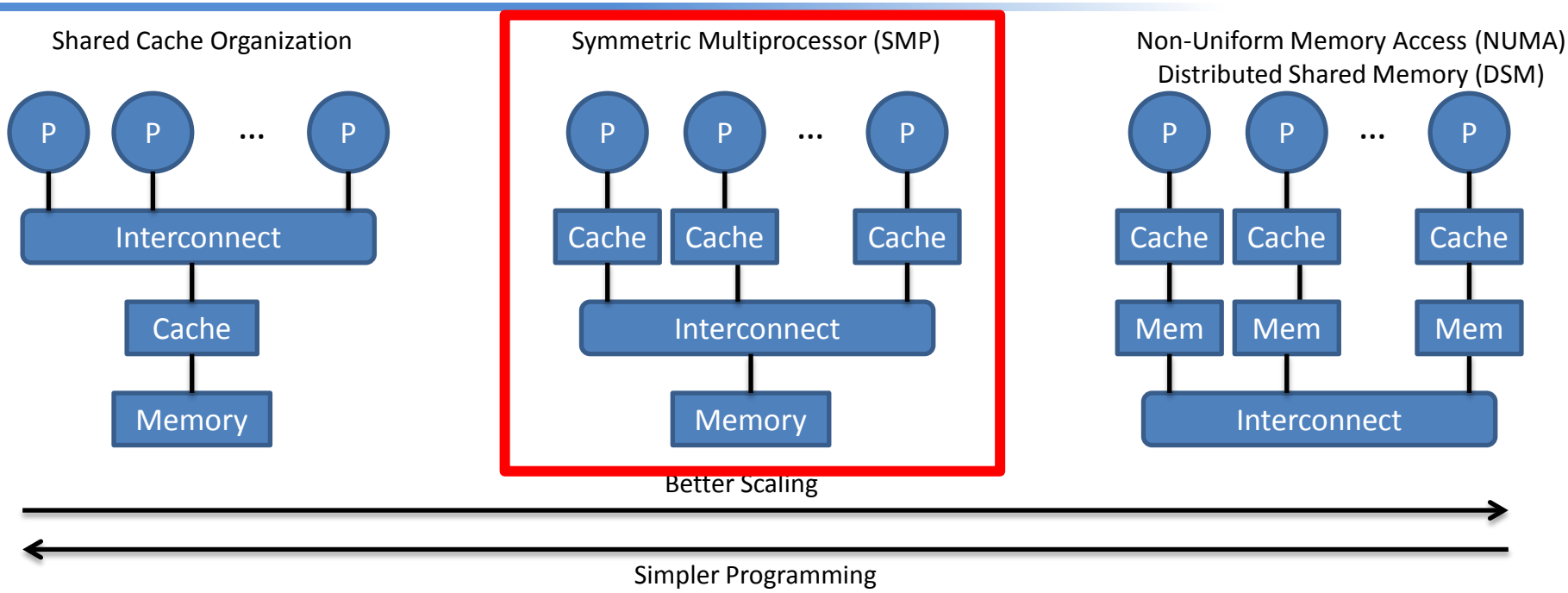
Processor Architecture

Shared Memory Multiprocessors

M. Schölzel

Organization of Shared Memory Multiprocessor Systems



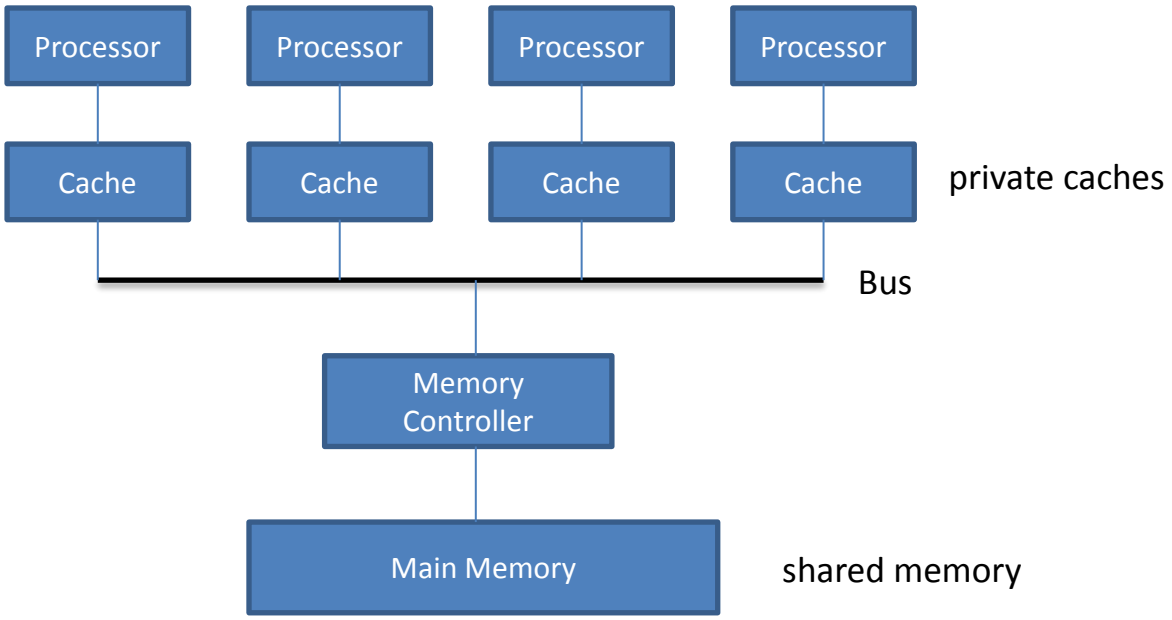
- Each processor can access any part of the cache
- No cache coherence problem
- Cache and interconnect must support many accesses at once (bad scaling)

- Faster Cache access possible
- Cache coherence problem occurs
- Better scaling of interconnects; memory traffic is already filtered
- If only 10% of the traffic is on the interconnect then 10x processors can be supported

- Scales very good
- More difficult to program
  - allocation of tasks to processors and
  - memory allocation policies of the OS should be taken into account

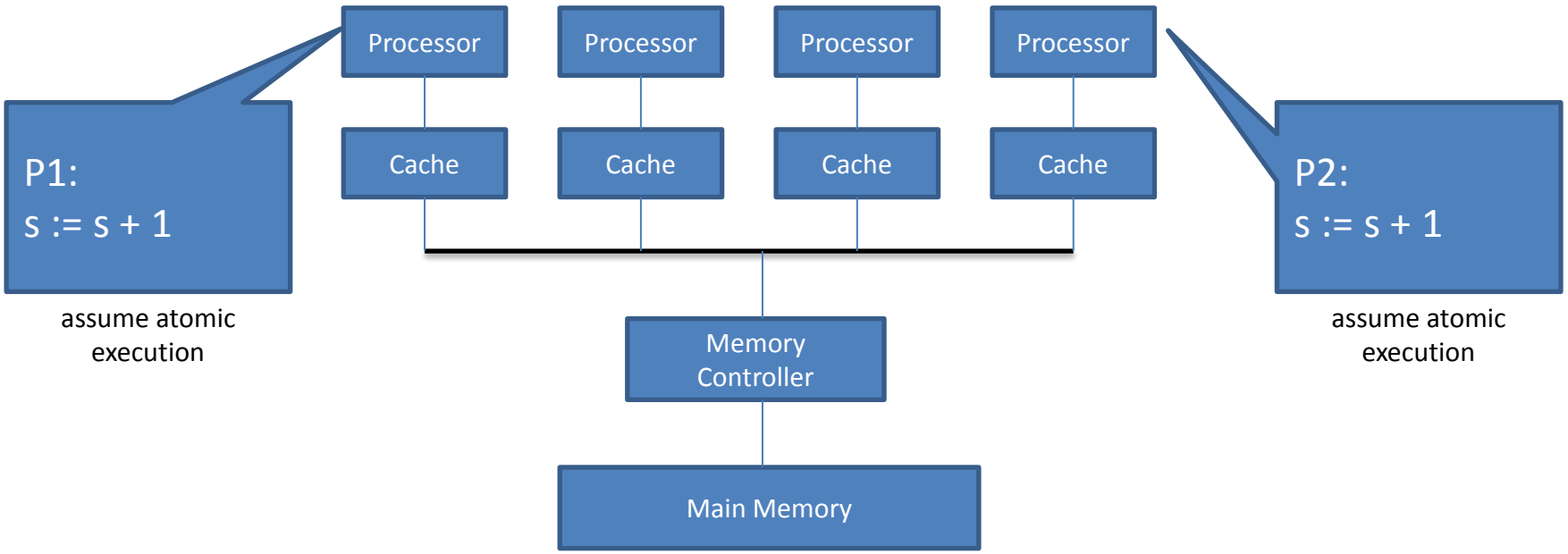
The Cache Coherence Problem

- Caches may contain local copies of the same memory address
- without proper coordination they work independently on their local copies



Example

- Both processors have a copy of the same process
- Without proper coordination both work on different copies
- If the block is evicted from both caches, then s is only incremented by 1
- no matter if the cache is write-back or write-through



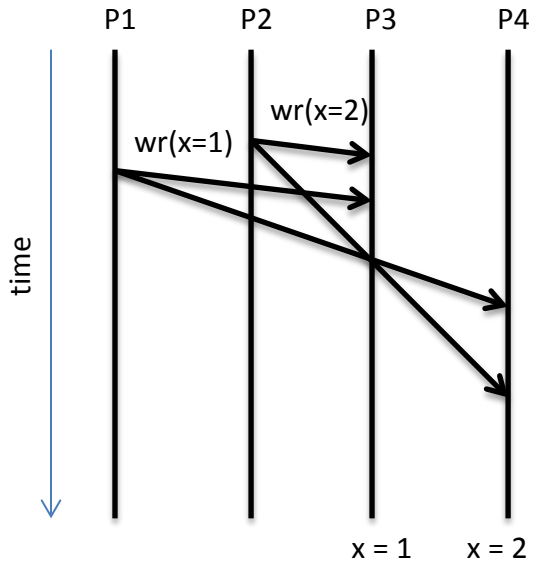
Shared Memory Support

Required Hardware Support for Shared Memory Multiprocessor Systems:

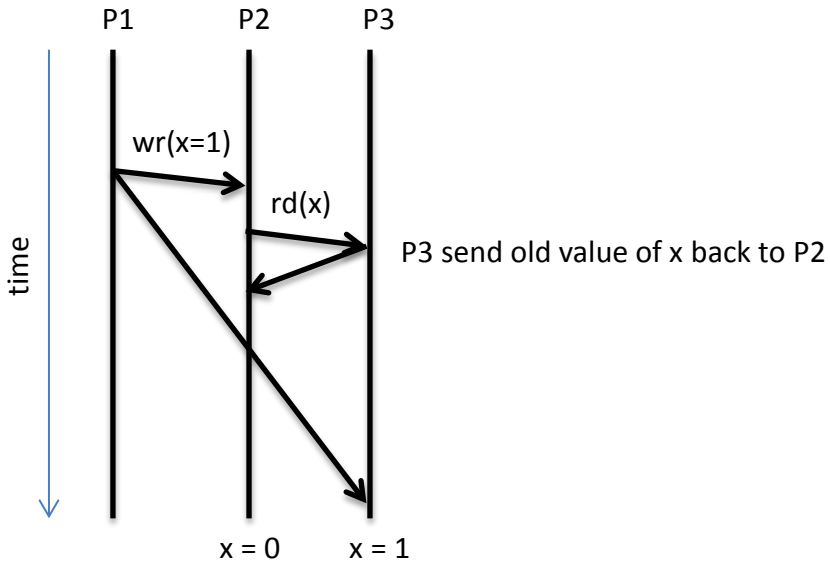
- Cache Coherence Protocol
  - coherent view on cached values by multiple processors
- Memory Consistency Models
  - consistent view by all processors on the sequence of memory operations
- Hardware Synchronization Support
  - for synchronization of multiple programs

Enable Coherent View

- Write-Propagation is required: All other caches must be notified by transactions that a value was changed in a local cache
- Serialization of these transactions is required: transaction serialization
  - When using a bus this serialization comes automatically



transaction serialization of write and write operations required



transaction serialization of write and read operations required

Classification of Cache Coherence Protocols

According to their write-propagation strategy

- Write-Update: updating all values in the other caches
- Write-Invalidate: invalidating the values in all other caches, forcing them to reload the block

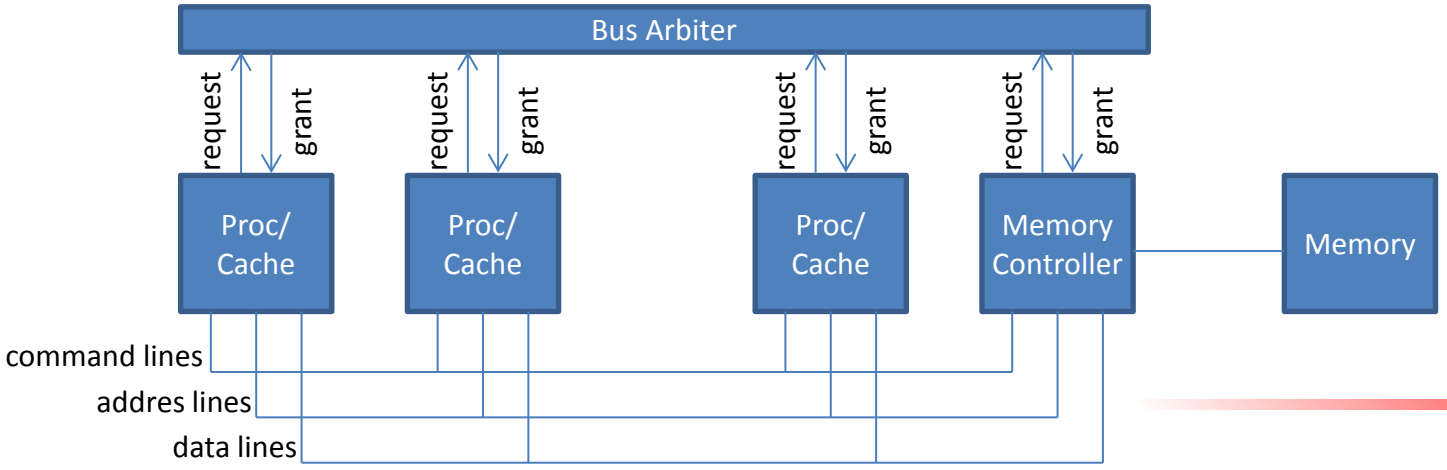
According to their transaction-propagation strategy

- Broadcast/Snoopy-protocols
  - Transaction is propagated to all caches
  - less scalable
- Directory-based-protocols
  - Only to selected Caches: directory must be maintained for keeping track of already updated caches
  - Disadvantages:
    - Storage overhead
    - indirection of requests (must be send first to the directory)

Basic Support for Bus-Based Multiprocessors (1)

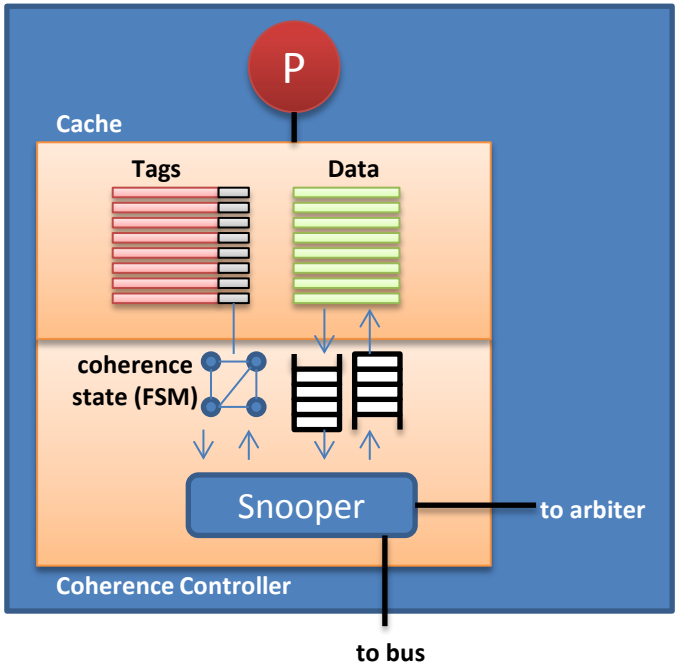
- The bus is a shared medium by all processors/memories
- Bus transactions are done in three phases
  - Arbitration
    - Processor/Cache requests the bus
    - Bus arbiter grants access to one of the requestor
  - Command: Requestor puts the command and address on the bus
  - Data Transfer: Data is send

  - In modern Systems split transactions are used:
    - bus is released between command and data transfer



Basic Support for Bus-Based Multiprocessors (1)

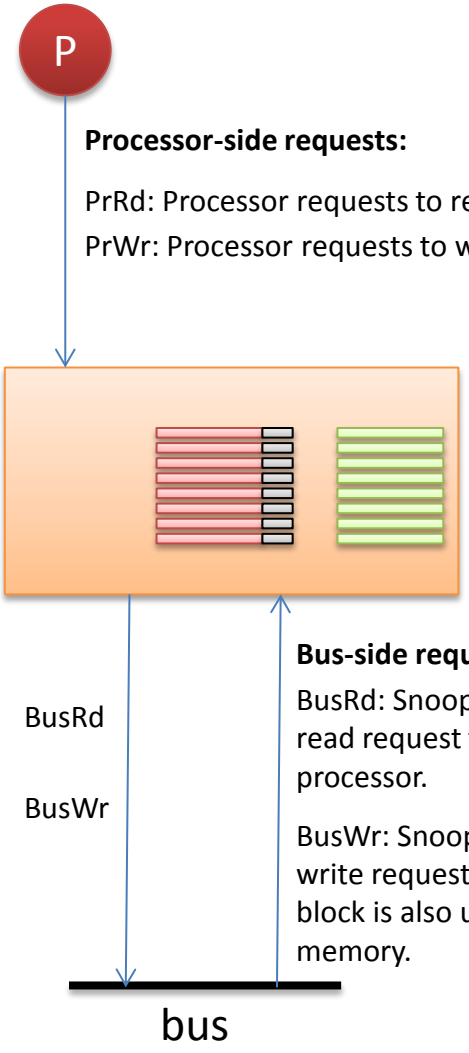
- Coherence is maintained at the granularity of cache blocks
- Bus snooper snoops each transaction on the bus
  - track operations (read/write) on blocks by other requestors
  - updates the local state of the block represented by a FSM accordingly
  - FSM changes for different cache coherence protocols



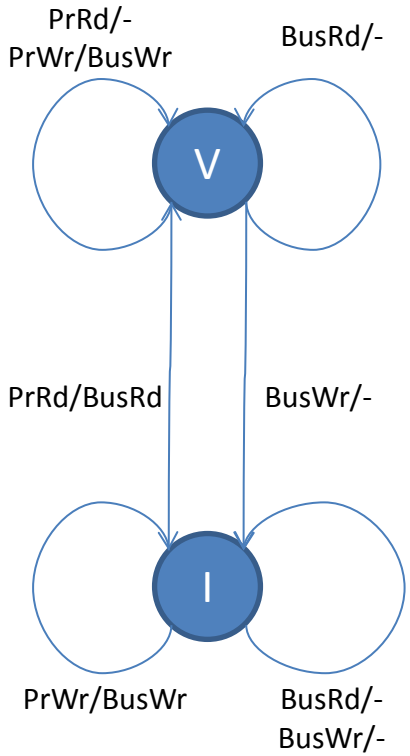
- For Write-Update-Protocols
  - Snooper updates local block
- For Write-Invalidate-Protocols
  - Snooper invalidates the local block

- Transaction serialization is achieved by the bus
  - if snooper looses competition for a bus-write it has to snoop the bus, if another access to the same block takes place and then update its own block accordingly, possibly canceling its own request

Cache Coherence Protocol for Write-Through Caches



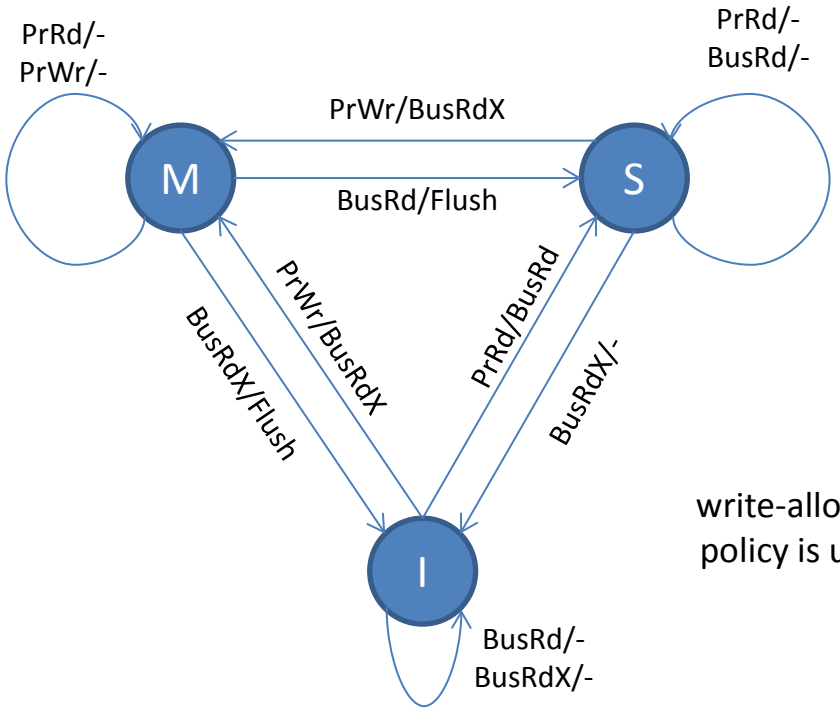
State Transition Diagram for the Cache Coherence Protocol
(write-no-allocate policy is used)



MSI Protocol for Write-Back-Caches

- Requests from processor-side:
  - PrRd: Request to read a cache block
  - PrWr: Request to write a cache block
- Requests from bus-side:
  - BusRD: snooped request, there is a read request to a block by another processor
  - BusRdX: snooped request, there is a read exclusive request (wants to write) to a cache block made by another processor
  - Flush: snooped request, the entire block is written back to the main memory by another processor

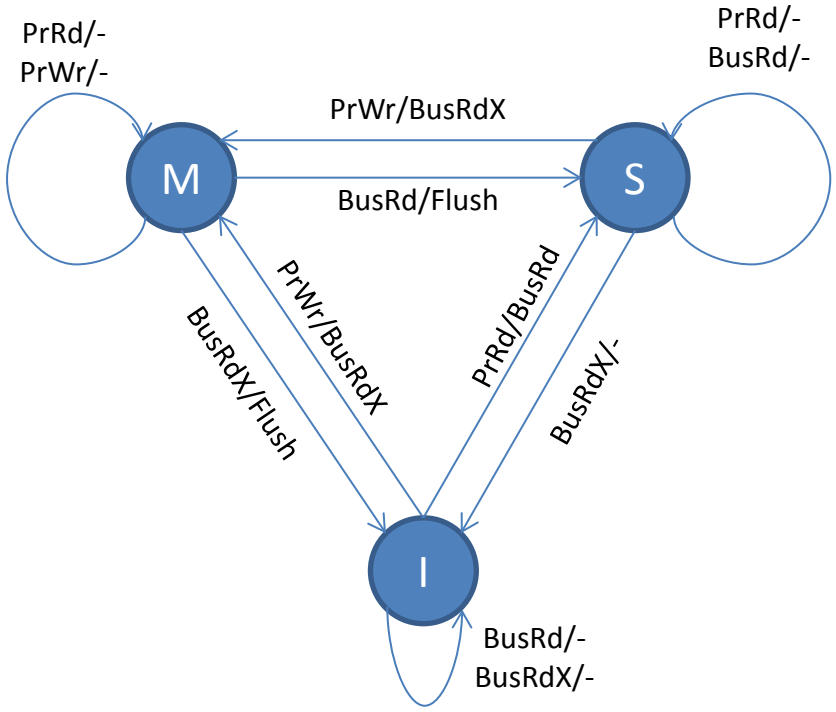
- FSM-States
  - Modified (M): the cache block is valid in only one cache and maybe dirty
  - Shared (S): the cache block is valid, maybe shared with other processors and not dirty
  - Invalid (I): the cache block is invalid



write-allocate policy is used

Example

	Request	P1	P2	P3	Bus Command	Data Supplier
0	-	I	I	I	-	-
1	R1	S	I	I	BusRd	Mem
2	W1	M	I	I	BusRdX	Mem
3	R3	S	I	S	BusRd, Flush (P3 receives block)	Cache P1
4	W3	I	I	M	BusRdX	Mem
5	R1	S	I	S	BusRd, Flush (P1 receives block)	Cache P3
6	R3	S	I	S	-	-
7	R2	S	S	S	BusRd	Mem



- Write-Propagation for a written block is ensured by two mechanisms:
  - BusRdX invalidates a block in other caches, forcing them
    - to reload the block, if it was not dirty (S)
    - to flush the block, if it was dirty
      - Processor that puts the BusRdX-command has to snoop the bus for the flush

Limitation of MSI

- Processor reads a block (not used by other processors) and writes then to the block
  - BusRd + BusRdX are generated, but BusRdX is useless, because the block is already exclusively used
  - Unnecessary traffic is generated on the bus; this may happens if
    - Process is running on a Processor and no other processors use the same block
    - Parallel program runs with multiple threads, which are tuned not to share the same blocks

- Flaw of MSI: It cannot be distinguished between an exclusive block that is not dirty and a shared block that is not dirty

- Solution: Split the state S into states S and E $\rightarrow$ MESI protocol

MESI Protocol for write-back caches

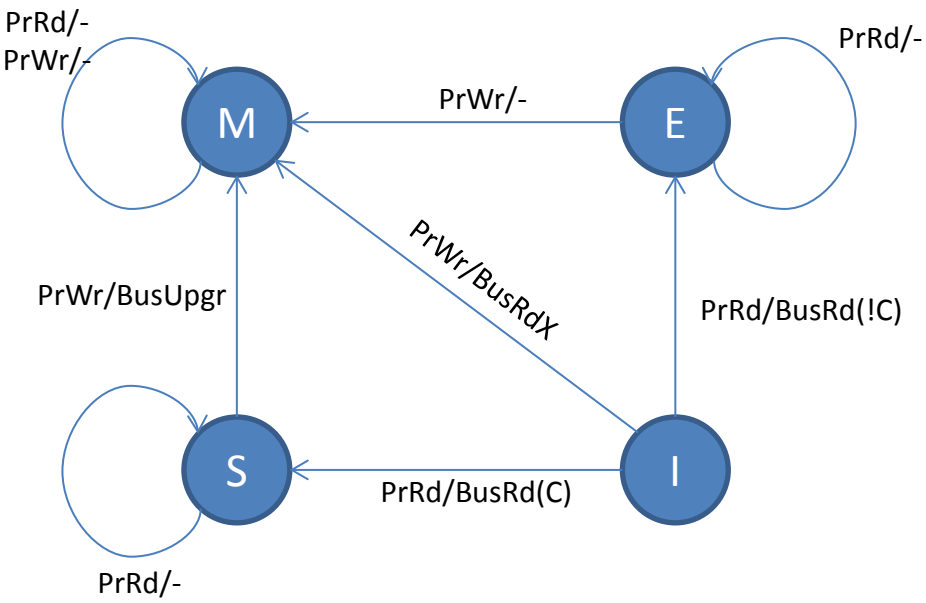
- Requests from processor side:
  - PrRd: Request to read a cache block
  - PrWr: Request to write a cache block
- Snooped requests from bus side:
  - BusRd: there is a read request to a block by another processor
  - BusRdX: there is a read exclusive (wants to write) request to a cache block made by another processor
  - BusUpgr: write request to a cache block that another processor already has in its cache
  - Flush: the entire block is written back to the main memory by another processor
  - FlushOpt: a block is posted on the bus in order to supply it to other processors

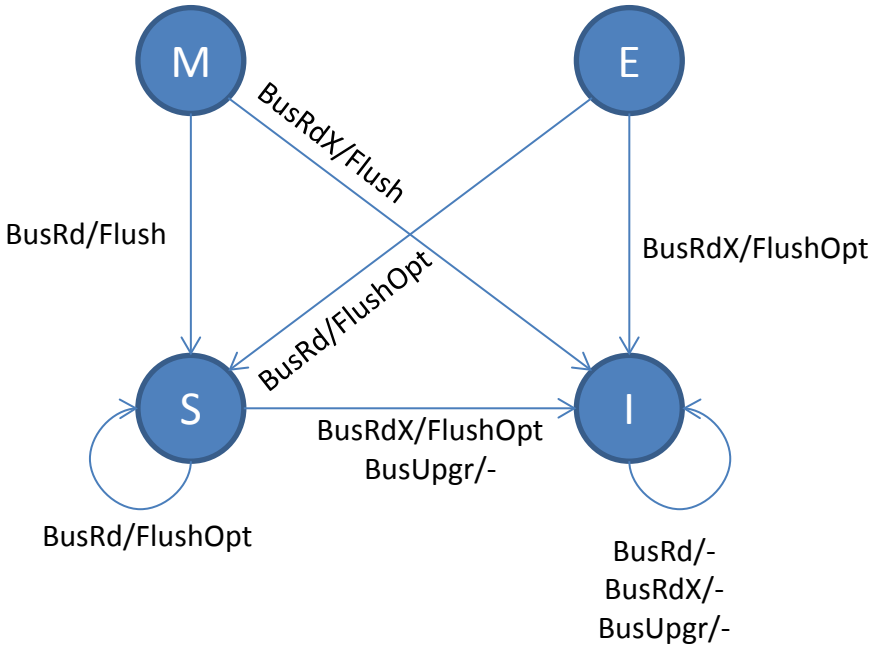
- FSM-States
  - Modified (M): the cache block is valid in only one cache and maybe dirty
  - Exclusive (E): the cache block is valid, clean, and resides in only one cache
  - Shared (S): the cache block is valid, maybe shared with other processors and not dirty
  - Invalid (I): the cache block is invalid

- If a cache loads a block, other caches can show by a new bus line (COPIES-EXIST), if they have the same block. Depending on that signal the block is loaded into state E or S

Used by Intel processors (e.g. Xeon)

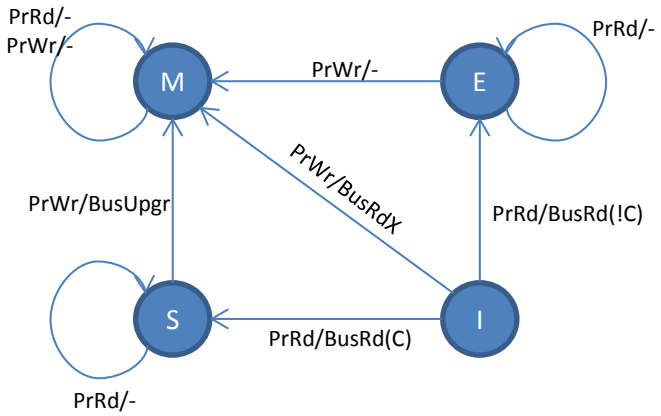
FSM for MESI

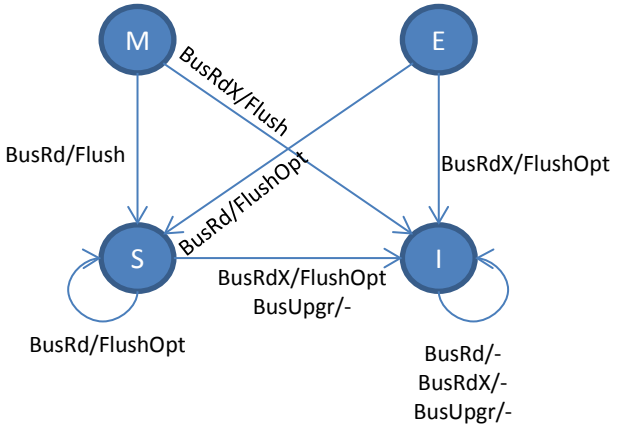

PrRd/-
PrWr/-
M
PrWr/-
E
PrRd/-
PrWr/BusUpgr
PrWr/BusRdX
PrRd/BusRd(!C)
S
PrRd/BusRd(C)
I
PrRd/-


M
BusRdX/Flush
E
BusRd/Flush
BusRdX/FlushOpt
BusRd/FlushOpt
S
BusRdX/FlushOpt
BusUpgr/-
I
BusRd/FlushOpt
BusRd/-
BusRdX/-
BusUpgr/-

Example MESI

	Request	P1	P2	P3	Bus Command	Data Supplier
0	-	I	I	I	-	-
1	R1	E	I	I	BusRd	Mem
2	W1	M	I	I	- (MSI: BusRdX)	- (MSI: Mem)
3	R3	S	I	S	BusRd, Flush (P3 receives block)	Cache P1
4	W3	I	I	M	BusUpgr (MSI: BusRdX)	- (MSI: Mem)
5	R1	S	I	S	BusRd Flush (P1 receives block)	Cache P3
6	R3	S	I	S	-	-
7	R2	S	S	S	BusRd	Cache P1/P3 (MSI: Mem)





Other Protocols for Write-Back-Caches

MOESI

- Allows for dirty sharing, which was not the case with MSI, MESI
- Dirty Sharing allows one processor to have the exclusive ownership of a modified block that is shared with other processors and quick supply of that block to other processors
- Used by AMD processors like the Opteron

MESIF

- Read misses on a block cannot be served by peer caches efficiently (neither in MESI nor MOESI)
- MESIF is used from Intel to solve this problem by an additional forward-state; but dirty sharing is not supported

The Dragon Protocol (1)

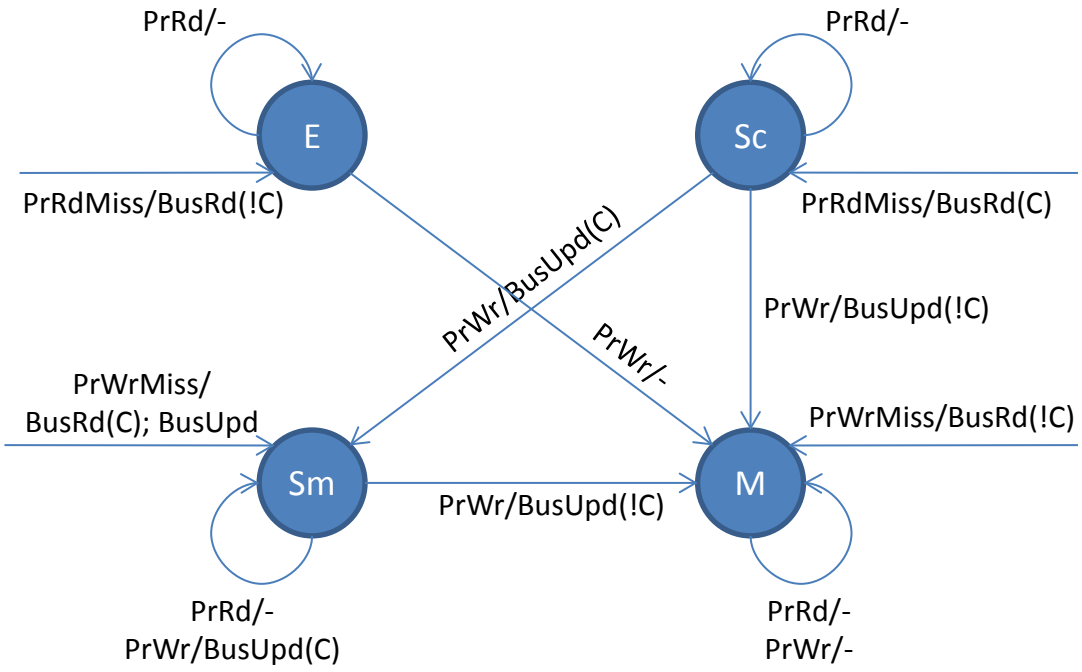
- In invalidate-based protocols read misses may occur due to invalidated blocks
  - invalidation was done to support write-propagate by reloading the block from the memory
  - write-propagation is done through the lower level memory hierarchy
- Update-based Protocols allow write-propagation without going through the lower level memories

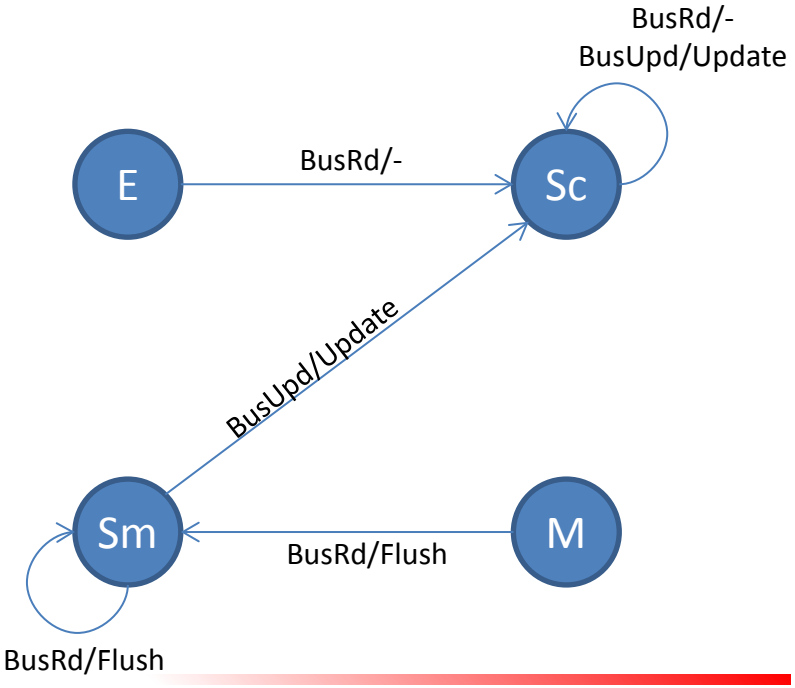
The Dragon Protocol (2)

- Requests from processor side:
  - PrRd: Request to read a cache block that is already in a cache
  - PrRdMiss: Request to read a cache block that is not in a cache so far
  - PrWr: Request to write a cache block that is already in a cache
  - PrWrMiss: Request to write a cache block that is not in a cache so far
- Snooped requests from bus side:
  - BusRd: there is a read request to a block by another processor
  - Flush: the entire block is written back to the main memory by another processor
  - BusUpd: a write to a word that is propagated to the bus for updating other caches

- FSM-States
  - Modified (M): the cache block is valid in only one cache and maybe dirty
  - Exclusive (E): the cache block is valid, clean, and resides in only one cache
  - Shared Modified(Sm): the cache block is valid, possibly dirty, maybe shared with other processors and not dirty there (it cannot be shared modified there)
  - Shared Clean (sc): cache block is valid, possibly shared with other caches, and maybe dirty in one of them

FSM for Dragon Protocol





Example Dragon

	Request	P1	P2	P3	Bus Command	Data Supplier
0	-	I	I	I	-	-
1	R1	E	I	I	BusRd	Mem
2	W1	M	I	I	-	-
3	R3	Sm	I	Sc	BusRd	Cache P1
4	W3	Sc	I	Sm	BusUpd	-
5	R1	Sc	I	Sm	-	-
6	R3	Sc	I	Sm	-	-
7	R2	Sc	Sc	Sm	BusRd	Cache P3