Multicores

Computer Systems

Johann Blieberger

Physical Cache Organization

United Cache Organization

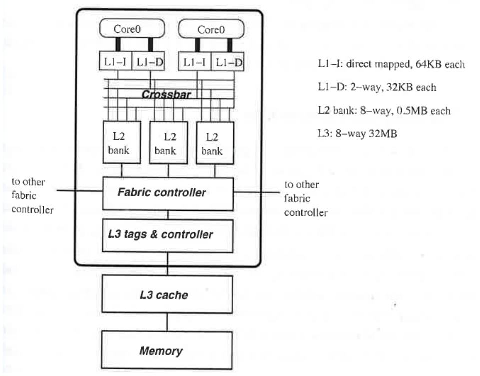

Abbildung: Dual-core chip architecture similar to IBM Power4 (taken from Solohin, Fundamentals of Parallel Multicore Architecture)

Details on how a crossbar is implemented follow later on.

Tiled Cache Organization

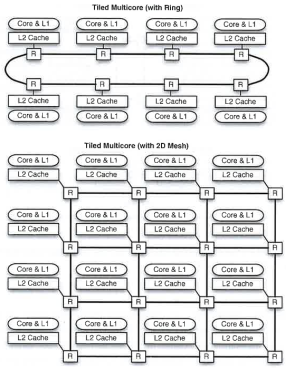

Abbildung: Tiled multicore with a ring (top) and a 2D mesh (bottom) interconnection (taken from Solohin, Fundamentals of Parallel Multicore Architecture)

R . . . Router, L2 cache tiles provide view of one single L2 cache

Hybrid Cache Organization

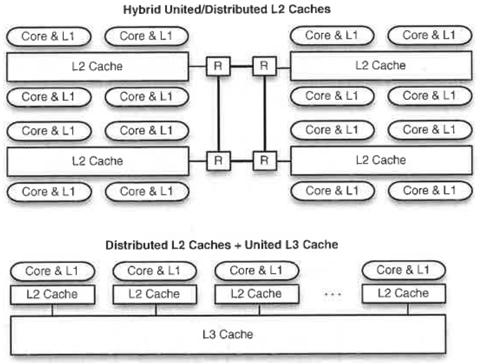
Hybrid United/Distributed L2 Caches
Core & L1
L2 Cache
R
Distributed L2 Caches + United L3 Cache
Core & L1
L2 Cache
. . .
L3 Cache

Abbildung: Hybrid physical configuration of L2 caches (top) and distributed L2 caches backed up by a united L3 cache (bottom) (taken from Solohin, Fundamentals of Parallel Multicore Architecture)

Logical Cache Organization

Logical Cache Organization

- in traditional multiprocessor systems not feasible to implement a shared cache organization since caches are located on different chips

Logical Cache Organization

- in traditional multiprocessor systems not feasible to implement a shared cache organization since caches are located on different chips
- in multicore CPUs cache tiles are located on a single die; hence accesses to remote caches perform quickly

Shared Memory Multiprocessors

Why shared memory multiprocessors?

Why shared memory multiprocessors?

- multi-threaded programs written for a single processor system, will work automatically on a shared memory multiprocessor

Problems

- cache coherence problem

Problems

- cache coherence problem
- memory consistency problem

Problems

- cache coherence problem
- memory consistency problem
- synchronization problem

Cache Coherence Problem

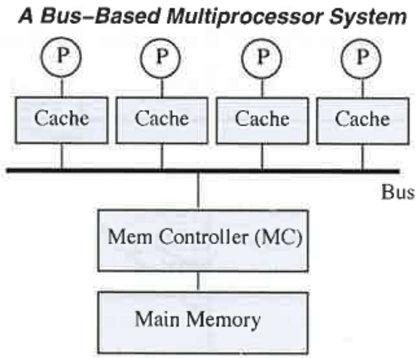

Abbildung: A simple bus-based multiprocessor system with four cores (taken from Solohin, Fundamentals of Parallel Multicore Architecture)

Can the abstraction of a single shared memory be automagically achieved?

Cache Coherence Problem

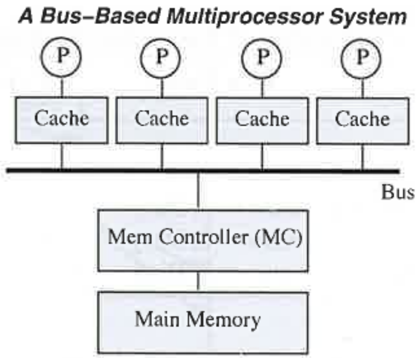

Abbildung: A simple bus-based multiprocessor system with four cores (taken from Solohin, Fundamentals of Parallel Multicore Architecture)

Can the abstraction of a single shared memory be automagically achieved? **No**

Example – Accumulate two values to a sum

$\texttt{sum} = 0$; $\texttt{a[0]} = 3$; $\texttt{a[1]} = 7$

Thread 0	**Thread 1**
`sum := sum + a[0];`	`sum := sum + a[1];`
`...`	
`/* after Thread 1 has finished */`	
`... := sum;`	

Assumption: access to `sum` occurs one at a time.

Example – Accumulate two values to a sum

`sum = 0; a[0] = 3; a[1] = 7`

Thread 0	Thread 1
`sum := sum + a[0];`	`sum := sum + a[1];`
`...`	
`/* after Thread 1 has finished */`	
`... := sum;`	

Assumption: access to `sum` occurs one at a time.

System without caches:

- Thread 0 reads `sum` from memory, adds 3, stores it back to memory.

Example – Accumulate two values to a sum

sum = 0; a[0] = 3; a[1] = 7

Thread 0	Thread 1
`sum := sum + a[0];`	`sum := sum + a[1];`
`...`	
`/* after Thread 1 has finished */`	
`... := sum;`	

Assumption: access to `sum` occurs one at a time.

System without caches:

- Thread 0 reads `sum` from memory, adds 3, stores it back to memory.
- Thread 1 reads `sum` from memory (=3), adds 7, stores 10 back to memory.

Example – Accumulate two values to a sum

sum = 0; a[0] = 3; a[1] = 7

Thread 0	Thread 1
`sum := sum + a[0];`	`sum := sum + a[1];`
`...`	
`/* after Thread 1 has finished */`	
`... := sum;`	

System with write back caches:

Action	Thread 0's Cache	Thread 1's Cache	Memory
Initially	—	—	sum = 0
Thread 0 reads sum	sum = 0	—	sum = 0
Thread 0 adds 3 to sum	sum = 3, Dirty	—	sum = 0
Thread 1 reads sum	sum = 3, Dirty	sum = 0	sum = 0
Thread 1 adds 7 to sum	sum = 3, Dirty	sum = 7, Dirty	sum = 0
Thread 0 reads sum	sum = 3, Dirty	sum = 7, Dirty	sum = 0

Example – Accumulate two values to a sum

sum = 0; a[0] = 3; a[1] = 7

Thread 0	Thread 1
`sum := sum + a[0];`	`sum := sum + a[1];`
`...`	
`/* after Thread 1 has finished */`	
`... := sum;`	

System with write through caches:

Does it resolve the problem?

Example – Accumulate two values to a sum

sum = 0; a[0] = 3; a[1] = 7

Thread 0	Thread 1
`sum := sum + a[0];`	`sum := sum + a[1];`
`...`	
`/* after Thread 1 has finished */`	
`... := sum;`	

System with write through caches:

Does it resolve the problem?
No.

Give it a try!

Conclusion

- write policy of caches dictates how a change of a value in a cached copy should be *propagated to the outer level* (e.g. main memory),

Conclusion

- write policy of caches dictates how a change of a value in a cached copy should be *propagated to the outer level* (e.g. main memory),
- but does not dictate how a change in a cached copy should be *propagated to other copies in peer caches*.

Conclusion

- write policy of caches dictates how a change of a value in a cached copy should be *propagated to the outer level* (e.g. main memory),
- but does not dictate how a change in a cached copy should be *propagated to other copies in peer caches*.

$\Longrightarrow$ **Cache Coherence Problem**

Cache Coherence Problem

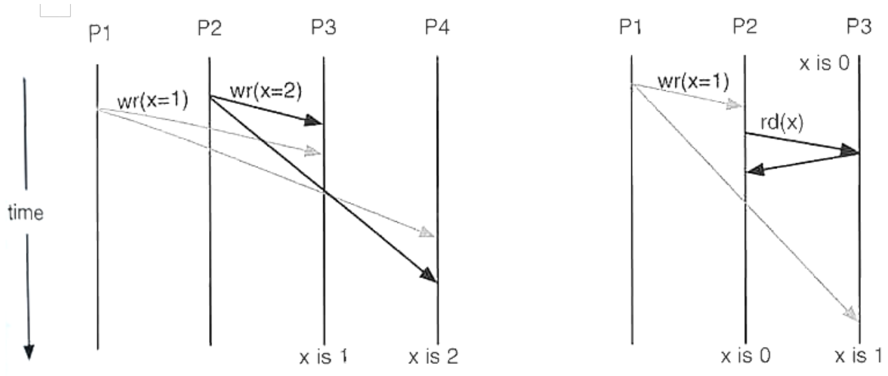

Abbildung: Illustrating the need for transaction serialization between writes (a) and between a write and a read (b)

Cache Coherence Protocol

- **Cache Coherence Protocol** has to solve Cache Coherence Problem
- **Cache Coherence Protocol** must ensure write propagation and transaction serialization

Cache Coherence Protocol

- **write update:** directly updating all cached values upon a write by a processor

Cache Coherence Protocol

- **write update:** directly updating all cached values upon a write by a processor advantageous when a write to a cache block tends to be followed by reads by other processors

Cache Coherence Protocol

- **write update:** directly updating all cached values upon a write by a processor advantageous when a write to a cache block tends to be followed by reads by other processors
- **write invalidate:** invalidating all other cached values via dirty bit

Cache Coherence Protocol

- **write update:** directly updating all cached values upon a write by a processor
  advantageous when a write to a cache block tends to be followed by reads by other processors
- **write invalidate:** invalidating all other cached values via dirty bit
  advantageous when a write to a cache block tends to be followed by subsequent writes to the same block;
  invalidation occurs only once and subsequent writes do not generate any more traffic

Cache Coherence Protocol

- requests are broadcast to all caches $\Longrightarrow$ *broadcast/snoopy* protocols

Cache Coherence Protocol

- requests are broadcast to all caches $\Longrightarrow$ *broadcast/snoopy* protocols
- requests sent only to select caches

Cache Coherence Protocol

- requests are broadcast to all caches $\Longrightarrow$ *broadcast/snoopy* protocols
- requests sent only to select caches
  require directory to keep track of which caches should be involved
  $\Longrightarrow$ *directory* protocols

Cache Coherence Protocol

- requests are broadcast to all caches $\Longrightarrow$ *broadcast/snoopy* protocols
- requests sent only to select caches
  require directory to keep track of which caches should be involved
  $\Longrightarrow$ *directory* protocols

For more details on how to implement cache coherence protocols see:
Yan Solohin, *Fundamentals of Parallel Multicore Architecture*, Chapman & Hall/CRC, Boca Raton, FL, 2016, ISBN: 978-0-367-57528-1

Memory Consistency Problem

Example: Producer – Consumer

	T_1	$(D, F, X) = (0, 0, 0)$		T_2
d:	`D := 42;`		if:	`if F=0 then goto if;`
f:	`F := 1;`		x:	`X := D;`

Memory Consistency Problem

Example: Producer – Consumer

	T_1	$(D, F, X) = (0, 0, 0)$		T_2
d:	`D := 42;`		if:	`if F=0 then goto if;`
f:	`F := 1;`		x:	`X := D;`

- *Out-of-order execution*: compiler or CPU may reorder statements `d` and `f` because there is no data dependency between them

Memory Consistency Problem

Example: Producer – Consumer

	T_1	$(D, F, X) = (0, 0, 0)$		T_2
d:	`D := 42;`		if:	`if F=0 then goto if;`
f:	`F := 1;`		x:	`X := D;`

- *Out-of-order execution*: compiler or CPU may reorder statements `d` and `f` because there is no data dependency between them
- Thus T_2 may find `F` equal to 1 before T_1 has set `D` to 42.

Memory Consistency Problem

Example: Producer – Consumer

	T_1	$(D, F, X) = (0, 0, 0)$		T_2
d:	`D := 42;`		if:	`if F=0 then goto if;`
f:	`F := 1;`		x:	`X := D;`

- *Out-of-order execution*: compiler or CPU may reorder statements `d` and `f` because there is no data dependency between them
- Thus T_2 may find `F` equal to 1 before T_1 has set `D` to 42.
- T_2 may assign 0 to `X`

Memory Consistency Problem

Example: Producer – Consumer

	T_1	$(D, F, X) = (0, 0, 0)$		T_2
d:	`D := 42;`		if:	`if F=0 then goto if;`
f:	`F := 1;`		x:	`X := D;`

- *Out-of-order execution*: compiler or CPU may reorder statements `d` and `f` because there is no data dependency between them
- Thus T_2 may find `F` equal to 1 before T_1 has set `D` to 42.
- T_2 may assign 0 to `X`

$\Longrightarrow$ **Memory Consistency Problem**

Memory Consistency Problem

Example: Producer – Consumer

T_1		$(D, F, X) = (0, 0, 0)$	T_2	
d:	`D := 42;`		if:	`if F=0 then goto if;`
f:	`F := 1;`		x:	`X := D;`

- *Out-of-order execution*: compiler or CPU may reorder statements `d` and `f` because there is no data dependency between them
- Thus T_2 may find `F` equal to 1 before T_1 has set `D` to 42.
- T_2 may assign 0 to `X`

$\Longrightarrow$ **Memory Consistency Problem**

- example shows problem without caching

Memory Consistency Problem

Example: Producer – Consumer

	T_1	$(D, F, X) = (0, 0, 0)$		T_2
d:	`D := 42;`		if:	`if F=0 then goto if;`
f:	`F := 1;`		x:	`X := D;`

- *Out-of-order execution*: compiler or CPU may reorder statements `d` and `f` because there is no data dependency between them
- Thus T_2 may find `F` equal to 1 before T_1 has set `D` to 42.
- T_2 may assign 0 to `X`

$\Longrightarrow$ **Memory Consistency Problem**

- example shows problem without caching
- problem may become worse if caches are involved

Synchronization Problem

- want to ensure that only one of several threads enters a so-called *critical section*

Synchronization Problem

- want to ensure that only one of several threads enters a so-called *critical section*
- implementation via *Lock* and *Unlock* operations

Synchronization Problem

- want to ensure that only one of several threads enters a so-called *critical section*
- implementation via *Lock* and *Unlock* operations
- if second thread tries to lock, the thread is blocked until the first thread unlocks

Synchronization Problem

Counter, initialized to 1

Lock:	Decrease counter by 1. If counter ≥ 0, thread may continue execution. If counter < 0, enqueue thread in a waiting queue & stop execution.
Unlock:	Increase counter by 1. If counter > 0, thread may continue execution. If counter ≤ 0, release 1st thread from waiting queue & start execution.

Synchronization Problem

Counter, initialized to 1

Lock:	Decrease counter by 1.
	If counter ≥ 0, thread may continue execution.
	If counter < 0, enqueue thread in a waiting queue
	& stop execution.
Unlock:	Increase counter by 1.
	If counter > 0, thread may continue execution.
	If counter ≤ 0, release 1st thread from waiting queue
	& start execution.

Race condition!

Synchronization Problem

Counter, initialized to 1

Lock:	Decrease counter by 1.
	If counter ≥ 0, thread may continue execution.
	If counter < 0, enqueue thread in a waiting queue
	& stop execution.
Unlock:	Increase counter by 1.
	If counter > 0, thread may continue execution.
	If counter ≤ 0, release 1st thread from waiting queue
	& start execution.

Race condition!

Everything that is **blue** must be executed atomically.

Cf. e.g. lecture on the topic of "Operating Systems"

Synchronization Problem

- Atomicity via HW instructions.

Synchronization Problem

- Atomicity via HW instructions.
- E.g.: **Read-Modify-Write** operations.

Synchronization Problem

- Atomicity via HW instructions.
- E.g.: **Read-Modify-Write** operations.
- Different instructions for different processors.