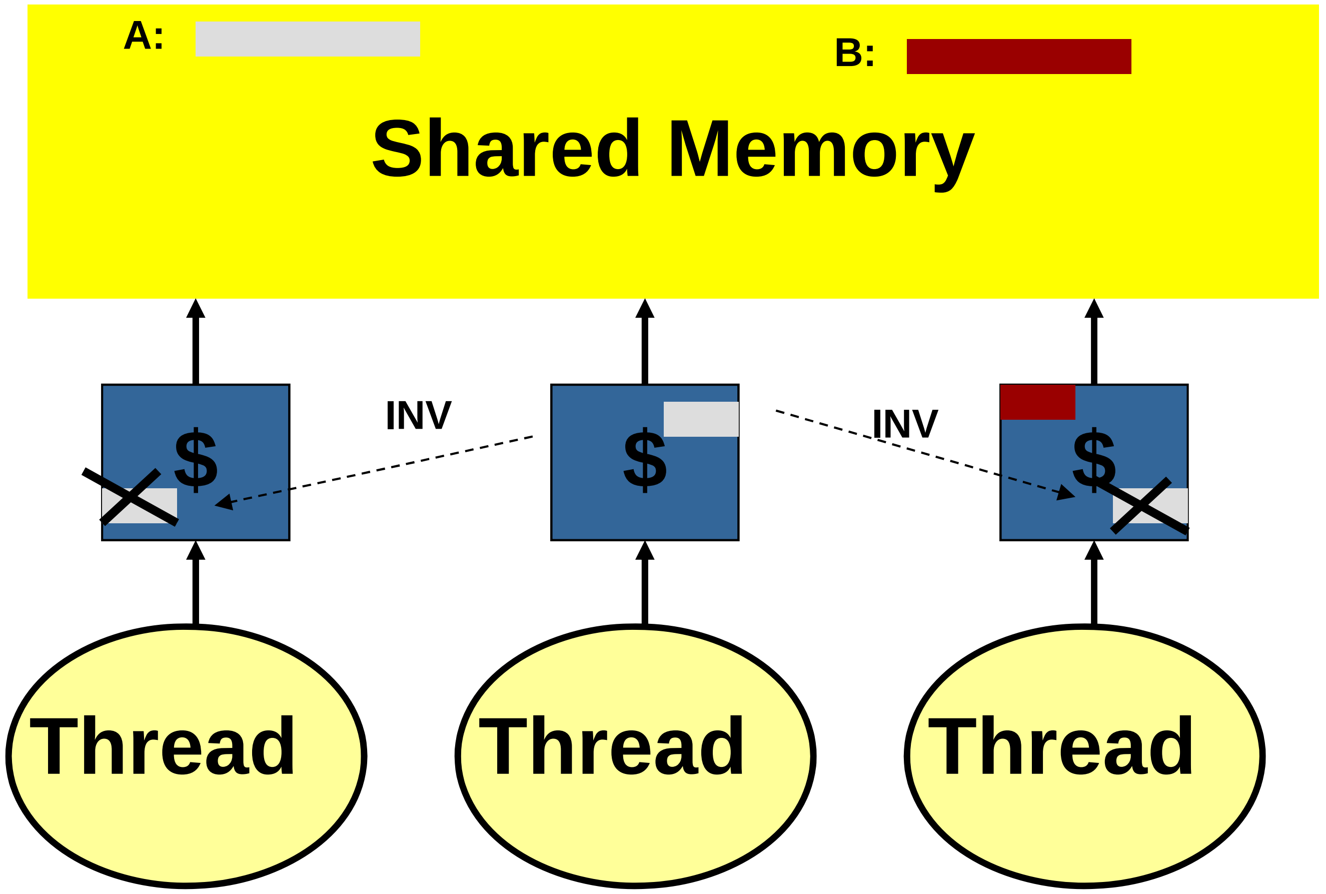
AVDARK 2013

Automatic Replication of Data



Read A	…	Read B
Read A	Read A	…
…	…	Read A
…		

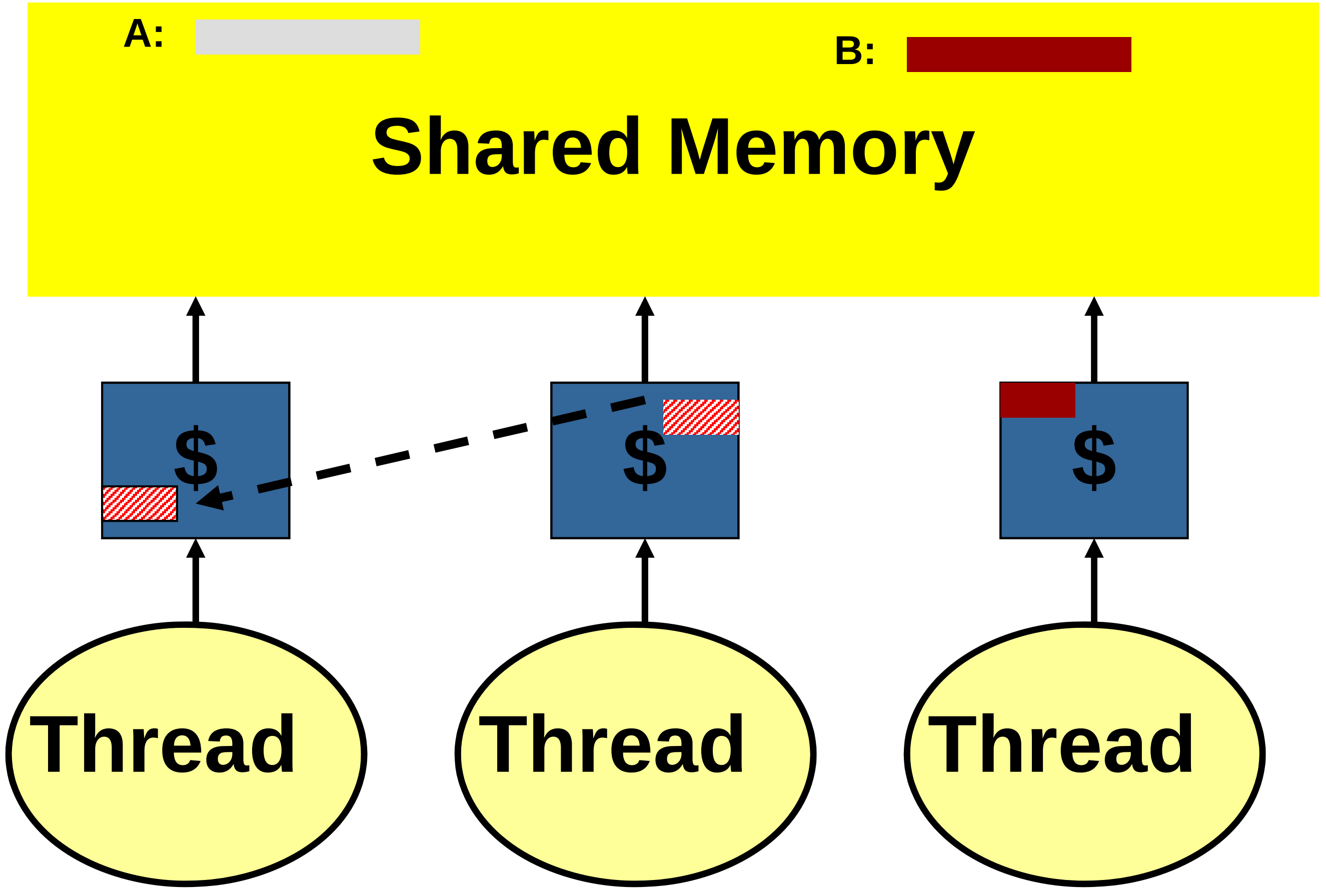
The Cache Coherent Memory System Coherent Write (Here: Write invalidate)



Read A	...	Read B
Read A	Read A	...
...	...	Read A
...	<u>Write A</u>	

The Cache Coherent Memory System Coherent Read & Write-back

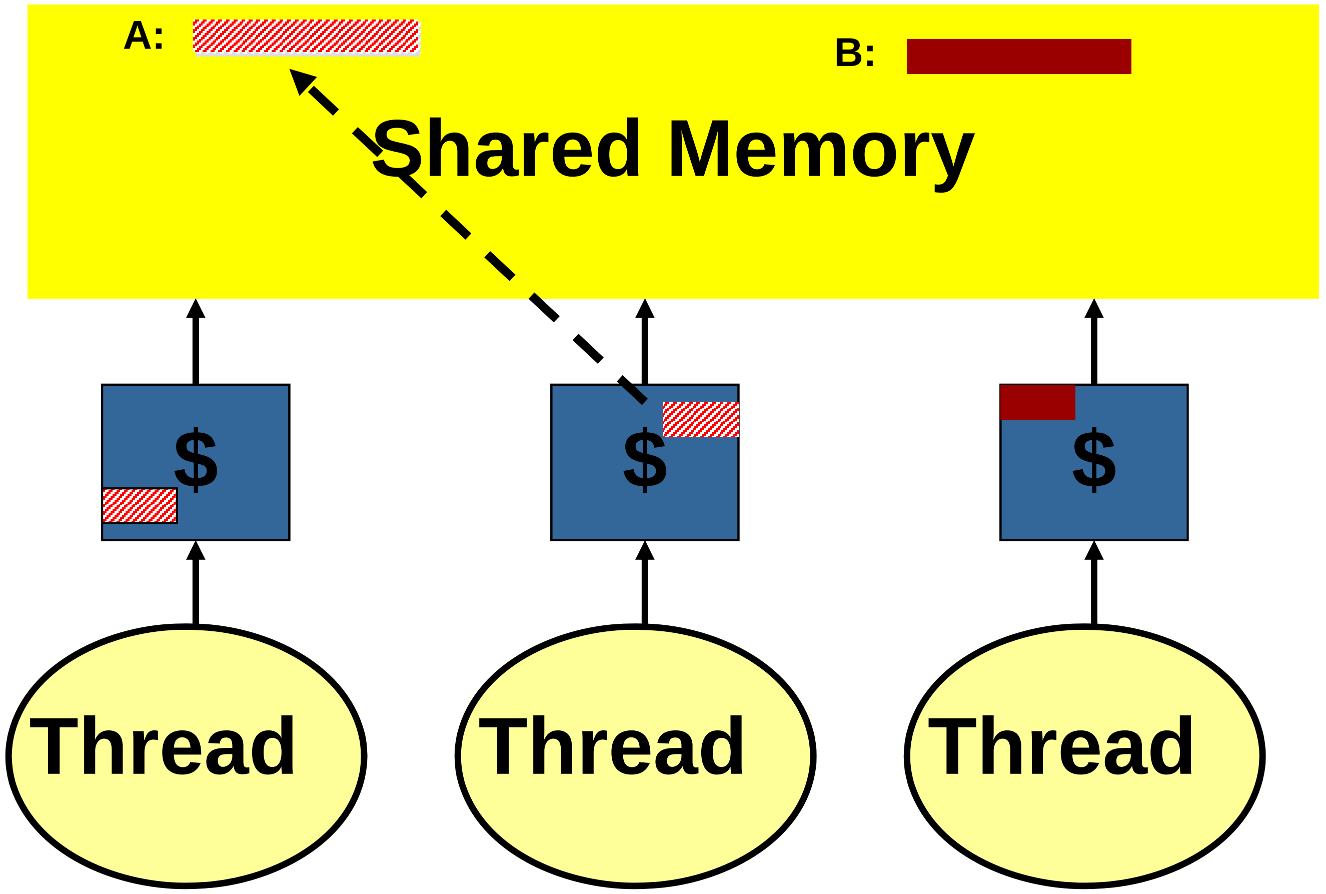
(Here: Cache to Cache Transfer)



Read A	…	Read B
Read A	Read A	…
…	…	Read A
…	Write A	…
Read A		Replace B

The Cache Coherent Memory System Coherent Read & Write-back

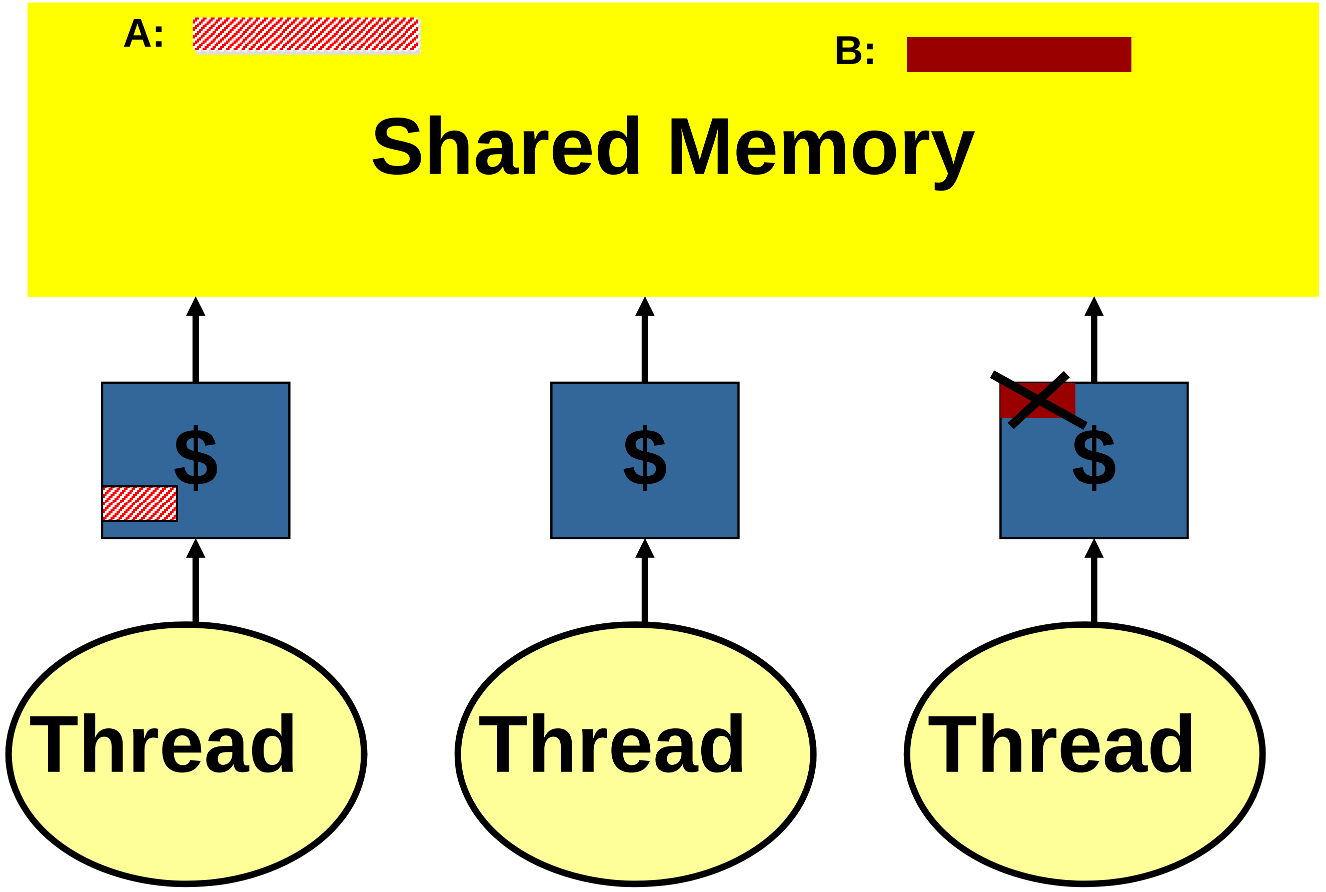
(Here: Write Back)



Read A	...	Read B
Read A	Read A	…
…	…	Read A
…	Write A	
Read A	...	
	Replace A	

The Cache Coherent Memory System Coherent Read & Write-back

(Here: Cache-to-Cache Transfer and Write Back)



Read A	…	Read B
Read A	Read A	…
…	…	Read A
…	Write A	…
Read A	…	Replace B
	Replace A	

UPPSALA UNIVERSITET

Summing up Coherence

Good intuition, but too strong definition!

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

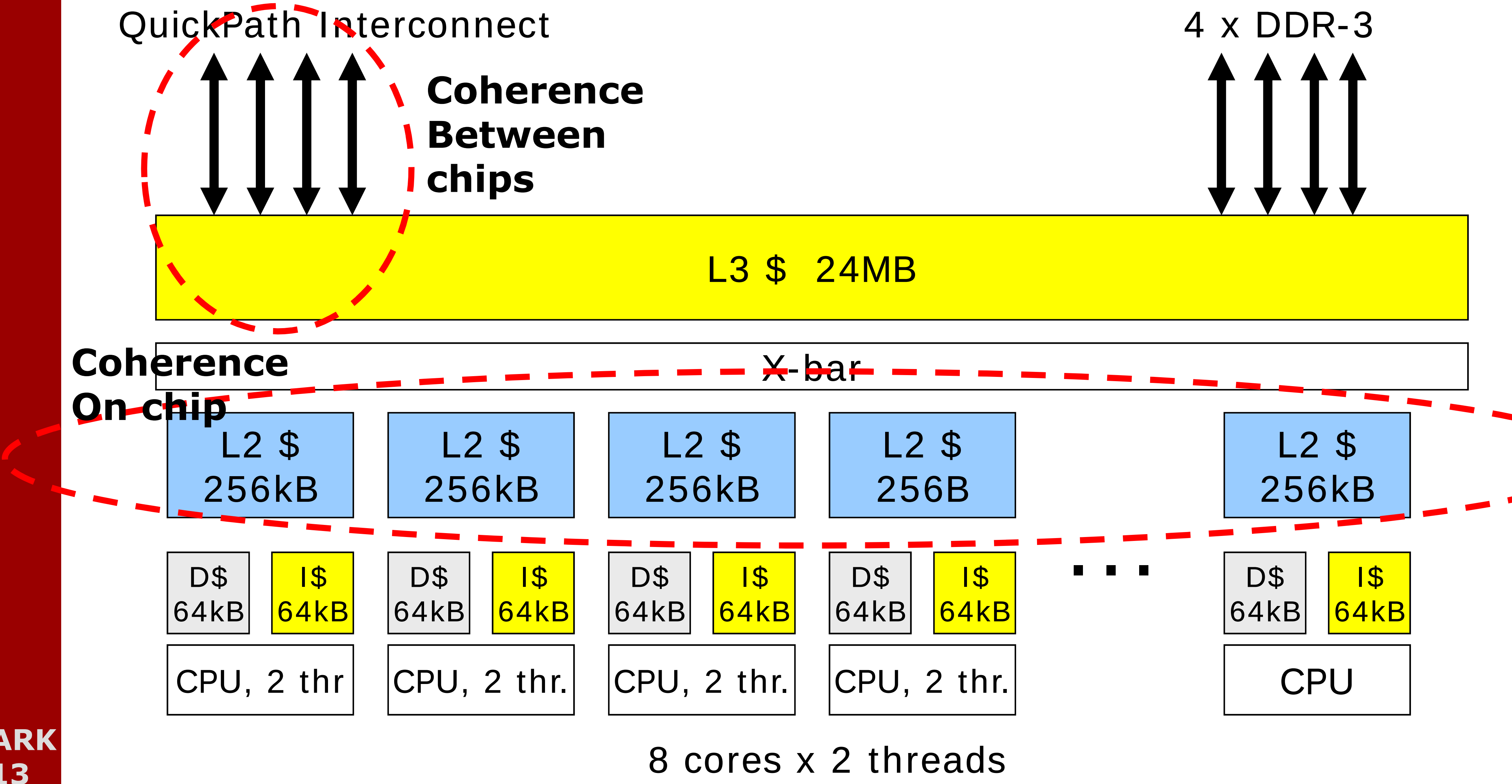
There is a single global order of value changes to each datum

After the computer stops, all copies should have the same value

Thread1={1,2,3,4,5,6,7...} Thread2={1,4,7...} ~~Thread3={1,8,7...}~~

AVDARK 2013

Where does coherence matter?



UPPSALA UNIVERSITET

Summary Coherence

- Coherent shared-memory programming model requires coherence
- (There is also non-coherent shared memory, e.g. single-sided MPI, PGAS)
- All threads can read and write shared data.
- Coherent view of the value of <u>a</u> datum
- Often: Coherence is kept per cache line.

AVDARK 2013

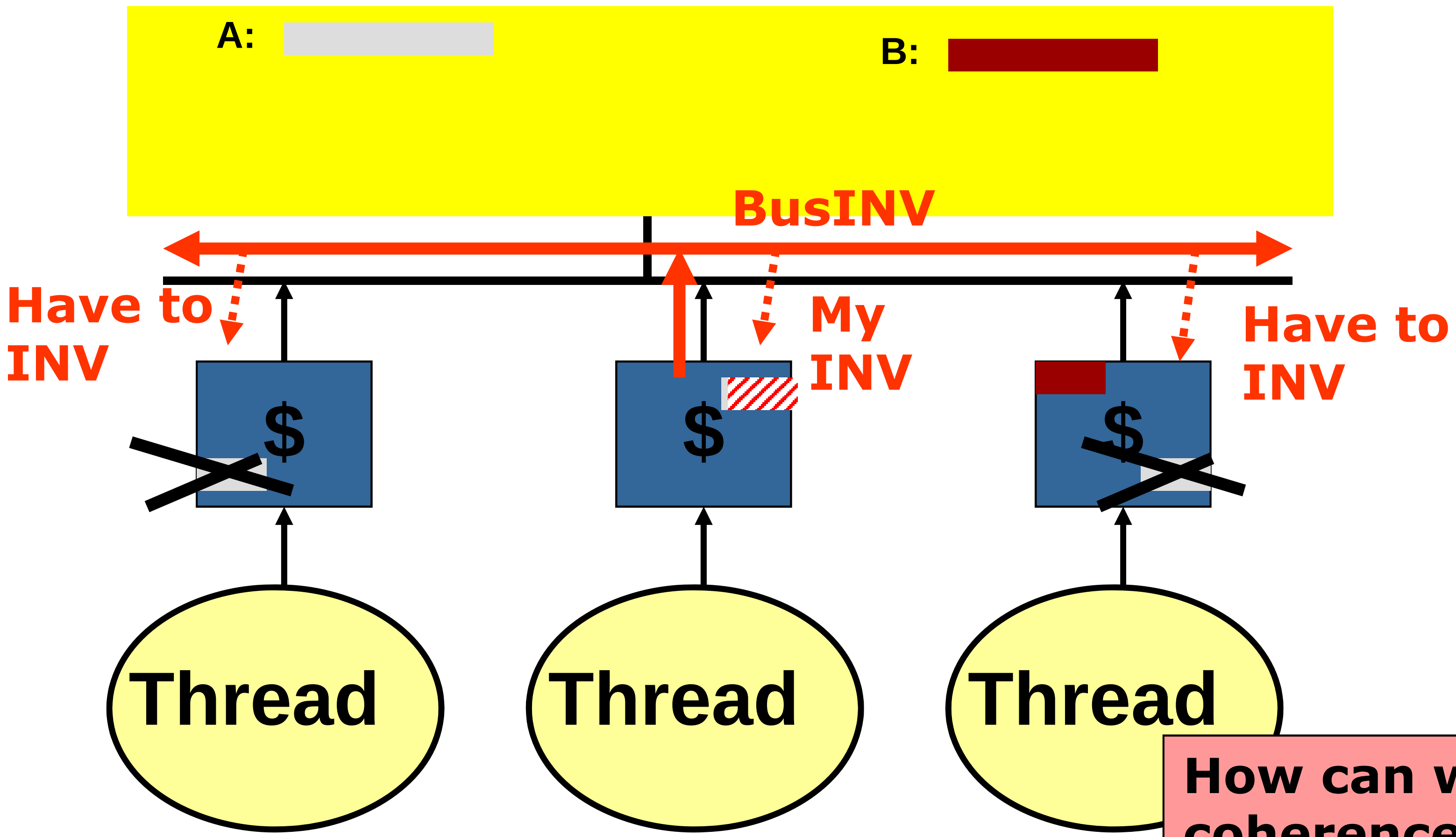
Snooping Coherence

Erik Hagersten
Uppsala University

Implementation options for coherence

- Two coherence options
  - Snoop-based (”broadcast protocol”)
  - Directory-based (”point to point protocol”)
- Different scalability
- Different latency

”Upgrade” in snoop-based



Read A
Read A
…
…

…
Read A
…
Write A

Read B
…
Read A

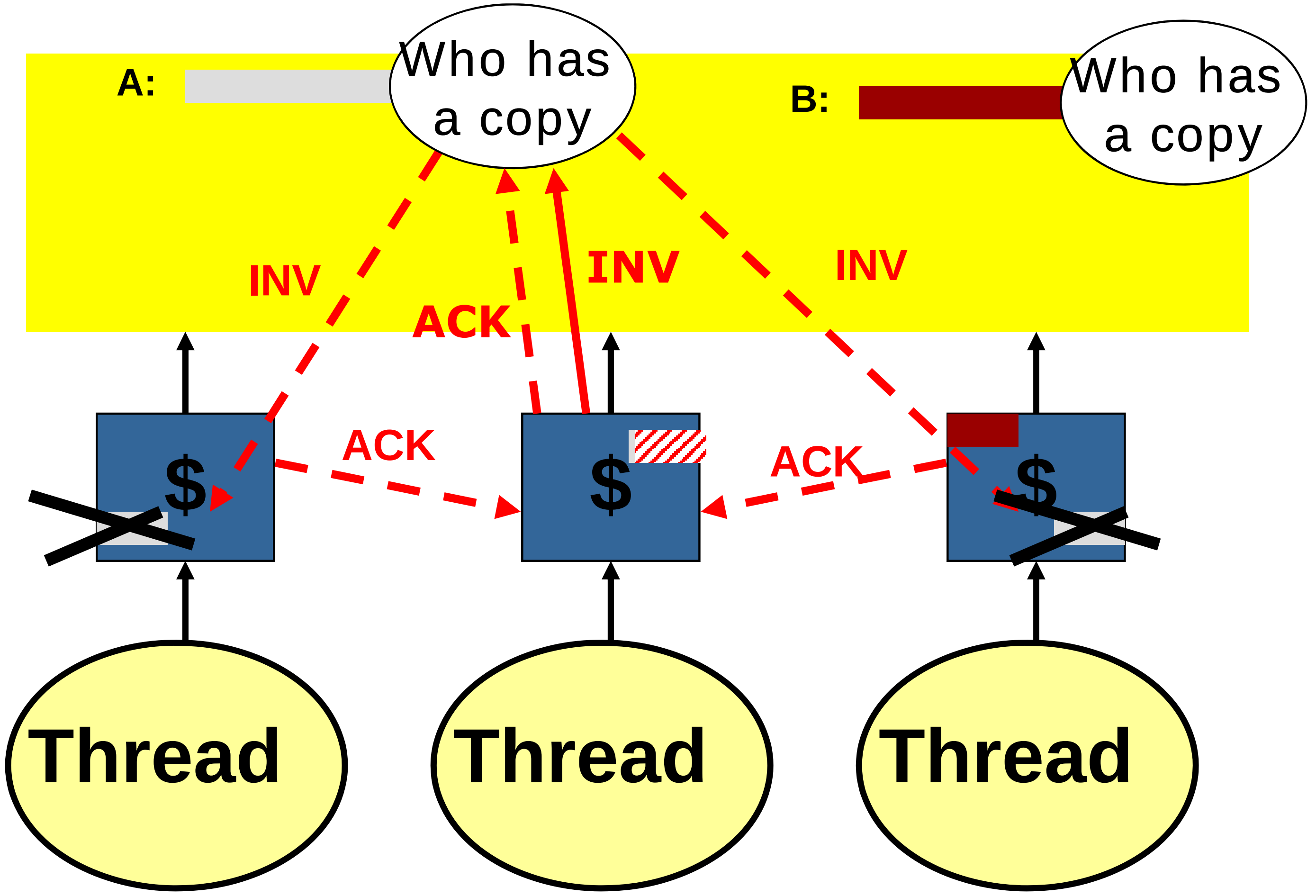
How can we implement coherence on a write?

- ❑ Invalidate A in mem.
- ❑ Invalidate A in left $
- ❑ Invalidate A in right $
- ❑ Invalidate A in both left and right $

UPPSALA UNIVERSITET

AVDARK 2013

"Upgrade" in dir-based

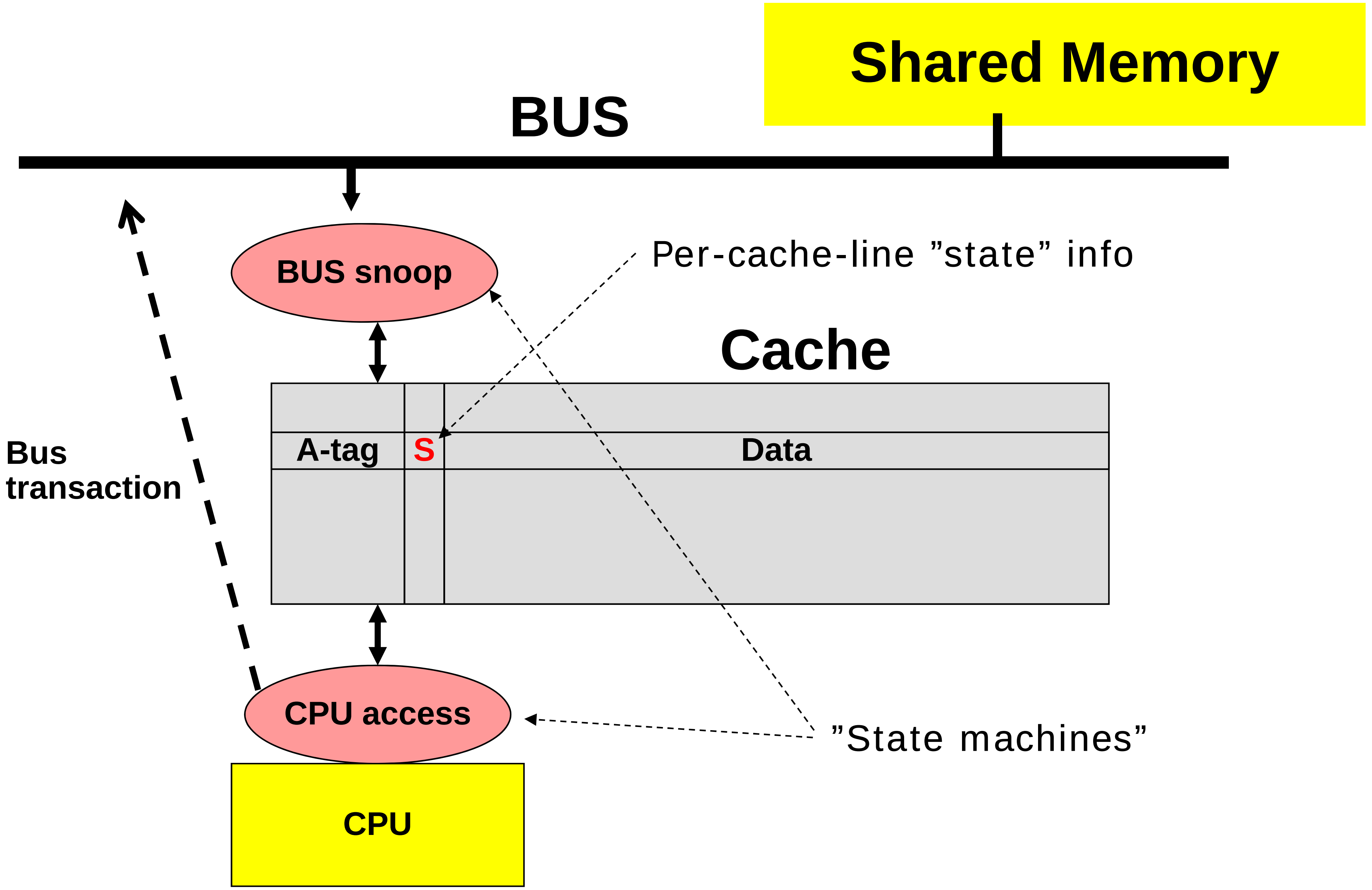


Read A
Read A
…
…

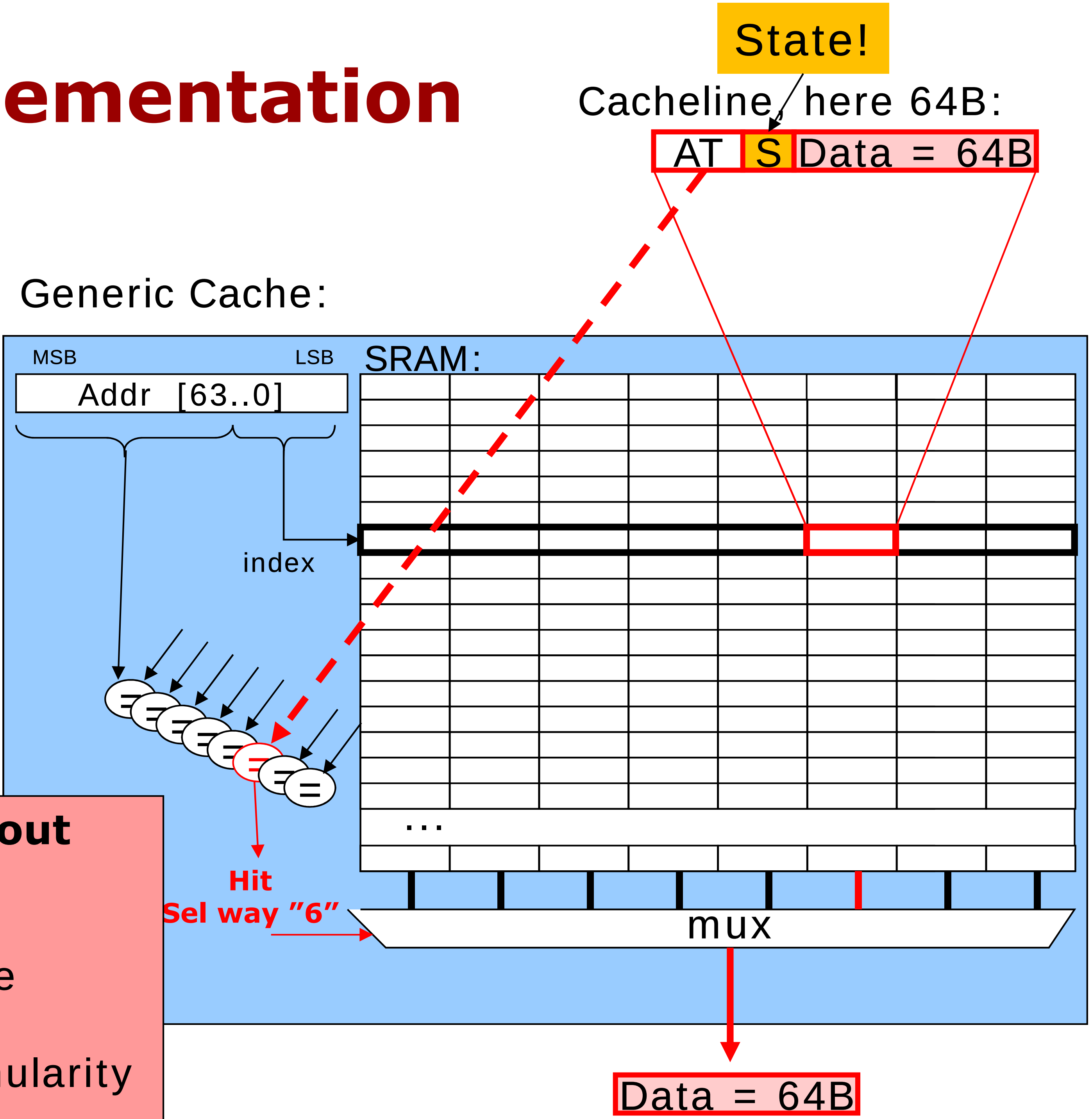
…
Read A
…
Write A

Read B
…
Read A

Snoop-based Protocol Implementation



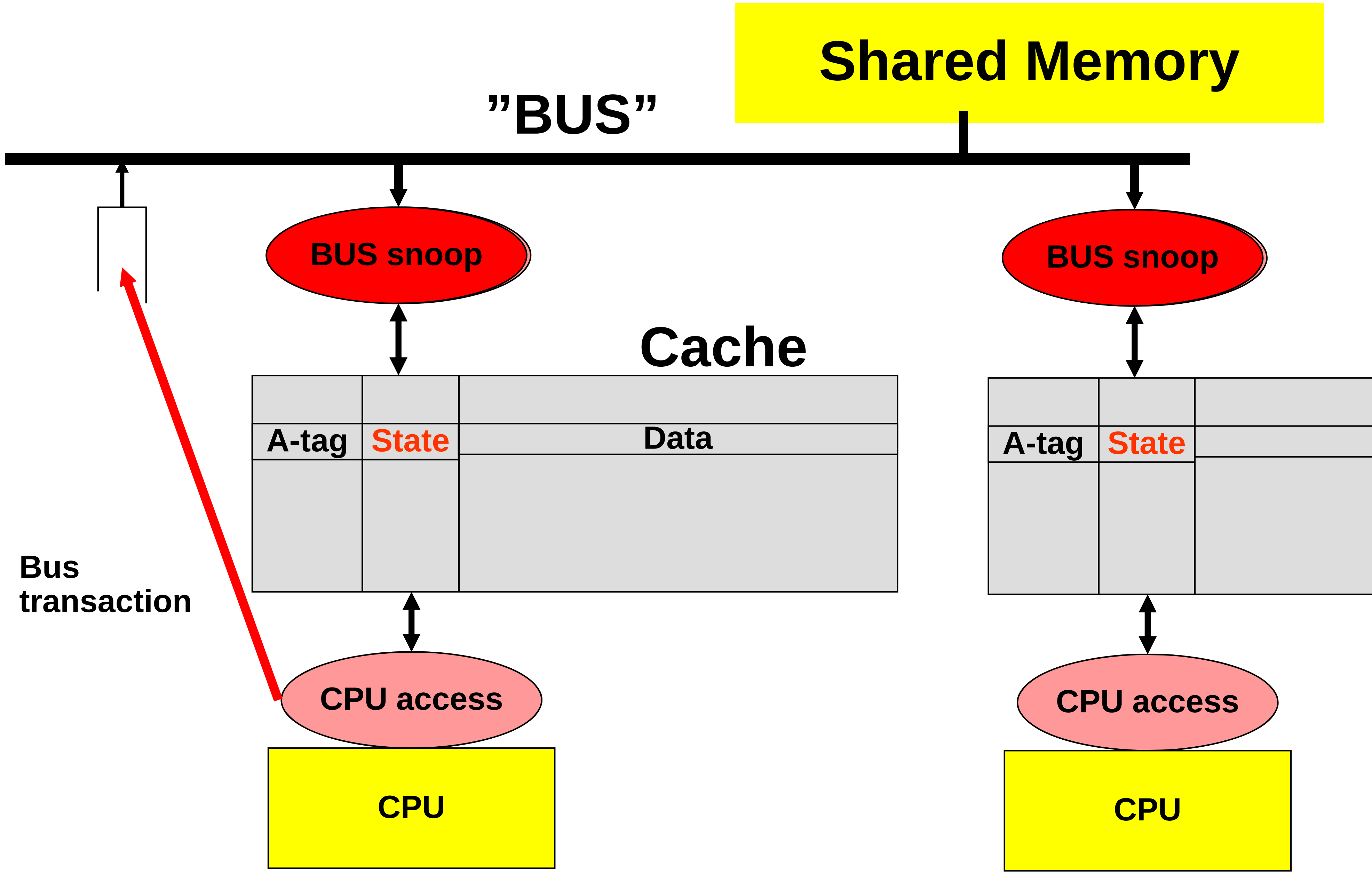
Cache implementation



Which statement is true about this cache?

- ❑ Coherence needs to be implemented on a cache line granularity
- ❑ Coherence with a word granularity can be implemented

Snoop-based Protocol Implementation



Example: MOSI Bus Snoop

Why is there no BUSinv arrow from M?
- No other can have a cached copy
- BUSinv is only used for writes

STATES:

M – Modified: My dirty* copy is the only cached copy

S – Shared: I have a clean copy, others may also have a copy

O – Owner: I have a dirty copy, others may also have a copy

I – Invalid: I have no valid copy in my cache (including cache miss)

BUS TRANSACTIONS FROM OTHERS:

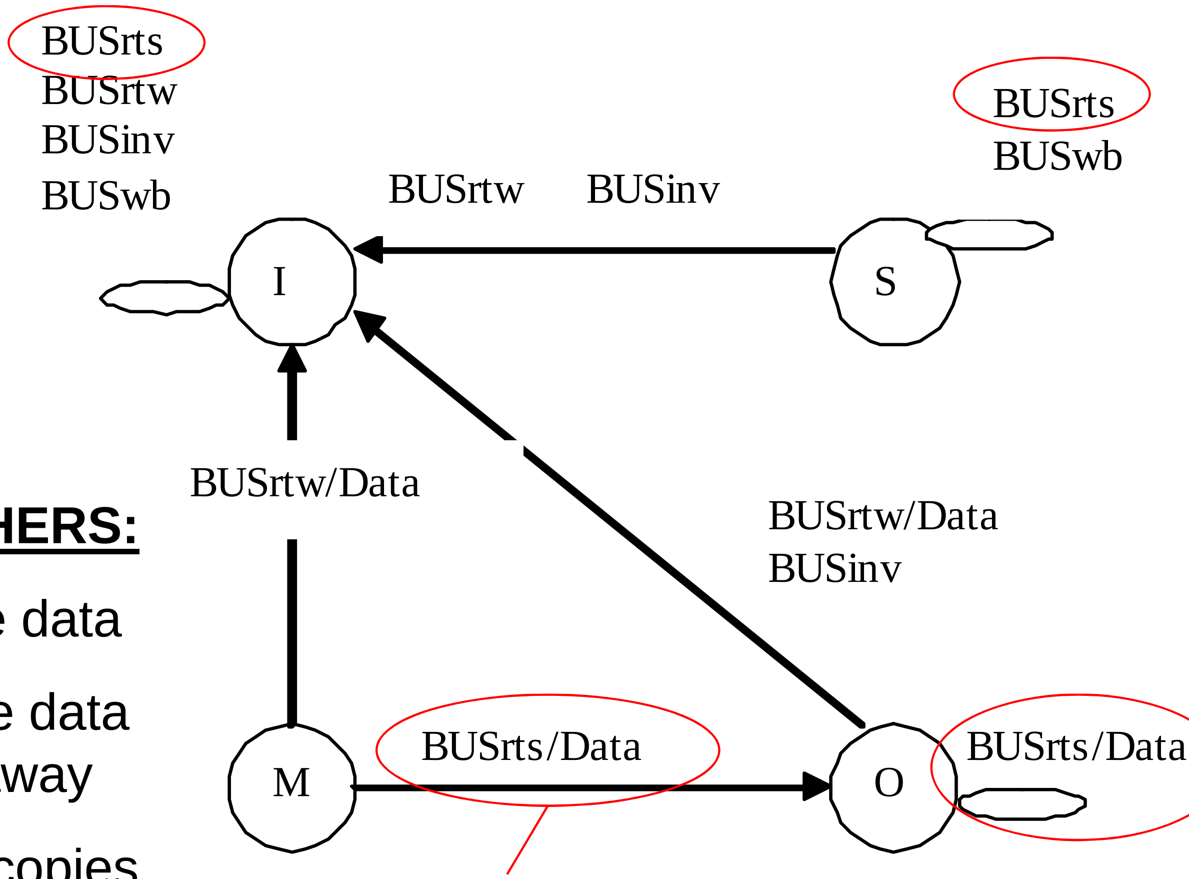
BUSrts ReadtoShare. Reading the data

BUSrtw: ReadToWrite. Reading the data with the intention to modify it right away

BUSinv: Invalidating other caches copies

BUSwb: Writing data back to memory

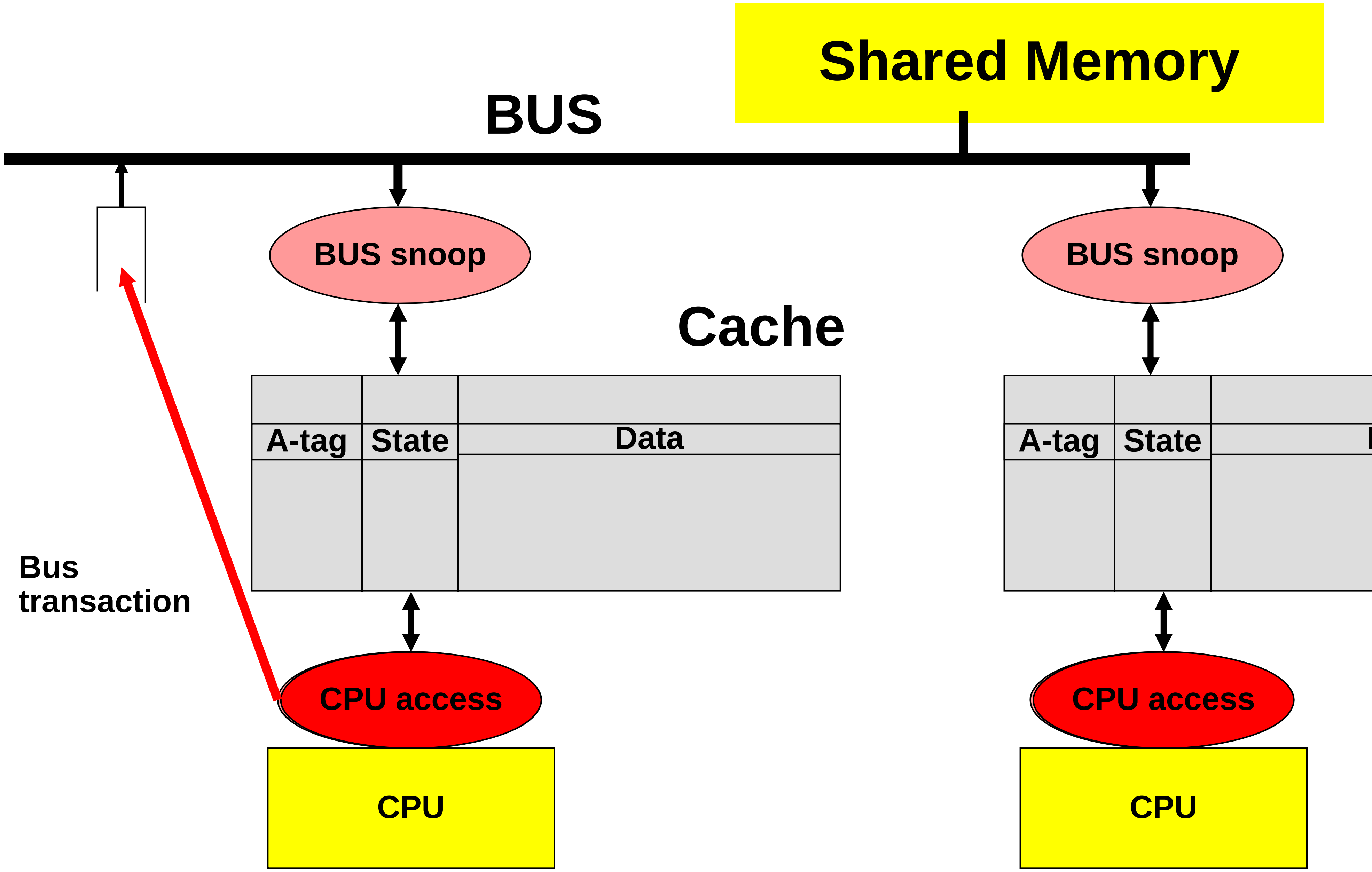
**Dirty: my value differs from the old value in mem*



Input-signal/Reply-signal
Meaning: If you are in state M and see BUSrts, goto state O and reply with Data

UPPSALA UNIVERSITET

Snoop-based Protocol Implementation



AVDARK 2013

UPPSALA UNIVERSITET

Example: CPU access MOSI

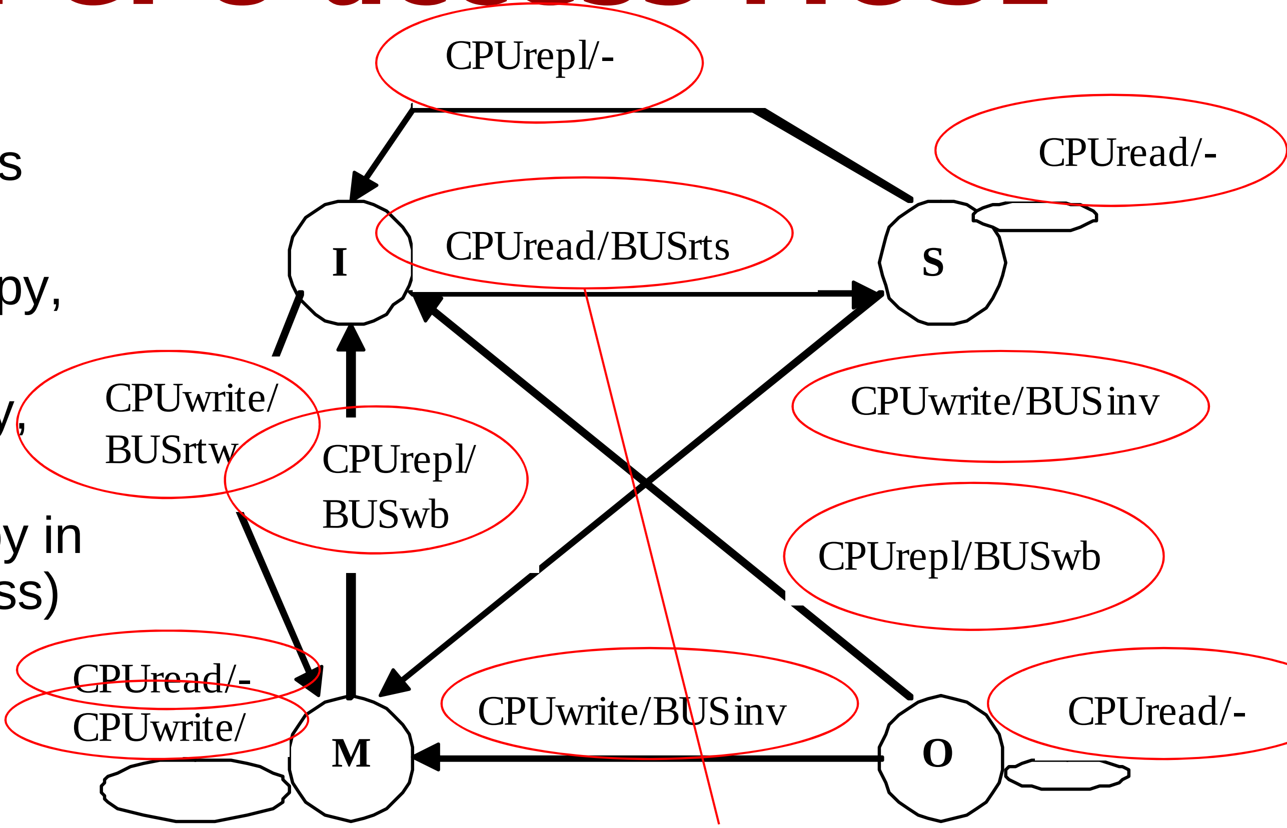
STATES:

M – Modified: My dirty* copy is the only cached copy

S – Shared: I have a clean copy, others may also have a copy

O – Owner: I have a dirty copy, others may also have a copy

I – Invalid: I have no valid copy in my cache (may be a cache miss)



Input-signal/Reply-signal
Meaning: If you are in state I and see CPUread, send a BUSrts and goto S

FROM MY CPU:

CPUread Caused by a Load instruction

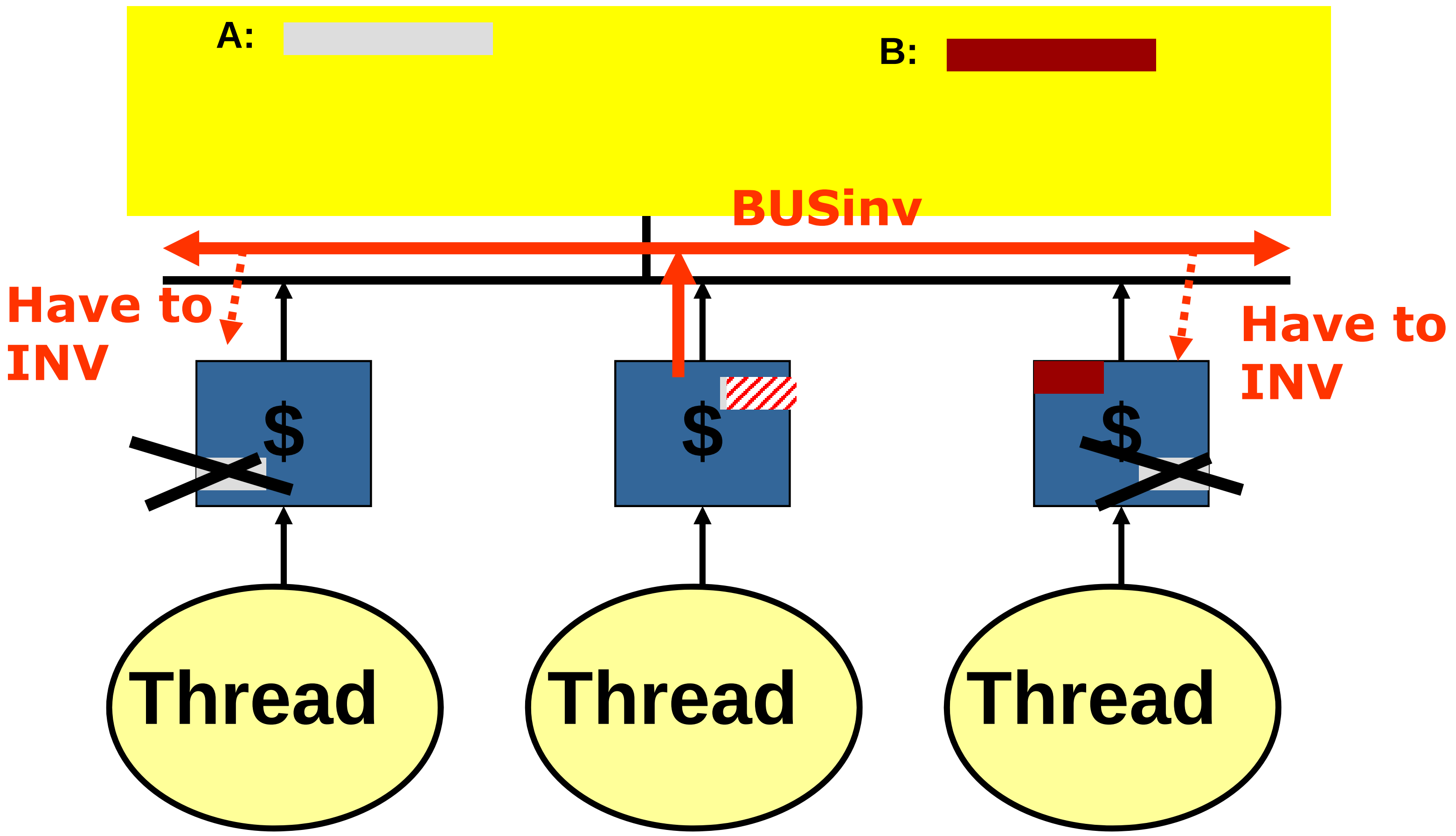
CPUwrite: Caused by a Store or Atomic instruction

CPUrepl: Caused by a replacement of this cachline (caused by murphy ☺)

AVDARK 2013

UPPSALA UNIVERSITET

”Upgrade” in snoop-based



Read A	…	Read B
Read A	Read A	…
…	…	Read A
…	Write A	

AVDARK 2013

UPPSALA UNIVERSITET

Summary

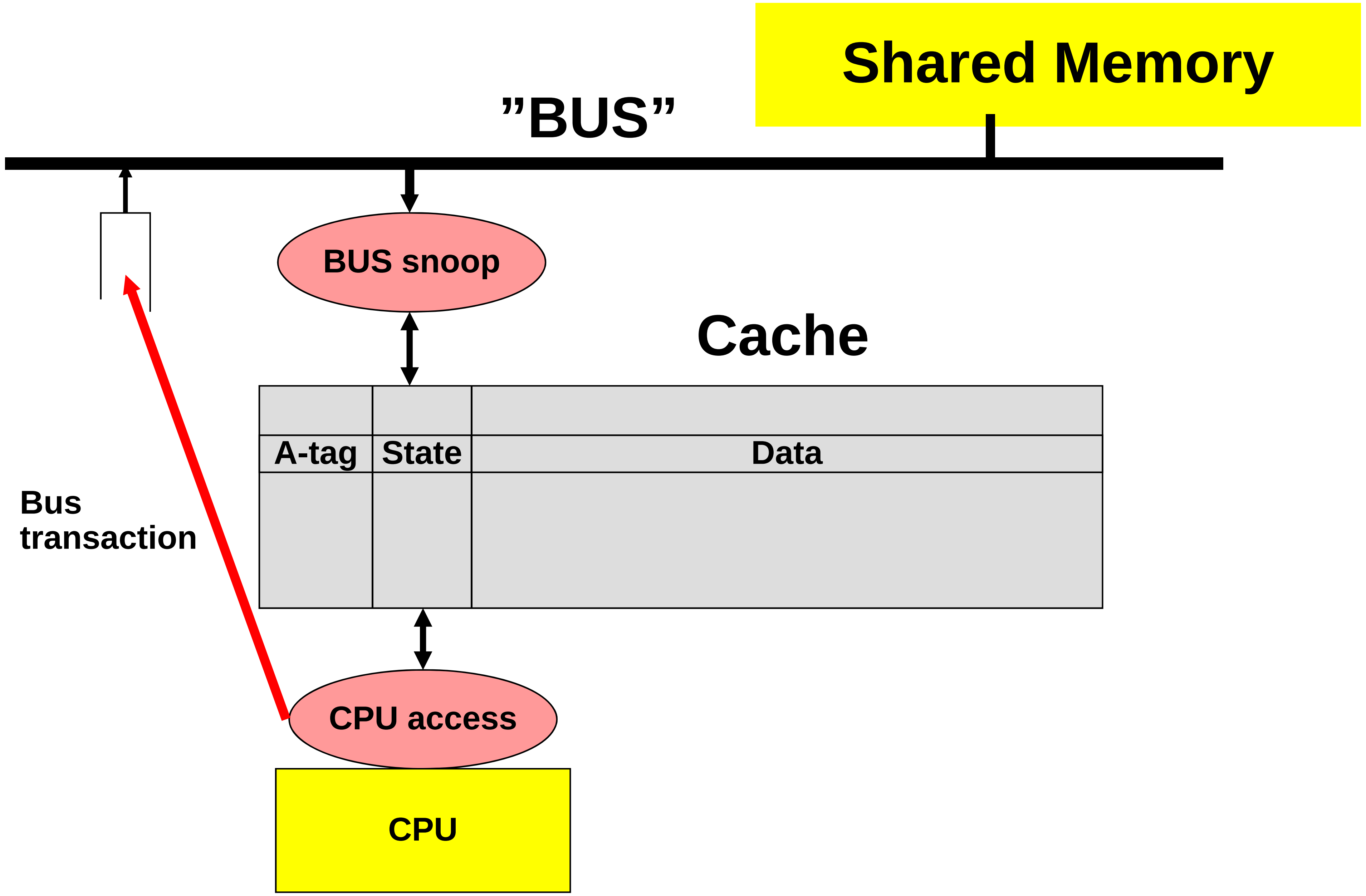
- Snooping was first suggested by Jim Goodman in ISCA, Stockholm 1984
- Effectively implements coherence through broadcast of ”read and write misses”
- Best suited for on-chip coherence between a small number of caches

MOSI Snooping Coherence Protocol Implementation

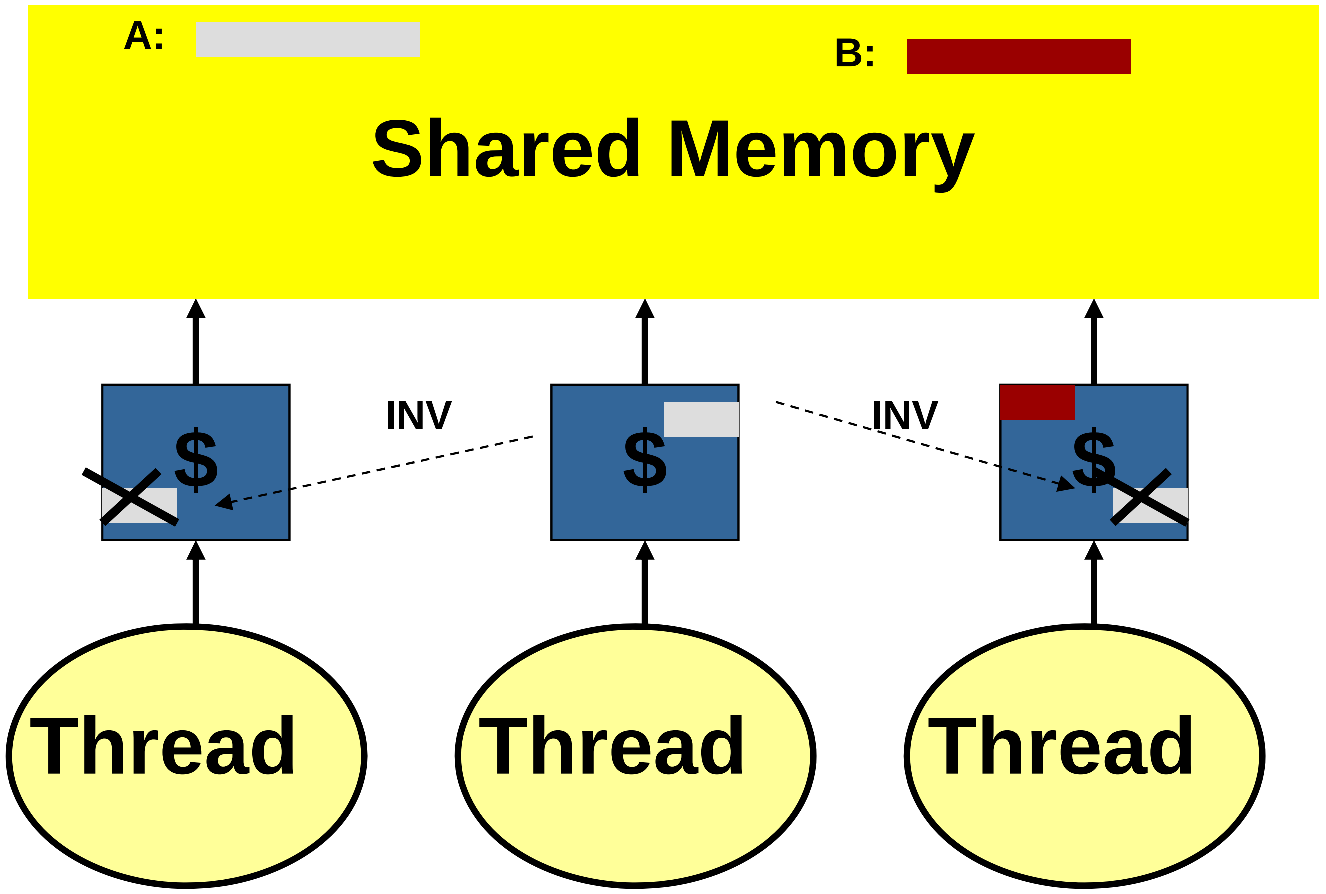
Erik Hagersten

Uppsala University

Snoop-based Protocol Implementation



Upgrade: Readable →Writable



Read A	…	Read B
Read A	Read A	…
…	…	Read A
…	Write A	

UPPSALA UNIVERSITET

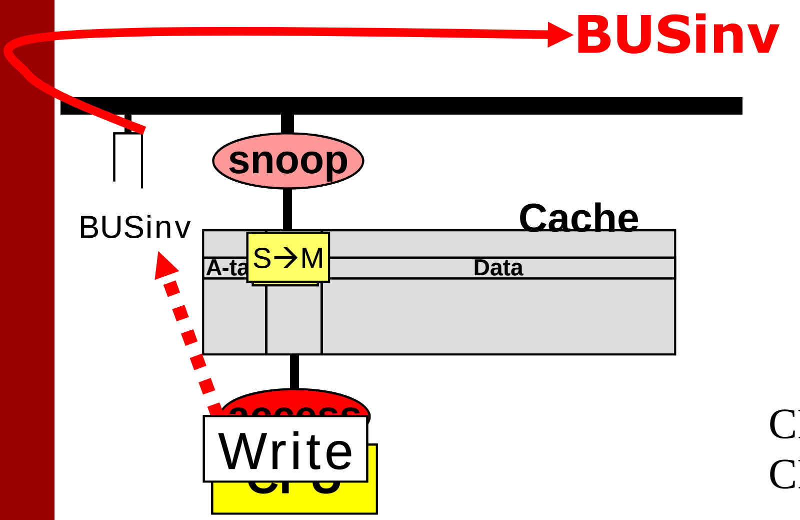
Upgrade – the requesting CPU

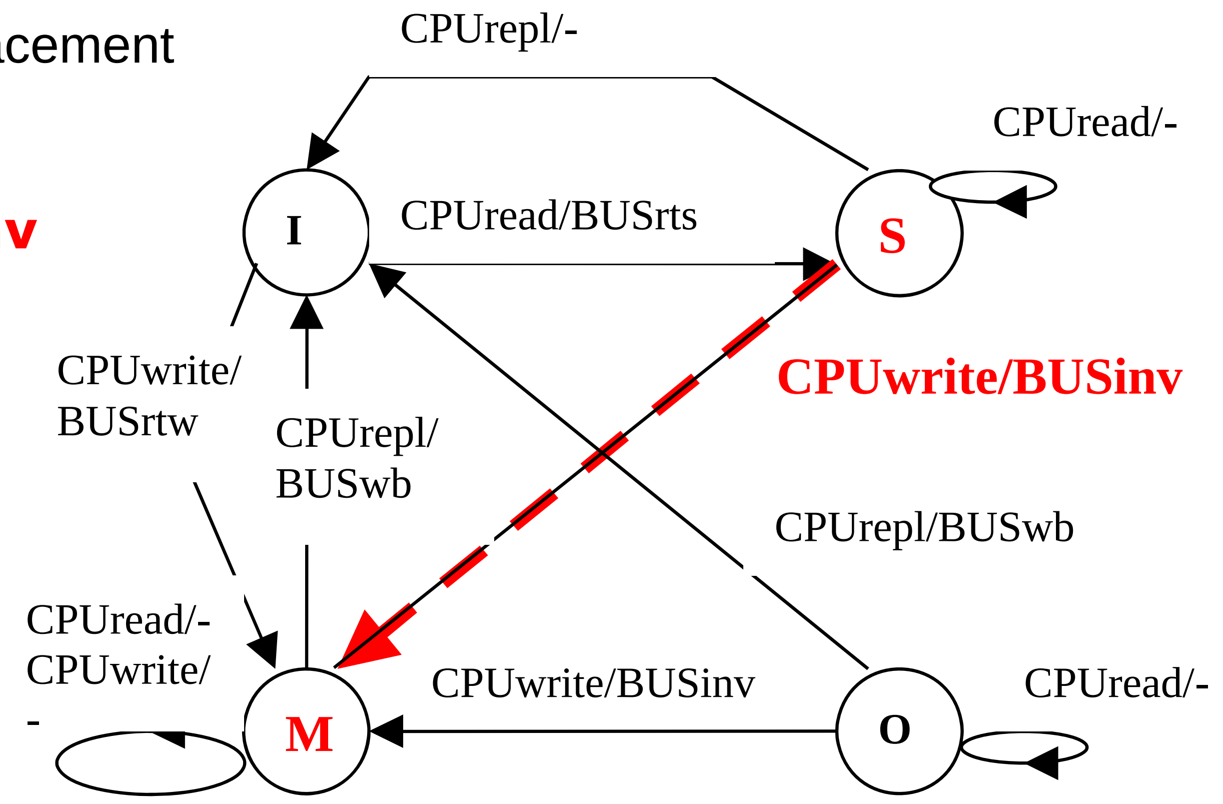
Defines action to CPU events:

CPUwrite: Caused by a store miss

CPUread Caused by a load miss

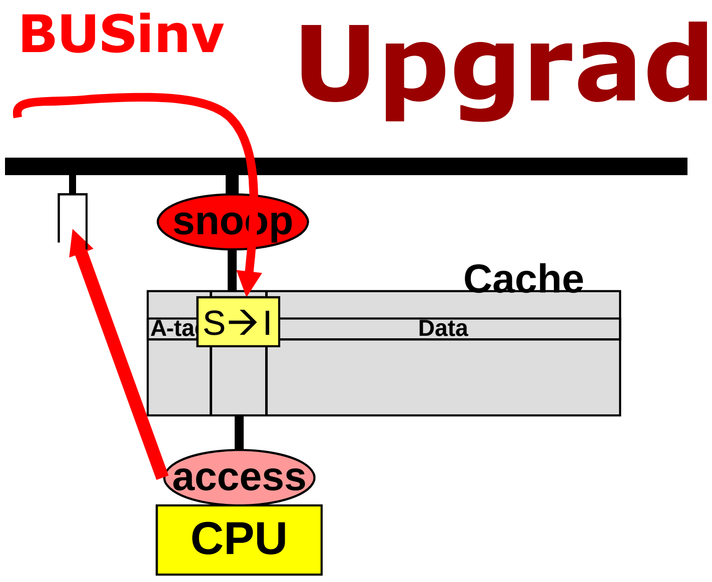
CPUrepl: Caused by a replacement

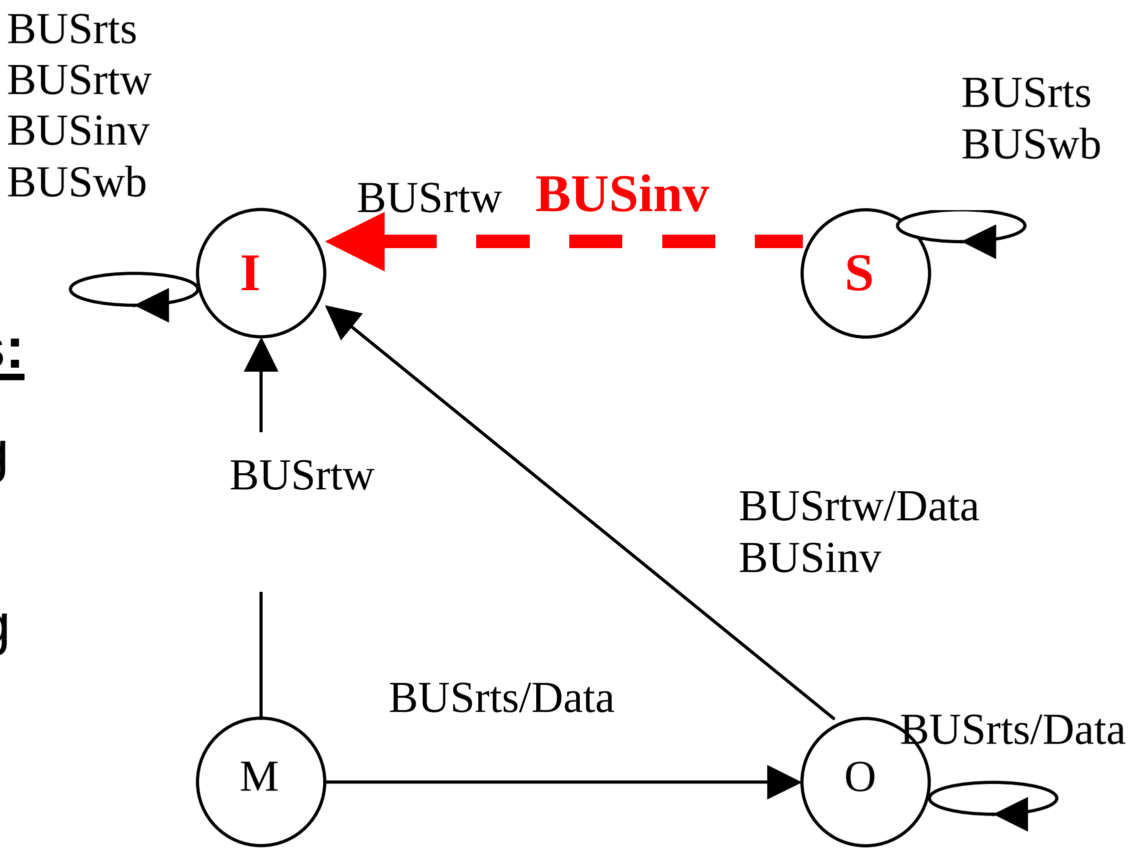




AVDARK 2013

Upgrade – the other CPUs





Defines action to Bus snoops:

BUSrts: ReadtoShare (reading the cacheline)

BUSrtw, ReadToWrite (reading the cacheline with the intention to modify it right away)

BUSwb: Writing a cacheline back to memory

BUSinv: Invalidating other caches copies of the cacheline

UPPSALA UNIVERSITET

More Cache Lingo

- **Capacity miss** – too small cache
- **Conflict miss** – limited associativity
- **Compulsory miss** – accessing data the first time
- **Coherence miss** – The cache would have had the data unless it had been invalidated by someone else
- **Upgrade miss:** (only for writes) – The cache would have had a writable copy, but answered a read request and "downgraded" itself to read-only state
- **False sharing:** Coherence/downgrade is caused by a shared cacheline and not by shared data:

False sharing example:

Read A	…
…	Read D
Write A	…
…	Write D
Read A	

cacheline:

A, B, C, D

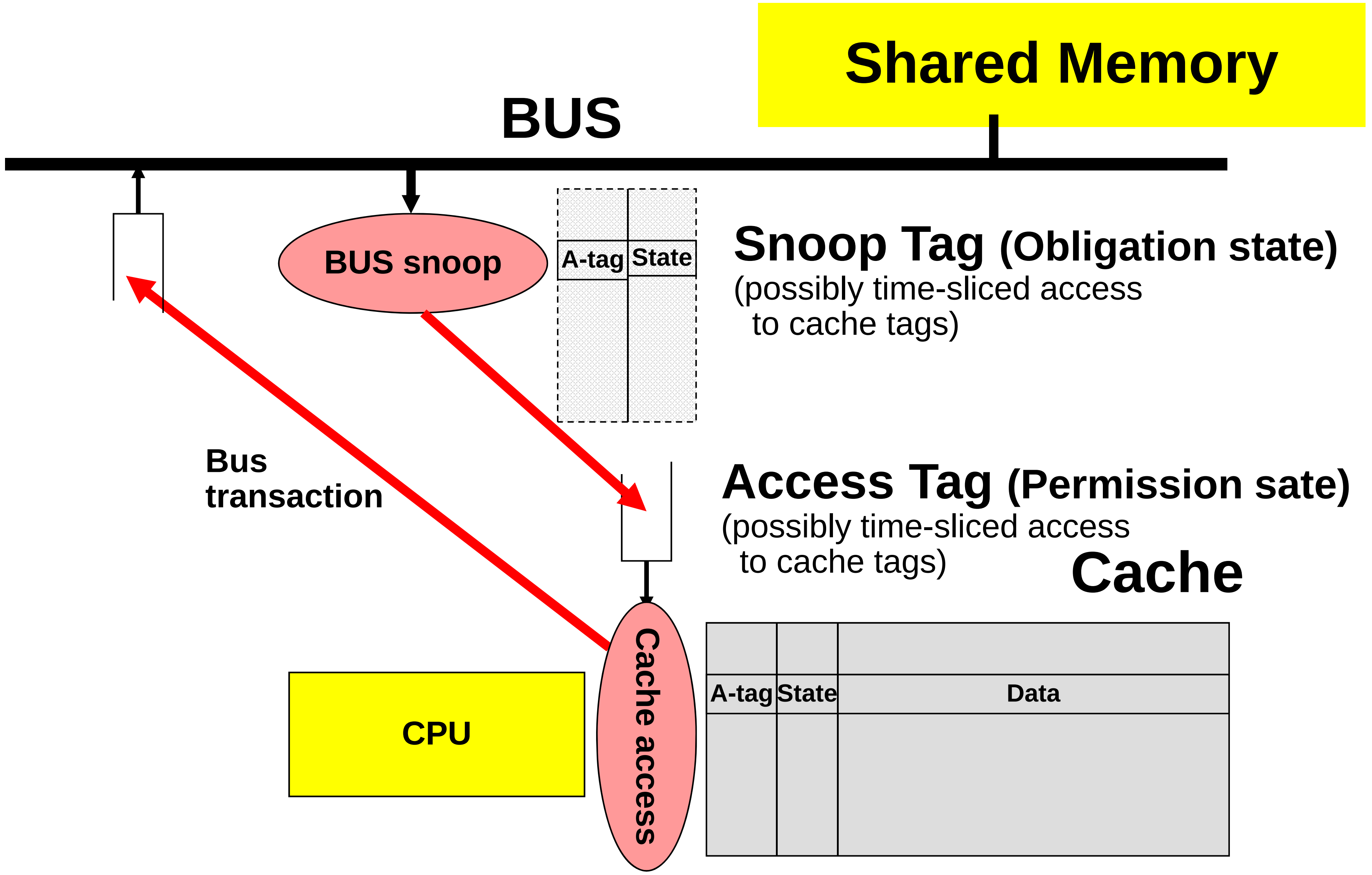
AVDARK 2013

Implementing Snooping. One example (Sun E6000, ≈Intel P6,)

Erik Hagersten

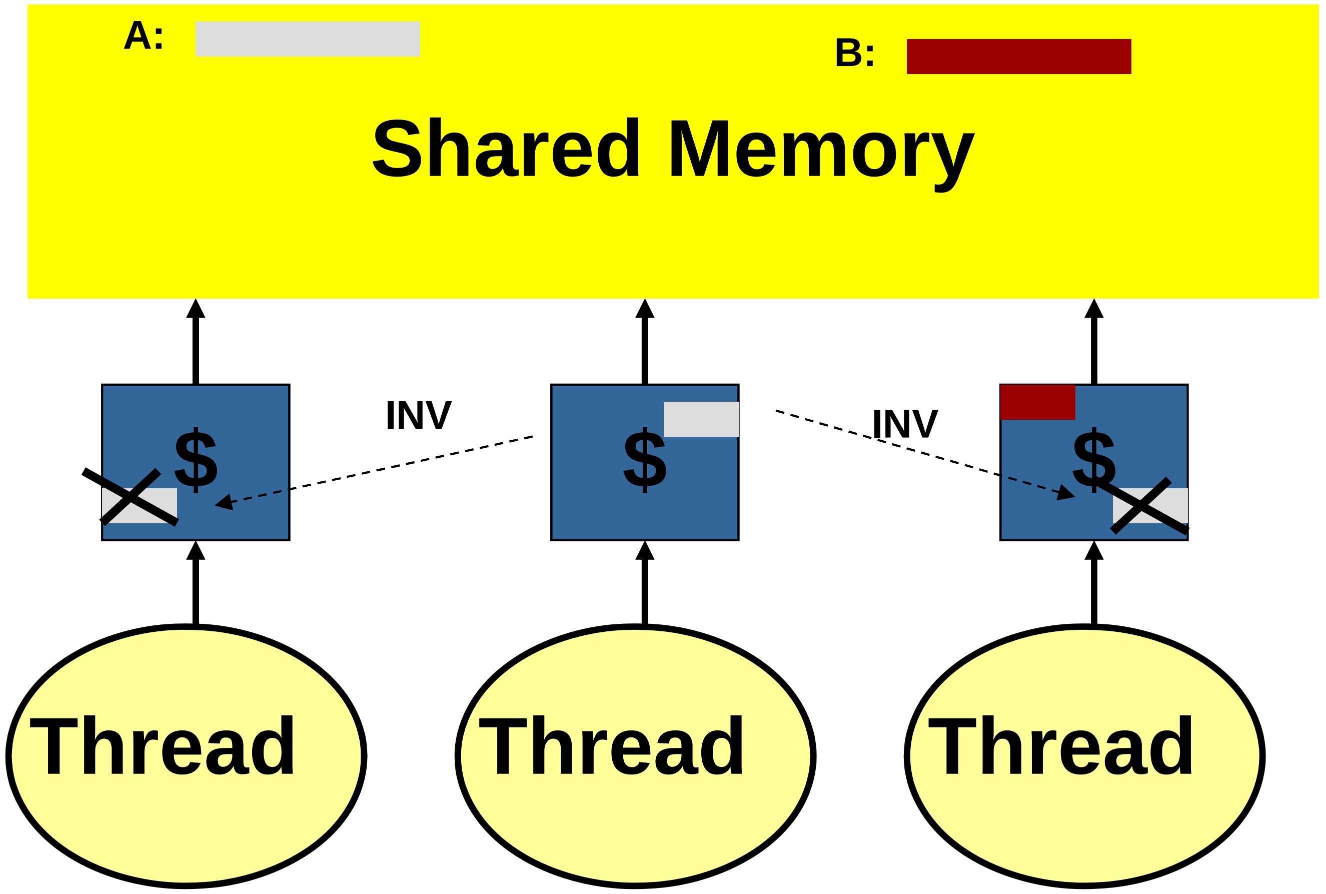
Uppsala University

Snoop-based architecture: Dual tags


Shared Memory
BUS
BUS snoop
A-tag
State
Snoop Tag (Obligation state)
(possibly time-sliced access to cache tags)
Bus transaction
Access Tag (Permission sate)
(possibly time-sliced access to cache tags)
Cache
CPU
Cache access
A-tag
State
Data

UPPSALA UNIVERSITET

The Cache Coherent Memory System
Coherent Write (Here: Write invalidate)



Read A
Read A
…
…

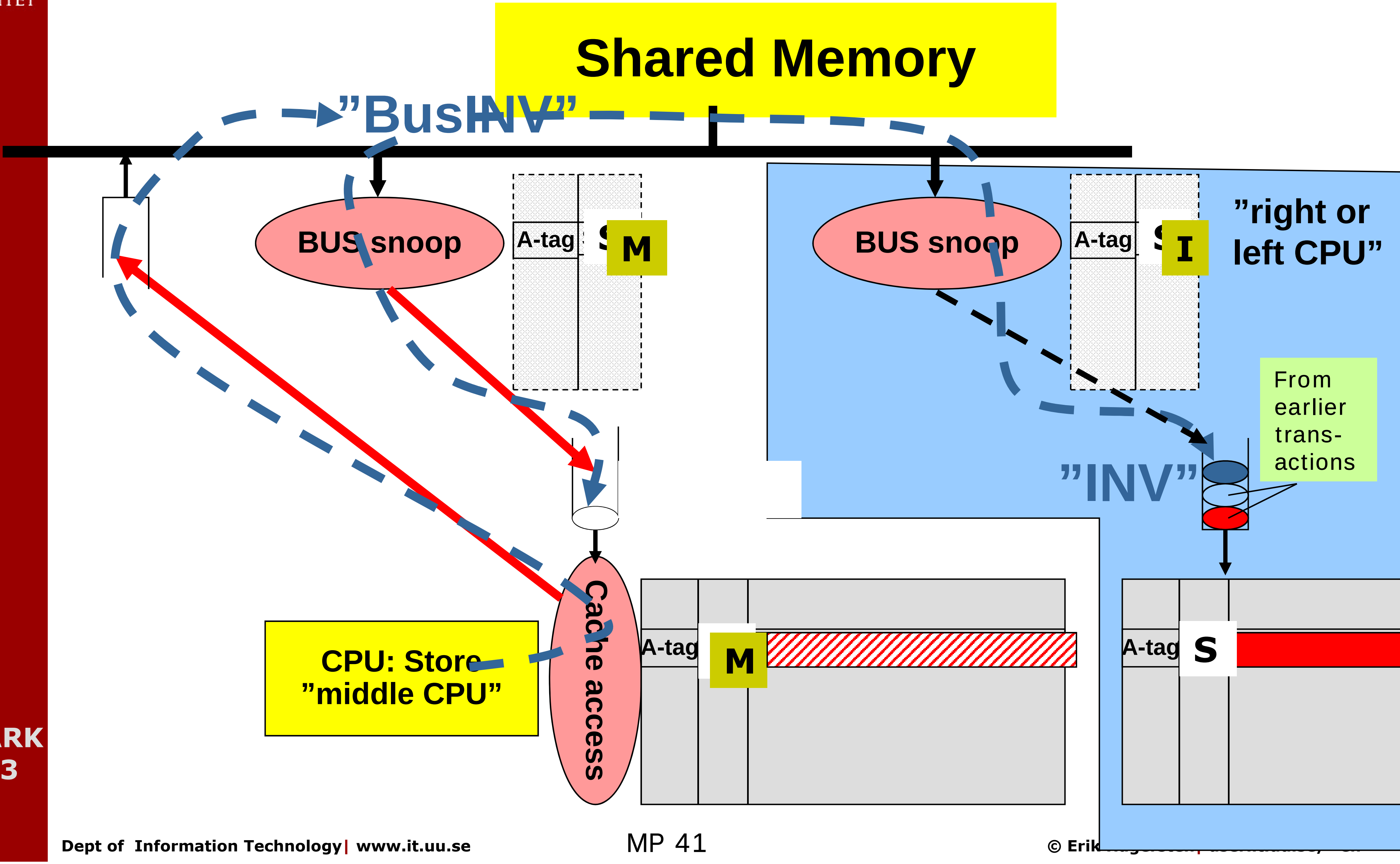
…
Read A
…
Write A

Read B
…
Read A

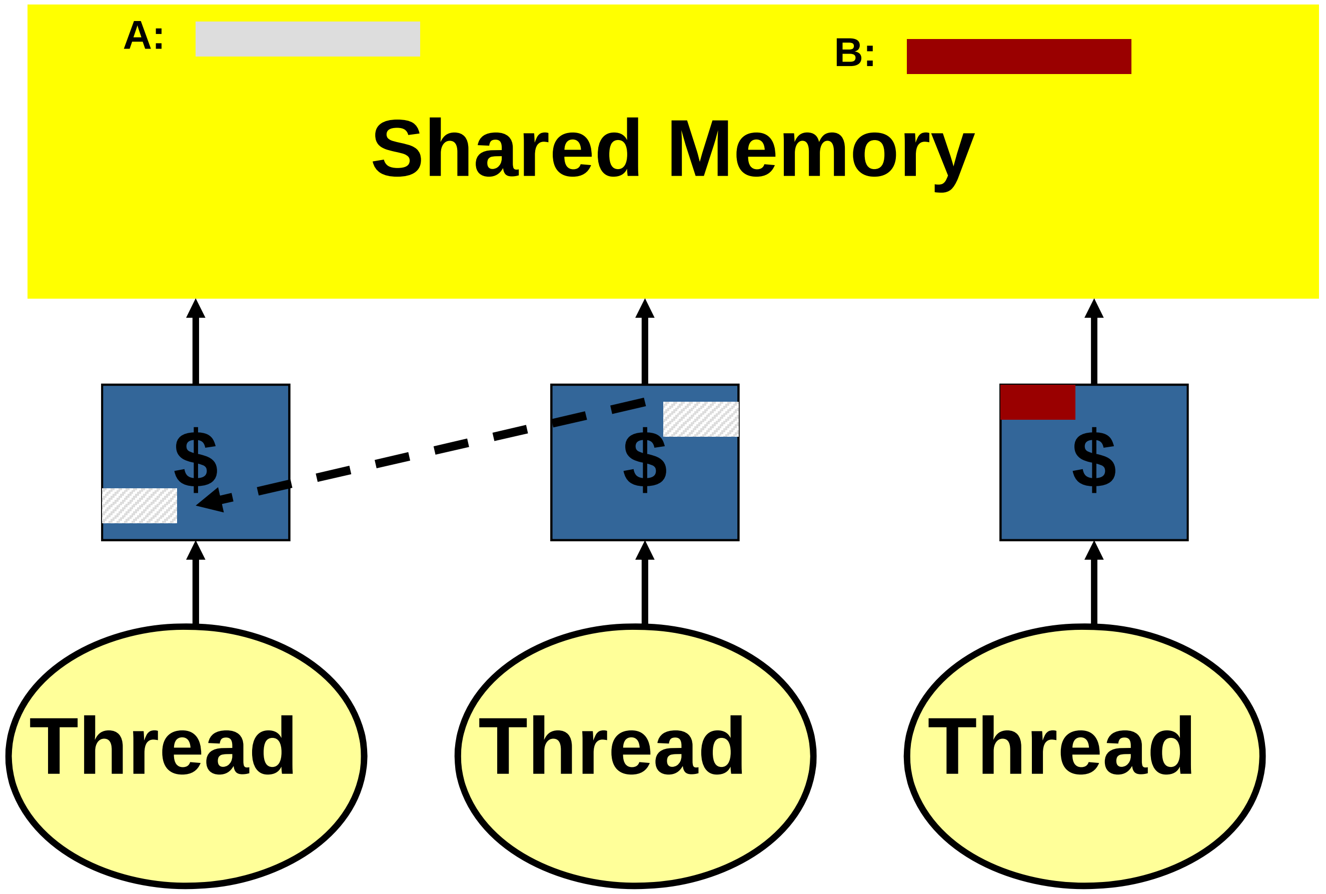
AVDARK 2013

UPPSALA UNIVERSITET

”Upgrade” in snooped-based



The Cache Coherent Cache-to-cache



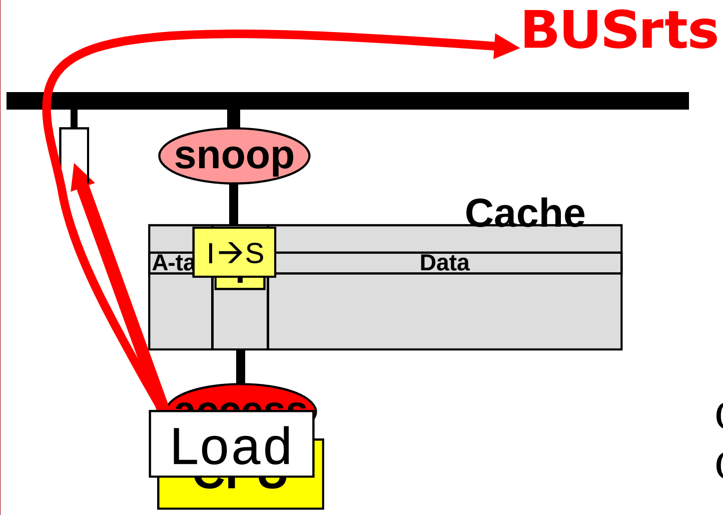
Read A	...	Read B
Read A	Read A	…
…	…	Read A
…	Write A	
<u>Read A</u>		

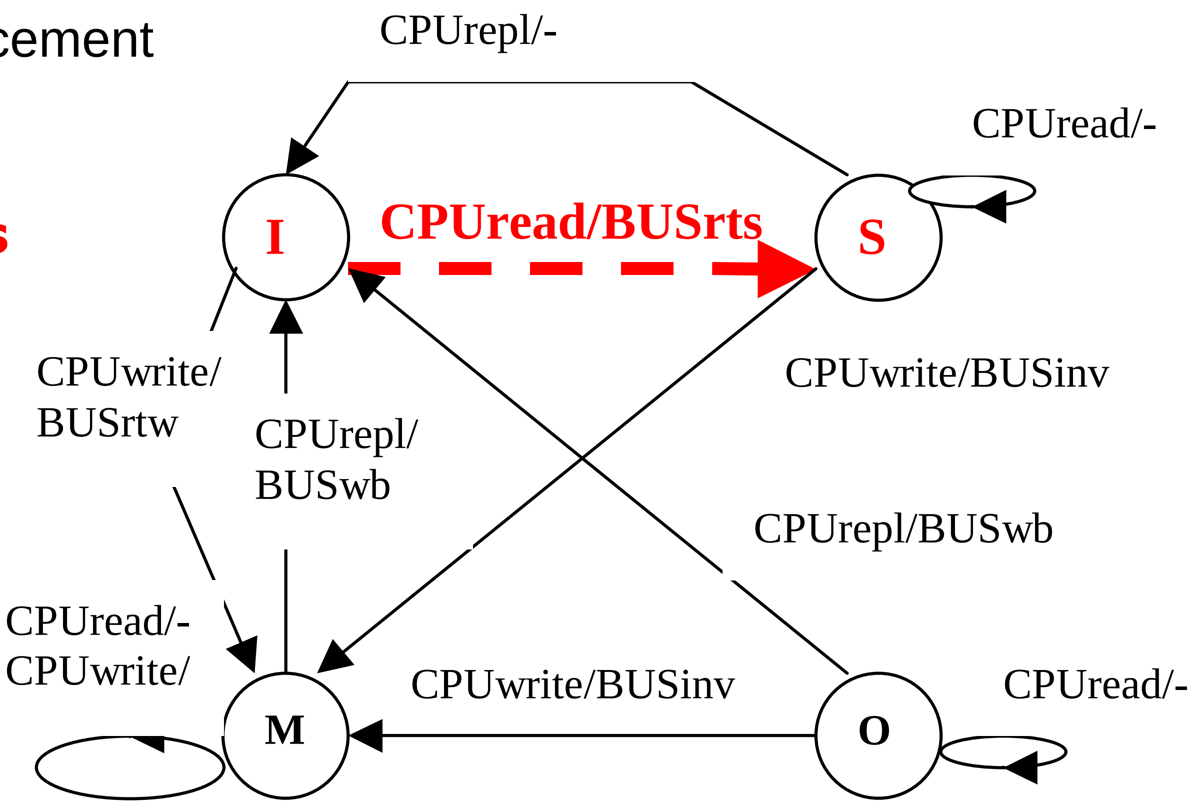
Cache2cache – the requesting CPU

CPUwrite: Caused by a store miss

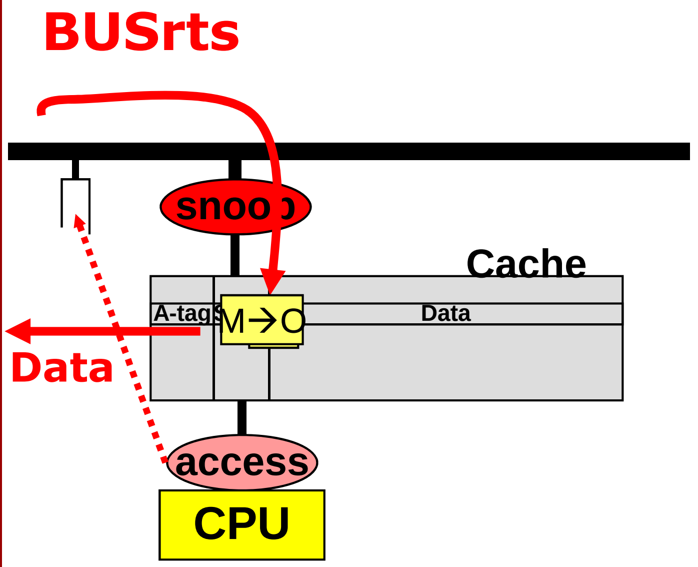
CPUread Caused by a loadmiss

CPUrepl: Caused by a replacement





Cache-to-cache – the other CPU

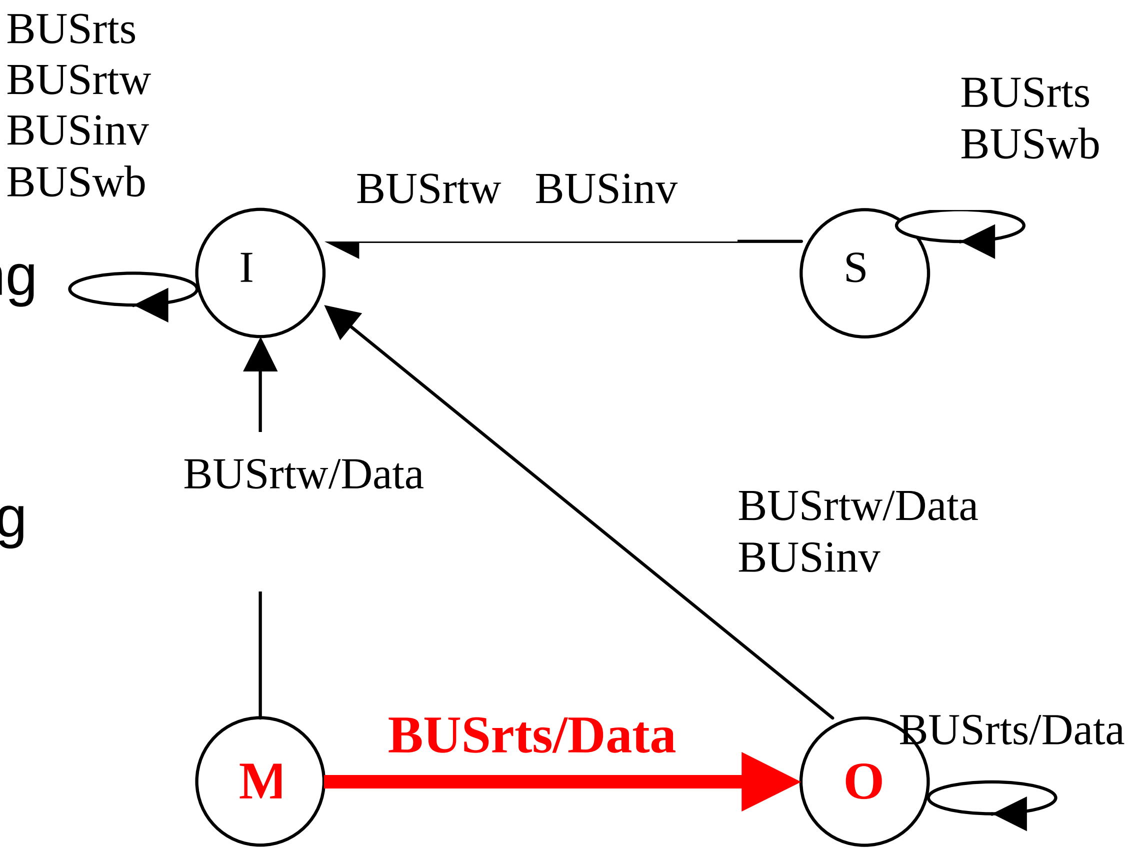


BUSrts: ReadToShare (reading the data with the intention to read it)

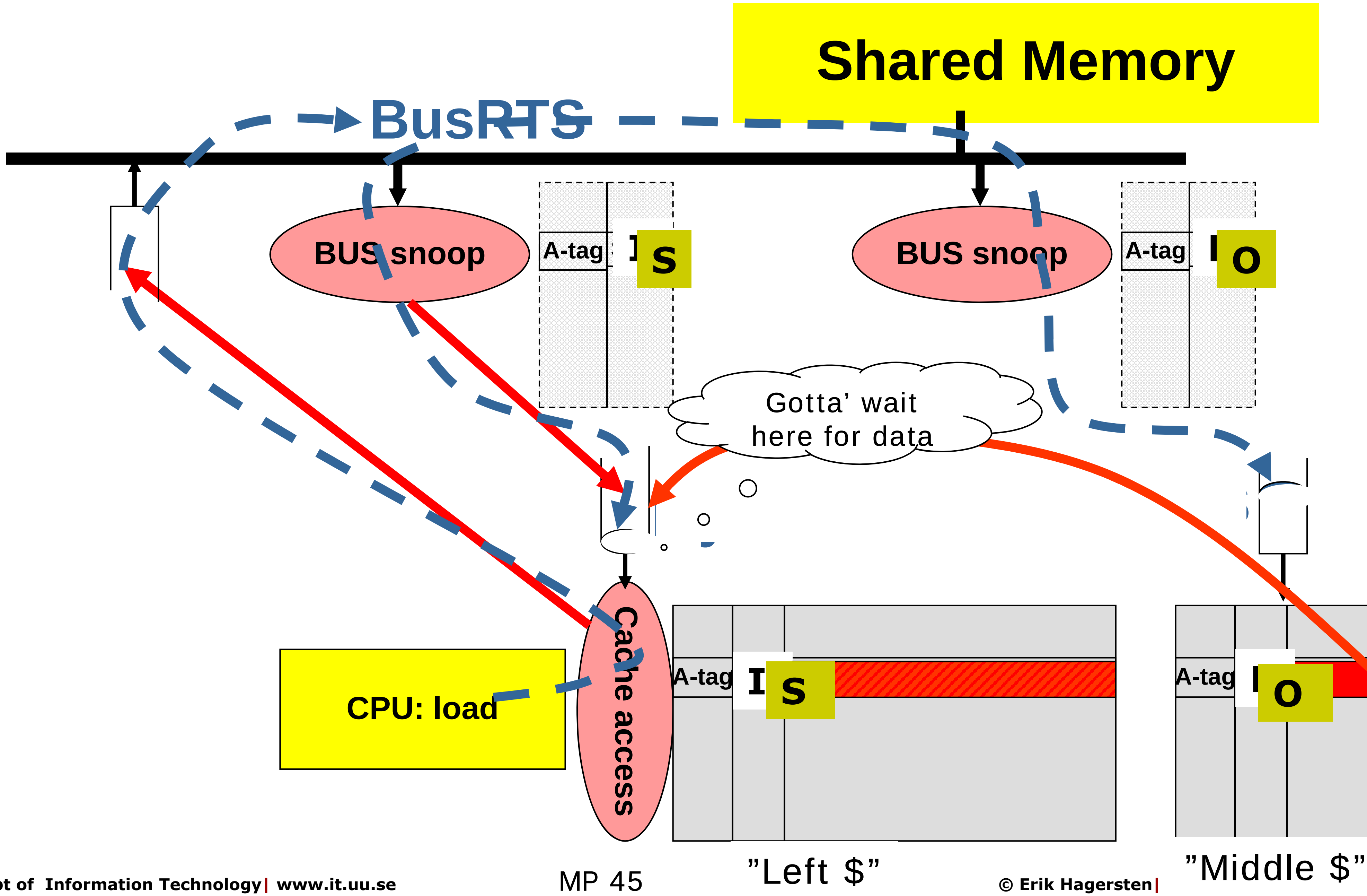
BUSrtw, ReadToWrite (reading the data with the intention to modify it)

BUSwb: Writing data back to memory

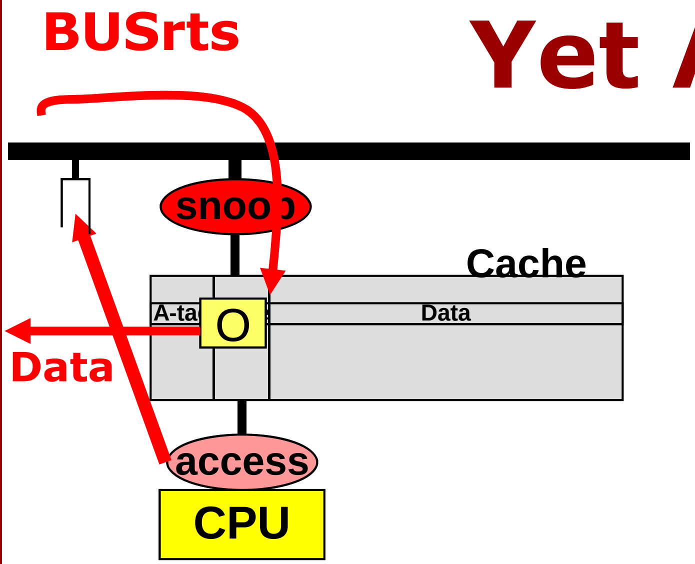
BUSinv: Invalidating other caches copies



Cache-to-cache in snoope-based



Yet Another Cache-to-cache

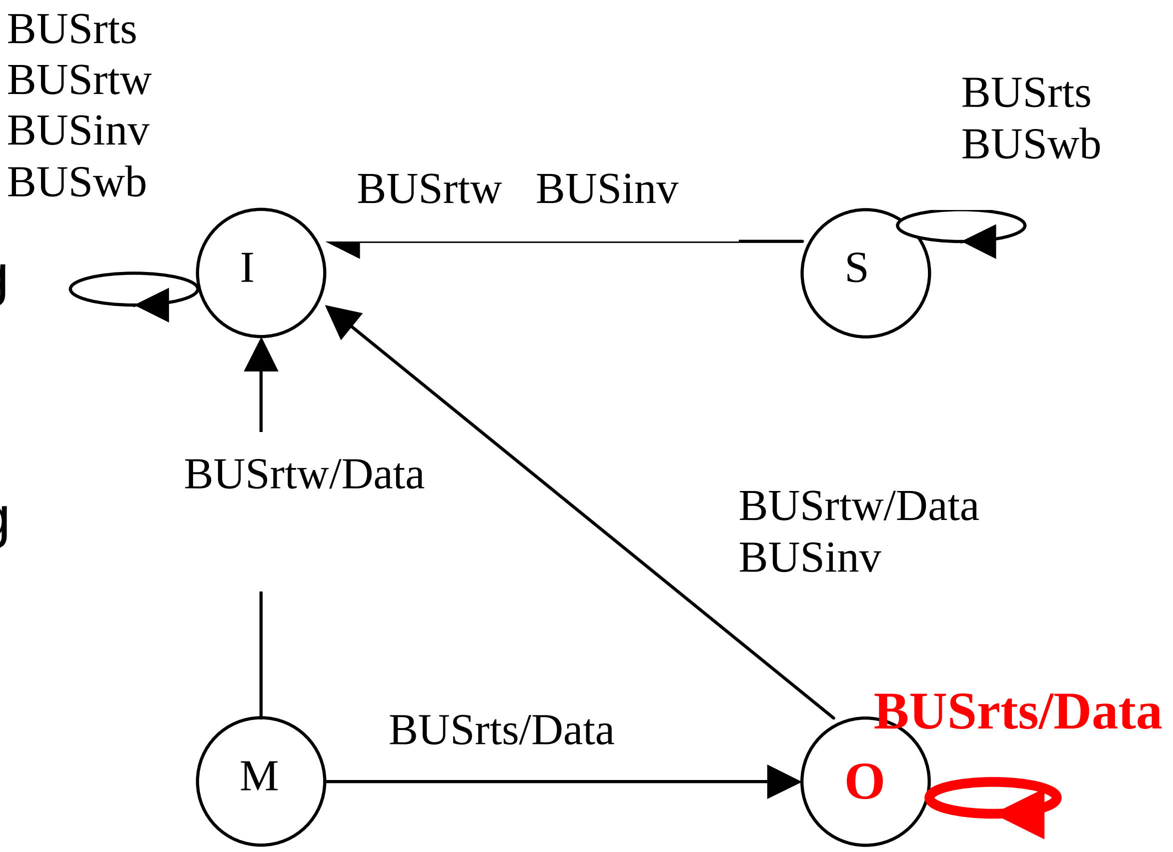


BUSrts: ReadtoShare (reading the data with the intention to read it)

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

BUSwb: Writing data back to memory

BUSinv: Invalidating other caches copies



UPPSALA UNIVERSITET

Summary

- Dual tags enable bus snoops and CPU lookups in parallel
- A datum may actually have several values ”at the same wall-clock time”
- … but not in ”logic time”: No software can detect that there are different values
- The value-change order maintained

AVDARK 2013

Other Coherence Alternatives

Erik Hagersten

Uppsala University

Common Cache States

- M – Modified
  My dirty (i.e. modified) 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

UPPSALA UNIVERSITET

Some Coherence Alternatives

- MOSI

Our first target

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

AVDARK 2013

Example A = A + 1

Initially A is only in mem

UPPSALA UNIVERSITET

MOSI:

CPU	BUS	State
LD A…	RTS(A)	S
ADD 1…	-	
ST A…	INV(A)	M
LD B	RTS(B)	S
ADD 1	-	
ST B	INV(B)	M
…		

MOESI:

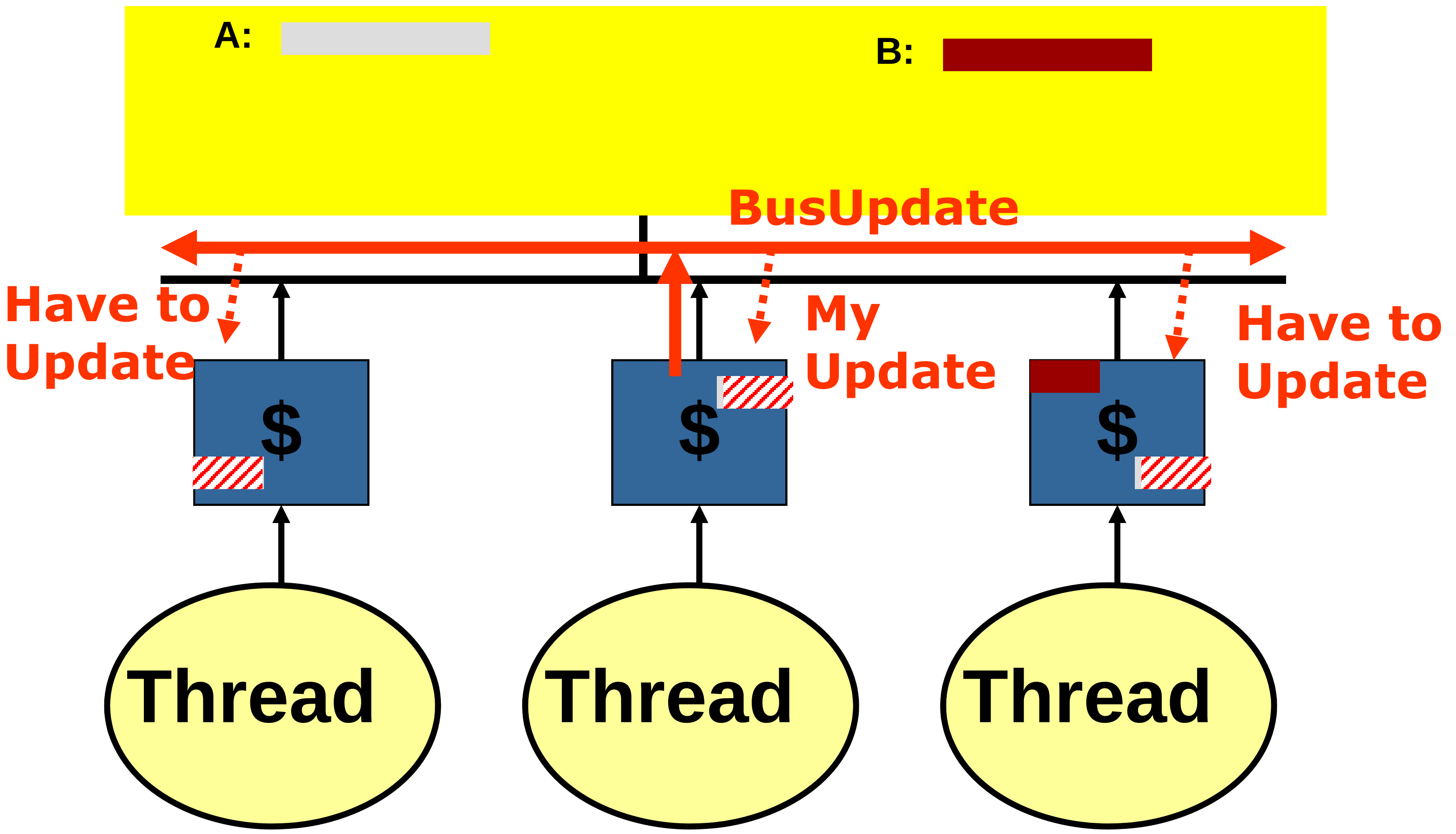
CPU	BUS	State
LD A	RTS(A)	E
ADD 1	-	
ST A	-	M
LD B	RTS(B)	E
ADD 1	-	
ST B	-	M
…		

AVDARK 2013

UPPSALA UNIVERSITET

AVDARK 2013

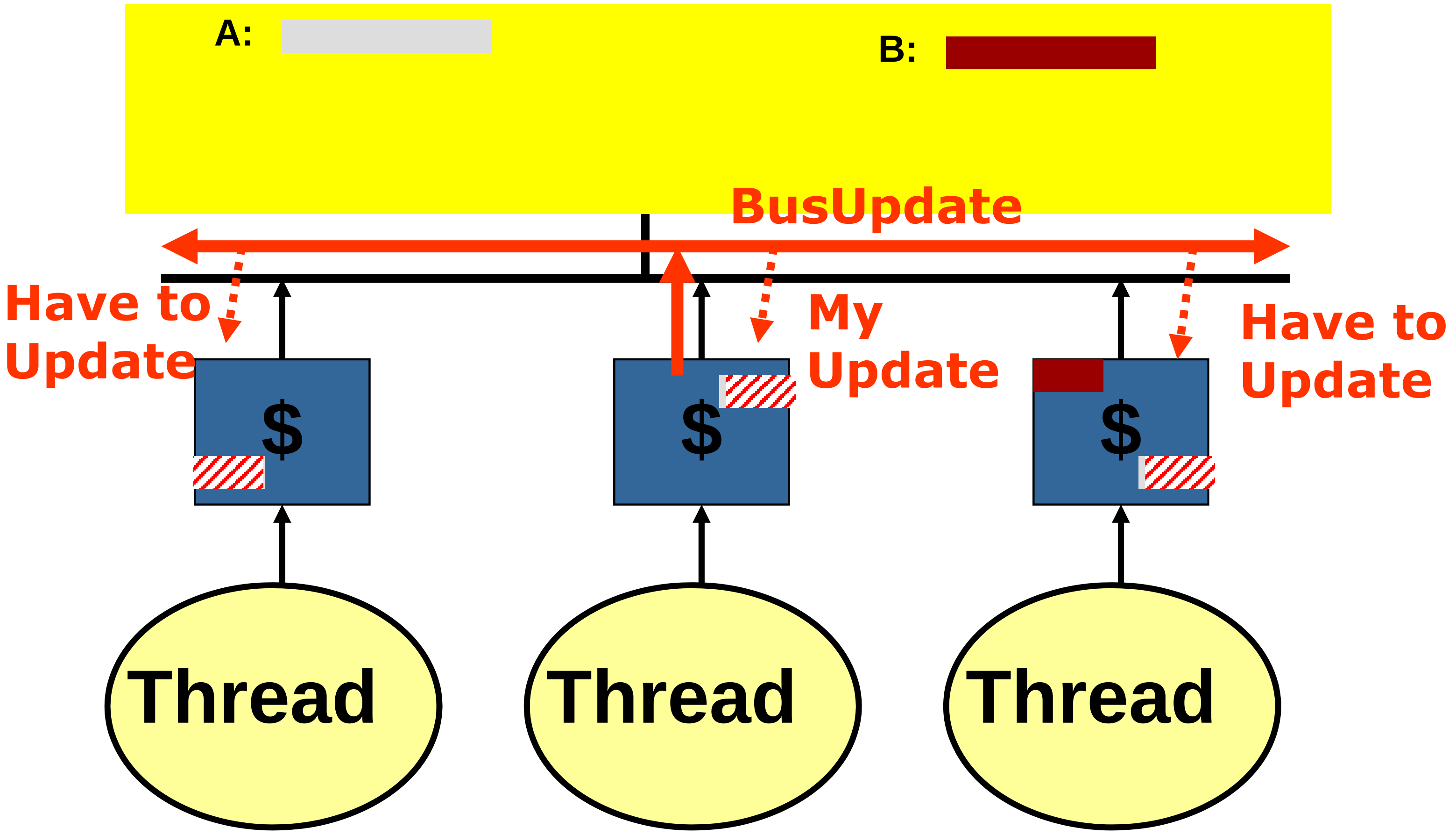
Update-based MOSI protocol



Read A	…	Read B
Read A	Read A	…
…	…	Read A
…	Write A	
Read A ➔ HIT		

UPPSALA UNIVERSITET

Update-based MOSI protocol: Next write



Read A	…	Read B
Read A	Read A	…
…	…	Read A
…	<u>Write A</u>	
Read A	<u>Write A</u>	

AVDARK 2013

Update-based Coherence

- Write the new value to the other caches holding a shared copy (instead of invalidating...)
- Can avoid coherence misses
- May consume a large amount of snoop bandwidth
- Hard to implement some ”memory models”
- Few commercial implementations: SPARCCenter2000, Xerox Dragon

Preparing for IRL

Example during next IRL Class:

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;
A = A + 1;              /* this is a primitive kind of unlock */
while (A != 4) {};      /* this is a primitive kind of barrier sync */
<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.
Assume that CPU3, 2 and 1 first make one memory reference (i.e, a load or a store) each and then repeats that interleaving.

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 <u>change</u> and/or value <u>change</u> 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).

UPPSALA UNIVERSITET

AVDARK 2013

Example of a state transition sheet:

CPU action	**Bus Transactio n (if any)**	**State/value after the CPU action**						**Data is provided by** [Cache 1, 2, 3 or Mem] (if any)
		CPU1 A	B	CPU2 A	B	CPU3 A	B	
Initially		I	I	I	I	I	I	
CPU3: LD A	RTS(A)					S/1		Mem
CPU2: LD A	RTS(A)			S/1				Mem
CPU1: LD A	RTS(A)	S/1						Mem
CPU3: LDA	—							—

What are Memory Models?

Erik Hagersten
Uppsala University
Sweden

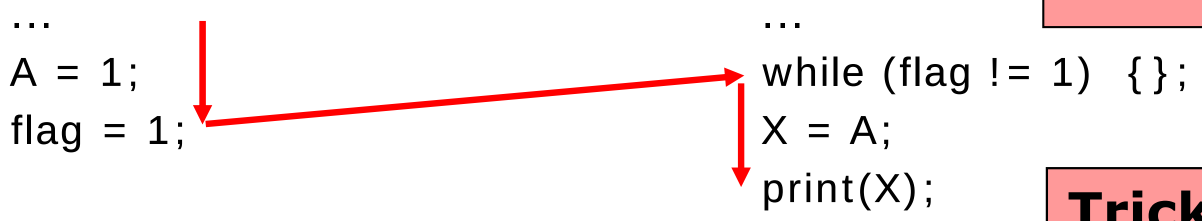
Where Memory Models Matter

Trick question
What value will be printed?

- ☐ 0
- ☐ 1
- ☐ Undefined (either 0 or 1)

- Flag synchronization

(initially flag = 0 and A = 0)



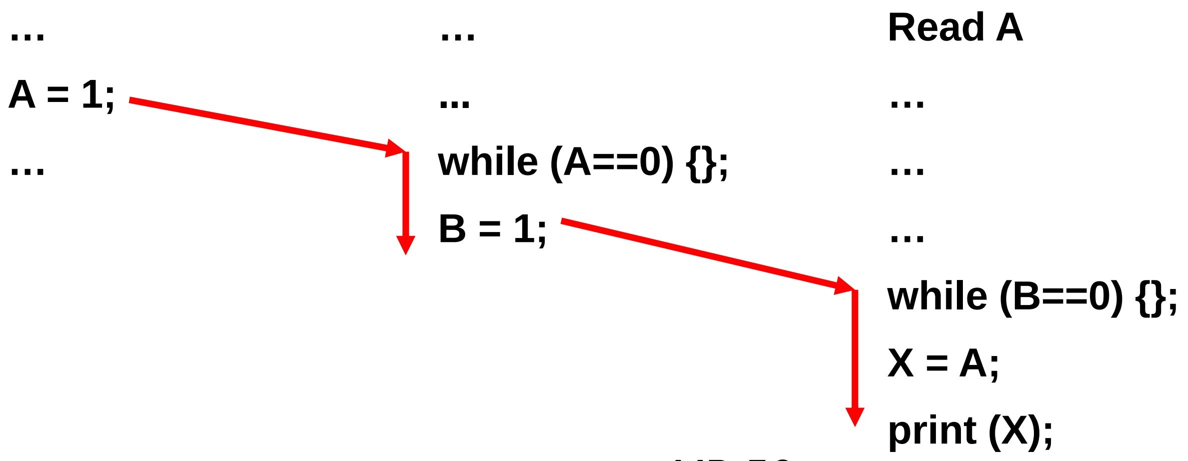
```
...                          ...
A = 1;                       while (flag != 1)  {};
flag = 1;                    X = A;
                             print(X);
```

Trick question
What value will be printed?

- ☐ 0
- ☐ 1
- ☐ Undefined (either 0 or 1)

- Causality (Causal correctness)

(Initially A = 0 and B = 0)



```
...             ...                 Read A
A = 1;          ...                 ...
...             while (A==0) {};    ...
                B = 1;              ...
                                    while (B==0) {};
                                    X = A;
                                    print (X);
```

UPPSALA UNIVERSITET

Dekker's Algorithm (mutual exclusion)

Initially A = B = 0

"fork"

```
A := 1
if (B== 0) print("A won")
```

```
B := 1
if (A == 0)  print("B won")
```

Trick question
Is it possible that both threads "win"?

- ☐ Yes
- ☐ No
- ☐ Undefined

AVDARK
2013

UPPSALA UNIVERSITET

Memory Ordering

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

AVDARK 2013

Memory Ordering

- Defines the guaranteed memory ordering
- Is a ”contract” between the HW and SW guys
- Without it, you may not be able to say much about the result of a parallel execution

Observing order in SW

In which order did A and B change value?

Value order

LD newA; LD oldB → *newA before newB*

ST newA; LD oldB →*newA before newB*

ST newA; ST newB → *newA before newB*

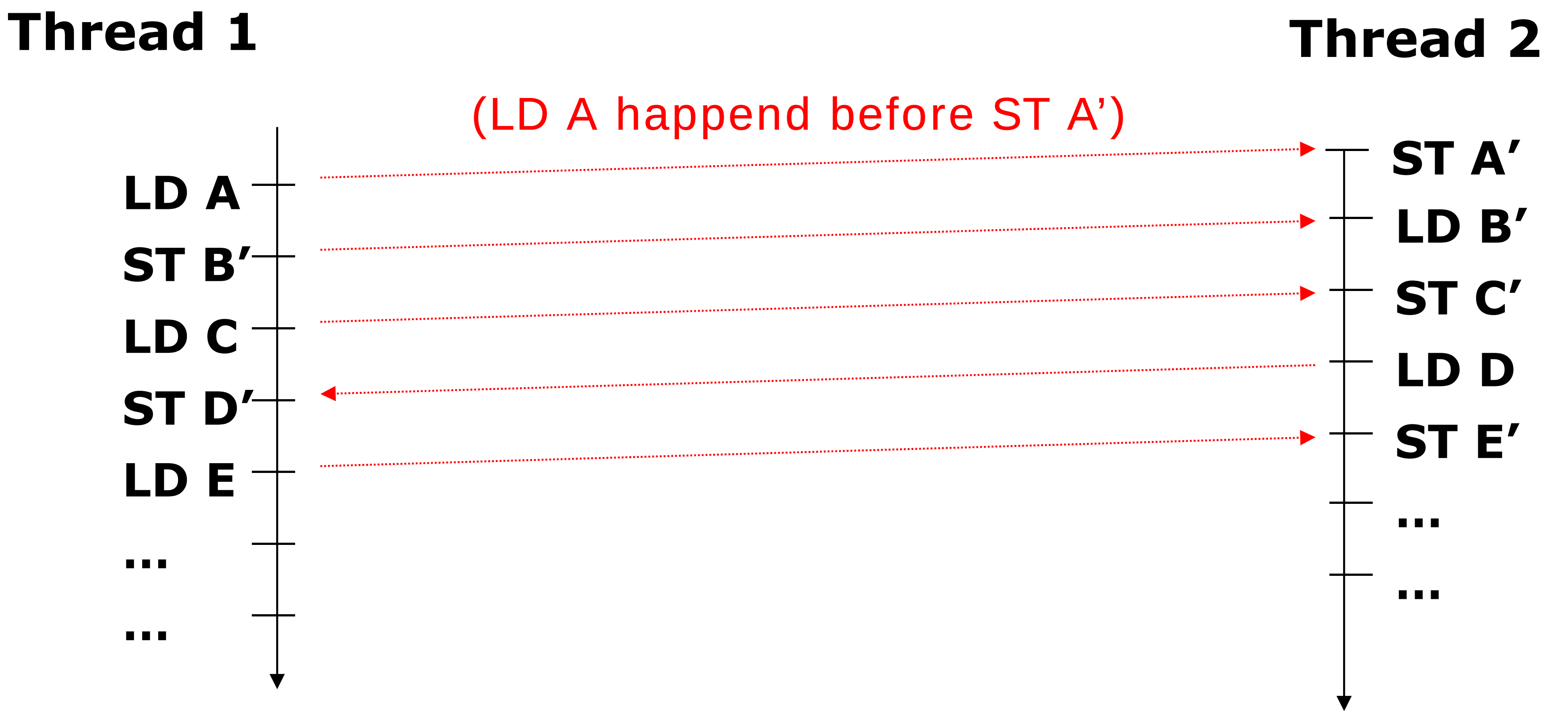
Program order

The order of program statements of each thread

UPPSALA UNIVERSITET

In which global order were these statements executed?

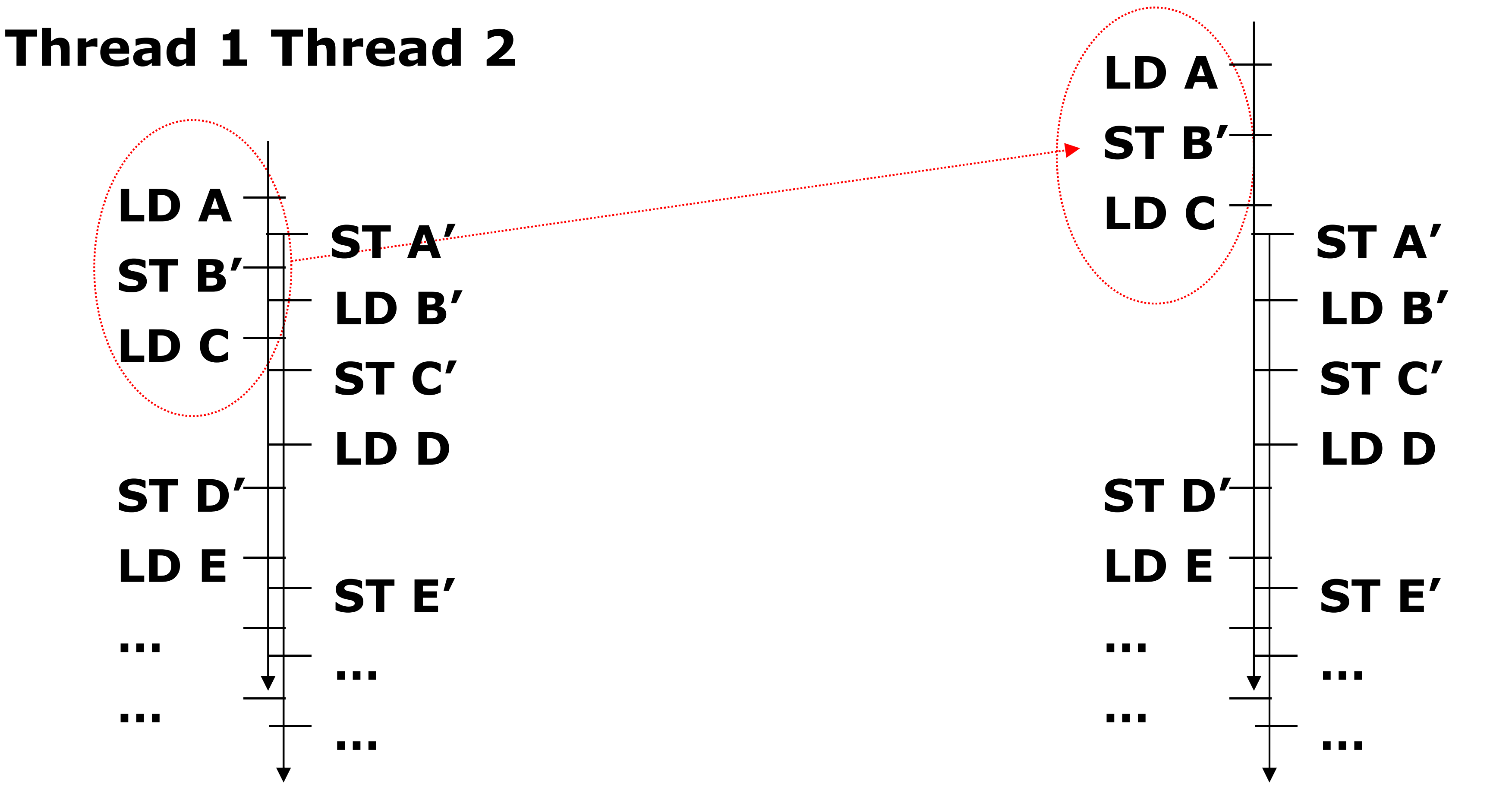
(A′ denotes a modified value to the data at addr A)



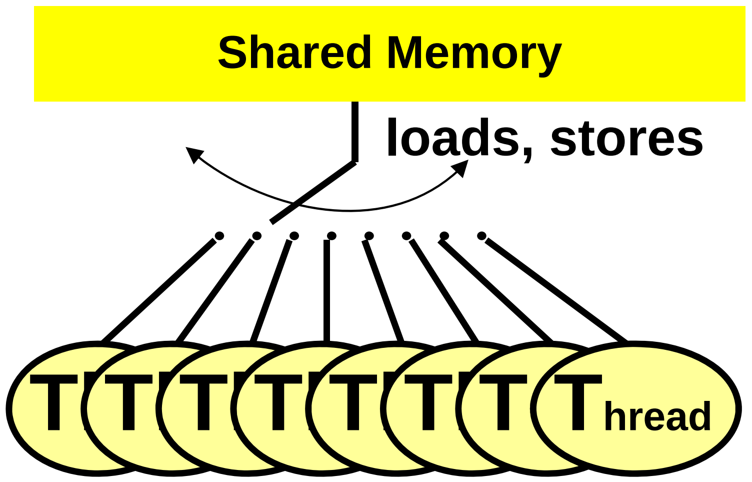
AVDARK 2013

One possible observed order

Another possible observed order



"The intuitive memory order" Sequential Consistency (Lamport)



- Global order achieved by *interleaving* all memory accesses from different threads
- SW should not be able to detect contradictive orders
- "Programmer's intuition is maintained"
- Unnecessarily restrictive ==> performance penalty

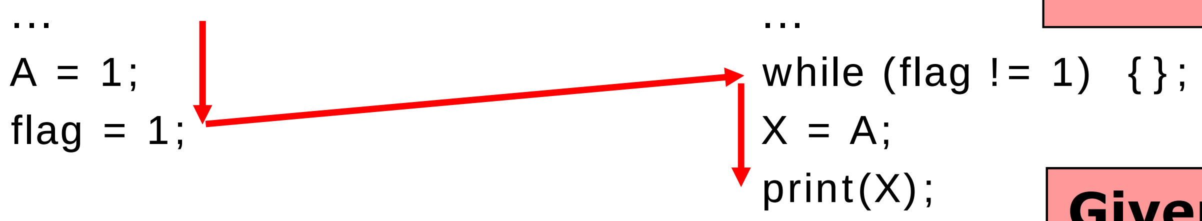
UPPSALA UNIVERSITET

AVDARK 2013

Where Memory Models Matters

- Flag synchronization

(initially flag = 0 and A = 0)



```
...                         ...
A = 1;                      while (flag != 1)  {};
flag = 1;                   X = A;
                            print(X);
```

Given Sequential Consistecy: What value will be printed?

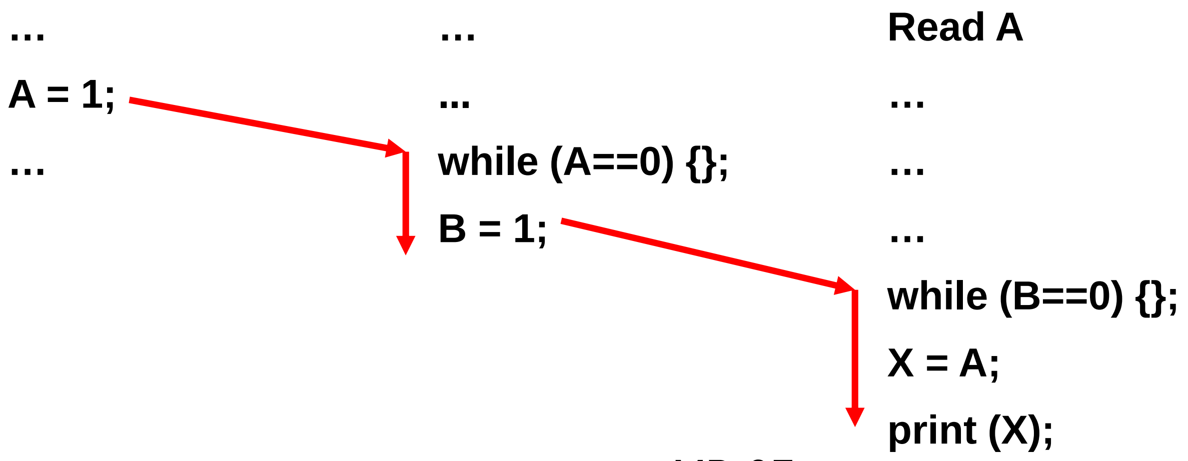
☐ 0

☐ 1

☐ Undefined (either 0 or 1)

- Causality (Causal correctness)

(Initially A = 0 and B = 0)



```
...               ...                Read A
A = 1;            ...                ...
...               while (A==0) {};   ...
                  B = 1;             ...
                                     while (B==0) {};
                                     X = A;
                                     print (X);
```

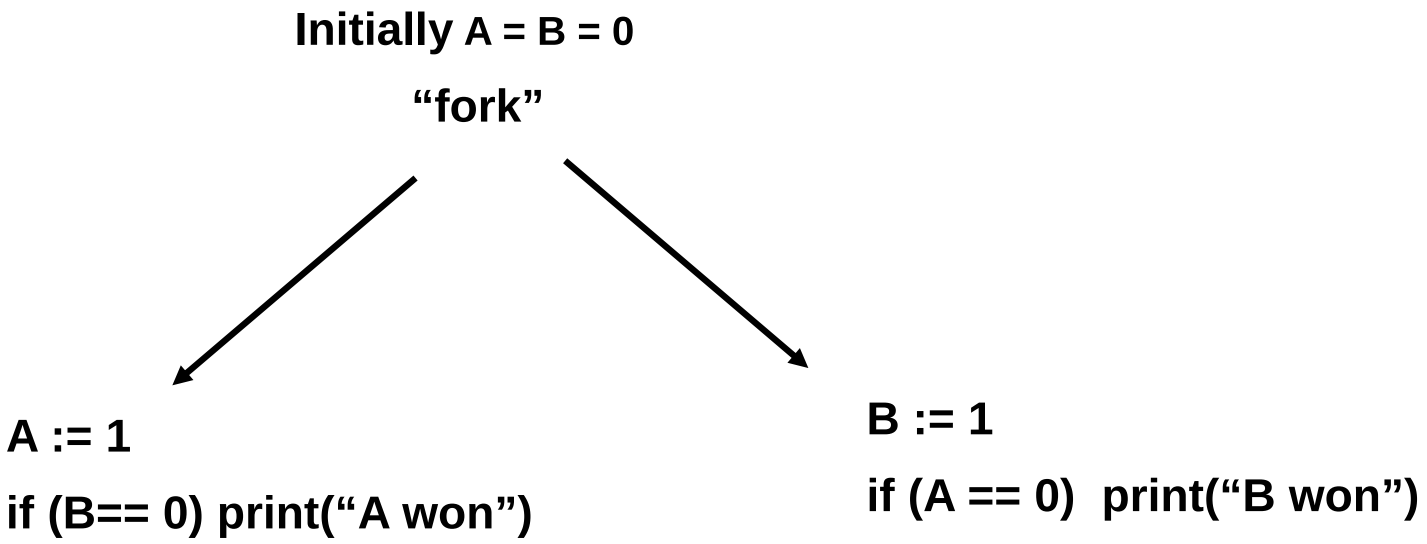
Given Sequential Consistecy: What value will be printed?

☐ 0

☐ 1

☐ Undefined (either 0 or 1)

Dekker’s Algorithm (mutual exclusion)



Initially A = B = 0

“fork”

```
A := 1
if (B== 0) print(“A won”)
```

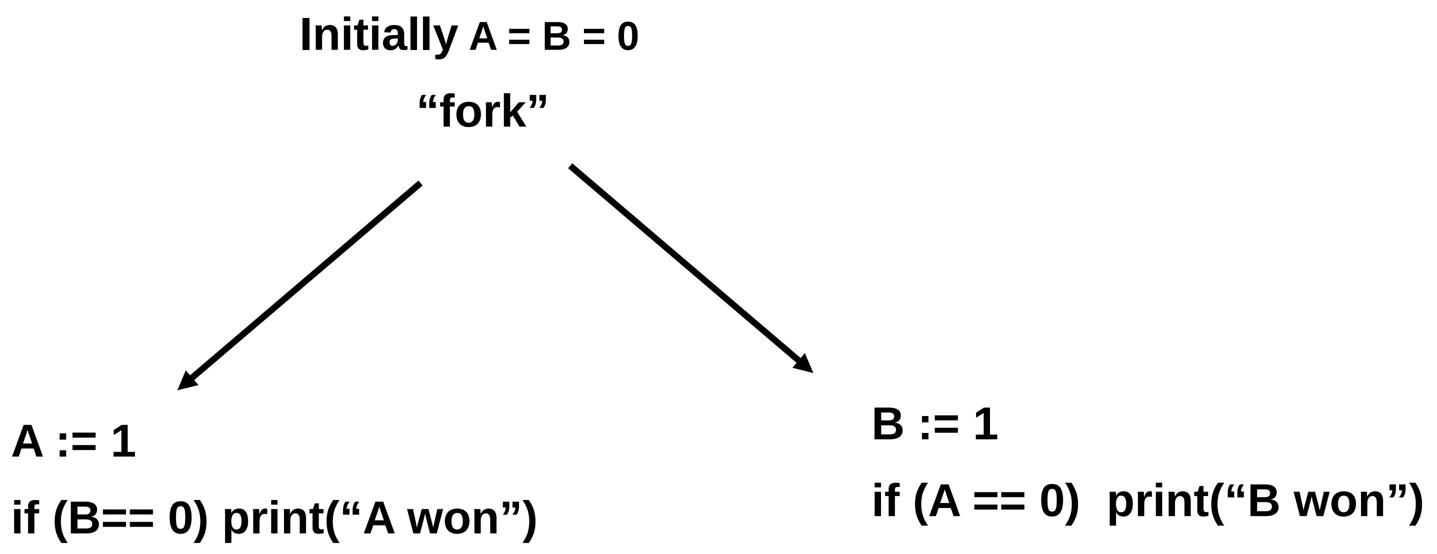
```
B := 1
if (A == 0)  print(“B won”)
```

Given Sequential Consistecy:
Is it possible that both threads ”win”?

- ☐ Yes
- ☐ No
- ☐ Undefined

UPPSALA UNIVERSITET

Proving Dekker's Algorithm under SC



Initially A = B = 0

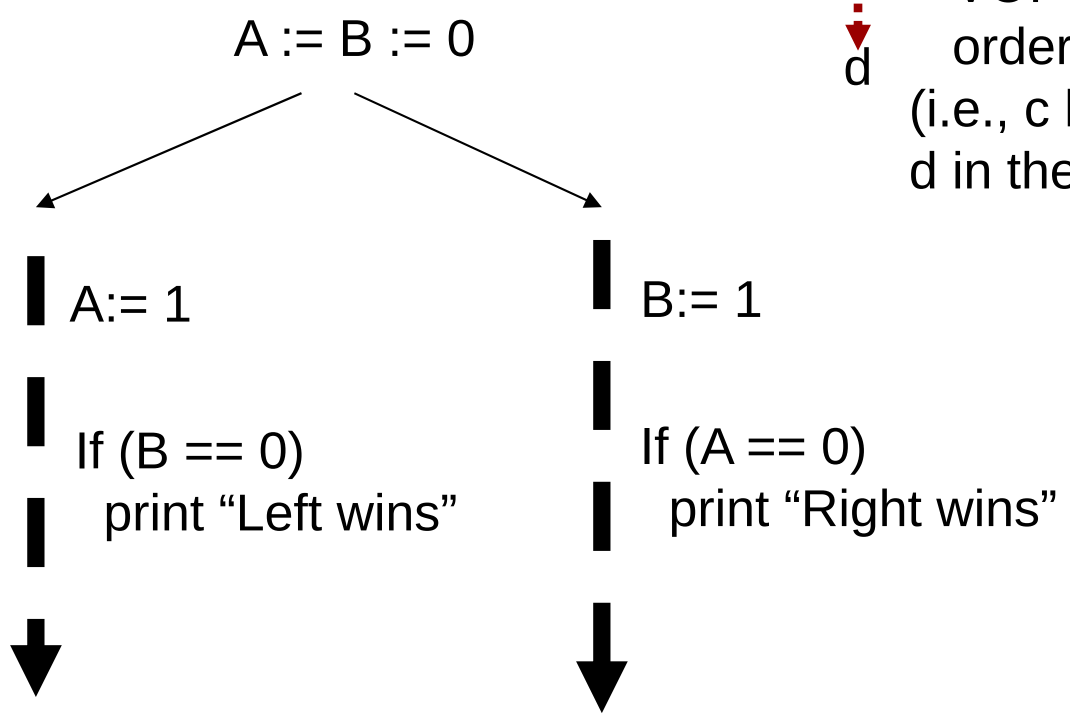
"fork"

```
A := 1
if (B== 0) print("A won")
```

```
B := 1
if (A == 0)  print("B won")
```

AVDARK 2013

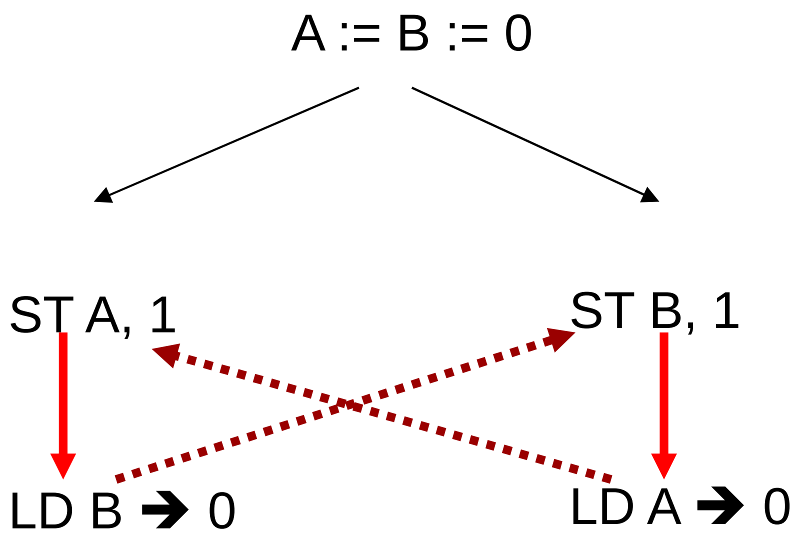
Can the case "both win" happen under SC?



Acess graph

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

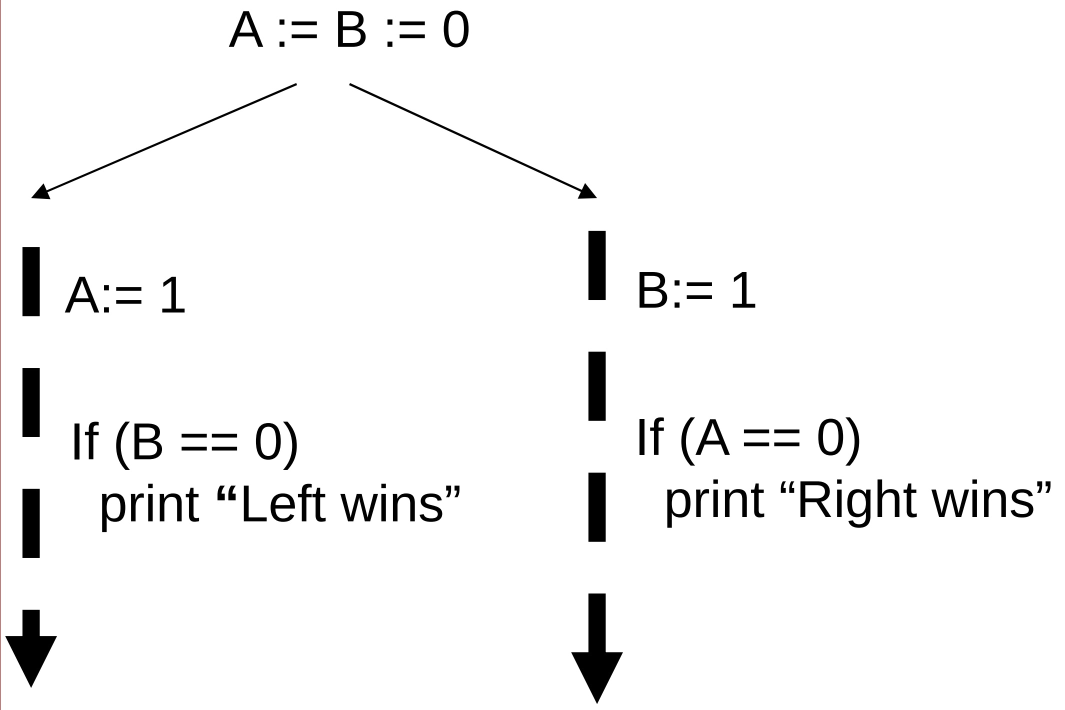
= PO: Program order: a < b (the order specified by the program)

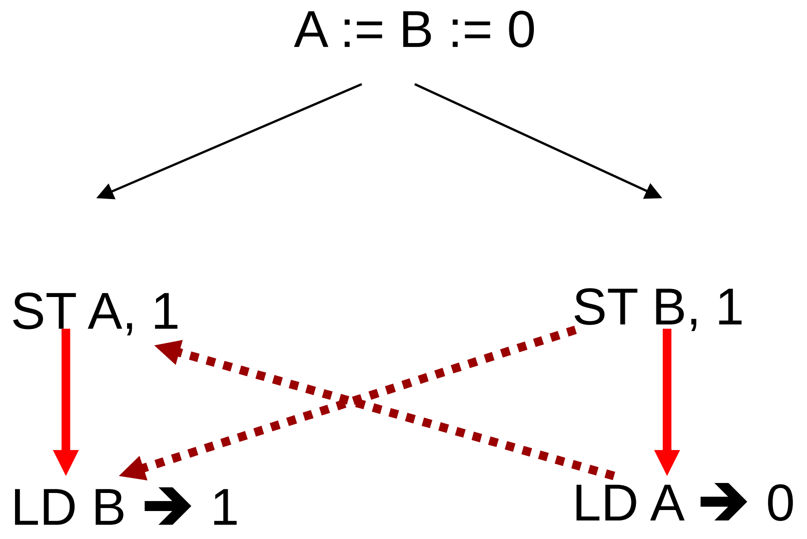


Cyclic access graph ➔ Not SC (there is no global order)

One thread wins under SC

Only Right wins ➔ SC is OK





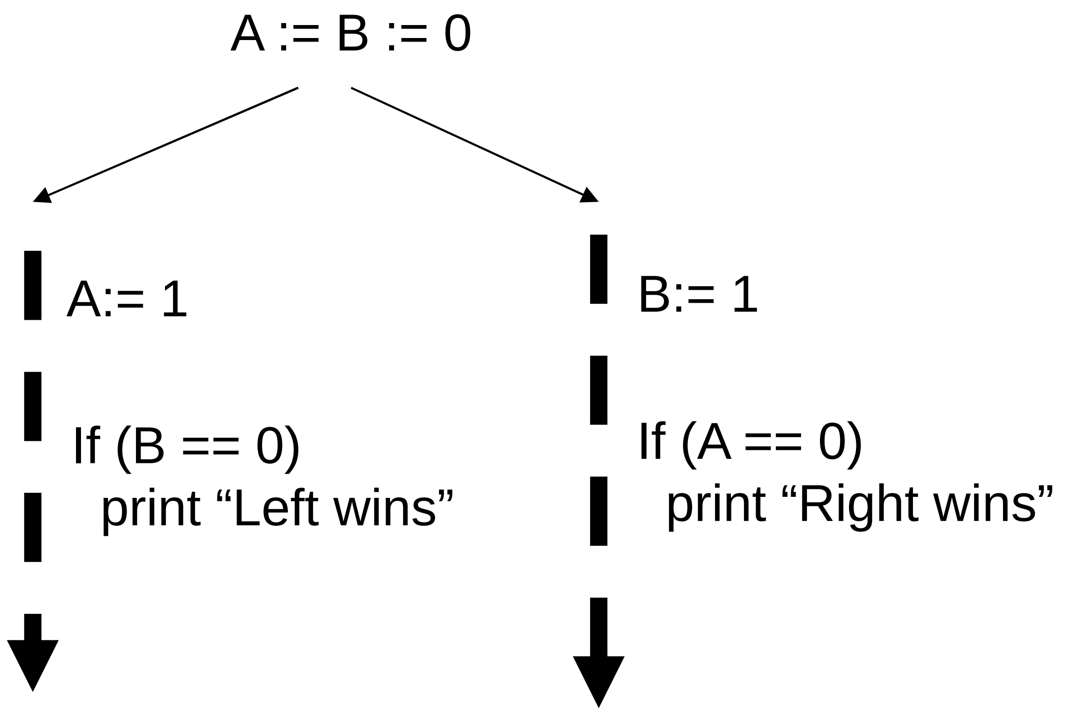
Not cyclic graph ➔ SC

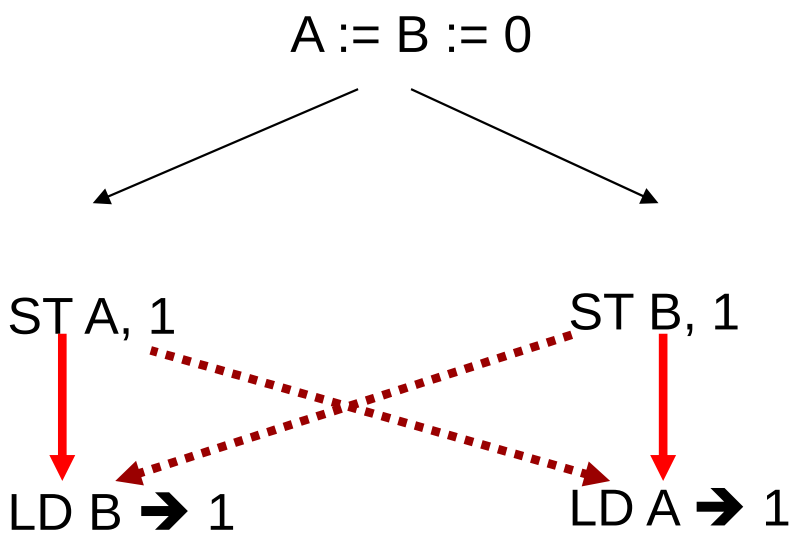
One global order:
STB < LDA < STA <LDB

UPPSALA UNIVERSITET

No thread wins under SC

No thread wins ➔ SC is OK





Not cyclic graph ➔ SC

Four Partial Orders, still SC

STB < LDA ; STA < LDA; STB < LDB ; STA < LDA

AVDARK 2013

UPPSALA UNIVERSITET

Summary

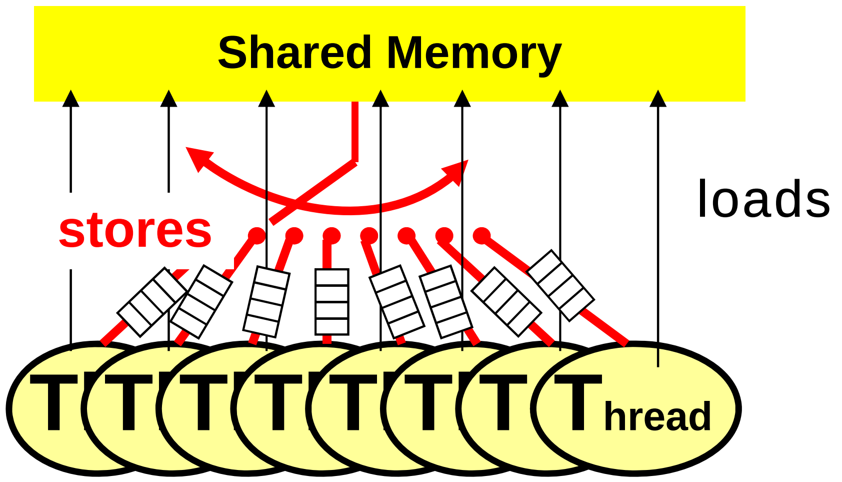
- Gives the ”illusion” of **one global order** between all memory accesses
- If two threads can observe two contratictive [partial] orders, SC is broken.
- Maintains human intuition
- … at the cost of performance (or complexity)

AVDARK 2013

Other Memory Models

Erik Hagersten
Uppsala University

“Almost intuitive memory model” Total Store Ordering [TSO] (P. Sindhu)

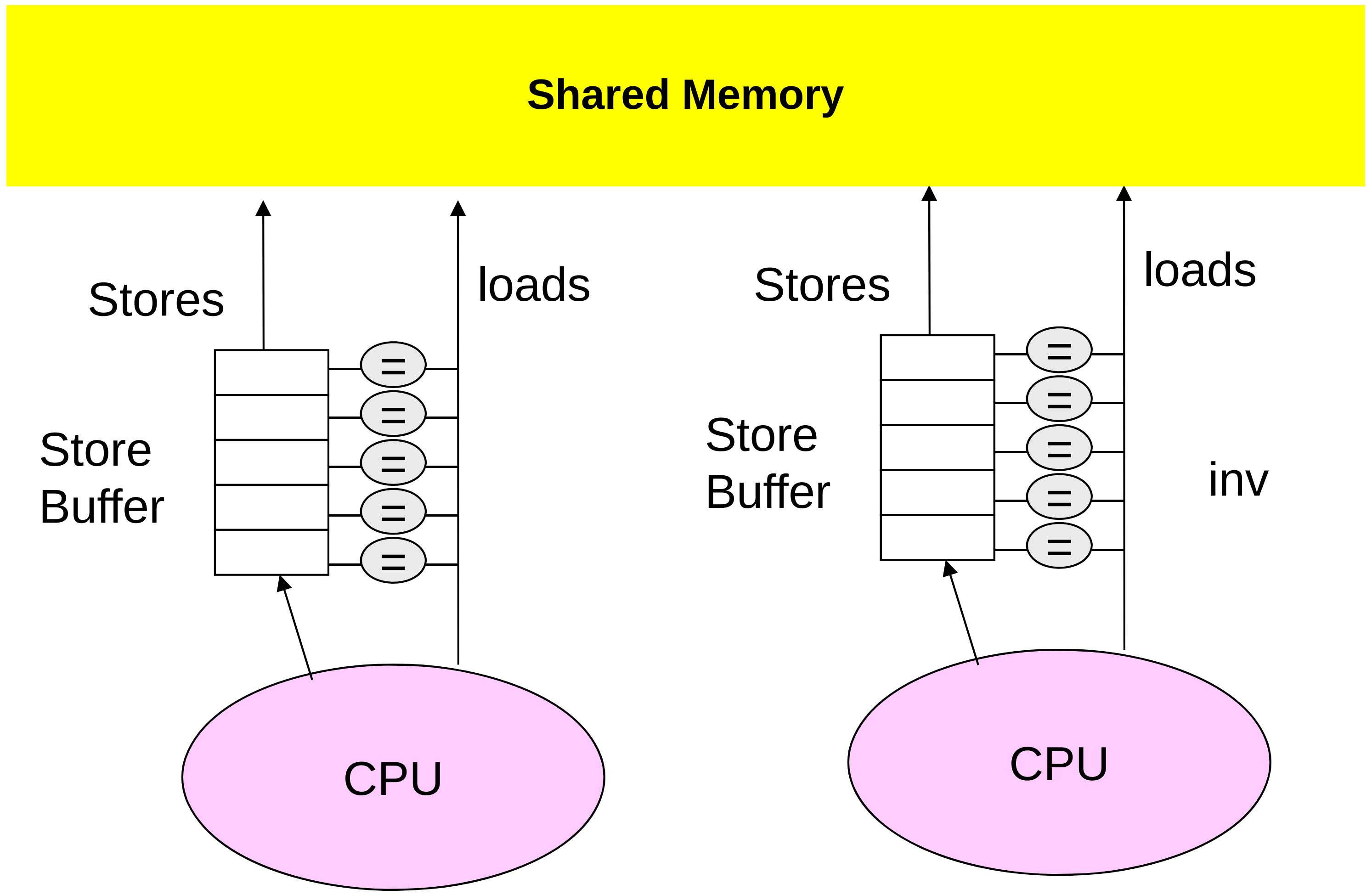


- Global *interleaving* [order] for all stores from different threads (own stores excepted)
- “Programmer’s intuition is almost maintained”
  - Flag synchronization? Yes
  - Store causality? Yes
  - Does Dekker work? No
- Unnecessarily restrictive ==> performance penalty

UPPSALA UNIVERSITET

AVDARK 2013

TSO HW Model



➔Stores are moved off the critical path

Coherence implementation can be the same as for SC

Where Memory Models Matters

- Flag synchronization

(initially flag = 0 and A = 0)

```
...                           ...
A = 1;                        while (flag != 1)  {};
flag = 1;                     X = A;
                              print(X);
```

Given Total Store Order:
What value will be printed?
- ☐ 0
- ☐ 1
- ☐ Undefined (either 0 or 1)

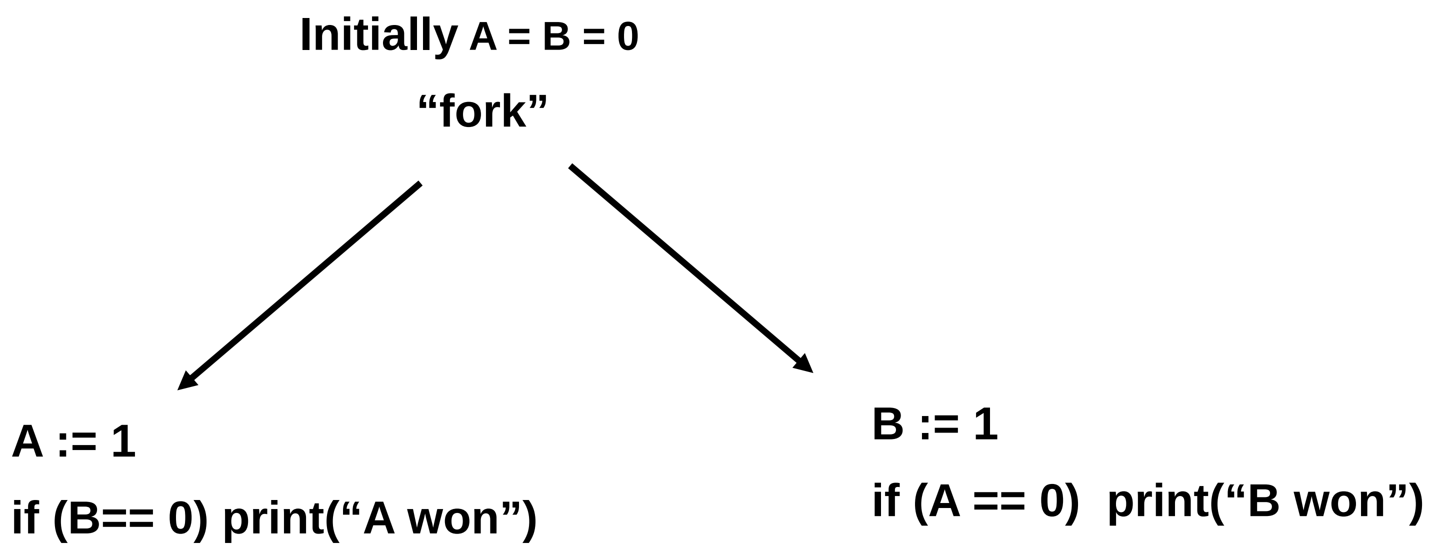
- Causality (Causal correctness)

(Initially A = 0 and B = 0)

```
...             ...                 Read A
A = 1;          ...                 ...
...             while (A==0) {};    ...
                B = 1;              ...
                                    while (B==0) {};
                                    X = A;
                                    print (X);
```

Given Total Store Order:
What value will be printed?
- ☐ 0
- ☐ 1
- ☐ Undefined (either 0 or 1)

Dekker’s Algorithm (mutual exclusion)



Initially A = B = 0

“fork”

```
A := 1
if (B== 0) print(“A won”)
```

```
B := 1
if (A == 0)  print(“B won”)
```

Does the write become globally visible before the read is performed?

Given Total Store Order:
Is it possible that both threads ”win”?

- ❑ Yes
- ❑ No
- ❑ Undefined

AVDARK 2013

Dekker’s Algorithm for TSO

Initially A = B = 0

“fork”

A := 1

Memory barrier

if (B== 0) print(“A won”)

B := 1

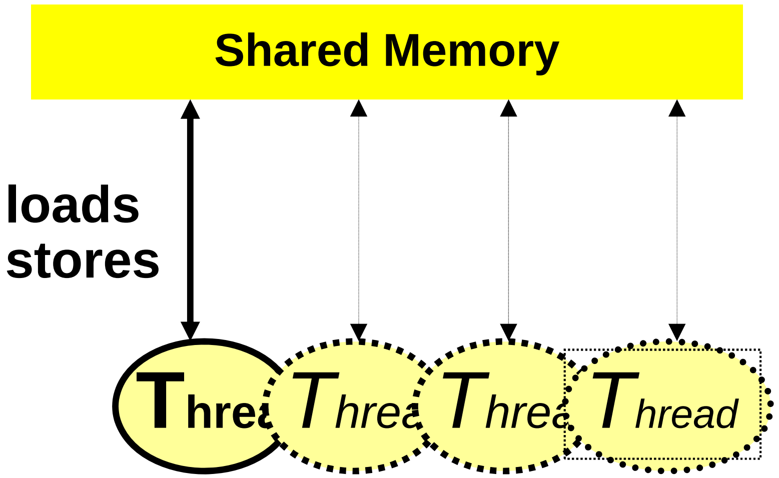
Memory barrier

if (A == 0) print(“B won”)

Memory barrier: Tells the HW to not start the LD until all previous stores have been ”globaly ordered”
➔behaves like SC
➔ Dekker’s algorithm works!

UPPSALA UNIVERSITET

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



- **Most accesses are unordered**
- **"Programmer's intuition is not maintained"**
  - Flag synchronization? No
  - Causal correctness? No
  - Dekker? No
- **Global order <u>only</u> established when the programmer explicitly inserts memory barrier instructions**

++ Better performance!!

-- Interesting bugs!!

AVDARK 2013

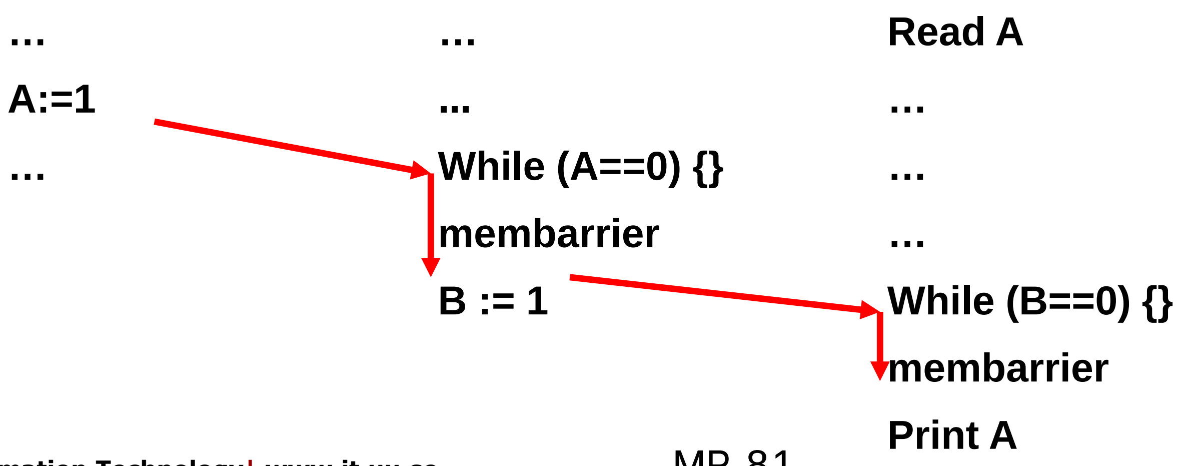
Weak/Release consistency

- 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

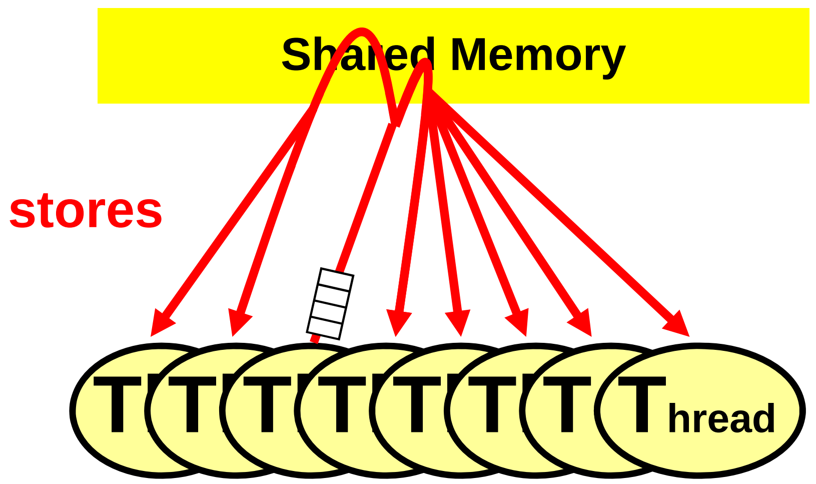
Initially A = B = 0



```
...            ...                  Read A
A:=1           ...                  ...
...            While (A==0) {}      ...
               membarrier           ...
               B := 1               While (B==0) {}
                                    membarrier
                                    Print A
```

UPPSALA UNIVERSITET

Processor Consistency [PC] (J. Goodman)



- PC: The stores from a processor appears to others in program order.
  - Flag synchronization? Yes
  - Causal correctness? Not clearly defined by Goodman. (yes, for PC "with causal correctness")
  - Dekker? No

AVDARK 2013

Learning more about memory models

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

RTFM: 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?
What does "volatile" declarations in C mean?

X86's new memory model

- Processor consistency with causual correctness for non-atomic memory ops
- TSO for atomic memory ops
- (Academia says the x86 mem model is TSO

- Link to the Intel manual (Section 8.2)

http://download.intel.com/products/processor/manual/325462.pdf

- Video presentation:

  http://www.youtube.com/watch?v=WUfvvFD5tAA&hl=sv

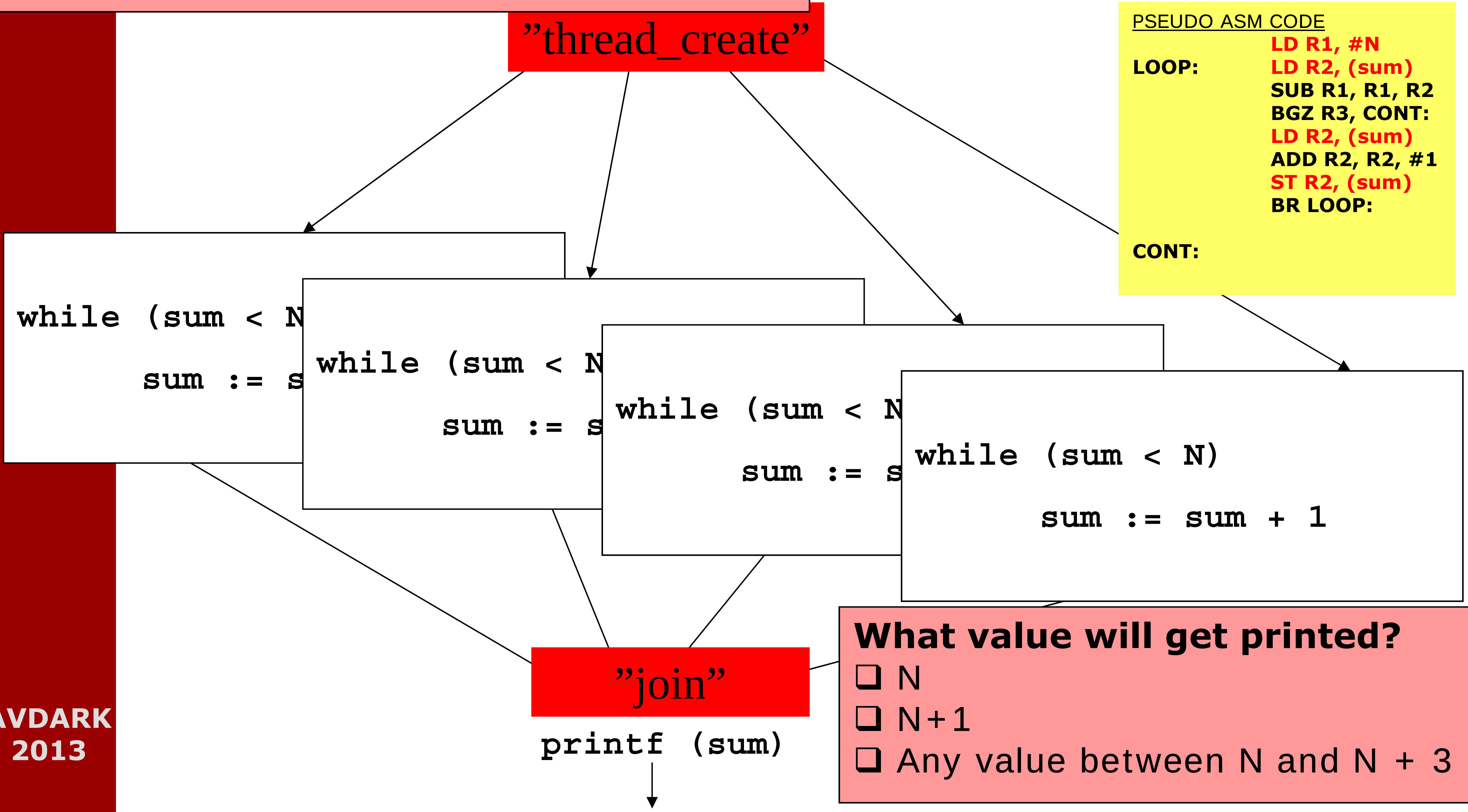
Synchronization

Erik Hagersten
Uppsala University
Sweden

How many additions will be executed?

- ❑ N
- ❑ Any value between N and N + 3
- ❑ Any value between N and N * 4

Execution on a sequentially consistent shared-memory machine:



PSEUDO ASM CODE

```
        LD R1, #N
LOOP:   LD R2, (sum)
        SUB R1, R1, R2
        BGZ R3, CONT:
        LD R2, (sum)
        ADD R2, R2, #1
        ST R2, (sum)
        BR LOOP:

CONT:
```

What value will get printed?

- ❑ N
- ❑ N+1
- ❑ Any value between N and N + 3

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

Components of a Synchronization Event

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

UPPSALA UNIVERSITET

Atomic Instruction to Acquire

Atomic example: test&set “TAS R1, (location)”

The value at Mem(location) is loaded into R1, and
the constant “1” atomically stored into Mem(location)
(Other constant could be implemented, e.g., SPARC: ”FF”)

Looks like a “store” to the coherence protocol

Implementation:

1. Get a writable exclusive copy of the cache line (state M in MOSI)
2. Make the atomic modification to that cached copy

Examples of other atomic primitives:

SWAP R1, (location): atomically swap the values of R1 with Mem(location)
CAS R1, R2, (location): (Compare&Swap) SWAP if Mem(location)=REG2

AVDARK 2013

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
- Lower overhead, but consumes processor resources
- Can cause coherence network traffic

Hybrid methods: busy-wait a while, then block

AVDARK 2013

UPPSALA UNIVERSITET

Release Algorithm

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

AVDARK 2013

UPPSALA UNIVERSITET

A Bad Example: ”POUNDING”

How is TAS treated by the coherence protocol?
- ❑ Like a CPU read operation
- ❑ Like a CPU write opreration
- ❑ By performing the ”SWAP” atomically in DRAM

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

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

If two threads are waiting for the lock
- ❑ They will both spin locally in their cache
- ❑ They will create coherence traffic by invalidating each other
- ❑ They will both block and need to be woken up by the OS

Assume: The function TAS[addr] returns the current memory value at addr and ***atomically*** *writes the busy pattern “1” to the memory*

Spinning threads produce traffic!

AVDARK 2013

UPPSALA UNIVERSITET

Optimistic Test&Set Lock ”spinlock”

```
proc lock(lock_variable) {
     while true {
       if (TAS[lock_variable] ==0)  break;      /* pound 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
}
```

If two threads are waiting for the lock

- They will mostly spin locally in their cache
- They will create coherence traffic all the time by invalidating each other
- They will both block and need to be woken up by the OS

Much less coherence traffic!!
-- still lots of traffic at lock handover!
More on this during Scalable Synchronization

AVDARK 2013

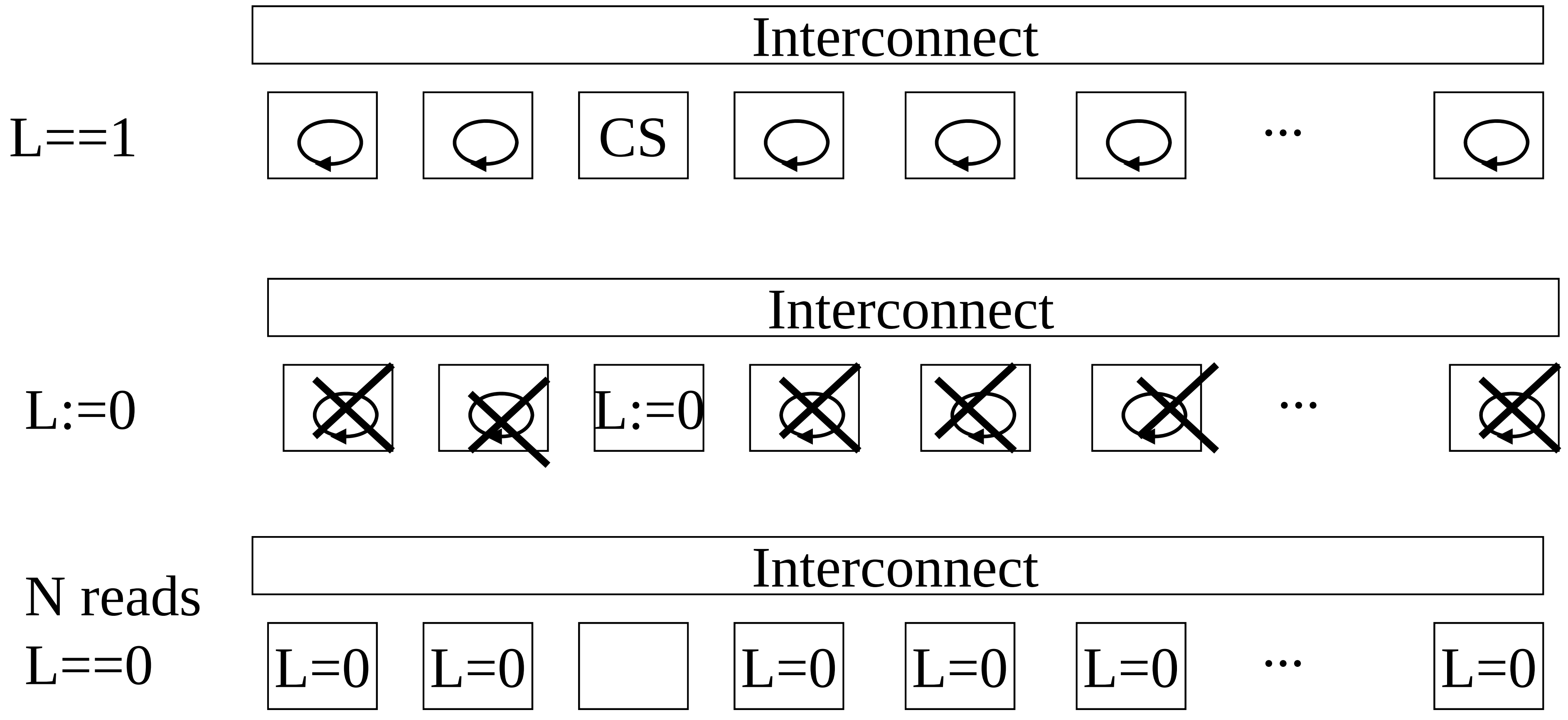
UPPSALA UNIVERSITET

Pesimistic Test&Set Lock ”spinlock”

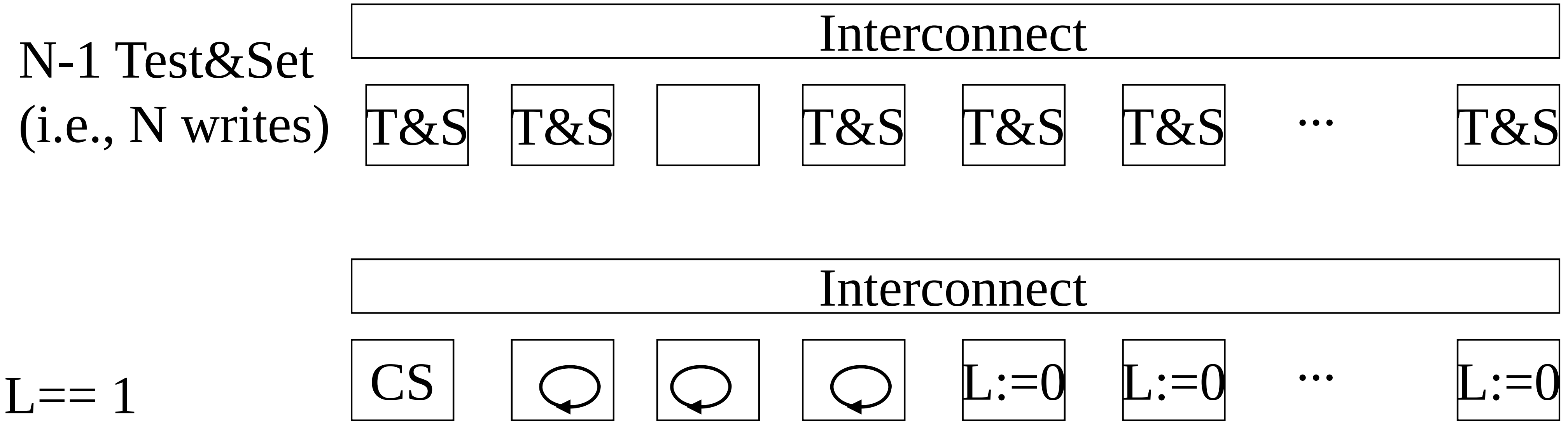
```
proc lock(lock_variable) {
    while true {
          while(lock_variable != 0) {}   /* spin locally in your cache until ”0” observed*/
           if (TAS[lock_variable] ==0)  break; /* pound on the lock once, done if TAS==0
    }
}

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

Slightly less traffic than Optimistic for contended locks
-- still lots of traffic at lock handover!
More solutions during Scalable Shared Memory

It could still get messy!



...messy (part 2)



potentially: ~N*N/2 reads :-(

Problem1: Contention on the interconnect slows down the CS execution
Problem2: The lock hand-over time is N*read_throughput
Fix1: Some back-off strategy, bad news for hand-over latency
Fix2: Queue-based locks

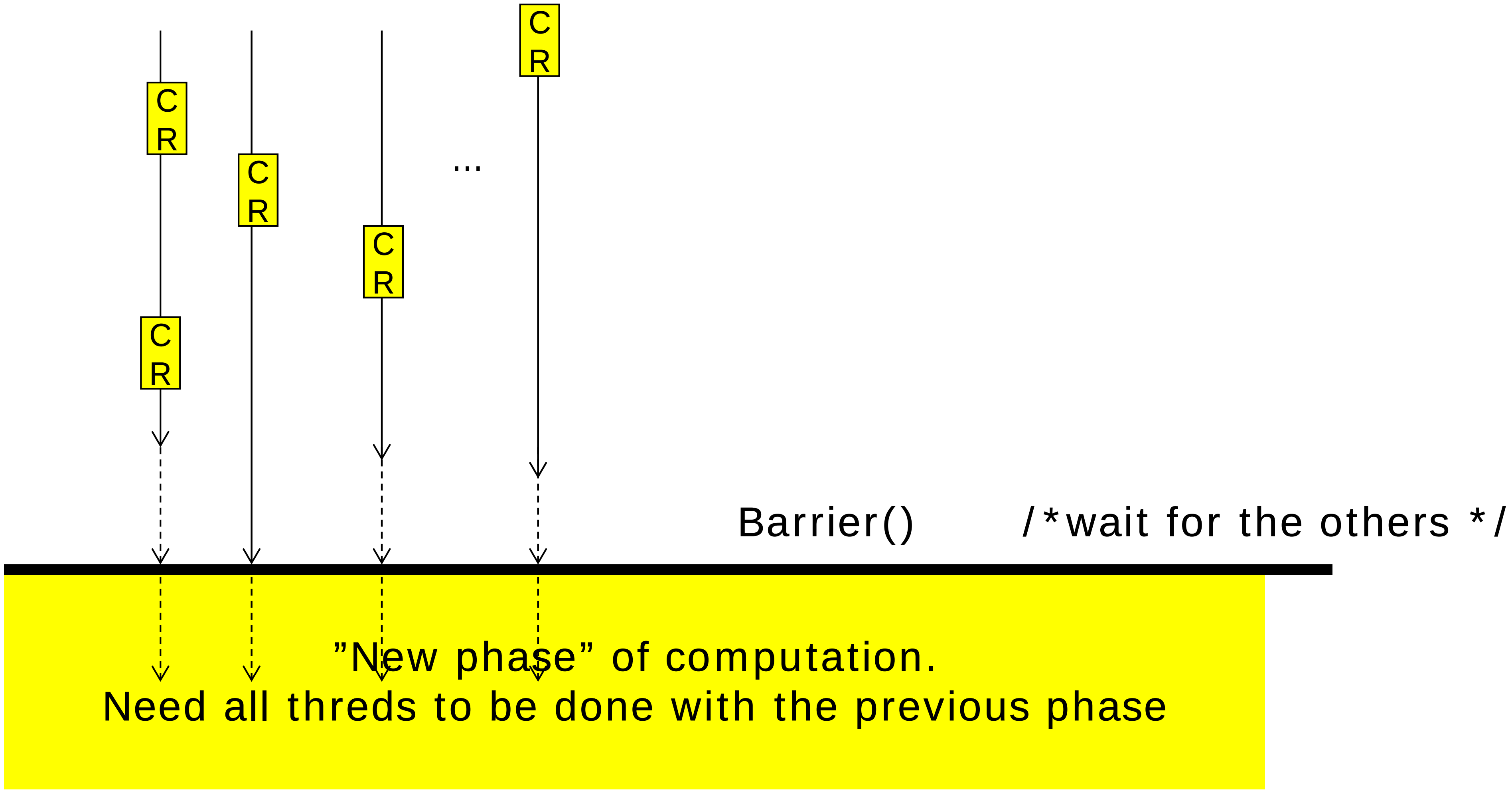
Barrier Synchronization

Erik Hagersten

Uppsala University

UPPSALA UNIVERSITET

Barrier Synchronization



C
R
C
R
C
R
C
R
C
R
...
Barrier()
/*wait for the others */
"New phase" of computation.
Need all threds to be done with the previous phase

AVDARK 2013

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

UPPSALA UNIVERSITET

A Naiive Centralized Barrier

```
BARRIER (bar_name, p) {

 LOCK(bar_name.lock) {
    if (bar_name.counter == p) bar_name.counter = 0; /* init count*/
    bar_name.counter++;                        /* globally increment the barrier count *
}
UNLOCK(bar_name.lock)

 while (bar_name.counter < p) {};      /* wait for the last thread */

}
```

AVDARK
2013

A More Complicated 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)}
      }
```