cudaMallocManaged

- `cudaMallocManaged` is a function in the CUDA API that allocates memory that can be accessed by both the CPU and GPU.

- **Unified Memory Allocation**
  - When you use `cudaMallocManaged`, the allocated memory is accessible from both the host (CPU) and the device (GPU).
  - This eliminates the need for explicit memory transfers like `cudaMemcpy` between the host and device.

- **Automatic Data Migration**:
  - The CUDA runtime automatically migrates data between the host and device as needed. When the CPU accesses the memory, the data is moved to the host memory, and when the GPU accesses it, the data is moved to the device memory. This migration is handled through page faults

- **Simplified Programming Model**:
  - This is particularly useful for applications with irregular memory access patterns

Multicore Architecture & Cache Coherence

References

- Chapters 5, 6, from Parallel Computer Architecture A Hardware/Software Approach, D. E. Culler, J. P. Singh, and A. Gupta, Morgan Kaufmann Publishers, Inc., ISBN-13: 9781558603431
- Intel performance analysis guide https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf
- Chapters 1, 4, from Introduction to High Performance Computing for Scientists and Engineers (Chapman & Hall/CRC Computational Science) Georg Hager and Gerhard Wellein.
- Intel: Optimizing Applications for NUMA https://software.intel.com/content/www/us/en/develop/articles/optimizing-applications-for-numa.html
- Intel: Avoiding and Identifying False Sharing Among Threads https://software.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html

Hardware determines (parallel) software performance

Why do you need to know about hardware architecture?

- In order to write efficient parallel software, you must be aware of the hardware design and constraints.
- Hardware parallelism is a function of cost and performance tradeoffs.
- It is is ultimately the hardware architecture that determines the cost (execution time) associated with each algorithmic motif e.g. CPU vs GPU

Taxonomy of parallel computing paradigms

A widely used taxonomy for describing the amount of concurrent control and data streams present in a parallel architecture was proposed by Flynn*.

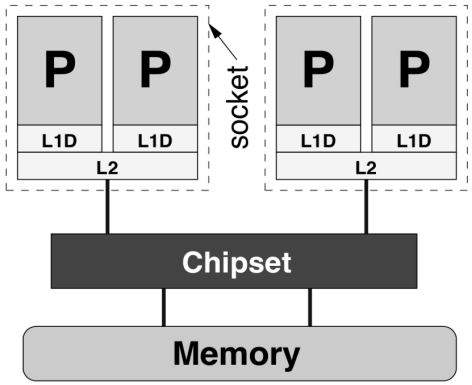
Single Instruction, Multiple Data (SIMD). A single instruction stream, either on a single processor (core) or on multiple compute elements, provides parallelism by operating on multiple data streams concurrently. Example: the SIMD capabilities (vectorization) of modern superscalar microprocessors

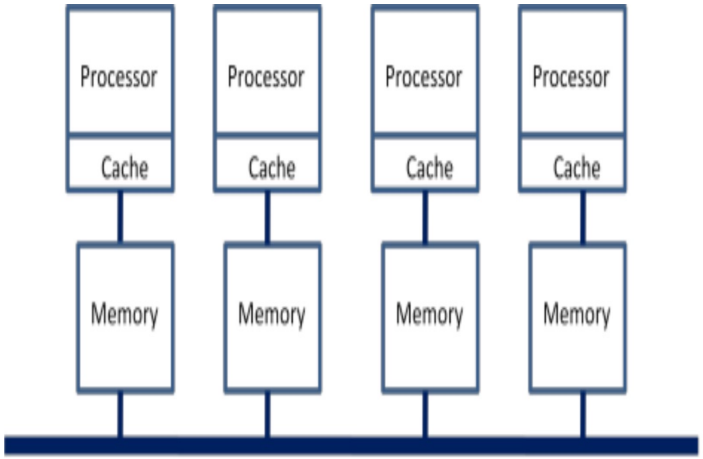
Multiple Instruction, Multiple Data (MIMD). Multiple instruction streams on multiple processors (cores) operate on different data items concurrently. Shared-memory and distributed-memory parallel computers are typical examples for the MIMD paradigm.

Single-stream ILP as employed in superscalar, pipelined execution is not included in this categorization.

Flynn, Michael J. (December 1966). "Very high-speed computing systems". Proceedings of the IEEE. **54 (12): 1901–1909. doi:10.1109/PROC.1966.5273.*

Shared-Memory Multiprocessors



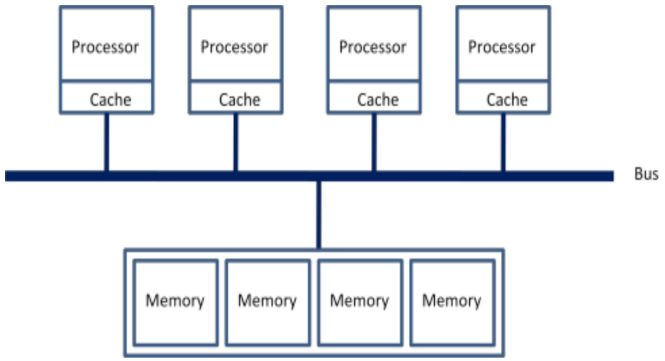


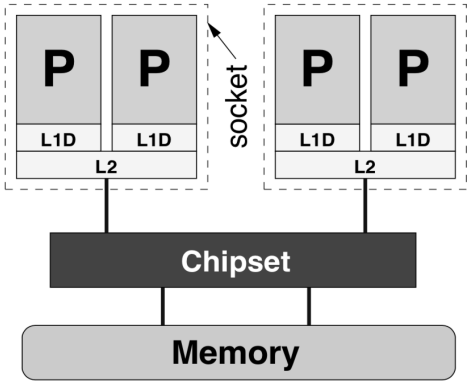
Single address space multiprocessors come in two forms: -

- **Uniform Memory Access (UMA) Multiprocessors**: Also known as Symmetric Multiprocessors (SMP), these systems have a memory architecture in which each processor in the multiprocessor system has uniform access time to memory.
- In other words, latency and bandwidth are the same for all processors and all memory locations. The latency to a word in memory does not depend on which processor/core asks for it. This architecture is common in multicore processor chips, such as those found in basic PCs or mobile phones.

- **Non-Uniform Memory Access (NUMA) Multiprocessors**: In these systems, memory access time depends on the memory location relative to a processor.
- The architecture is used in multiprocessor systems and aims to improve system performance by allowing a processor to access its local memory faster than non-local memory (memory local to another processor or memory shared between processors)

Shared-Memory Multiprocessors

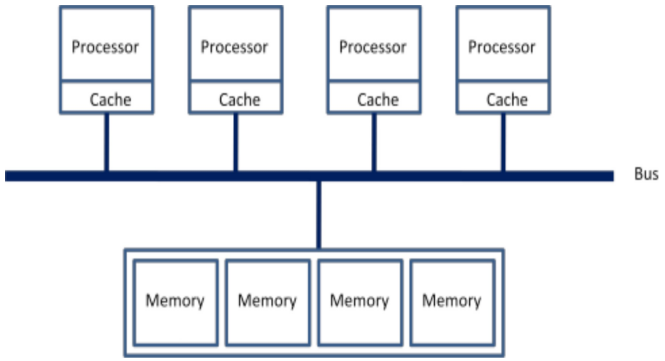




- **A shared memory multiprocessor** (SMMP) is one that offers the programmer a single physical address space across all processors – which is nearly always the case for multicore chips – although a more accurate term would have been shared-address multiprocessor.

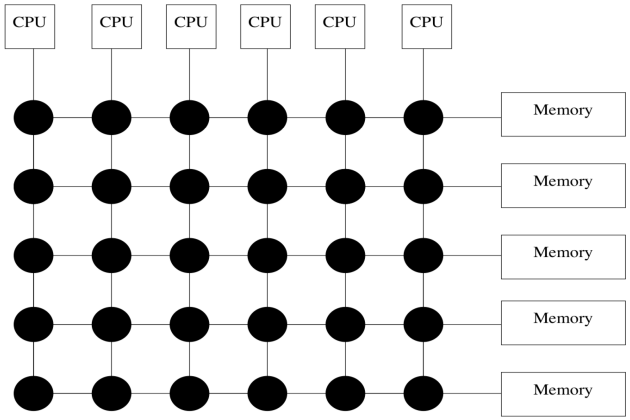
- **Uniform Memory Access** (UMA) multiprocessors. Latency and bandwidth are the same for all processors and all memory locations. This is also called a *symmetric multiprocessor* (SMP). The latency to a word in memory does not depend on which processor/core asks for it.
- This architecture covers almost all single multicore processor chips, *e.g.* 4 or more cores on basic PC or in your mobile phone.

Shared-Memory Multiprocessors



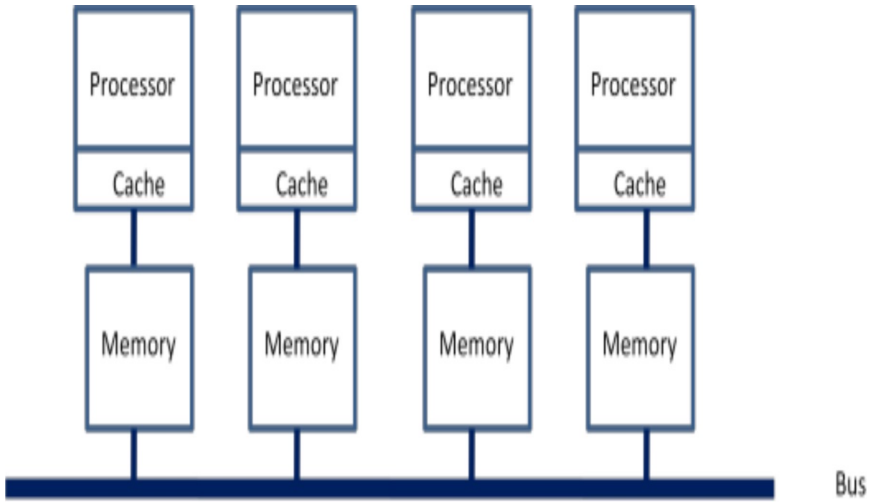
Uniform Memory Access (UMA) multiprocessors.

- Bandwidth bottlenecks are bound to occur when the number of sockets (or Front Side Buses) is larger than a certain limit. Does not scale due to the blocking nature of the buses.



- Performance is improved using nonblocking networks such as *crossbar switches* that establish point-to-point connections between sockets and memory modules. Does not scale because nonblocking networks quickly become too expensive.

Shared-Memory Multiprocessors



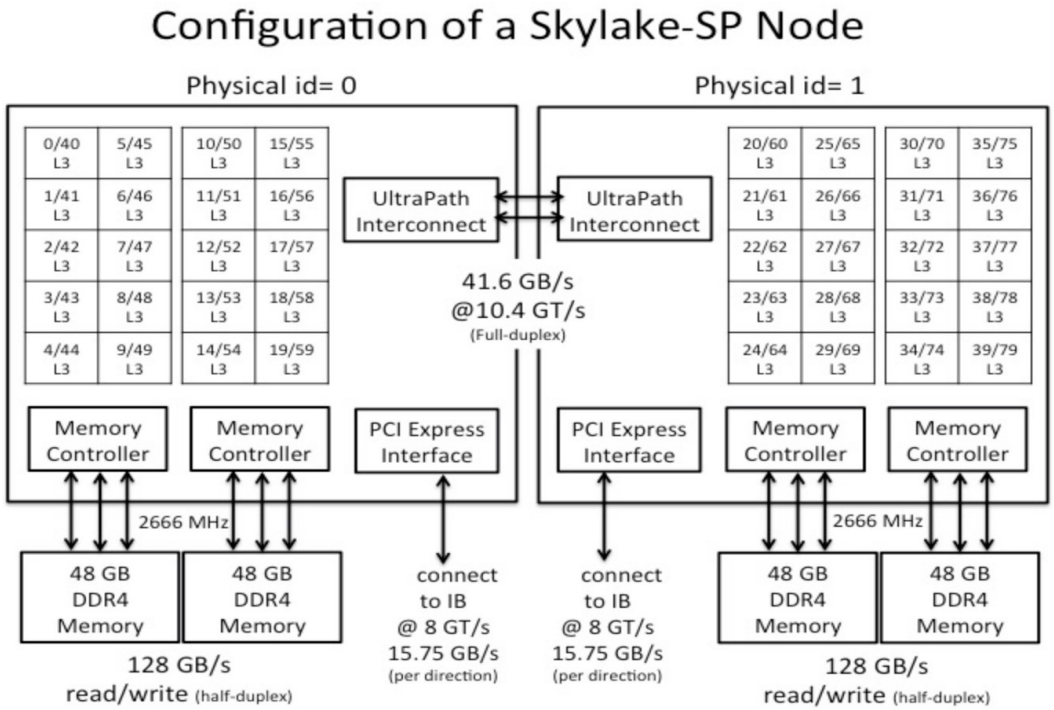
- **cache coherent Non-Uniform Memory Access** (ccNUMA)
  - Each processor has its own local memory module that it can access directly with a distinctive performance advantage.
  - At the same time, it can also access any memory module belonging to another processor using a shared bus (or some other type of interconnect).
  - A locality domain (LD) or *NUMA node* is a set of processor cores together with their locally connected memory

Shared-Memory Multiprocessors

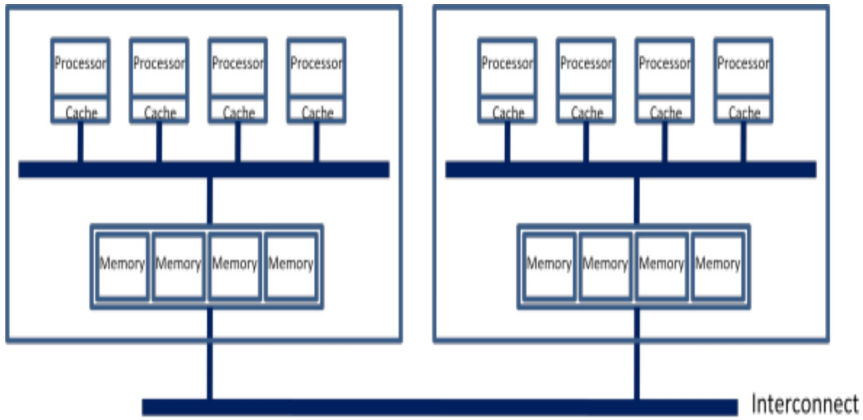
Shared memory multiprocessors (SMMP) come in two styles.

cache coherent Non-Uniform Memory Access (ccNUMA)

Intel SkyLake architecture



Modern Shared-Memory Multiprocessors



- **cache coherent Non-Uniform Memory Access** (ccNUMA)
  - Modern multiprocessor systems mix UMA locality domains within an overall NUMA architecture
  - The LDs are linked via a *coherent* interconnect, which allows access from any processor to any other processor's memory (single address space).
  - The interconnect provides a high-speed connection. QuickPath (QPI) current technologies favored by Intel, were replaced on Skylake by Intel Ultra Path Interconnect (UPI)

The Cache Coherence Problem

Cache line design (review)

Cache write hit policies:-

- **Write-through:** This policy writes data to the cache and the main memory at the same time. It ensures consistency as the main memory always contains the same data as the cache. However, it can be slower because every write to the cache requires a write to the main memory.

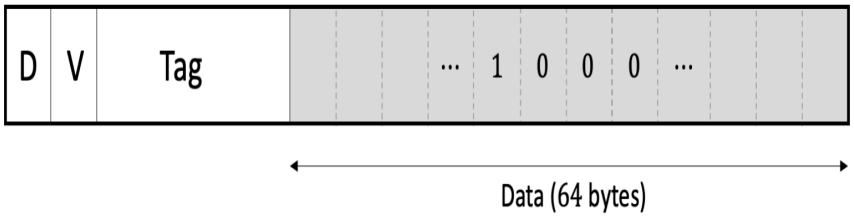
- **Write-back:** In this policy, only the cache location is updated during a write operation. The main memory is updated only when the word is replaced from the cache. This policy can reduce the number of references to main memory, thereby improving performance. However, it may lead to inconsistency between the cache and main memory.

Cache write miss policies: -

- **Write-no-allocate** or **write-around:** This policy bypasses cache entry allocation in case of a cache miss. This policy helps avoid cache pollution (filling the cache with entries that are not frequently used) but may result in higher latency for read operations if the recently written data is accessed soon.
- **Write-allocate:** Under this policy, the cache line is loaded into the cache, followed by a write operation. This is done with the hope that subsequent writes (or even reads) will be made to that location, leading to a hit. In other words, a cache block is first allocated before performing the write action.

Cache line design (review)

Let's assume our data is `int x = 1`



When evicting (removing data from the cache)

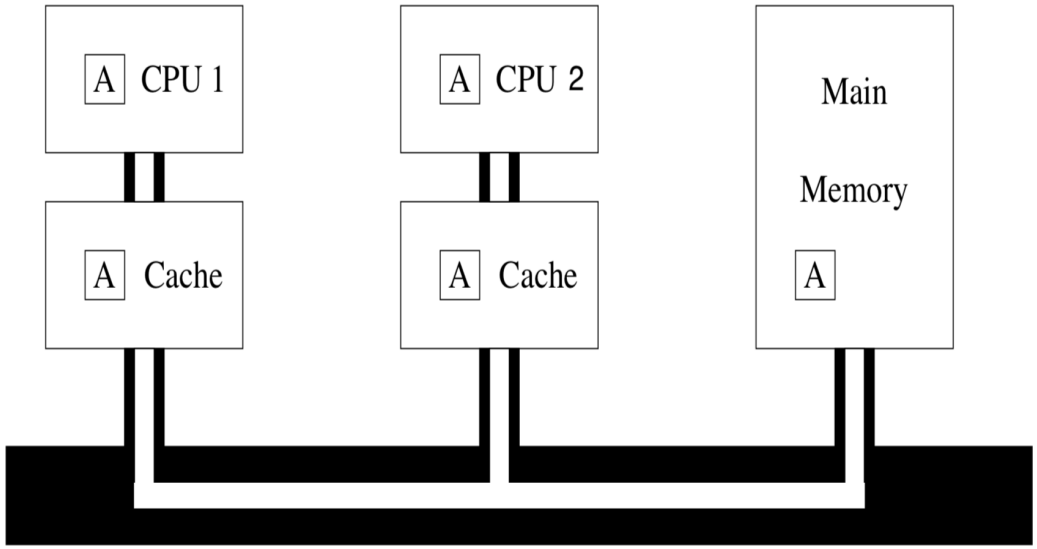
D =?, *V* =?

- The "D" stands for "Dirty" bit. If D=1, it means that the data in the cache has been modified (is "dirty") and needs to be written back to the main memory before eviction. If D=0, the data has not been modified (is "clean") and can be safely evicted without a write-back.

- The "V" stands for "Valid" bit. If V=1, it means that the data in the cache line is valid. If V=0, then the cache line is either empty or has been invalidated and should not be read.

Parallel caching

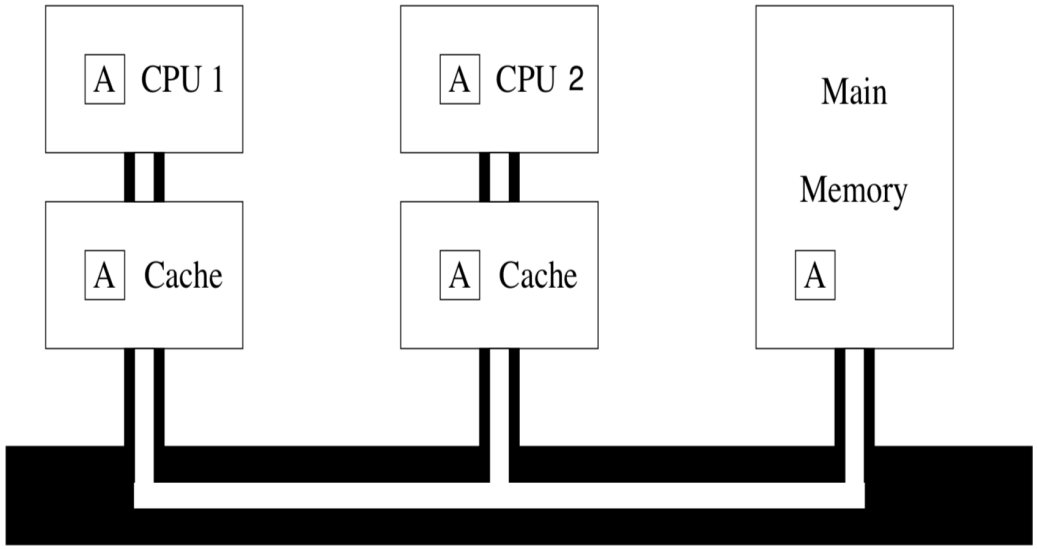
What happens if two processors (cores) want to read and write to the same location in memory (*e.g.* our `int A`)?



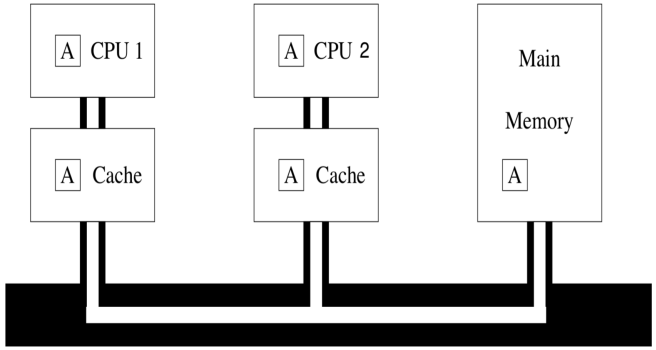
Parallel caching

What happens if multiple processors (cores) want to read and write to the same location in memory (*e.g.* our `int A`)?

- Reading `A` should return the last value written to its address by any processor.



Parallel caching

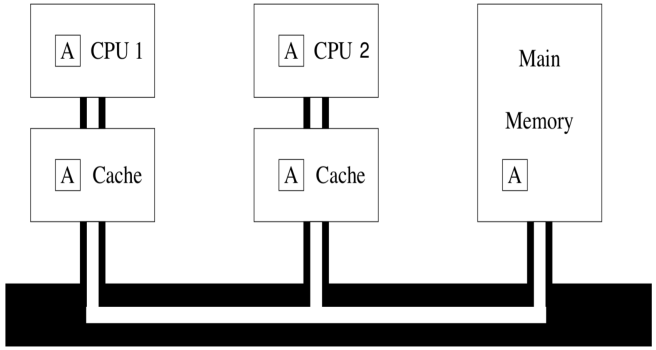


- We have variable `A` stored in main memory at address `&A`; it's initial value is $A = 0$.
- We assume write-back cache policy. **Write-back:** In this policy, only the cache location is updated during a write operation.

time	Action	P1$	P2$	P3$	P4$	Mem[&A]
t_0						0
t_1	P1 loads A	0 miss				0
t_2	P2 loads A	0	0 miss			0

t1: Processor P1$ loads data A into its cache. This is a miss because the data A was not initially present in the cache of P1$.
t2: Processor P2$ also loads data A into its cache. This is also a miss because the data A was not initially present in the cache of P2$.

Parallel caching

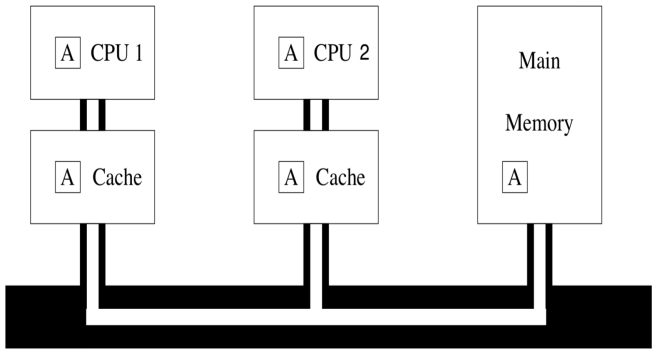


- We have variable `A` stored in main memory at address `&A`; it's initial value is $A = 0$.
- We assume write-back cache policy.

time	Action	P1$	P2$	P3$	P4$	Mem[&A]
t_0						0
t_1	P1 loads A	0 miss				0
t_2	P2 loads A	0	0 miss			0
t_3	P1 stores A	1	0			0
t_4	P3 loads A	1	0	0 miss		0

t3: Processor P1$ stores data A. The value in the cache of P1$ is updated to 1, indicating that the data A has been modified.
t4: Processor P3$ loads data A into its cache. This is a miss because the data A was not initially present in the cache of P3$.

Parallel caching

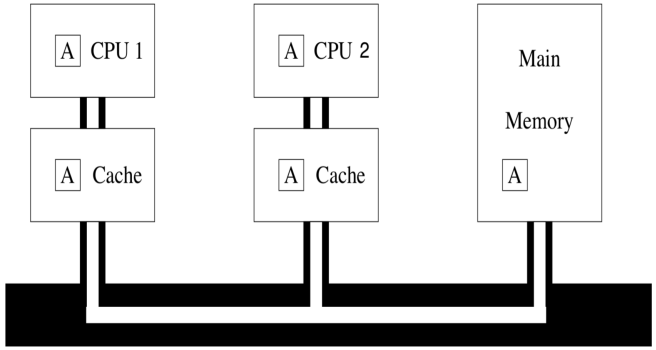


- We have variable `A` stored in main memory at address `&A`; it's initial value is $A = 0$.
- We assume write-back cache policy.

time	Action	P1$	P2$	P3$	P4$	Mem[&A]
t_0						0
t_1	P1 loads A	0 miss				0
t_2	P2 loads A	0	0 miss			0
t_3	P1 stores A	1	0			0
t_4	P3 loads A	1	0	0 miss		0
t_5	P3 stores A	1	0	2		0
t_6	P2 loads A	1	0 hit	2		0

t5: Processor P3$ stores data A. The value in the cache of P3$ is updated to 2, indicating that the data A has been modified.
t6: Processor P2$ loads data A. This is a hit because the data A is already present in the cache of P2$.

Parallel caching

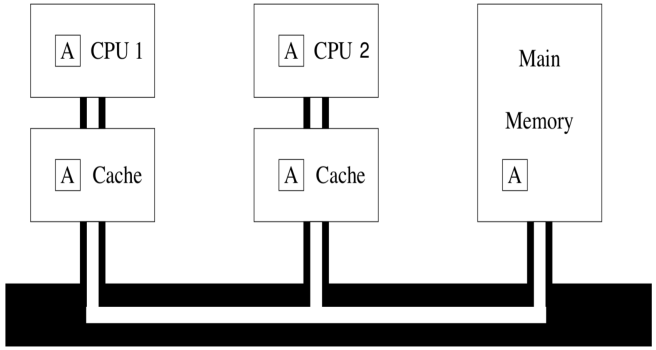


- We have variable `A` stored in main memory at address `&A`; it's initial value is $A = 0$.
- We assume write-back cache policy.

time	Action	P1$	P2$	P3$	P4$	Mem[&A]
t_0						0
t_1	P1 loads A	0 miss				0
t_2	P2 loads A	0	0 miss			0
t_3	P1 stores A	1	0			0
t_4	P3 loads A	1	0	0 miss		0
t_5	P3 stores A	1	0	2		0
t_6	P2 loads A	1	0 hit	2		0
t_7	P1 loads B (causes eviction of A)	B miss	0	2		1

t7: Processor P1$ loads data B. This is a miss because the data B was not initially present in the cache of P1$. The miss causes eviction of data A from the cache of P1$ to make room for data B.

Parallel caching



time	Action	P1$	P2$	P3$	P4$	Mem[&A]
t_0						0
t_1	P1 loads A	0 miss				0
t_2	P2 loads A	0	0 miss			0
t_3	P1 stores A	1	0			0
t_4	P3 loads A	1	0	0 miss		0
t_5	P3 stores A	1	0	2		0
t_6	P2 loads A	1	0 hit	2		0
t_7	P1 loads B (causes eviction of A)	B miss	0	2		1

- This situation is a result of the *cache coherence problem* in multiprocessor systems
- Each processor has its own private cache, and without a mechanism to ensure coherence, the view of memory can become inconsistent across processors

Can we eliminate this problem by using locks?

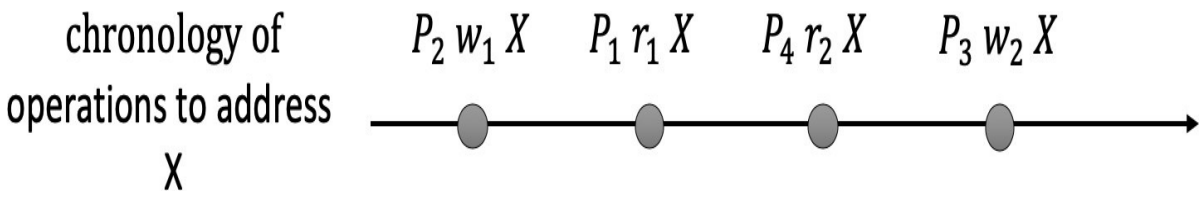
Memory coherence

- **A multiprocessor memory system is *coherent* if: -**
  - The results of any execution of a program for a given location *X* , are such that it is possible to construct a hypothetical serial order of all operations (all processors) to *X* that is consistent with the results of the execution and in which:
    - operations issued by any threads occur in the order in which they were issued to the memory system by that thread;
    - the value returned by each read operation is the value returned by the *last* write to that location in the serial order.
- **This definition guarantees two properties: -**
  - *write propagation*: writes become visible to other threads (note we are not specifying when);
  - *write serialization*: writes to a location (from the same or different threads) are seen in the same order by all threads.

Memory coherence

A multiprocessor memory system is *coherent* if: -

1. It preserves the program order: a read at *X* should provide the last value written to *X* by any processor.

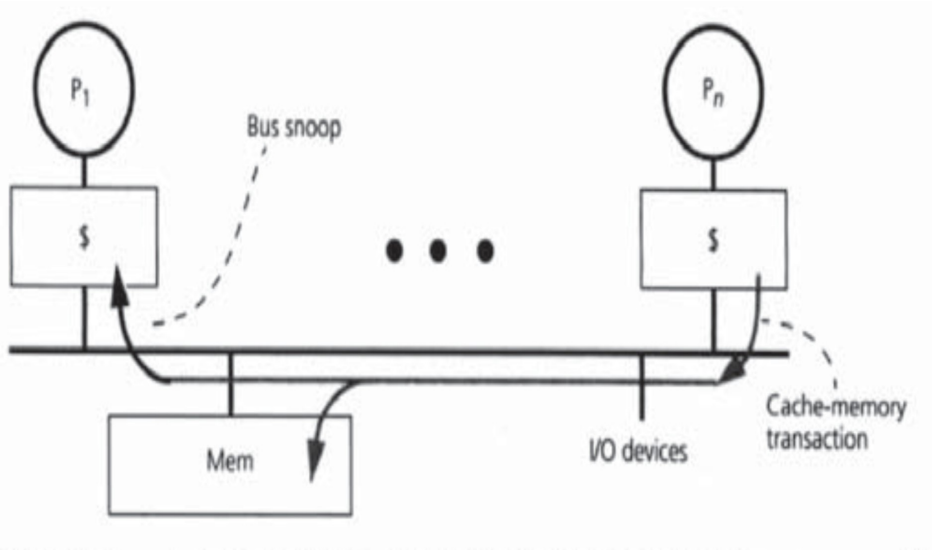


2. A write from P_1 to X will propagate to all other processors P_n, eventually.

3. Writes to the same address by different processors are serialized.

Cache Coherence through Bus Snooping

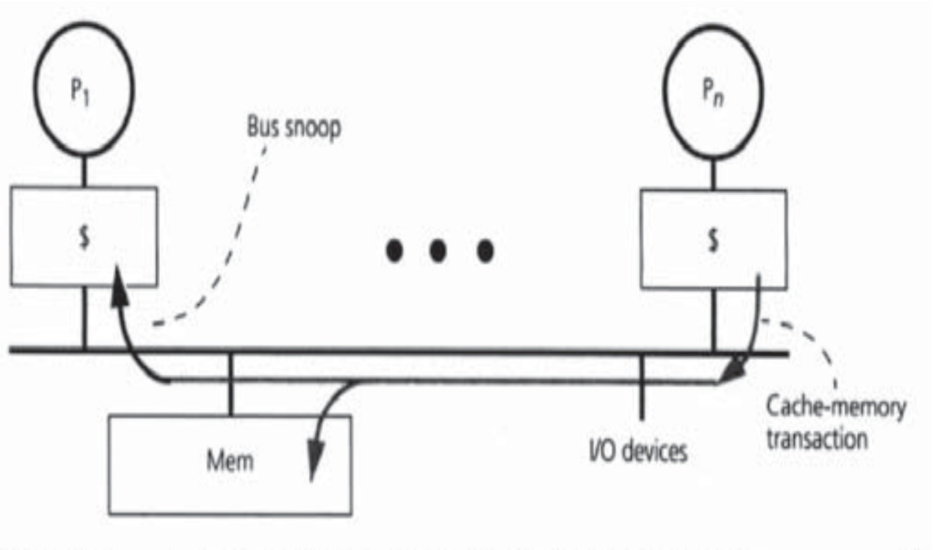
- Multiple processors with private caches (indicated with $) are placed on a shared bus.
- All coherence-related activity is broadcast to all processor caches through the shared bus.
- Each cache controller “snoops” on the bus watching for *relevant* transactions and updates its state suitably to keep its cache coherent.



Cache Coherence through Bus Snooping

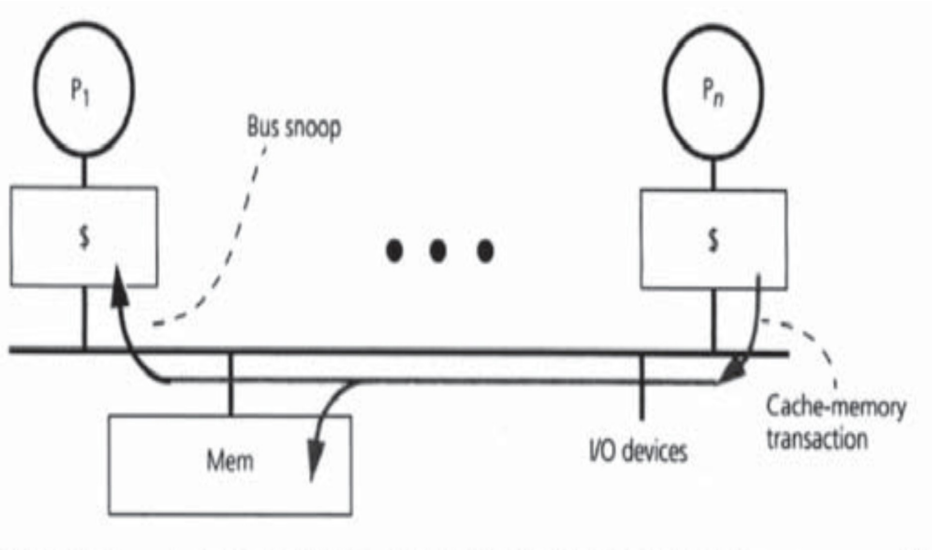
Multiple processors with private caches (indicated with $) are placed on a shared bus.

- Each cache controller "snoops" (monitors) on the bus watching for relevant transactions and updates its state suitably to keep its cache coherent.
- Here relevant transaction means that it involves a memory block of which it has a copy in its cache.
- For example, P_1 may take the action of invalidating or updating one of its cache lines if it sees a write from P_3 that maps to the same memory block.



Cache Coherence through Bus Snooping

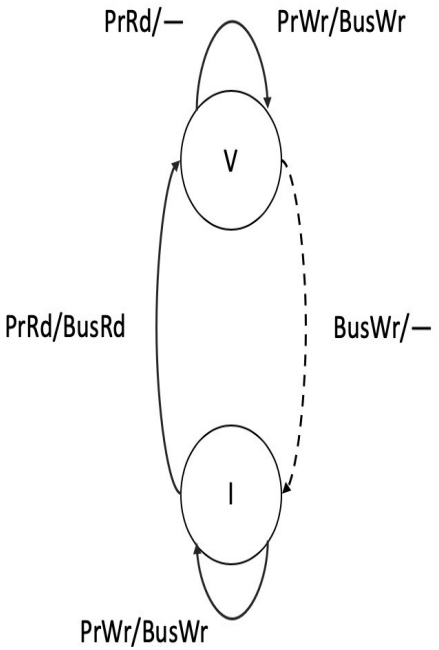
- Since cache lines are the granularity of allocation in the cache, they are also the granularity of coherence protocols.
- In order to support cache coherence a bus must be designed such that: -
  - All transactions that appear on the bus are visible to all cache controllers.
  - Transactions are visible to all controllers in the same order.



- A **cache coherence protocol** is a set of rules that guarantees that all necessary transactions appear on the bus, in response to memory operations, and that controllers take the appropriate actions in response.
- *This is implemented at the hardware level.*

The Write-Through Invalidation Protocol

- Coherence protocols are represented by a collection (state diagram) of state machines and associated transitions.
- By state here we mean the **state of a cache line**
- Actions are denoted as combinations like "PrRd/BusRd" to represent a processor read leading to a bus read action.



Assumptions

- Write-through, write-no-allocate cache
- Bus is atomic
- Memory operations are atomic
- Proc waits until previous memory operation is complete before issuing a new one
- Invalidation happens during bus transaction (as part of the invalidation broadcast)

A/B: if A is observed, then B is generated

——→ Processor-initiated

- - - - - - - - → Bus-snooper-initiated

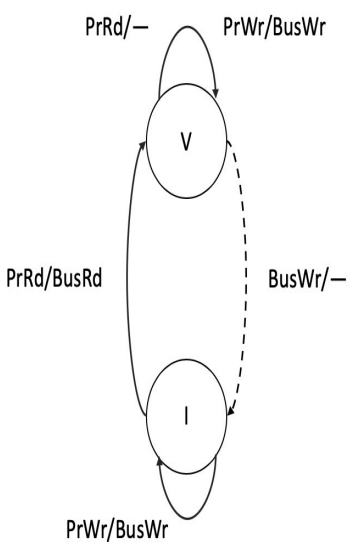
The Write-Through Invalidation Protocol

States:

V (Valid): This state indicates that the data in the cache is valid and matches the corresponding data in the main memory.
I (Invalid): This state indicates that the data in the cache is invalid, either because it doesn't match the corresponding data in the main memory or because it has been explicitly invalidated.

Operations (*action/response*):

PrRd/—: This represents a processor read operation. If the data is in the Valid state, the read operation is a hit and the data is fetched from the cache. If the data is in the Invalid state, the read operation is a miss and the data is fetched from the main memory.
PrWr/BusWr: This represents a processor write operation. In a write-through, write-no-allocate cache, the data is written to both the cache (if it's in the Valid state) and the main memory.
PrRd/BusRd: This represents a processor read operation that results in a miss (because the data is in the Invalid state). The data is fetched from the main memory and the cache state transitions to Valid.
BusWr/I—: This represents a write operation initiated by another processor (or device) on the bus. The cache controller snoops the bus, detects that the write operation affects the data in the cache, and invalidates the cache line



Assumptions

- Write-through, write-no-allocate cache
- Bus is atomic
- Memory operations are atomic
- Proc waits until previous memory operation is complete before issuing a new one
- Invalidation happens during bus transaction (as part of the invalidation broadcast)

A/B: if A is observed, then B is generated

——→ Processor-initiated

- - - - - - - - → Bus-snooper-initiated

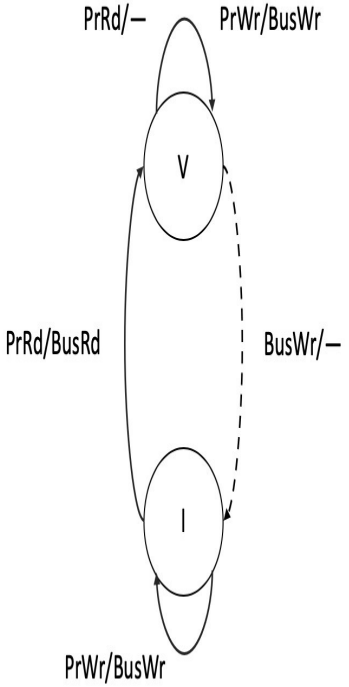
A Write-Through Invalidation Example

Write-through cache : This policy writes data to the cache and the main memory at the same time.
+ Write-no-allocate cache: This policy bypasses cache entry allocation *in case of a cache miss*.

t1: Processor P1 is reading data, but there is a miss in the cache (C1). A BusRd action is initiated to read the data from memory. The state of C1 changes to 'V' (valid), while P2 and C2 remain in the 'I' (invalid) state. The memory content remains unchanged.

t2: Processor P2 also attempts to read data and encounters a miss in the cache (C2). Another BusRd action occurs. Now, both C1 and C2 are in the 'V' state, indicating that they have valid copies of the data. The memory content still remains unchanged.

time	Action	P1	P2	C1 state	C2 state	Mem[A]
t_0				I	I	0
t_1	P1Rd A→ BusRd	0 miss		V	I	0
t_2	P2Rd A→ BusRd	0	0 miss	V	V	0
t_3	P1Wr A→ BusWr	1				
t_5						
t_5						



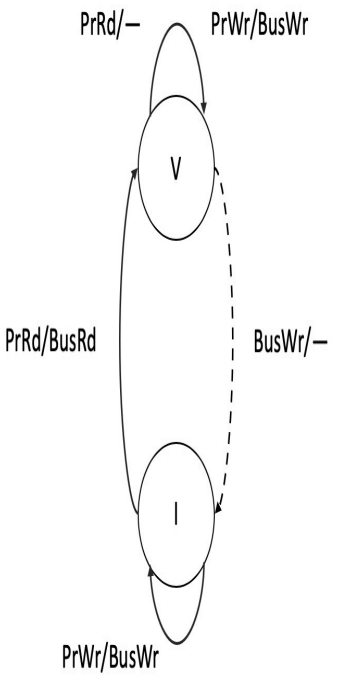
A Write-Through Invalidation Example

Write-through, write-no-allocate cache

t3: Processor P1 writes data, changing its cache state to modified ('M') and invalidating C2's copy of the data (changing its state to 'I'). A BusWr action updates the memory with new content.

t5: Processor P2 attempts another read but encounters a miss since its cache was invalidated at t3. It initiates a BusRd action to get updated data from memory

time	Action	P1	P2	C1 state	C2 state	Mem[&A]
t_0				I	I	0
t_1	P1Rd A→ BusRd	0 miss		V	I	0
t_2	P2Rd A→ BusRd	0	0 miss	V	V	0
t_3	P1Wr A→ BusWr	1	0	V	I	1
t_5	P2Rd A→ BusRd					
t_5						



The MSI Write-Back Invalidation Protocol

Are we happy with write-through caches?

The protocol uses three states to distinguish valid blocks that are unmodified (clean) from those that are modified (dirty):

- *Modified*: also called dirty or exclusive means that only this cache has a valid copy of the cache line, and the copy in main memory is stale.
- *Shared*: the cache line is presented in unmodified state in this cache, main memory is up-to-date, and zero or more caches may also have an up-to-date copy.
- *Invalid:* Not present *or* invalidated by a bus request.

The MSI Write-Back Invalidation Protocol

Key tasks of the protocol

- Ensuring processor obtains exclusive access for a write
- Locating most recent copy of cache line's data on a cache miss

Two processor operations(triggered by local CPU)

- PrRd(read)
- PrWr(write)

Three coherence-related bus transactions (from remote caches)

- BusRd: obtain copy of line with no intent to modify
- BusRdX: obtain copy of line with intent to modify
- BusWB: write dirty line out to memory

The MSI Write-Back Invalidation Protocol

The protocol uses three states to distinguish valid blocks that are unmodified (clean) from those that are modified (dirty):

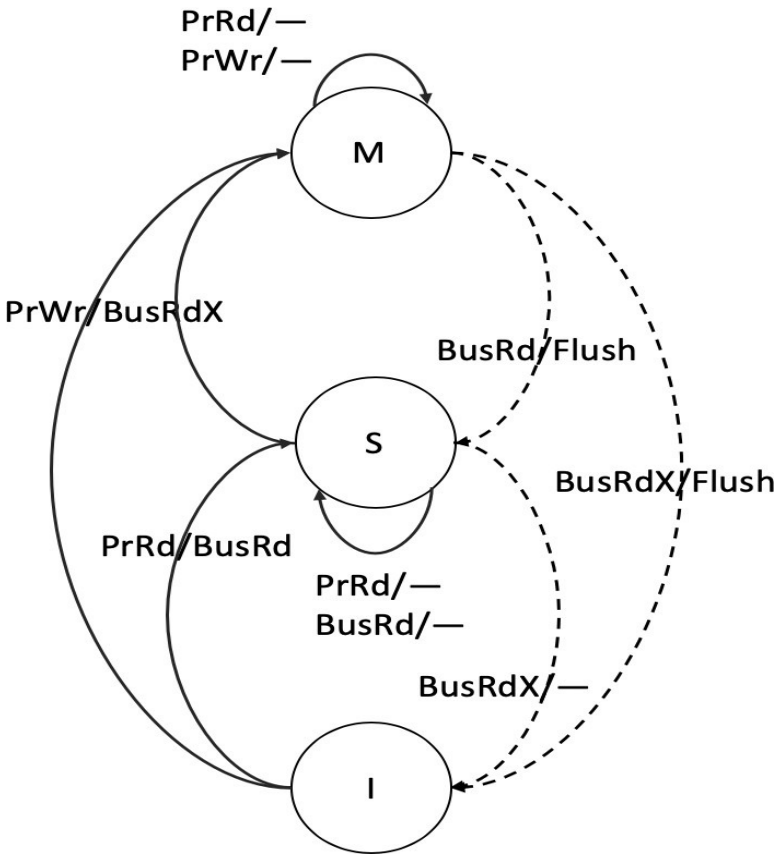
- *Modified*: also called dirty or exclusive, means that only this cache has a valid copy of the cache line, and the copy in main memory is stale.
- *Shared*: the cache line is presented in unmodified state in this cache, main memory is up-to-date, and zero or more caches may also have an up-to-date copy.
- *Invalid:* Not present or invalidated by a bus request.

Corollary rule: Before a shared or invalid copy can be written and placed in the modified state, all other potential copies must be invalidated via a *read-exclusive* bus transaction.

Why do we need this rule?

The MSI Write-Back Invalidation Protocol

- *BusRd*: (Bus Read) The cache controller asks for a copy (cache line) that it does not intend to modify.
- *BusRdX*: (Bus Read exclusive) The cache controller asks for an *exclusive copy that it intends to modify*. The memory system supplies the data. All other caches are invalidated.
- *BusRdX/Flush*: (also known as *BusWB = Bus writeback)* The processor does not know about it and does not expect a response. The main memory is updated with the latest content.



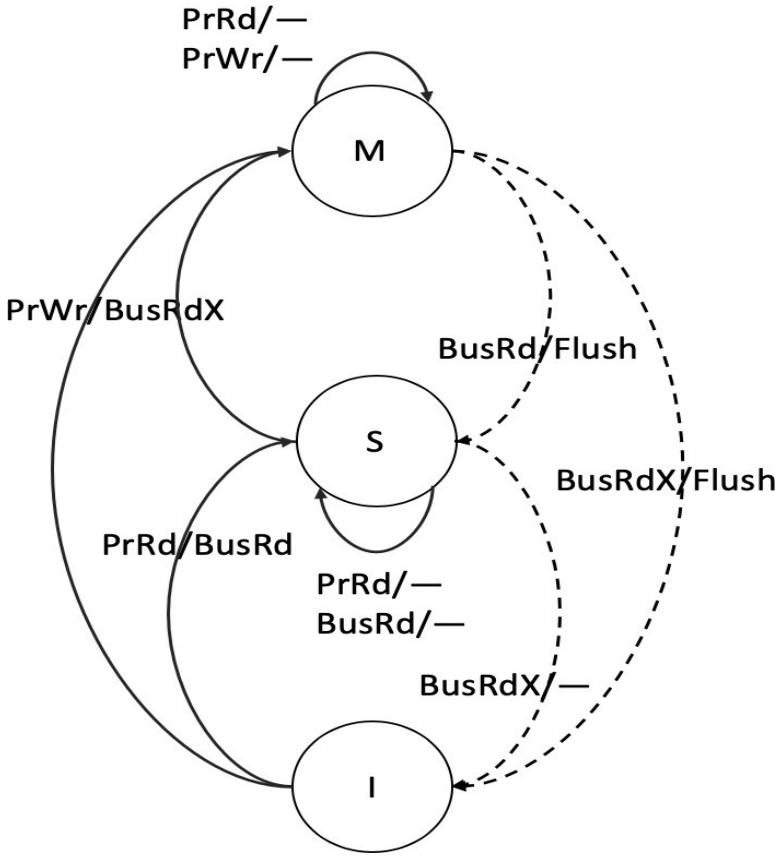
A / B: if action A is observed by cache controller, action B is taken

MSI Write-Back Invalidation Protocol

Do you think we can improve this protocol?

MSI requires *two* interconnect transactions for the common case of reading an address, then writing to it: -

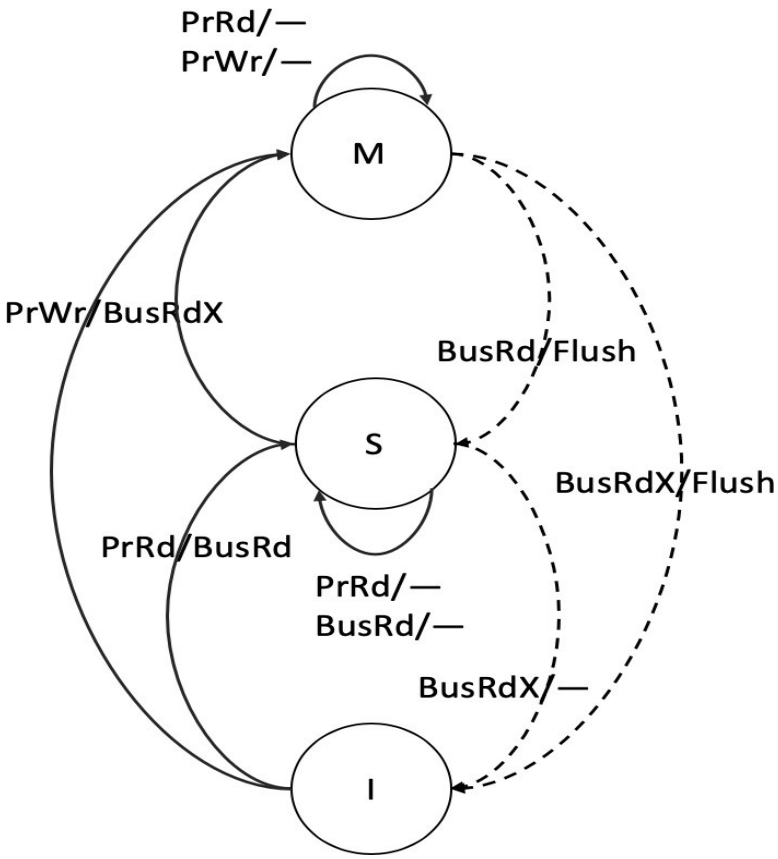
- Assume that we perform a PrRd/BusRd and only the local processor has a copy of that memory block (cache line).
  - BusRd to move from I to S state
- Then assume that the processor wants to write (PrWr): do we need to broadcast this information to any of the remote cache controllers?
  - BusRdX to move from S to M state in the local cache, indicating that this cache now has the most recent copy of the memory block.
- We need to broadcast to the other processors as they might have a shared copy of the same memory block, and we need to ensure that they invalidate their copies to maintain cache coherence in the local cache, indicating that this cache now has the most recent copy of the memory block.



A / B: if action A is observed by cache controller, action B is taken

An MSI Example

time	Action	P1	P2	P3	C1 state	C2 state	C3 state	Mem[&A]
t_1	P1Rd A→ BusRd	[0] miss	–		S	–	–	0
t_2	P3Rd A→ BusRd	0	–	[0] miss	S	–	S	0
t_3	P3Wr A→ BusRdX	0	–	1	I	–	M	0
t_5	P1Rd A→ BusRd	[1] miss	–	1	S	–	S	1
t_5	P2Rd A→ BusRd	1	[1] miss	1	S	S	S	1



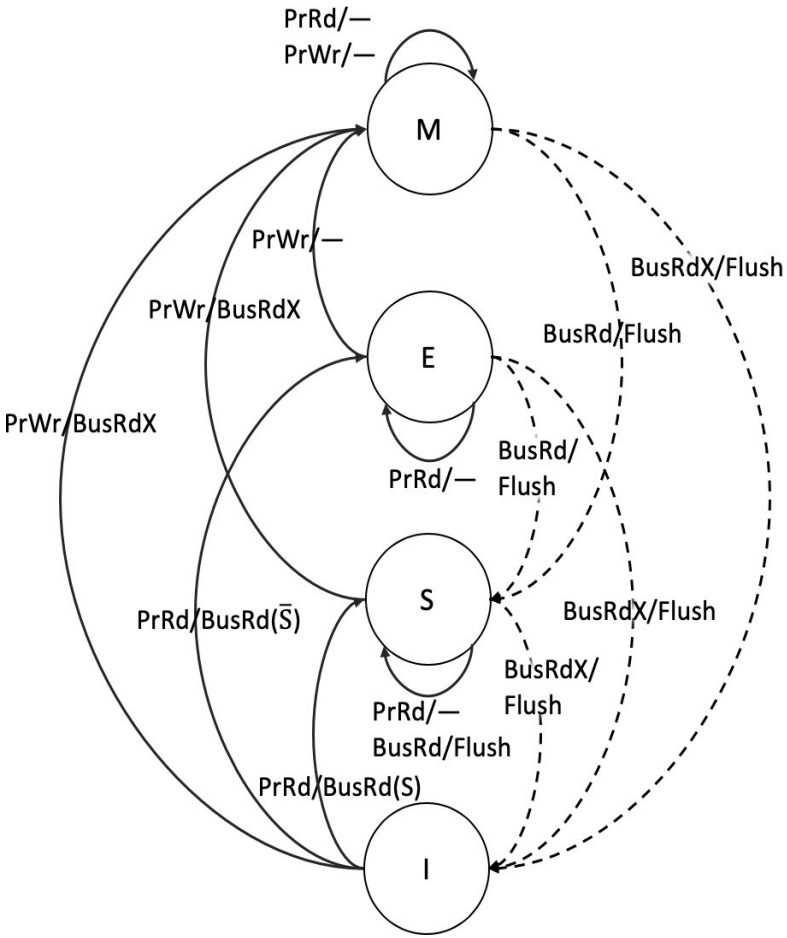
A / B: if action A is observed by cache controller, action B is taken

The MESI Protocol

M and I have the same semantics as before.

***Exclusive* (E)** or exclusive-clean state means that only one cache (this cache) has a copy of the cache line and it has not been modified (main memory is up-to-date).

***Shared* (S)** now means that potentially two or more processors have this block in their cache in an unmodified state.



A / B: if action A is observed by cache controller, action B is taken

More Complex Snooping Protocols

Modern multiprocessors tend to implement slightly more complex protocols than MESI

- 5-stage MOESI, MESIF, which for example include the possibility of cache-to-cache transfers.

- Directory-based cache-coherence protocols that significantly reduce the overhead and serialization associated with bus transactions.
  - Instead of broadcasting coherence traffic to all L2's, only send coherence messages to L2's that contain the line

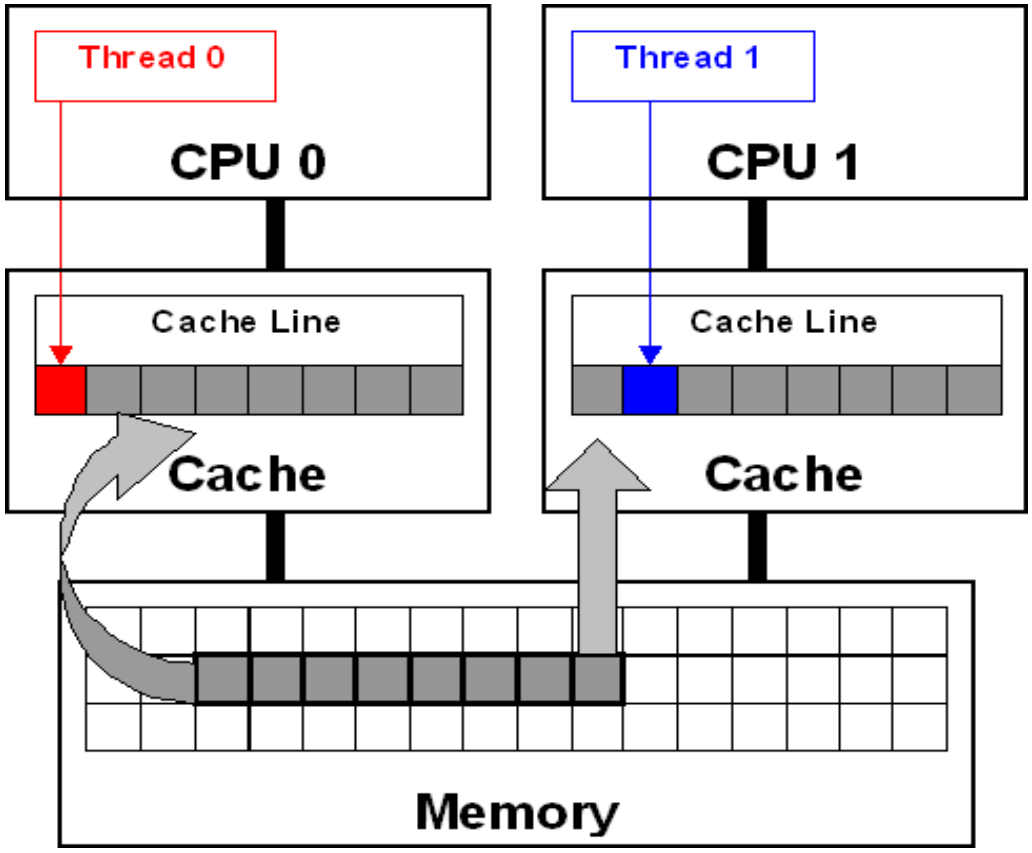
Flaws of Cache Coherence Protocols (so far)

All modern multiprocessors (specifically CPUs, *GPUs do not*) implement cache coherence. What are the drawbacks?

- More complicated caches and interconnects (*e.g.* bus).
- Increased bus traffic (what are the implication? Remember Amdahl's law ...)
  - This can significantly impinge on performance, especially for large core counts.
- Cache line thrashing via false sharing.

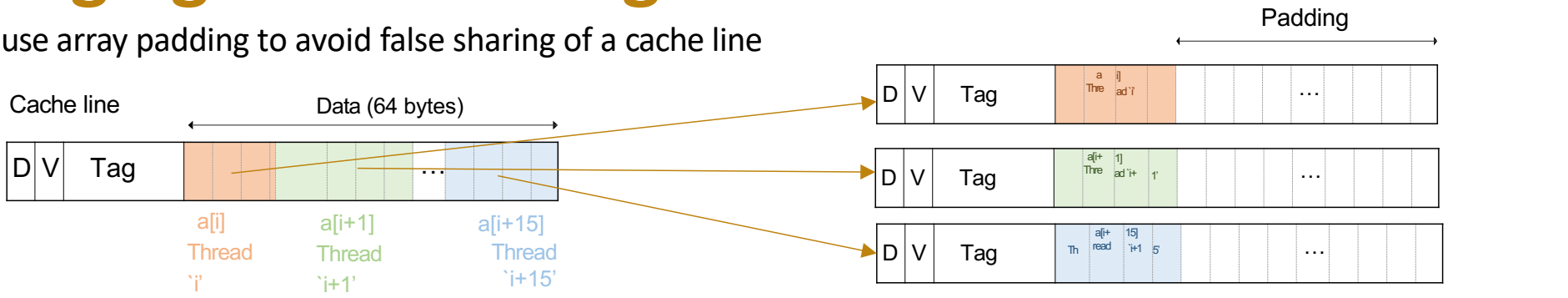
False Sharing

- Condition where two processors write to different addresses, but addresses map to the same cache line
- Cache line "ping-pongs" between caches of writing processors, generating significant communication due to coherence protocol (high serialization overhead)
- No inherent communication, **this is entirely artifactual communication (cachelines > 4B)**
- False sharing can severely affect parallel performance.



Managing False Sharing

You can use array padding to avoid false sharing of a cache line




```
float a[ NTHREADS ] = { 0.0 };
#pragma omp parallel for
for ( int i = 0; i < NTHREADS ; i++ ) {
  for ( int j = 0; j < BigN ; j++ ) {
    a[ i] += ( float ) rand ( );
  }
}
```

```
size_t FL_CLINE_WORDS = 64/ sizeof ( float );
float a[ NTHREADS ][ FL_CLINE_WORDS ] = { 0.0 };
#pragma omp parallel for
for ( int i = 0; i < NTHREADS ; i++ ) {
 for ( int j = 0; j < BigN ; j++ ) {
   a[ i ][0] += ( float ) rand ( );
 }
}
```

- `FL_CLINE_WORDS` gives the number of floats that can fit into a cache line.
- The 2D array `a[NTHREADS][FL_CLINE_WORDS]` ensures that each row of the array can fit into a single cache line
- The `#pragma omp parallel` for directive is used to parallelize the outer loop. This means that the iterations of the outer loop will be distributed among the available threads.