



# Shared Memory Architecture Concepts and Performance Issues

TDDD56 Lecture 3 / TDDC78 Lecture 2

Christoph Kessler

PELAB / IDA  
Linköping university  
Sweden

2014



## Outline – TDDD56

Lecture 1: Multicore Architecture Concepts

Lecture 2: Parallel programming with threads and tasks

**Lecture 3: Shared memory architecture concepts and performance issues**

- Memory hierarchy
- Consistency issues and coherence protocols
- Performance issues, e.g. false sharing
- Optimizations for data locality (briefly, more in Lecture 8)

Lecture 4: Non-blocking synchronization

Lecture 5/6: Design and analysis of parallel algorithms

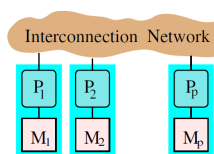
Lecture 6/7: Parallel Sorting Algorithms

...

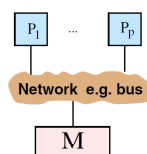
2



## Distributed Memory vs. Shared Memory



Distributed memory system



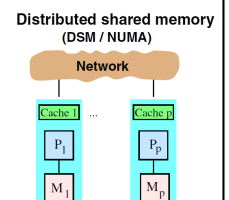
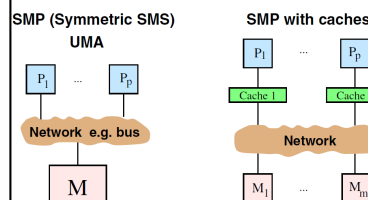
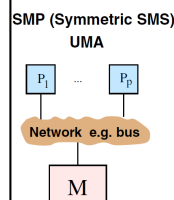
Shared memory system

4



## Shared Memory – Variants

- Single shared memory module (UMA) quickly becomes a performance bottleneck
- Often implemented with caches to leverage access locality
  - As done for single-processor systems, too
- Can even be realized on top of distributed memory system (NUMA – non-uniform memory access)



5



## Cache

**Cache** = small, fast memory (SRAM) between processor and main memory, today typically on-chip

- contains copies of main memory words
  - cache hit = accessed word already in cache, get it fast.
  - cache miss = not in cache, load from main memory (slower)
- **Cache line** holds a copy of a block of adjacent memory words
  - size: from 16 bytes upwards, can differ for cache levels
- Cache-based systems profit from
  - spatial access locality
    - ▶ access also other data in same cache line
  - temporal access locality
    - ▶ access same location multiple times
- HW-controlled cache line replacement
  - dynamic adaptivity of cache contents
- suitable for applications with high (also dynamic) data locality

6



## Cache (cont.)

**Mapping** memory blocks → cache lines / page frames:

direct mapped:  $\forall j \exists! i : B_j \mapsto C_i$ , namely where  $i \equiv j \bmod m$ .

fully-associative: any memory block may be placed in any cache line  
set-associative

**Replacement strategies** (for fully- and set-associative caches)

LRU least-recently used  
LFU least-frequently used

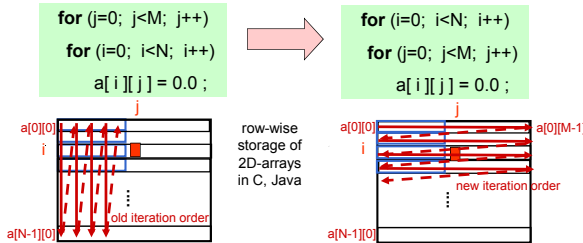
...

7

## Optimizing Programs for Improved Access Locality

### Example:

Loop Interchange



- Can improve spatial locality of memory accesses (fewer cache misses / page faults)

## Caches: Memory Update Strategies

### Write-through

- + consistency
- slow, write stall (→ write buffer)

### Write-back

- + update only cache entry
- + write back to memory only when replacing cache line
- + write only if modified, marked by “dirty” bit for each  $C_i$
- not consistent,
  - DMA access (I/O, other procs) may access stale values
  - must be protected by OS, write back on request

## Memory Hierarchy Example

Memory hierarchy of the Itanium2 processors in SGI Altix 3700 Bx2 (Mozart)

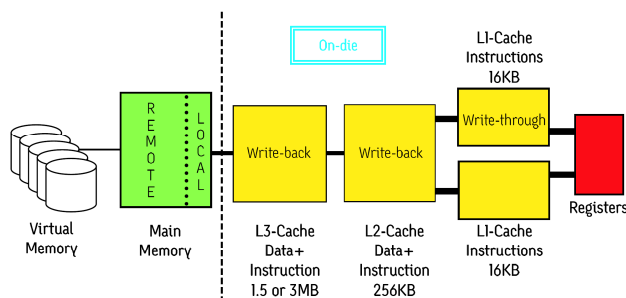


Fig. 2. Itanium 2 memory and cache hierarchy diagram

Source: Michael Woodacre et al.: The SGI Altix™ 3000 Global Shared-Memory Architecture. White Paper, SGI, 2003.

## Cache Coherence and Memory Consistency

Caching of (shared) variables leads to **consistency problems**.

A cache management system is called **coherent**

if a read access to a (shared) memory location  $x$  reproduces always the value corresponding to the most recent write access to  $x$ .

→ no access to **stale** values

A memory system is **consistent** (at a certain time)

if all copies of shared variables in the main memory and in the caches are identical.

Permanent cache-consistency implies cache-coherence.

## Cache Coherence – Formal Definition

What does “most recent write access” to  $x$  mean?

Formally, 3 conditions must be fulfilled for **coherence**:

- Each processor sees its *own* writes and reads in program order  
 $P_1$  writes  $v$  to  $x$  at time  $t_1$ , reads from  $x$  at  $t_2 > t_1$ ,  
 no other processor writes to  $x$  between  $t_1$  and  $t_2$   
 → read yields  $v$
- The written value is eventually visible to *all* processors.  
 $P_1$  writes to  $l$  at  $t_1$ ,  $P_2$  reads from  $l$  at  $t_2 > t_1$ ,  
 no other processor writes to  $l$  between  $t_1$  and  $t_2$ ,  
 and  $t_2 > t_1$  sufficiently large, then  $P_2$  reads  $x$ .
- All processors see one total order of all write accesses.  
 (total store ordering)

## Cache Coherence Protocols

Inconsistencies occur when modifying only the copy of a shared variable in a cache, not in the main memory and all other caches where it is held.

### Write-update protocol

At a write access, all other copies in the system must be updated as well.  
 Updating must be finished before the next access.

### Write-invalidate protocol

Before modifying a copy in a cache,  
 all other copies in the system must be declared as “invalid”.

Most cache-based SMPs use a write-invalidate protocol.

Updating / invalidating straightforward in bus-based systems (**bus-snooping**)  
 otherwise, a **directory mechanism** is necessary

## Details: Write-Invalidate Protocol



Implementation: multiple-reader-single-writer sharing

At any time, a data item (usually, entire cache line blocks) may either be:  
 accessed in **read-only mode** by one or more processors  
 read and written (**exclusive mode**) by a single processor

Items in read-only mode can be copied indefinitely to other processors.

Write attempt to read-only-mode data  $x$ :

1. broadcast invalidation message to all other copies of  $x$
2. await acknowledgements before the write can take place
3. Any processor attempting to access  $x$  is blocked if a writer exists.
4. Eventually, control is transferred from the writer and other accesses may take place once the update has been sent

→ all accesses to  $x$  processed on first-come-first-served basis.

Achieves sequential consistency.

## Write-Invalidate Protocol (cont.)



- + parallelism (multiple readers)
- + updates propagated only when data are read
- + several updates can take place before communication is necessary
- Cost of invalidating read-only copies before a write can occur
  - + ok if read/write ratio is sufficiently high
  - + for small read/write ratio: single-reader-single-writer scheme (at most one process gets read-only access at a time)

15

## Write-Update Protocol



Write  $x$ :

done locally + broadcast new value to all who have a copy of  $x$   
 these update their copies immediately.

Read  $x$ :

read local copy of  $x$ , no need for communication.  
 → multiple readers  
 → several processors may write the same data item at the same time (multiple-reader-multiple-writer sharing)

Sequential consistency if broadcasts are totally ordered and blocking

- all processors agree on the order of updates.
- the reads between writes are well defined

+ Reads are cheap

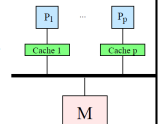
– totally ordered broadcast protocols quite expensive to implement

## Bus-Snooping



For bus-based SMP with caches and write-through strategy.

All relevant memory accesses go via the central bus.



Cache-controller of each processor listens to addresses on the bus:

write access to main memory is recognized  
 and committed to the own cache.

– bus is performance bottleneck → poor scalability

17

## Write-back invalidation protocol (MSI-protocol)



A block held in cache has one of 3 states:

**M (modified)**

only this cache entry is valid, all other copies + MM location are not.

**S (shared)**

cached on one or more processors, all copies are valid.

**I (invalid)**

this cache entry contains invalid values.

18

## MSI-Protocol: State Transitions



State transitions:

triggered by bus operations and local processor reads/writes

**Bus read (BusRd)**

read access caused a cache miss

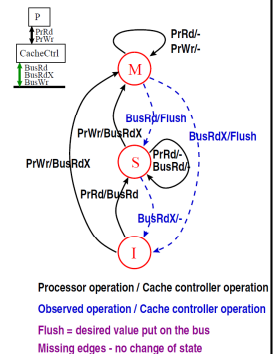
**Bus read exclusive (BusRdX)**

write attempt to non-modifiable copy  
 → must invalidate other copies

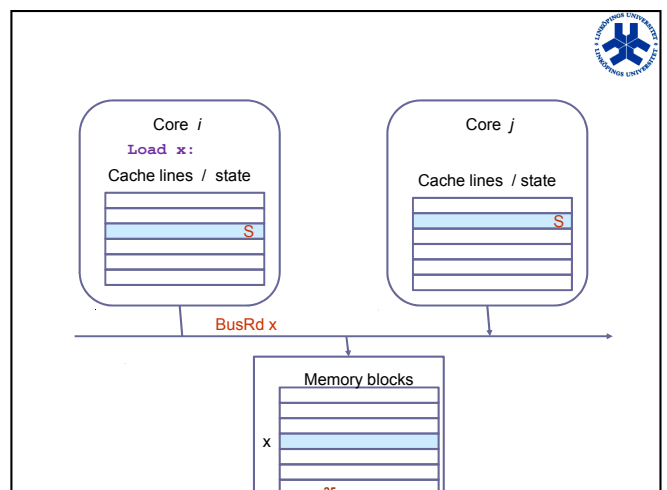
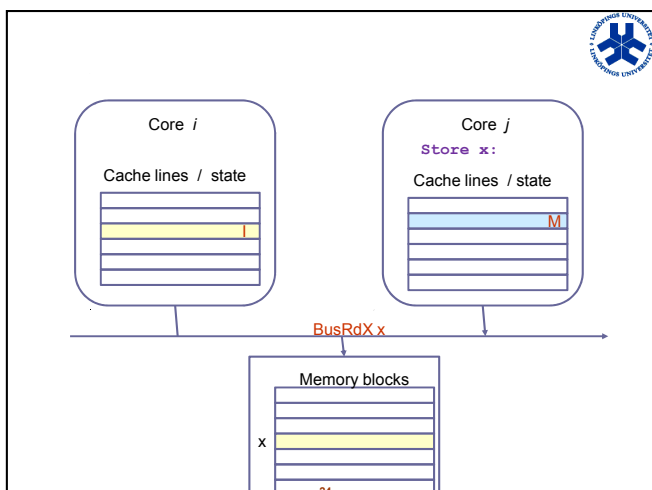
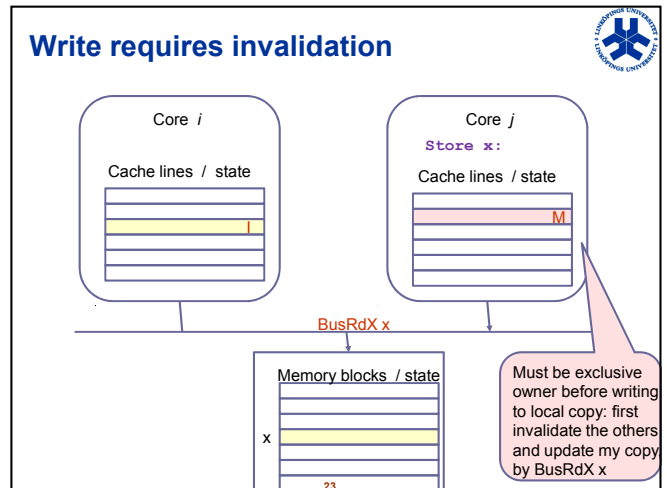
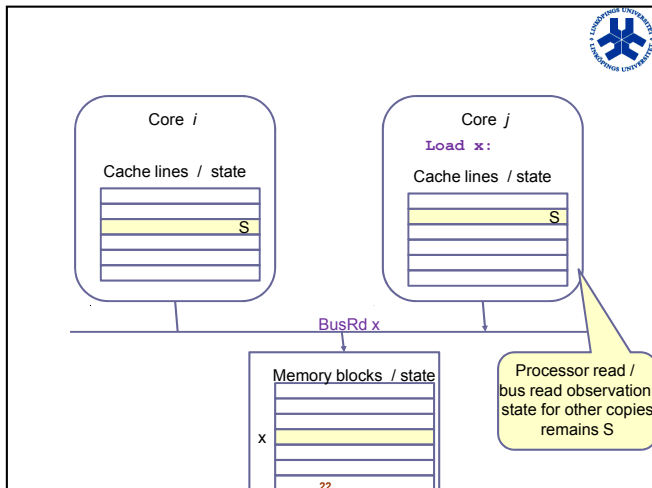
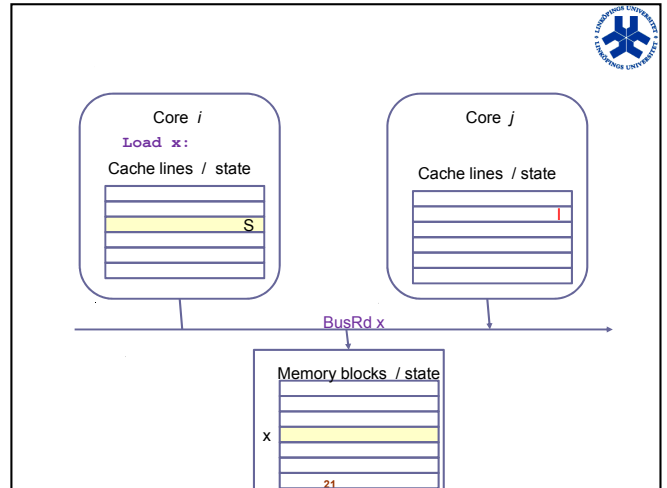
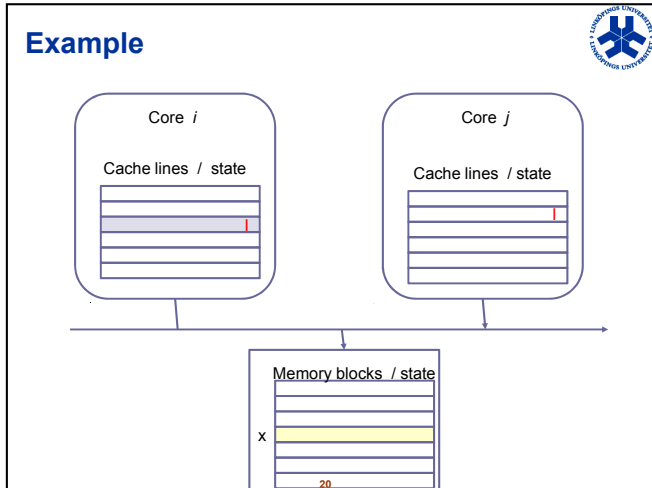
**Write back (BusWr)**, due to replacement

Processor reads (PrRd)

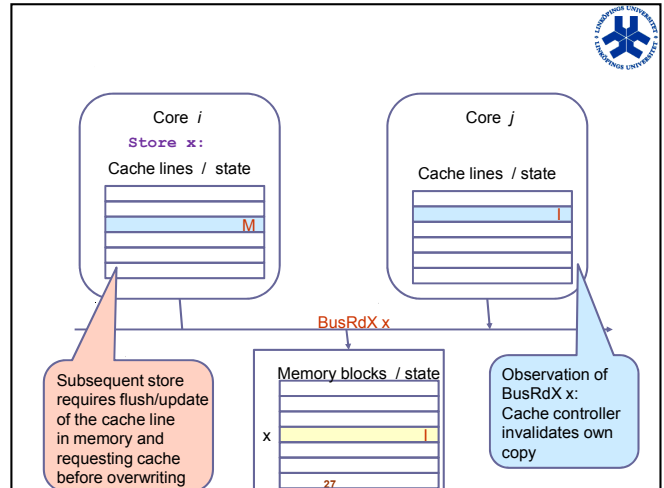
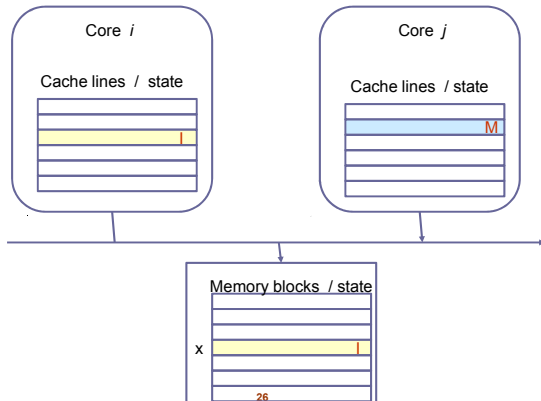
Processor writes (PrWr)



## Example



## Another scenario:



## MSI-Protocol: State Transitions

State transitions:

triggered by bus operations and local processor reads/writes

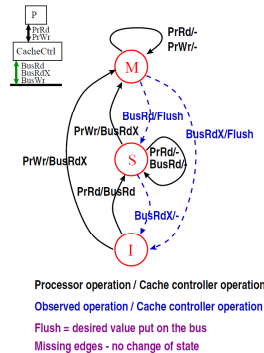
**Bus read (BusRd)**  
read access caused a cache miss

**Bus read exclusive (BusRdX)**  
write attempt to non-modifiable copy  
→ must invalidate other copies

**Write back (BusWr)**, due to replacement

Processor reads (PrRd)

Processor writes (PrWr)



## MESI-Protocol

MSI protocol:

2 bus operations (BusRd, BusRdX) required  
if a processor first reads (→ S), then writes (→ M) a memory location,  
even if no other processor works on this program.

→ generalization to MESI-protocol with new state

**E (exclusive)**

no other cache has a copy of this block,  
and this copy is not modified.

Modifications in MSI-protocol:

- + PrRd to a non-cached address (BusRd): → E (not S)
- + PrWr to E-address: local change to M, write (no bus operation)
- + read access from another processor to E-address (BusRd/Flush): → S

MESI supported by Intel Pentium, MIPS R4400, IBM PowerPC, ...

## CC-NUMA: Directory based Protocol for non-bus-based Architectures

No central medium:

- (a) → no cache coherence (e.g. Cray T3E)
- (b) → directory lookup

Directory keeps the copy set for each memory block  
e.g. stored as bitvectors

1 presence bit per processor

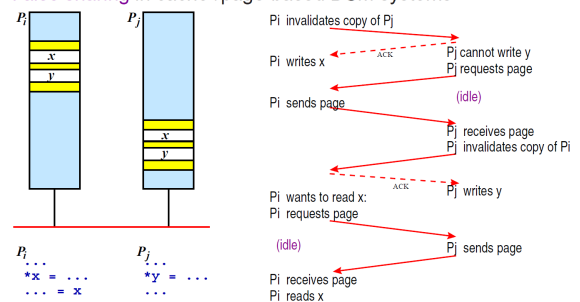
status bits

e.g. dirty-bit for the status of the main memory copy

See [Culler'98, Ch. 8]

## Performance Issue: False Sharing

False sharing in cache-/page-based DSM systems



Cache lines / pages treated as units → sequentialization, thrashing

## How to Avoid False Sharing?

- Smaller cache lines / pages  
→ false sharing less probable, but  
→ more administrative effort
- Programmer or compiler gives hints for data placement  
→ more complicated
- Time slices for exclusive use:  
each page stays for  $\geq d$  time units at one processor

### How to reduce performance penalty of false sharing?

- Use weaker consistency models  
→ programming more complicated/error-prone

32

## Shared Memory Consistency Models

Strict consistency  
Sequential consistency  
Causal consistency  
Superstep consistency  
“PRAM” consistency  
Weak consistency  
Release consistency / Barrier consistency  
Lazy Release consistency  
Entry consistency  
Others (processor consistency, total/partial store ordering etc.)

[Culler et al.'98, Ch. 9.1], [Gharachorloo/Adve'96]

33

## Consistency Models: Strict Consistency

### Strict consistency:

Read( $x$ ) returns the value that was most recently ( $\rightarrow$  global time) written to  $x$ .

realized in classical uniprocessors and SB-PRAM

in DSM physically impossible without additional synchronization

$P_1$   $\leftarrow 3m \text{ distance} \rightarrow$   $P_2$   
 $t_1: *x = \dots$   
 $\dots$   $t_1 + 1ns: \dots = x$

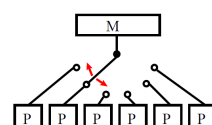
Transport of  $x$  from  $P_1$  to  $P_2$  with speed  $10c$  ???

34

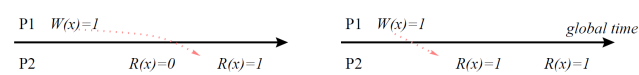
## Consistency Models: Sequential Consistency

### Sequential consistency [Lamport'79]

- + all memory accesses are ordered in *some* sequential order
- + all read and write accesses of a processor appear in program order
- + otherwise, arbitrary delays possible



Not deterministic:



## Consistency Models: Weak Consistency

- + Classification of shared variables (and their accesses):  
**synchronization variables** (locks, semaphores)  
→ always consistent, atomic access  
**other shared variables**  
→ kept consistent by the user, using synchronizations
- + Accesses to synchronization variables are sequentially consistent
- + All pending writes committed before accessing a synchr. variable
- + Synchronization before a read access to obtain most recent value

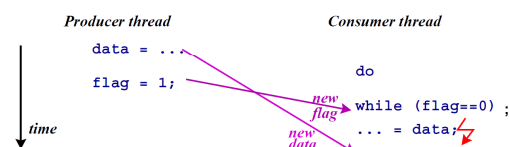


36

## Consistency Models: Weak Consistency in OpenMP

OpenMP implements weak consistency. Inconsistencies may occur due to

- + register allocation
- + compiler optimizations
- + caches with write buffers



Need explicit “memory fence” to control consistency: **flush** directive

- write back register contents to memory
- forbid code moving compiler optimizations
- flush cache write buffers to memory
- re-read flushed values from memory

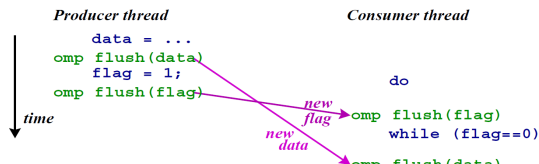
## Consistency Models: Weak Consistency in OpenMP (cont.)



```
!$omp flush ( shvarlist )
```

creates for the executing processor a consistent memory view  
for the shared variables in *shvarlist*.

If no parameter: create consistency of all accessible shared variables



A flush is implicitly done  
at barrier, critical, end critical, end parallel,  
and at end do, end section, end single  
if no `nowait` parameter is given

## Questions?



## Further Reading



- D. Culler et al. *Parallel Computer Architecture, a Hardware/Software Approach*. Morgan Kaufmann, 1998.
- J. Hennessy, D. Patterson: *Computer Architecture, a Quantitative Approach*, Second edition (1996) or later. Morgan Kaufmann.
- S. Adve, K. Gharachorloo: Shared memory consistency models: a tutorial. IEEE Computer, 1996.