A Primer on Modern Hardware Architectures

Alessandro Pellegrini

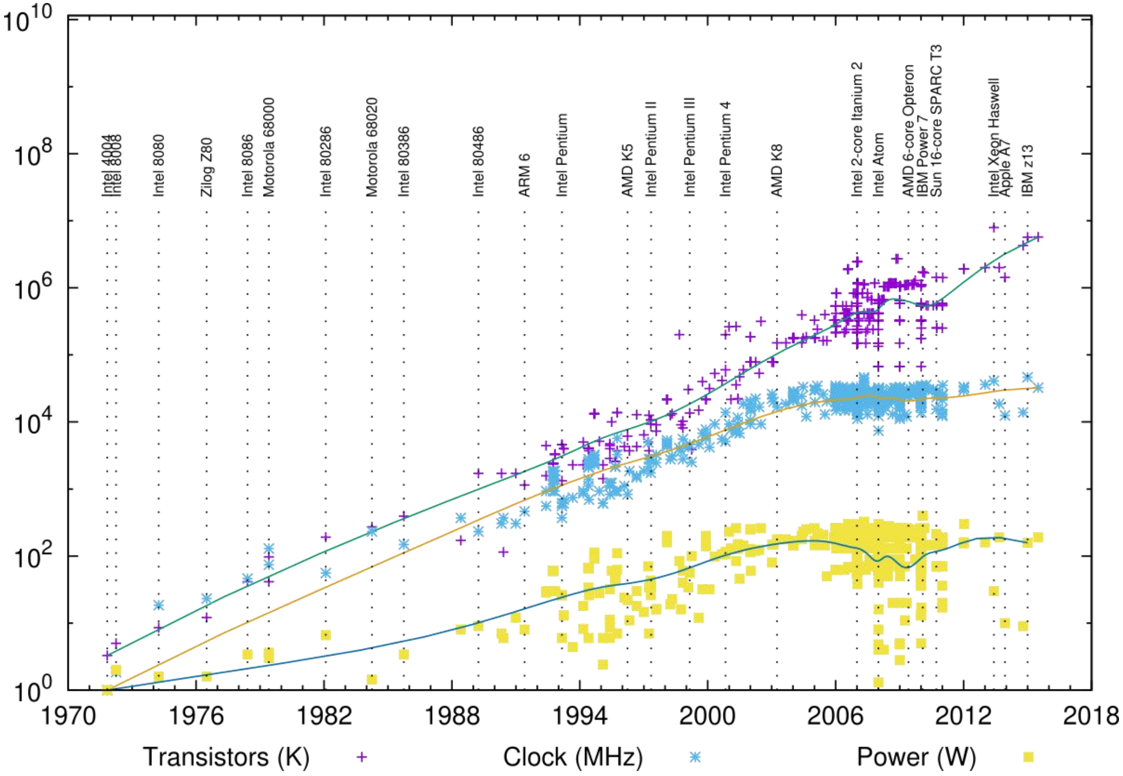
A.Y. 2019/2020

Moore's Law (1965)

The number of transistors in a dense integrated circuit doubles approximately every two years

— Gordon Moore, Co-founder of Intel

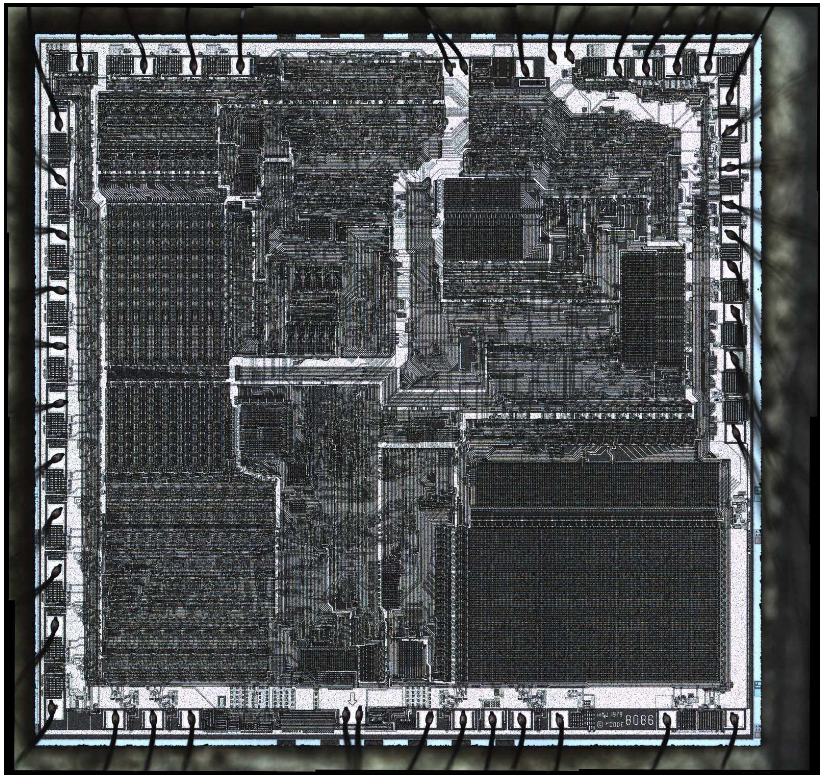
Effects of this Technological Trend



- Implications of
- Moore’s Law have changed since 2003
- 130W is considered an upper bound (the *power wall*)

$$P=ACV^2f$$

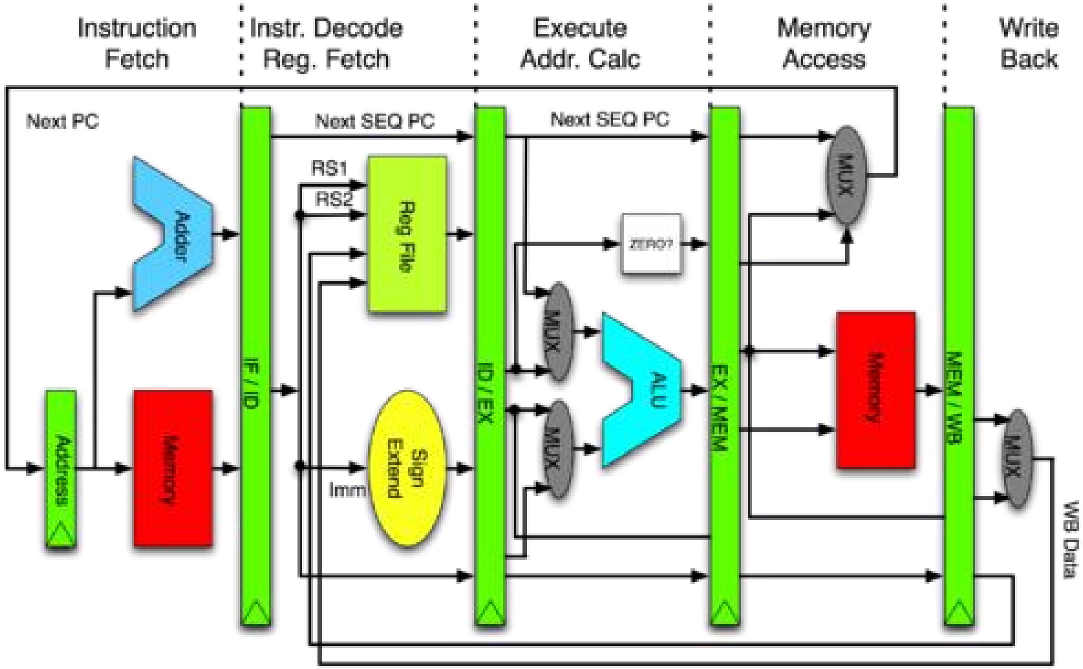
Single Cores



- Multicycle single core CPUs were already complex...
- ...but they were slow!

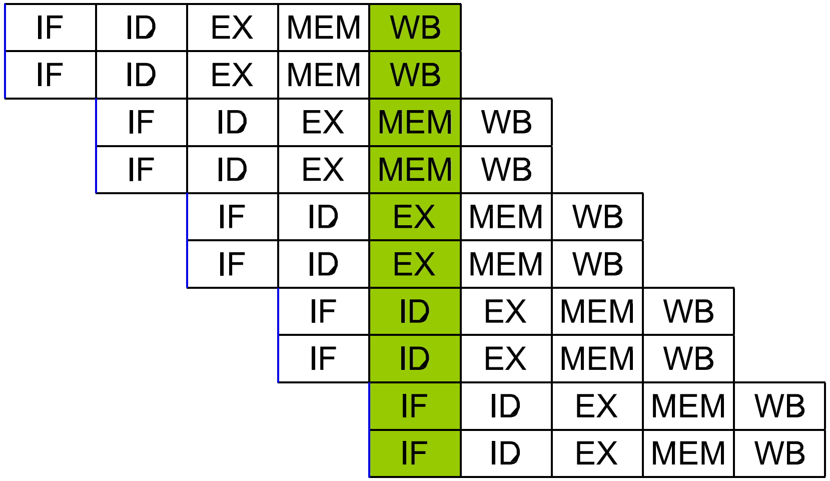
Trying to speedup: the pipeline (1980s)

- Temporal parallelism
- Number of stages increases with each generation
- Maximum Cycles Per Instructions (CPI)=1



Superascalar Architecture (1990s)

- More instructions are simultaneously executed on the same CPU
- There are redundant functional units that can operate in parallel
- Run-time scheduling (in contrast to compile-time)



IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

Speculation

- In what stage does the CPU fetch the next instruction?
- If the instruction is a conditional branch, when does the CPU know whether the branch is taken or not?
- *Stalling* has a cost: *nCycles · branchFrequency*
- A guess on the outcome of a compare is made
  - if wrong the result is discarded
  - if right the result is flushed

```
$a \leftarrow b + c$
if $a \geq 0$ then
    $d \leftarrow b$
else
    $d \leftarrow c$
end if
```

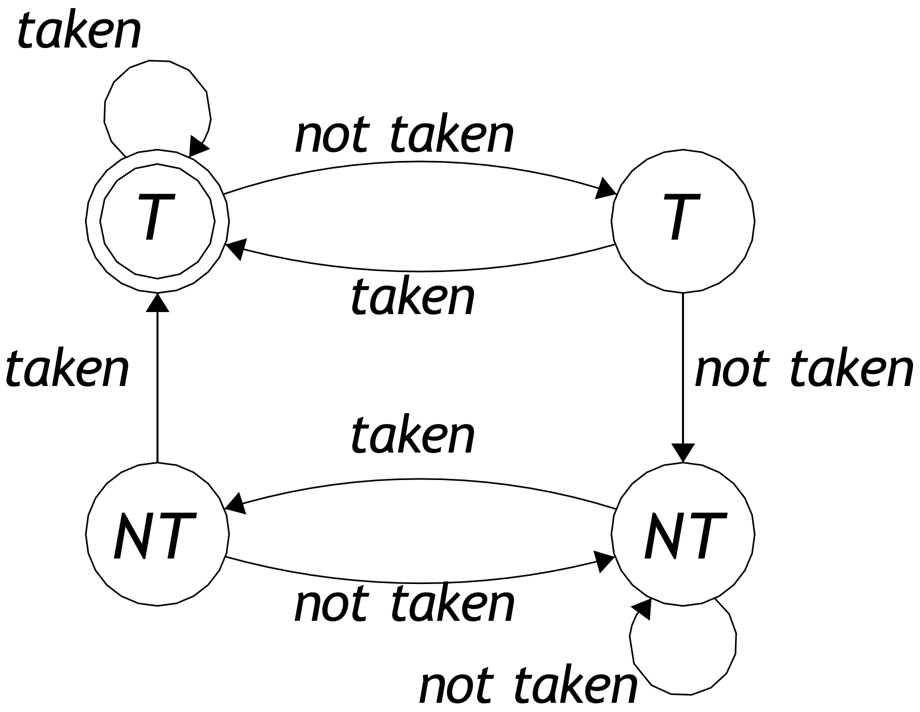
Branch Prediction

- Performance improvement depends on:
  - whether the prediction is correct
  - how soon you can check the prediction
- Dynamic branch prediction
  - the prediction changes as the program behaviour changes
  - implemented in hardware
  - commonly based on branch history
    - predict the branch as taken is it was taken previously
- Static branch prediction
  - compiler-determined
  - user-assisted (e.g., `likely` in kernel's source code; `0x2e`, `0x3e` prefixes for Pentium 4)

Branch Prediction Table

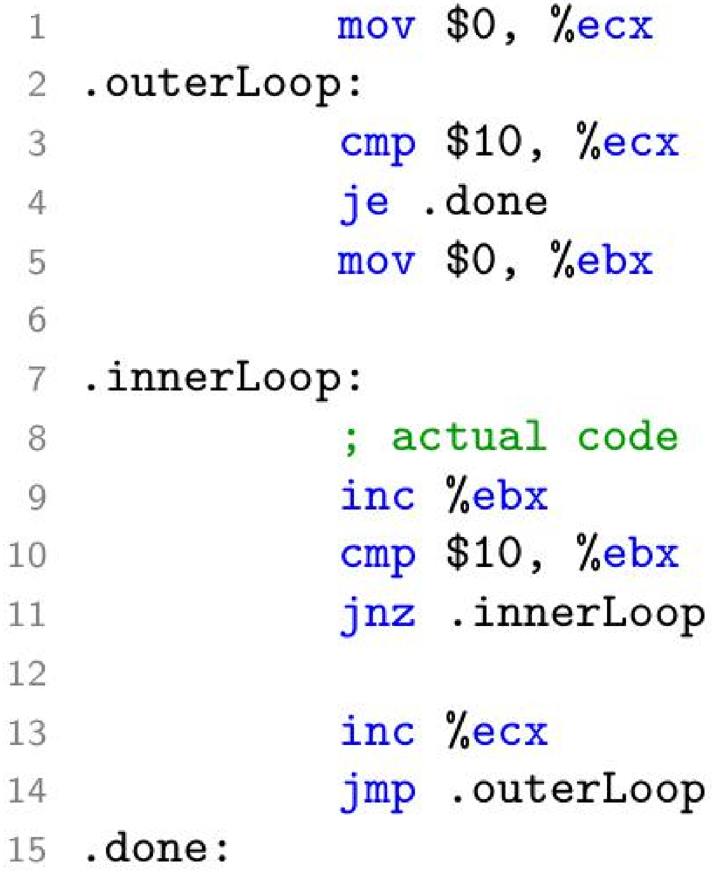
- Small memory indexed by the lower bits of the address of conditional branch instruction
- Each instruction is associated with a prediction
  - *Take* or *not take* the branch
- If the prediction is *take* and it is correct:
  - Only one cycle penalty
- If the prediction is *not take* and it is correct:
  - No penalty
- If the prediction is *incorrect*:
  - Change the prediction
  - Flush the pipeline
  - Penalty is the same as if there were no branch prediction

Two-bit Saturating Counter



- How well does it work with nested loops?

A Nested Loop Example

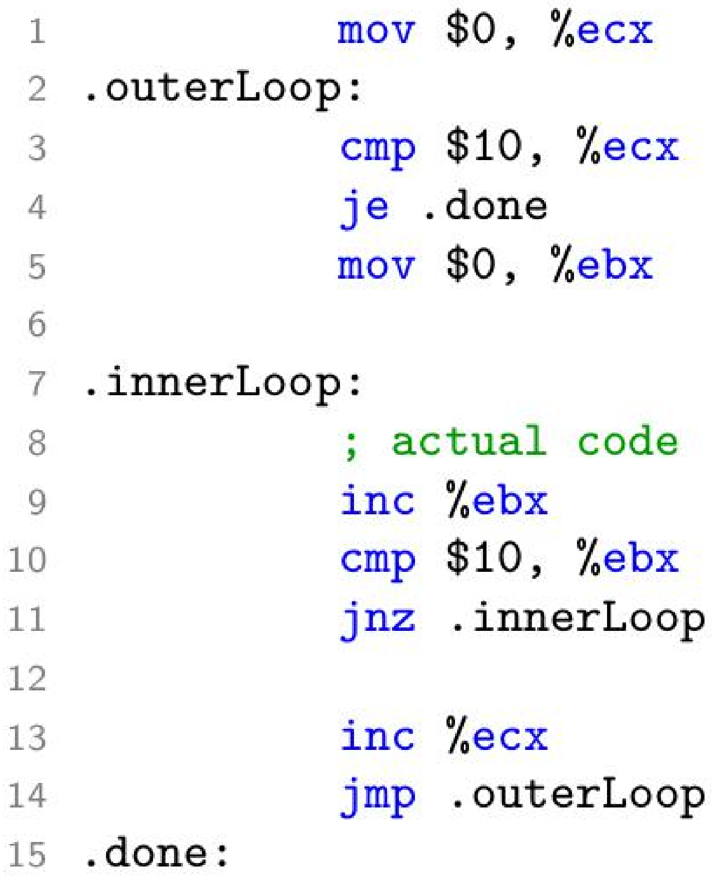


```
          mov $0, %ecx
.outerLoop:
          cmp $10, %ecx
          je .done
          mov $0, %ebx

.innerLoop:
          ; actual code
          inc %ebx
          cmp $10, %ebx
          jnz .innerLoop

          inc %ecx
          jmp .outerLoop
.done:
```

A Nested Loop Example



```
            mov $0, %ecx
.outerLoop:
            cmp $10, %ecx
            je .done
            mov $0, %ebx

.innerLoop:
            ; actual code
            inc %ebx
            cmp $10, %ebx
            jnz .innerLoop

            inc %ecx
            jmp .outerLoop
.done:
```

- 2-bit saturating counters do not work well with nested loops
- It could mispredict the first and last iteration due to the inner loop

Branch Prediction is Important

- Conditional branches are around 20% of the instructions in the code
- Pipelines are deeper
  - A greater misprediction penalty
- Superscalar architectures execute more instructions at once
  - The probability of finding a branch in the pipeline is higher
- Object-oriented programming
  - Inheritance adds more branches which are harder to predict

- Two-bits prediction is not enough
  - Chips are denser: more sophisticated hardware solutions could be put in place

How to Improve Branch Prediction?

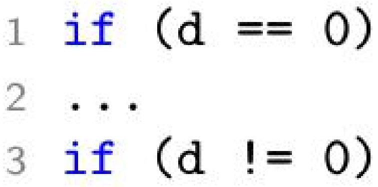
- Improve the prediction
  - Correlated (two-levels) predictors [Pentium]
  - Hybrid local/global predictor [Alpha]

- Determine the target earlier
  - Branch target buffer [Pentium, Itanium]
  - Next address in instruction cache [Alpha, UltraSPARC]
  - Return address stack [Consolidated into all architecture]

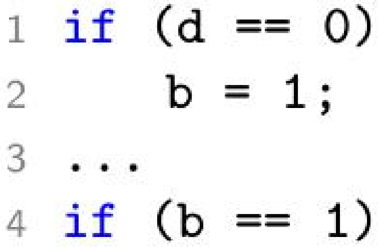
- Reduce misprediction penalty
  - Fetch both instruction streams [IBM mainframes]

Correlated Predictor

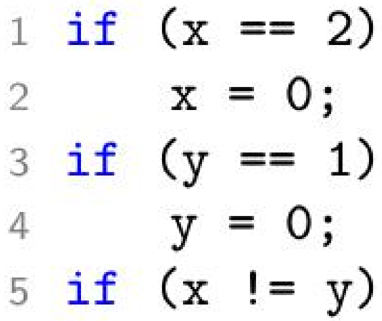
- Predictions cannot depend on only one branch
- Some branch outcomes are correlated



```
if (d == 0)
...
if (d != 0)
```



```
if (d == 0)
    b = 1;
...
if (b == 1)
```

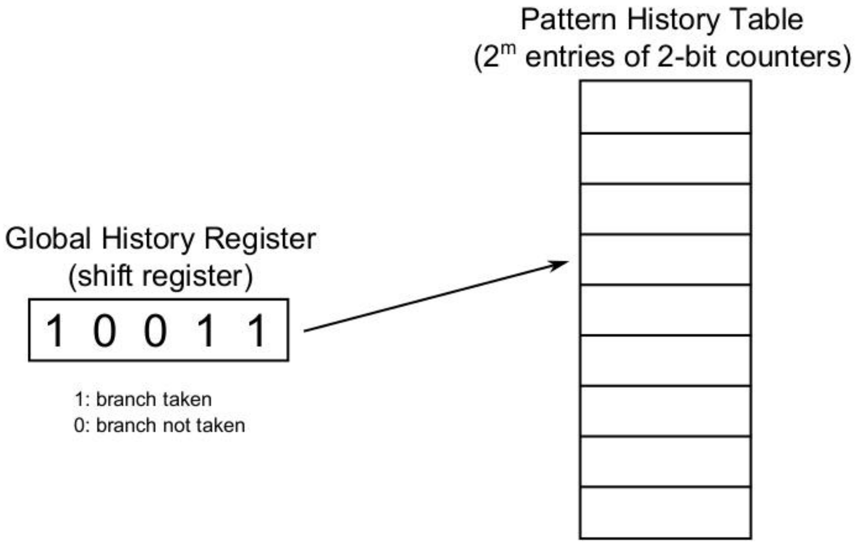


```
if (x == 2)
    x = 0;
if (y == 1)
    y = 0;
if (x != y)
```

- Use a *history* of past *m* branches, representing a path through the program

Pattern History Table (PHT)

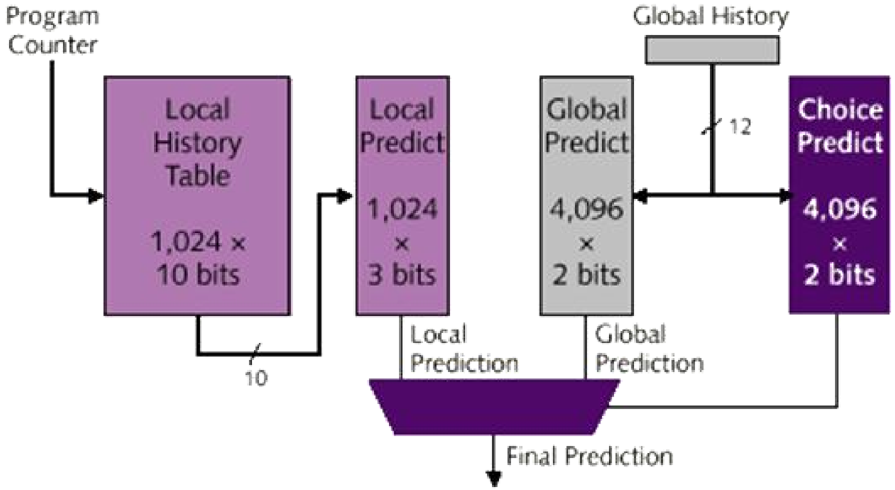
- Put the global branch history in a global history register
- Use its value to access a PHT of 2-bit saturating counters



Tournament Predictor

- Combines different branch predictors
- A *local predictor*, selected using the branch address
- A *correlated predictor*, based on the last *m* branches, accessed by the local history
- An indicator of which has been the best predictor for this branch
    - A 2-bit counter, which is increased for one and decreased for the other

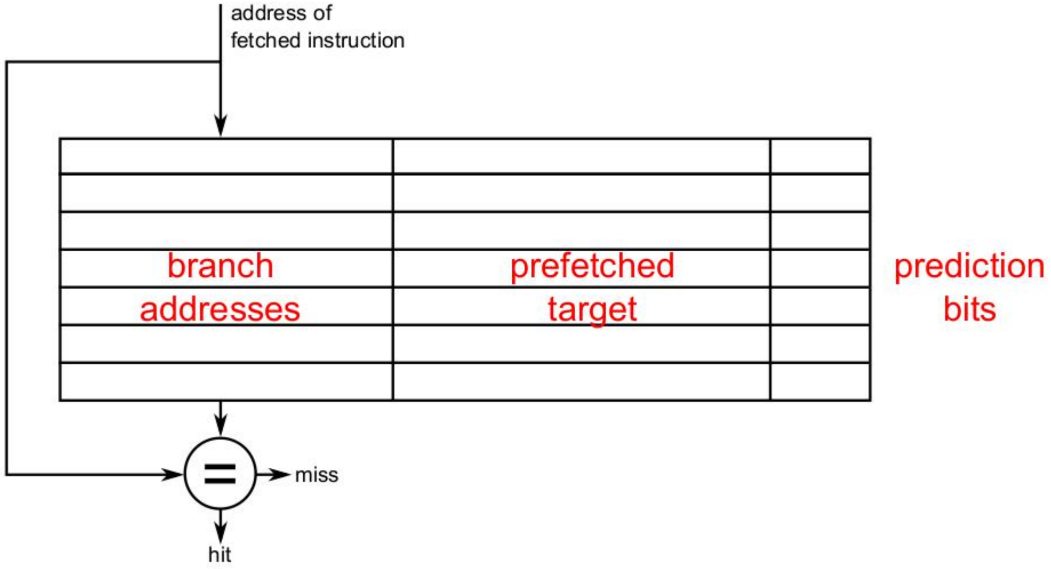
DEC Alpha 21264 BPU



- Tournament Branch Prediction Algorithm
- 35Kb of prediction information
- 2% of total die size
- Claim 0.7-1.0% misprediction

Branch Target Buffer

- Indirect jumps are hard to predict (e.g., `jmp * rax`, `ret`)
- It is a small cache memory, accessed via PC during the fetch phase
- Prediction bits are coupled with prediction target



Return Address Stack

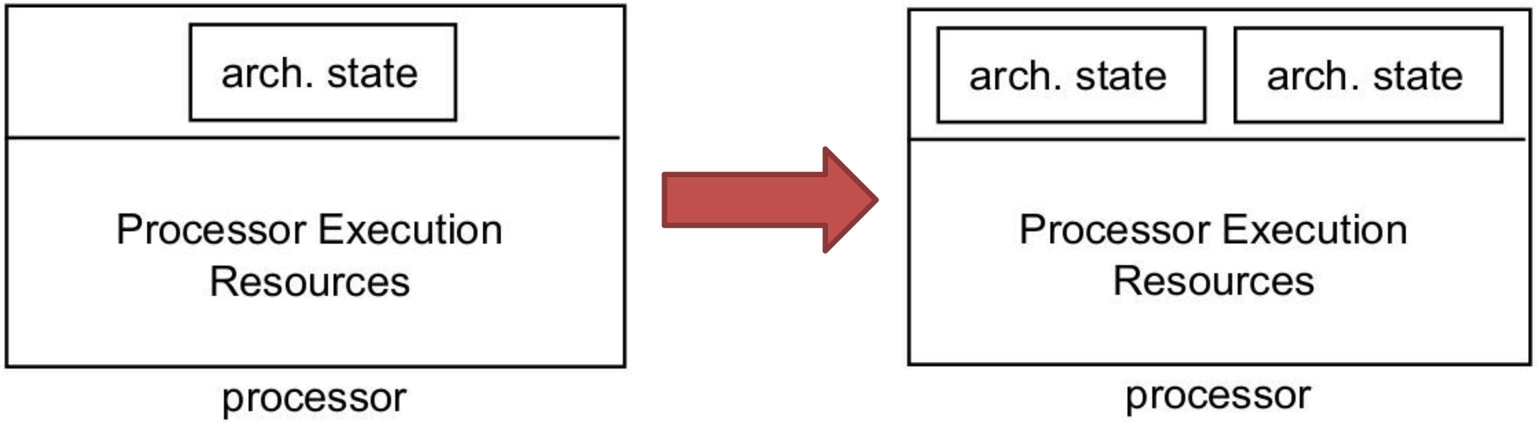
- Registers are accessed several stages after instruction’s fetch
- Most of indirect jumps (85%) are function-call returns
- Return address stack:
  - it provides the return address early
  - this is pushed on call, popped on return
  - works great for procedures that are called from multiple sites
    - BTB would would predict the address of the return from the last call

Fetch Both Targets

- This technique fetches both targets of the branch
- Does not help in case of multiple targets (e.g., `switch` statements)
- Reduces the misprediction penalty
- Requires a lot of I-cache bandwidth

Simultaneous Multi-Threading—SMT (2000s)

- A physical processor appears as multiple logical processors
- There is a copy of the architecture state (e.g., control registers) for each logical processor
- A single set of physical execution resources is shared among logical processors
- Requires less hardware, but some sort of arbitration is mandatory



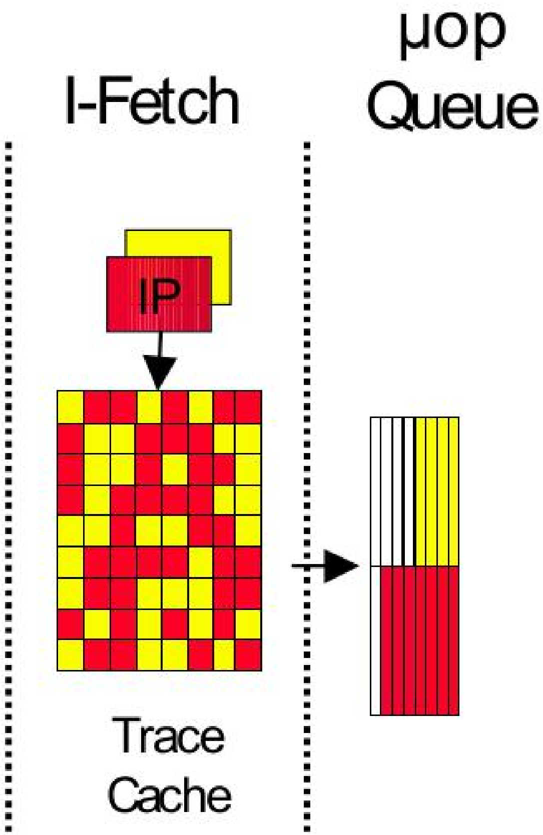
The Intel case: Hyper-Threading on Xeon CPUs

- **Goal 1**: minimize the die area cost (share of the majority of architecture resources)
- **Goal 2**: when one logical processor stalls, the other logical process can progress
- **Goal 3**: in case the feature is not needed, incur in no cost penalty
- The architecture is divided into two main parts:
  - Front end
  - Out-of-order execution engine

Xeon Front End

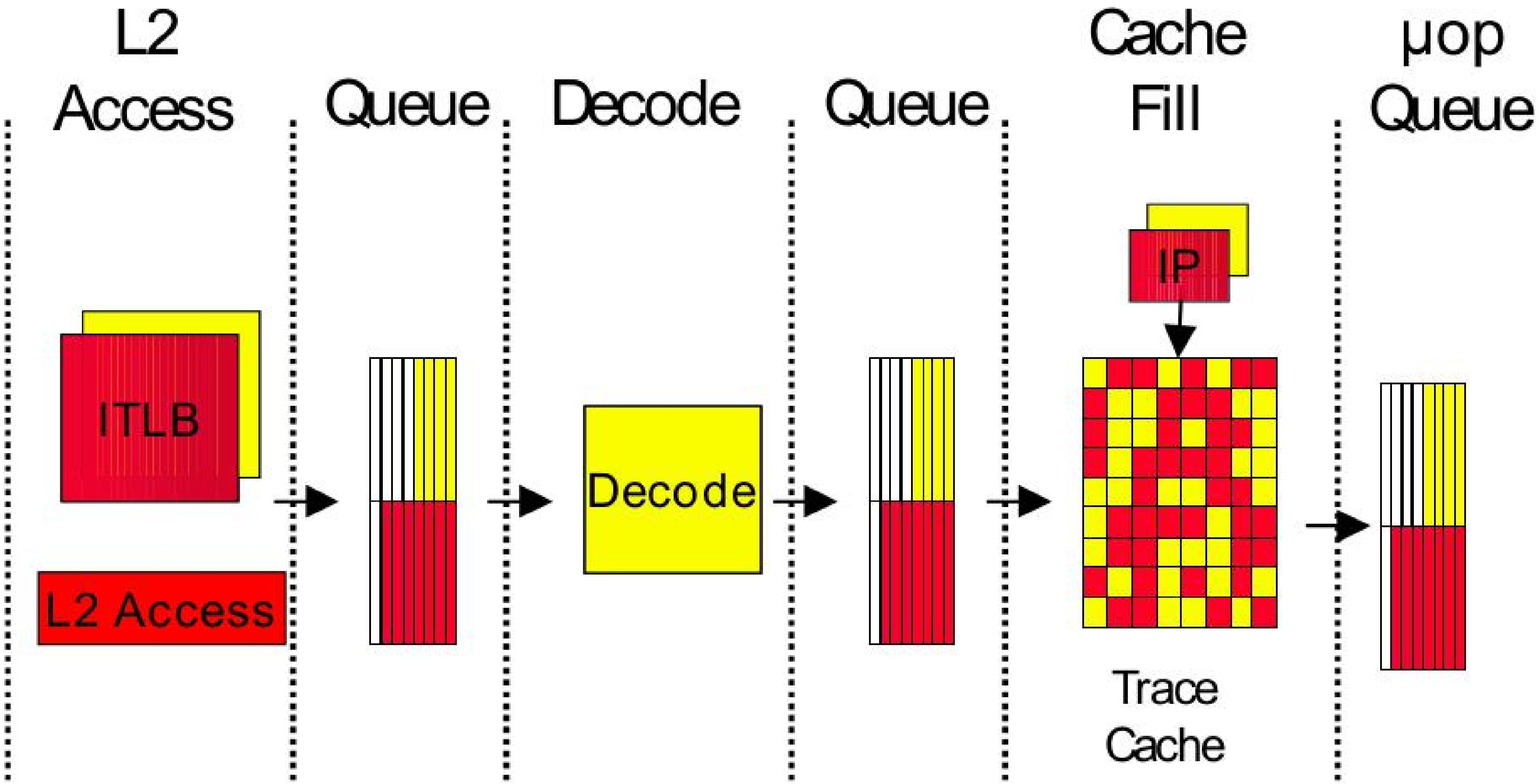
- The goal of this stage is to deliver instruction to later pipeline
- stages
- Actual Intel’s cores do not execute CISC instructions
  - Intel instructions are cumbersome to decode: variable length, many different options
  - A Microcode ROM decodes instructions and converts them into a set of semantically-equivalent RISC μ-ops
- μ-ops are cached into the Execution Trace Cache (TC)
- Most of the executed instructions come from the TC

Trace Cache Hit



- Two sets of next-instruction pointers
- Access to the TC is arbitrated among logical processors at each clock cycle
- In case of contention, access is alternated
- TC entries are tagged with thread information
- TC is 8-way associative, entries are replaced according to a LRU scheme

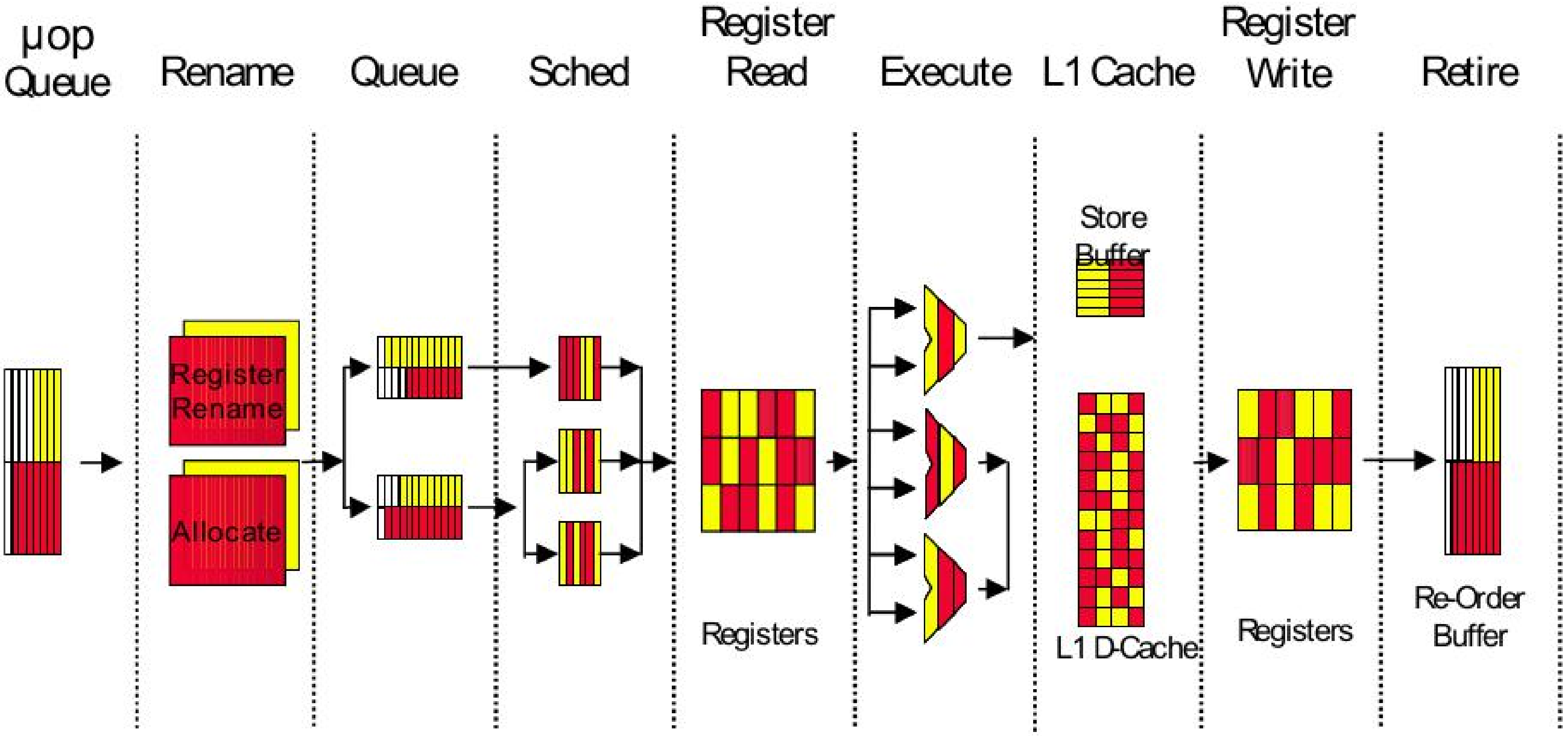
Trace Cache Miss



ITLB and Branch Prediction

- The *Instruction Tranlsation Lookaside Buffer* (ITLB) receives a request from TC to deliver new instructions
- The next-instruction pointer is translated to a physical address
  - A request is sent to L2 cache
  - Per-logical-processor access is arbitrated on a FIFO basis, with at least one slot for each logical processor
- Instruction bytes are stored in a 64-byte streaming buffer
- Duplicated branch-prediction structures:
  - Return stack buffer
  - Branch history register
- Shared branch-prediction structures:
  - Pattern history table, with per-logical processor entry tags

Xeon Out-of-order Pipeline



μ-op Queue

- This queue decouples the front end from the out-of-order execution engine
- Allows logical processors to make progress independently of front-end stalls
- μ-ops are executed as quickly as their inputs are ready
- The original program order is not taken into account

Allocator

- The allocator take μ-ops from the μ-op queue
- Its goal is to allocate machine buffers needed for μ-op execution:
  - 126 re-order buffer entries
  - 128 integer and 128 floating-point physical registers
  - 48 load and 24 store buffer entries
- Some of these buffers are partitioned among logical processors, to ensure progress and fairness
- The allocator alternates selecting μ-ops associated with logical processors at each clock cycle
- If partitioned resources are all used by a logical processor, it is stalled

Register Rename

- IA-32 has 8 general-purpose registers, while Intel Xeon has 128 physical registers
- Logical registers are expanded to use all the available physical registers
- A Register Alias Table (RAT) tracks the latest version of each IA-32 register to tell the next instruction where to find input operands
- There is one RAT for each logical processor
- After register rename, μ-ops are placed into two separate queues:
  - A memory-operation queue (for load/stores)
  - A queue for all other operations
- These queue are duplicated for each logical processor

Instruction Scheduling

- Five μ-op schedulers are used to schedule different types of μ-ops for the various execution units
- They can dispatch up to 6 μ-ops at each clock cycle
- The two queues (memory and other instructions) send as fast as they can μ-ops to the scheduler, handling logical processors in a round-robin fashion, if possible
- Each scheduler has its own queue, of up to 12 entries
- μ-ops are taken from these queues independently of the logical processor (schedulers are logical-processor-oblivious)
- μ-ops are evaluated depending on the availability of their input operands

Execution Units

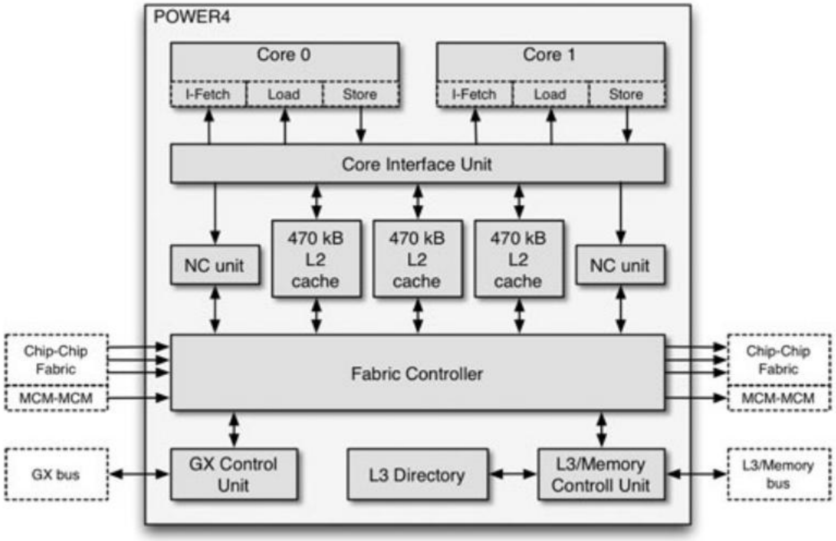
- They are oblivious of the logical processors as well
- The availability of many physical registers reduces contention
- Standard forward propagation is used to deliver results to other executing μ-ops
- After the execution, μ-ops are placed in the re-order buffer
- This buffer decouples execution from the retirement stage

Retirement

- μ-op retirement commits the architecture state in program order
- μ-ops are tracked, and the firmware determines when the execution can be considered completed
- If possible, retirement is enforced in a round-robin fashion which accounts for logical processors
- Once stores have retired, the store data is written on L1 cache

Multicores (2000s)

- First multicore chip: IBM Power4 (1996)
- 1.3 GHz dual-core PowerPC-based CPU



- How do they access the data?

Cache Coherence (CC)

- CC defines the correct behaviour of caches, *regardless of how they are employed by the rest of the system*
- Typically programmers don’t see caches, but caches are usually part of shared-memory subsystems

t	**Core C1**	**Core C2**	*C1 Cache*	*C2 Cache*
0	load r1, A		A: 42	
1		load r1, A	A: 42	A: 42
2	add r1, r1, $1			
	store r1, A		A: 43	A: ??

- What is the value of A in C2?

Strong Coherence

- Most intuitive notion of CC is that cores are *cache-oblivious*:
  - All cores see at any time the same data for a particular memory address, *as they should have if there were no caches in the system*
- Alternative definition:
  - All memory read/write operations to a single location A (coming from all cores) must be executed in a *total order* that respects the order in which each core commits its own memory operations

Strong Coherence

- A sufficient condition for strong coherence is jointly given by implementing two *invariants*:
  1. Single-Writer/Multiple-Readers (SWMR)
     - For any memory location A, at any given epoch, either a single core may read and write to A or some number of cores may only read A
  2. Data-Value (DV)
     - The value of a memory location at the start of an epoch is the same as its value at the end of its latest read-write epoch

Weaker Coherence

- Weaker forms of coherence may exist for performance purposes
  - Caches can respond faster to memory read/write requests
- The SWMR invariant might be completely dropped
  - Multiple cores might write to the same location A
  - One core might read A while another core is writing to it
- The DV invariant might hold only *eventually*
  - Stores are guaranteed to propagate to all cores in d epochs
  - Loads see the last value written only after *d* epochs
- The effects of weak coherency are usually visible to programmers
  - Might affect the memory consistency model (see later)

No Coherence

- The fastest cache is *non-coherent*
  - All read/write operations by all cores can occur simultaneously
  - No guarantees on the value observed by a read operation
  - No guarantees on the propagation of values from write operations
- Programmers must explicitly coordinate caches across cores
  - Explicit invocation of coherency requests via C/Assembly APIs

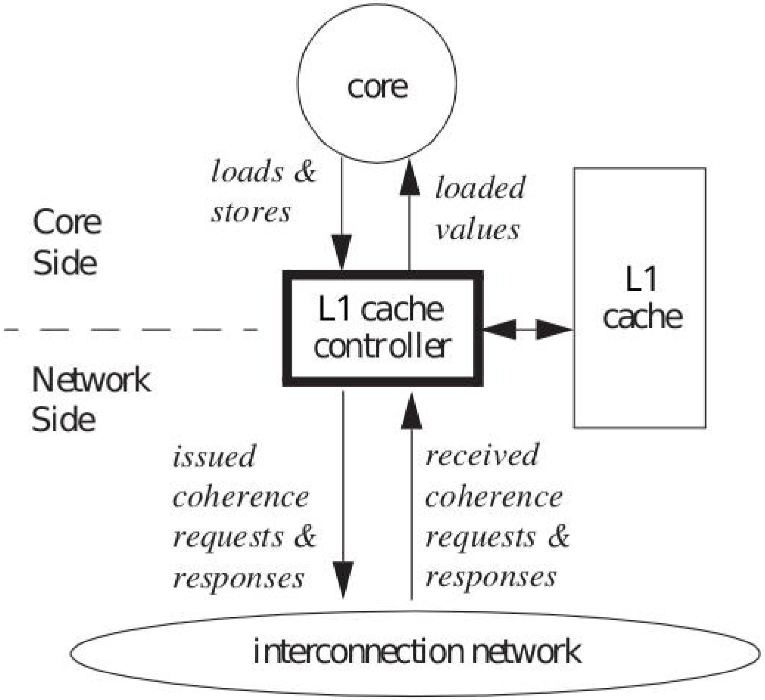
CC Protocols

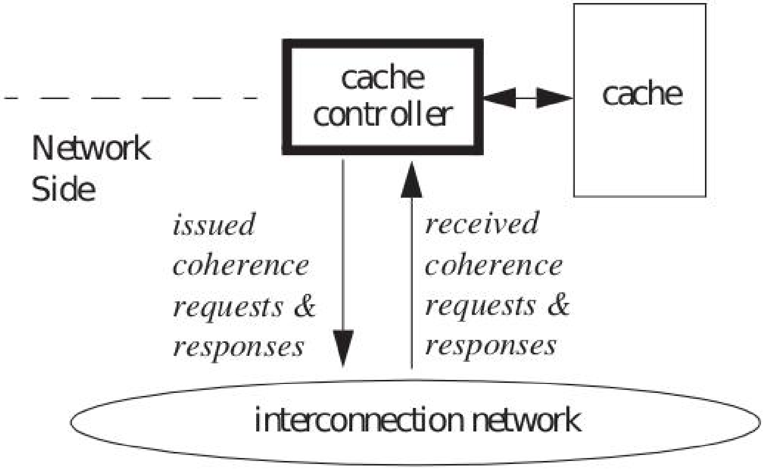
- A CC protocol is a distributed algorithm in a message-passing distributed system model
- It serves two main kinds of memory requests
  - *Load(A)* to read the value of memory location A
  - *Store(A, v)* to write the value v into memory location A
- Itinvolves two main kinds of actors
  - Cache controllers (i.e., L1, L2, ..., LLC)
  - Memory controllers
- It enforces a given notion of coherence
  - Strong, weak, no coherence

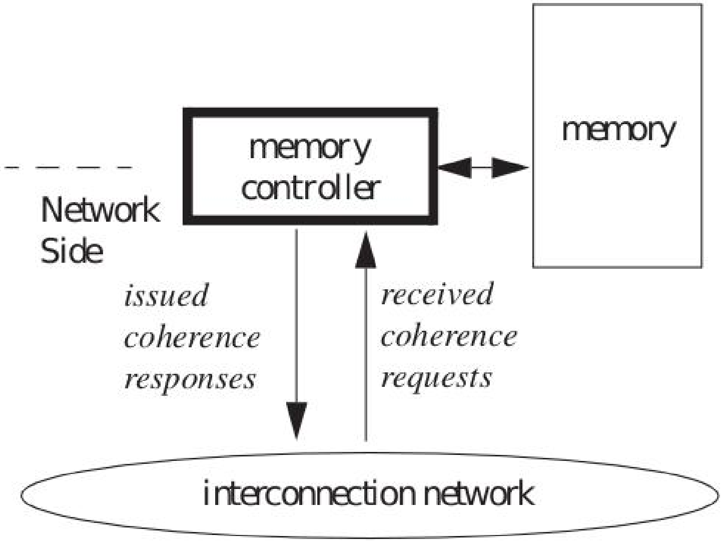
Coherency Transactions

- A memory request may traduce into some *coherency transactions* and produce the exchange of multiple *coherence messages*
- There are two main kinds of coherency transactions:
  - *Get*: Load a cache block *b* into cache line *l*
  - *Put*: Evict a cache block *b* out of cache line *l*

Cache and Memory Controllers




cache controller
cache
Network Side
issued coherence requests & responses
received coherence requests & responses
interconnection network



Finite-State Machines

- Cache controllers manipulate local finite-state machines (FSMs)
- A single FSM describes the state of a copy of a block (not the block itself)
- States:
  - Stable states, observed at the beginning/end of a transaction
  - Transient states, observed in the midst of a transaction
- Events:
  - Remote events, representing the reception of a coherency message
  - Local events, issued by the parent cache controller
- Actions:
  - Remote action, producing the sending of a coherency message
  - Local actions, only visible to the parent cache controller

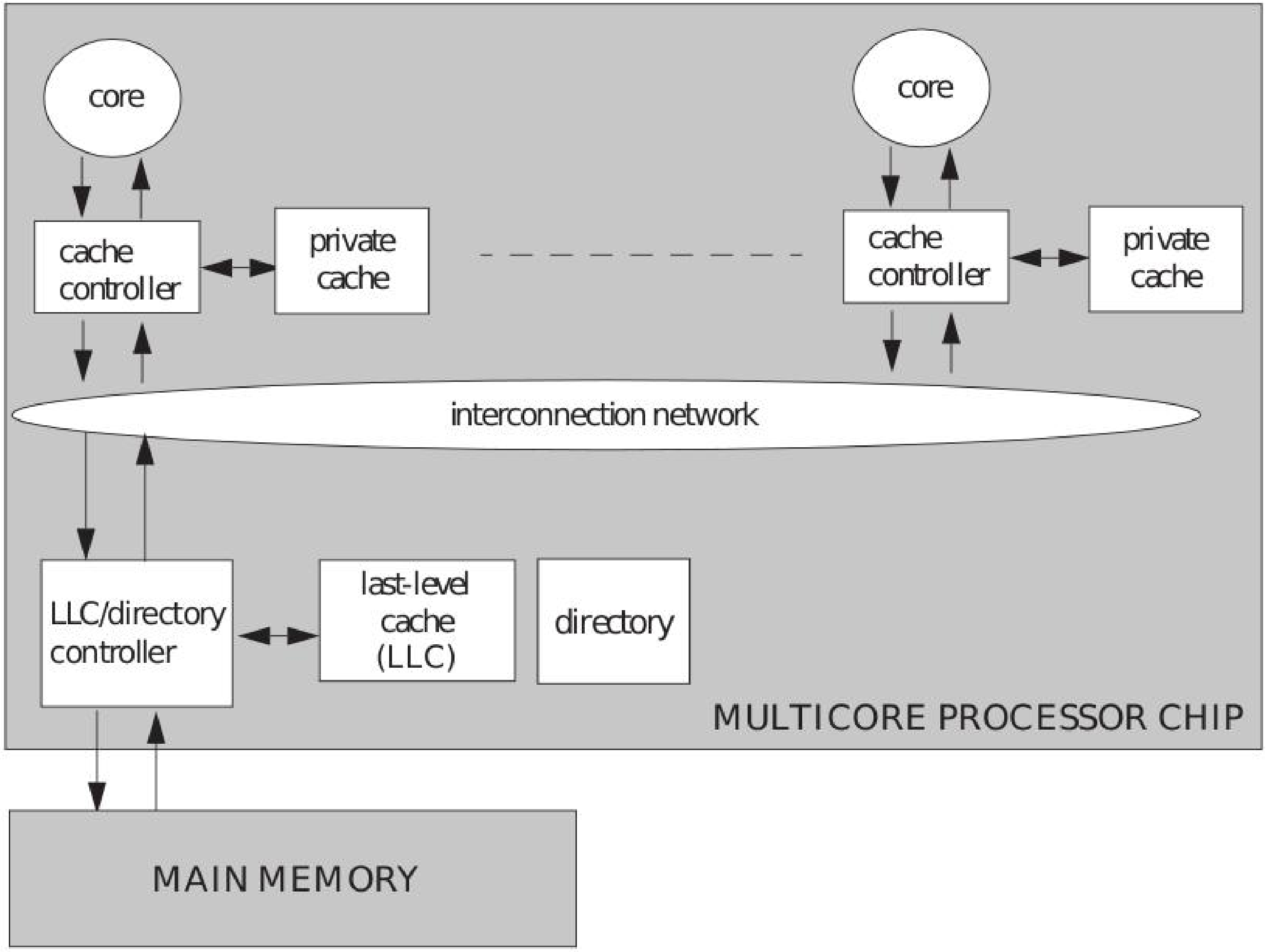
Families of Coherence Protocols

- **Invalidate protocols**:
  - When a core writes to a block, all other copies are invalidated
  - Only the writer has an up-to-date copy of the block
  - Trades latency for bandwidth
- **Update protocols:**
  - When a core writes to a block, it updates all other copies
  - All cores have an up-to-date copy of the block
  - Trades bandwidth for latency

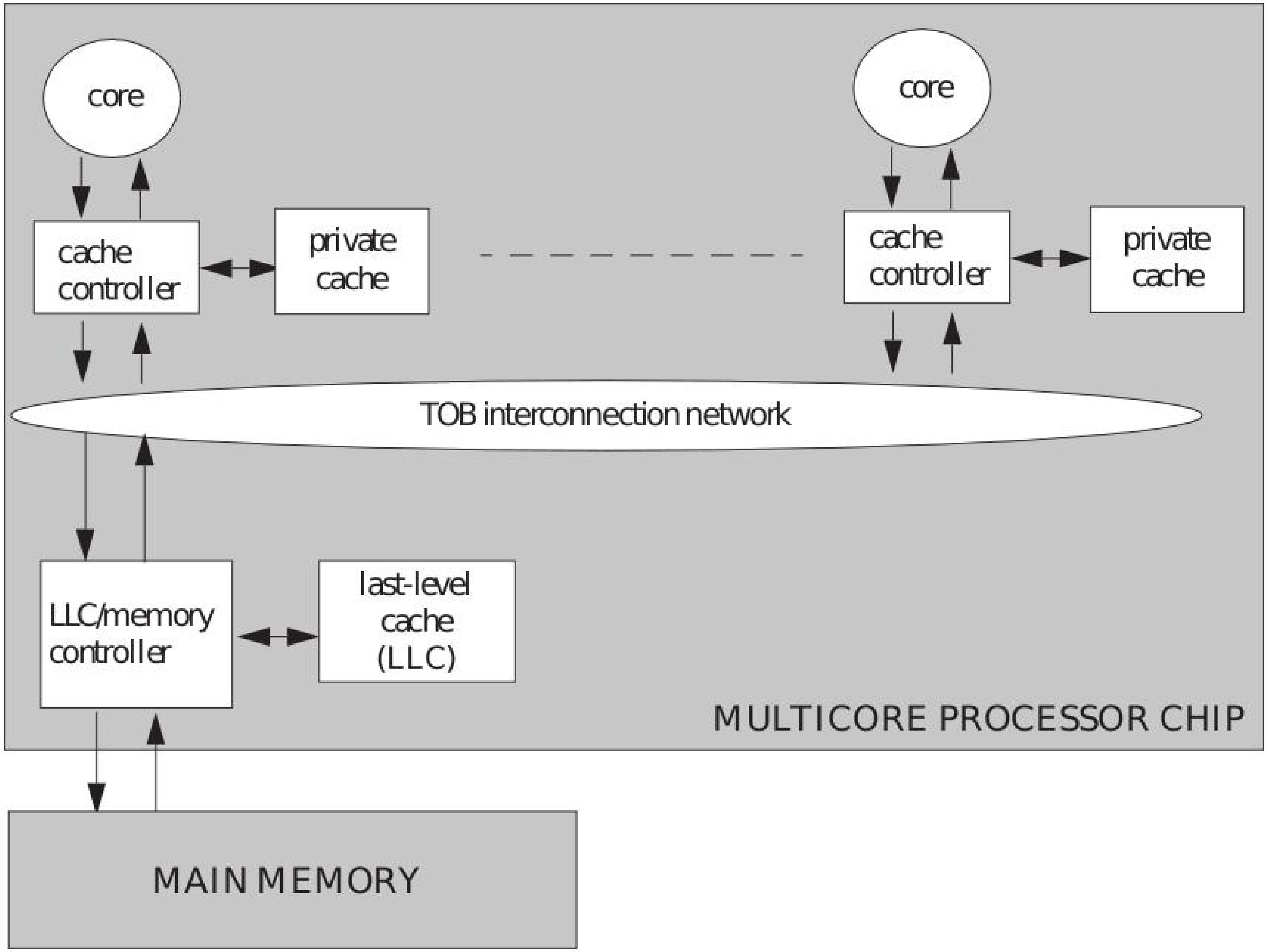
Families of Coherence Protocols

- **Snooping Cache:**
  - Coherence requests for a block are broadcast to all controllers
  - Require an interconnection layer which can total-order requests
  - Arbitration on the bus is the serialization point of requests
  - Fast, but not scalable
- **Directory Based:**
  - Coherence requests for a block are unicast to a directory
  - The directory forwards each request to the appropriate core
  - Require no assumptions on the interconnection layer
  - Arbitration at the directory is the serialization point of requests
  - Scalable, but not fast

Directory System Model



Snooping-Cache System Model

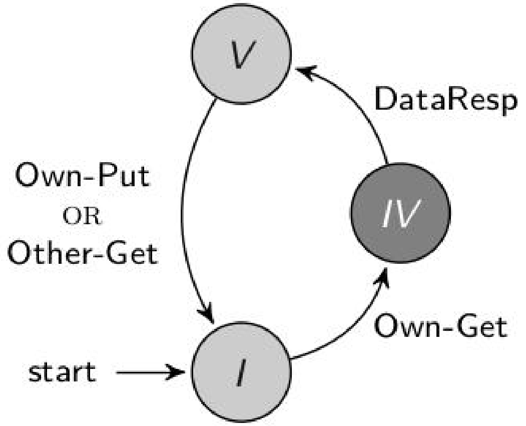


The VI Protocol

- Only one cache controller can read and/or write the block in any epoch
- Supported transactions:
  - Get: to request a block in read-write mode from the LLC controller
  - Put: to write the block’s data back to the LLC controller
- • List of events:
  - `Own-Get`: Get transaction issued from local cache controller
  - `Other-Get`: Get transaction issued from remote cache controller
  - `Any-Get`: Get transaction issued from any controller
  - `Own-Put`: Put transaction issued from local cache controller
  - `Other-Put`: Put transaction issued from remote cache controller
  - `Any-Put`: Put transaction issued from any controller
  - `DataResp`: the block’s data has been successfully received

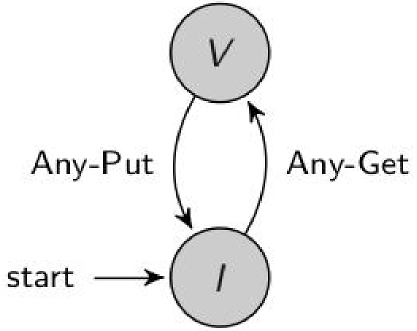
The VI Protocol

Cache controllers



V	Valid read-write copy
IV	Waiting for an up-to-date copy (transient state)
I	Invalid copy

LLC/Memory controller



V	One valid copy in L1
I	No valid copies in L1

The VI Protocol

- It has an implicit notion of *dirtiness* of a block
  - When in state V , the L1 controller can either read-write or just read the block (can't distinguish between the two usages)
- It has an implicit notion of *exclusiveness* for a block
  - When in state V , the L1 controller has exclusive access to that block (no one else has a valid copy)
- It has an implicit notion of ownership of a block
  - When in state V , the L1 controller is responsible for transmitting the updated copy to any other controller requesting it
  - In all other states, the LLC is responsible for the data transfer
- This protocol has minimal space overhead (only a few states), but it is quite inefficient—why?

What a CC Protocol should offer

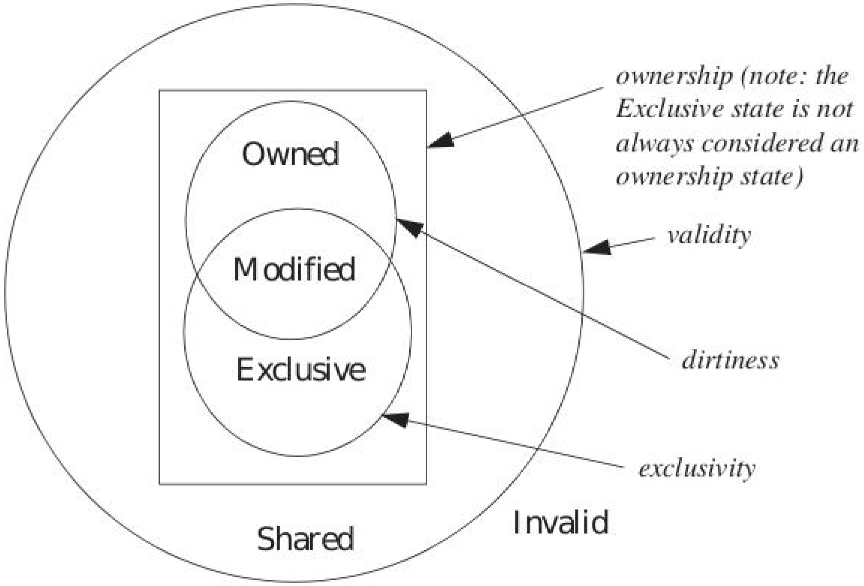
- We are interested in capturing more aspects of a cache block
  - *Validity*: A valid block has the most up-to-date value for this block. The block can be read, but can be written only if it is exclusive.
  - *Dirtiness*: A block is dirty if its value is the most up-to-date value, and it differs from the one stored in the LLC/Memory.
  - *Exclusivity*: A cache block is exclusive if it is the only privately cached copy of that block in the system (except for the LLC/Memory).
  - *Ownership*: A cache controller is the owner of the block if it is responsible for responding to coherence requests for that block.

- In principle, the more properties are captured, the more aggressive is the optimization (and the space overhead!)

MOESI Stable States

- **Modified (M)**: The block is valid, exclusive, owned and potentially dirty. It can be read or written. The cache has the only valid copy of the block.
- **Owned (O):** The block is valid, owned, and potentially dirty, but not exclusive. It can be only read, and the cache must respond to block requests.
- **Exclusive (E):** The block is valid, exclusive and clean. It can be only read. No other caches have a valid copy of the block. The LLC/Memory block is up-to-date.
- **Shared (S):** The block is valid but not exclusive, not dirty, and not owned. It can be only read. Other caches may have valid or read-only copies of the block.
- **Invalid (I):** The block is invalid. The cache either does not contain the block, or the block is potentially stale. It cannot be read nor written.

MOESI Stable States


Owned
Modified
Exclusive
Shared
Invalid
ownership (note: the Exclusive state is not always considered an ownership state)
validity
dirtiness
exclusivity

- Many protocols spare one bit and drop the Owned state (MESI)
- Simpler protocols drop the Exclusive state (MSI)

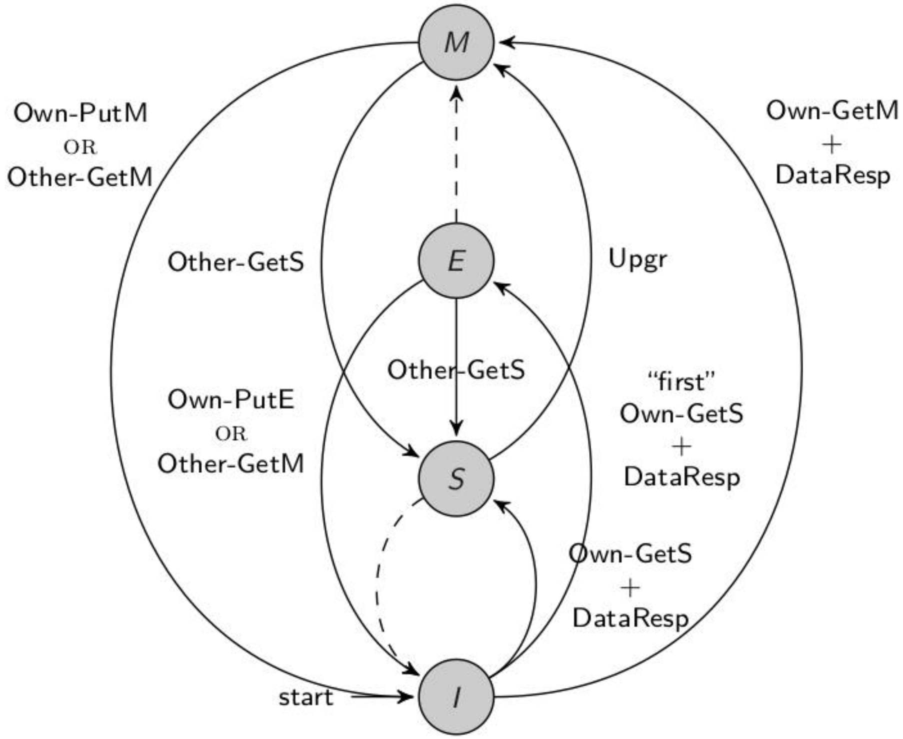
MOESI Transactions

- *GetS*: Obtain a block in Shared (read-only) state
- *GetM*: Obtain a block in Modified (read/write) state
- *Upgr*: Upgrade block from Shared/Owned (read-only) to Morified (read/only)—no data transfer needed

- *PutS*: evict block in Shared state
- *PutE*: evict block in Exclusive state
- *PutO*: evict block in Owned state
- *PutM*: evict block in Modified state

The MESI Protocol

- **Cache controllers**

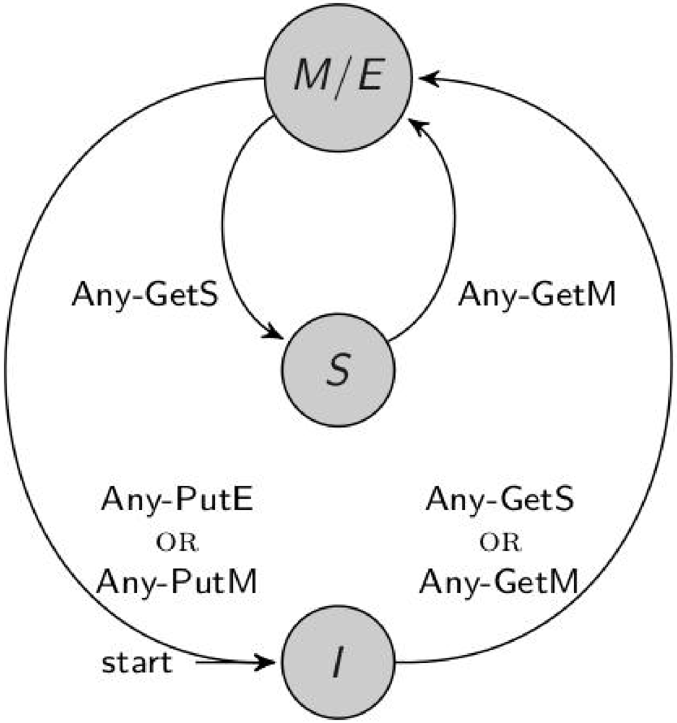


M	Valid read-write copy
E	Valid read copy (exclusive copy)
S	Valid read copy
I	Invalid copy

Dashed lines indicate "silent" transactions

The MESI Protocol

- **LLC/Memory controller**



M	One valid read-write copy in L1
E	One valid read-only copy in L1
S	Zero or more valid read-only copy in L1
I	No valid copies in L1

The MESI Protocol

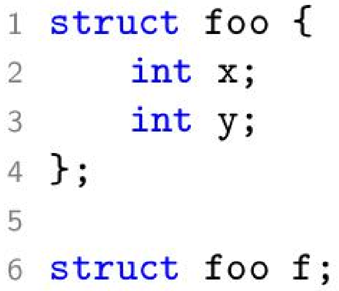
- The notion of exclusiveness is explicit (E state)
  - E to M transition in L1 controller is silent
- The notion of ownership is implicit (M/E states)
- Counting on S copies is avoided
  - S to I transition in L1 controller is silent
  - S to I transition in LLC is disabled
- An Other-GetS message in state M or E requires the LLC to update its copy and become the owner

CC and Write-Through Caches

- Stores immediately propagate to the LLC
  - States M and S collapse into V (no dirty copies)
  - Eviction only requires a transition from V to I (no data transfer)
- Write-through requires more bandwidth and power to write data

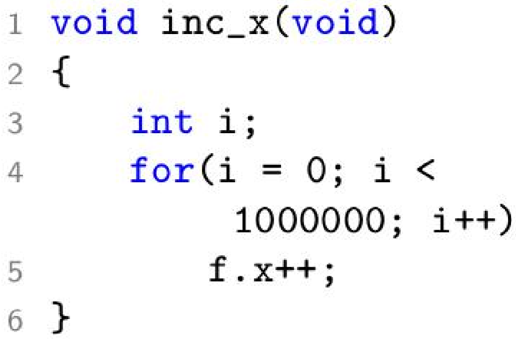
CC and False Cache Sharing

- This problem arises whenever two cores access different data items that lie on the same cache line (e.g., 64B–256B granularity)
- It produces an invalidation although accessed data items are different

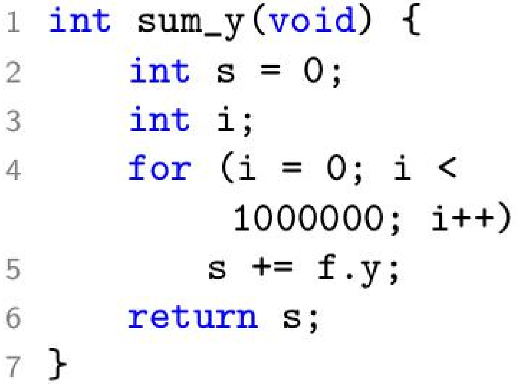


```c
struct foo {
    int x;
    int y;
};

struct foo f;
```



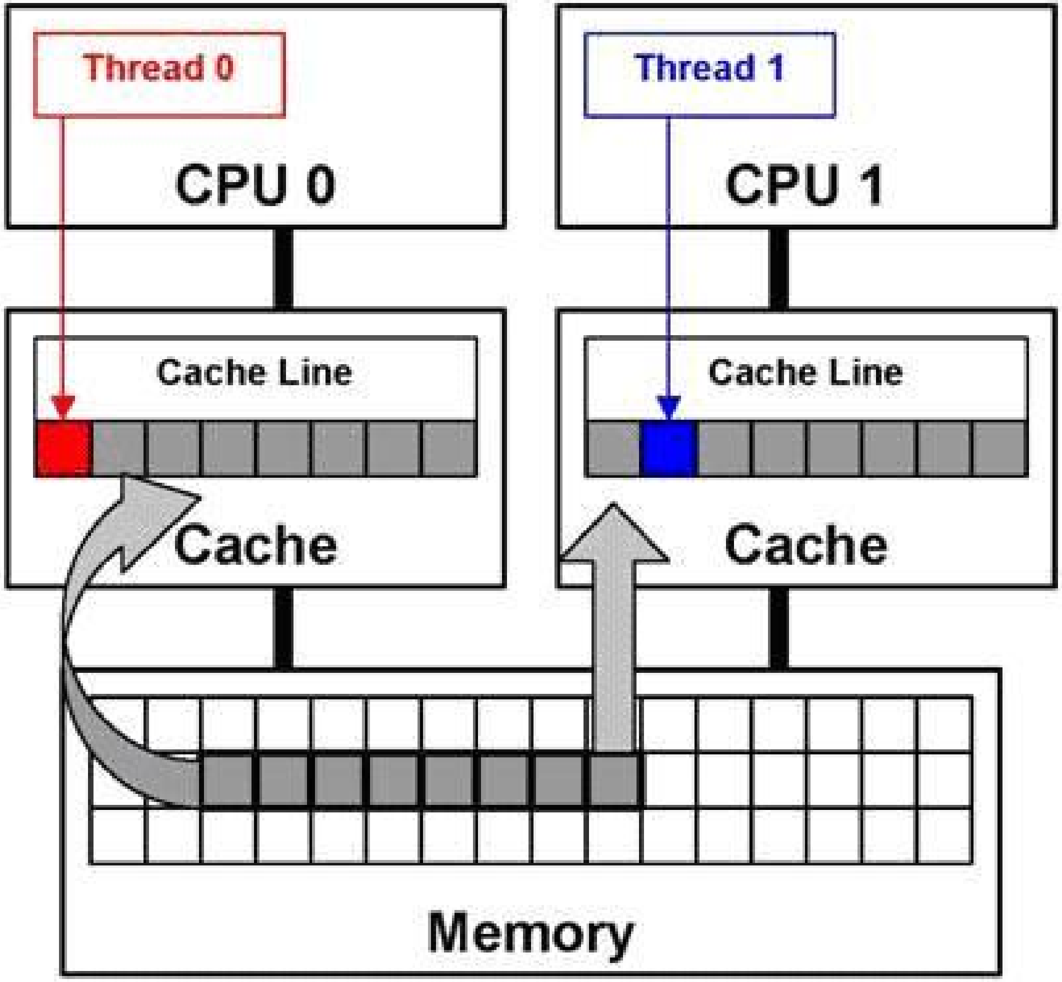
```c
void inc_x(void)
{
    int i;
    for(i = 0; i <
            1000000; i++)
           f.x++;
}
```



```c
int sum_y(void) {
    int s = 0;
    int i;
    for (i = 0; i <
            1000000; i++)
          s += f.y;
    return s;
}
```

- Can be solved using sub-block coherence or speculation
- Better if prevented by good programming practices

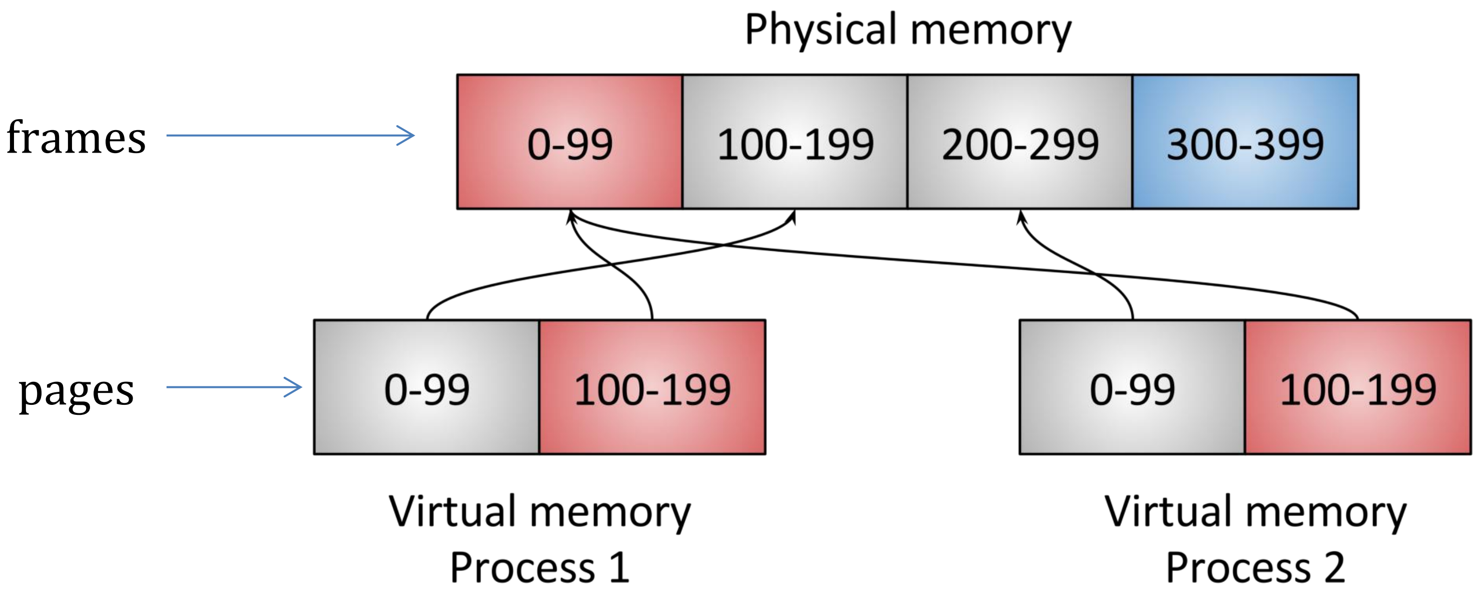
The False Cache Sharing Problem



Virtual vs Physical Memory Addressing

- A technique supported by CPUs since the 70's
  - A layer of abstraction between the memory address layout that most software sees and the physical devices backing that memory
- It allows applications to utilize more memory than the machine actually has
- The virtual-to-physical translation is supported by hardware/firmware (Translation Lookaside Buffer—TLB) cooperating with the Operating System

Virtual vs Physical Memory Addressing



Physical memory

frames

0-99

100-199

200-299

300-399

pages

0-99

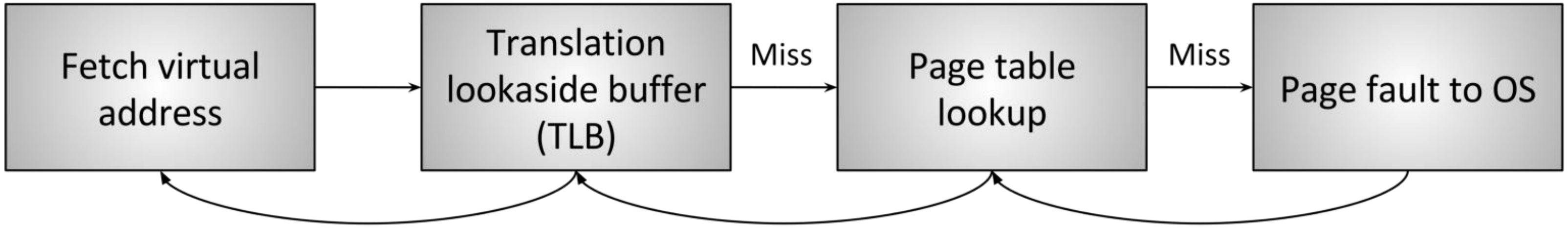
100-199

Virtual memory
Process 1

0-99

100-199

Virtual memory
Process 2



Fetch virtual address

Translation lookaside buffer (TLB)

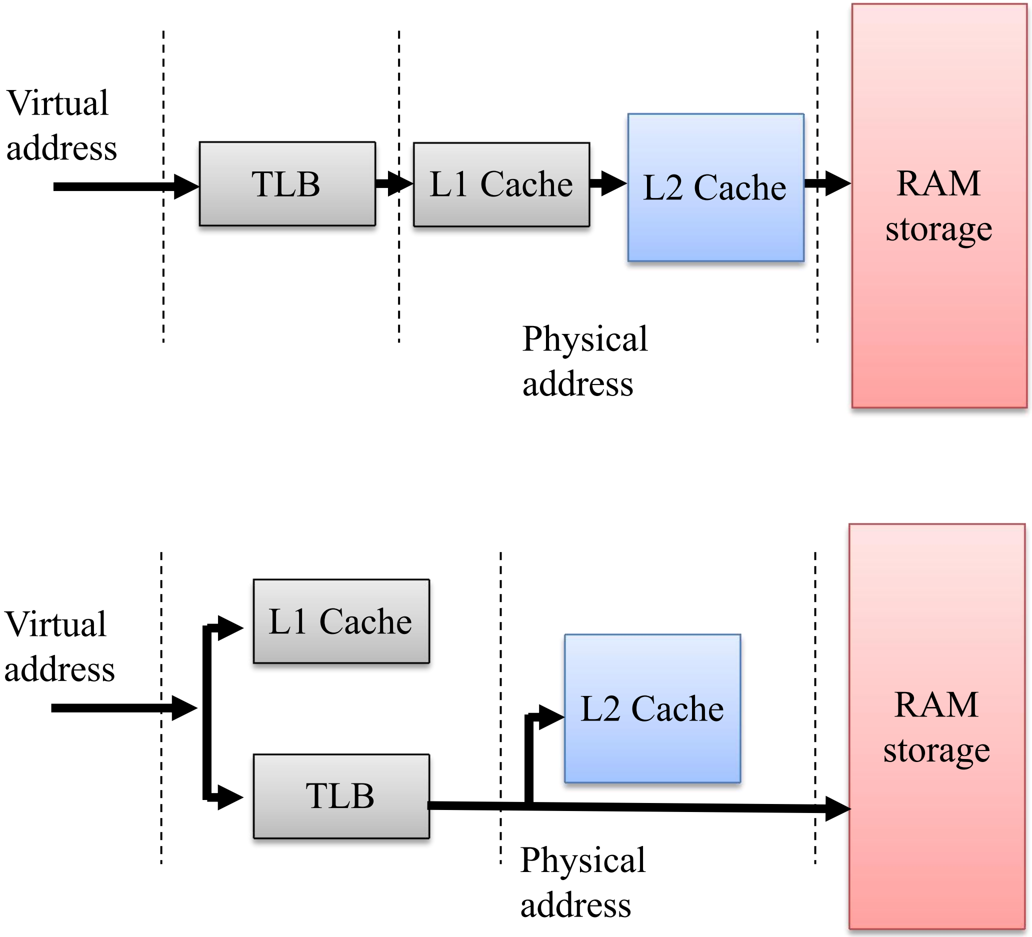
Miss

Page table lookup

Miss

Page fault to OS

Where to put the L1 cache?



- With “virtual caches”:
  - one-step process in case of a hit
  - cache must be flushed upon context switch
  - *aliasing* problems occur

x86 Caches

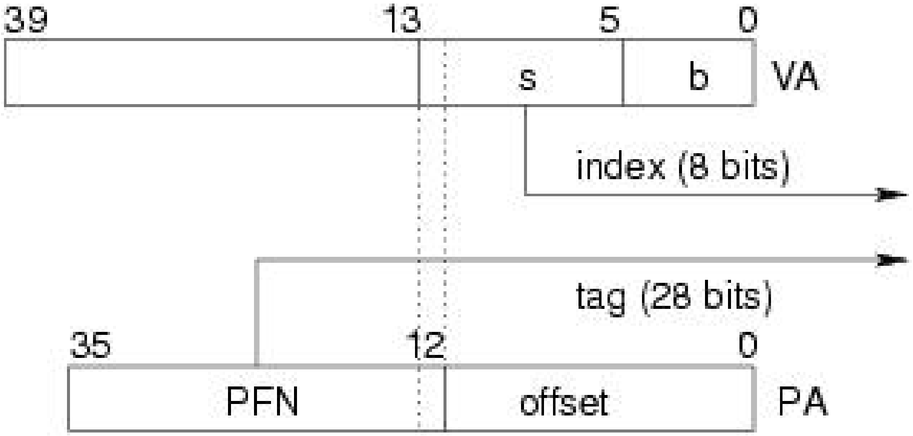
- On x86 architectures, caches are physically indexed and physically tagged (except for small L1 caches)
- Virtual address associated with any memory map is filtered by the MMU before real access to the memory hierarchy is performed

ARM Caches

- On ARMv4 and ARMv5 processors, cache is organized as a virtual-indexed, virtual-tagged (VIVT) cache
- Both the index and the tag are based on the virtual address.
  - Cache lookups are faster (TLB is not involved in matching cache lines for a virtual address)
  - The same physical address can be mapped to multiple virtual addresses

MIPS R4x00 Caches

- Virtually-indexed, physically tagged on-chip L1 cache
  - 16kB cache with 32B lines
  - 4kB (base) page size
- Location of data in cache is determined by the index
- The tag only confirms whether it's a hit



Virtual Aliasing

- This is an anomaly occurring when the cache (at some level) is indexed via virtual addresses (e.g. Sparc64)
- This leads to cache coherency issues
- Two families of problems:
  - Homonyms
  - Synonyms (alias)

Virtual Aliasing: Homonyms

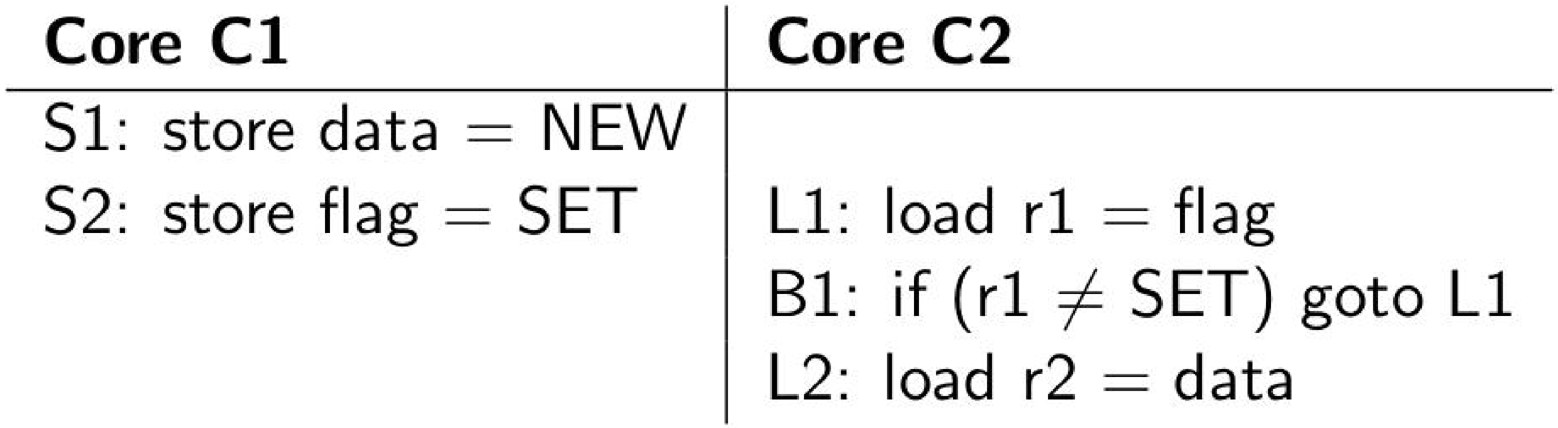
- Same VA corresponds to several PAs
  - standard situation in multitasking systems
- Problem: tag may not uniquely identify cache data!
- Homonyms lead to cache accessing the wrong data!
- Homonym prevention:
  - tag virtual address with address-space ID (ASID)
    - disambiguates virtual addresses (makes them globally valid)
  - use physical tags
  - flush cache on context switch

Virtual Aliasing: Synonyms

- Several VAs map to the same PA
  - frames shared between processes
  - multiple mappings of a frame within an address space
- Synonyms in cache may lead to accessing stale data:
  - same data may be cached in several lines
  - on write, one synonym is updated
  - a subsequent read on the other synonym returns the old value
- Physical tagging doesn't help, ASIDs don't help either
- Solutions:
  - hardware synonym detection (disallow aliases to coexist in cache)
  - flush cache on context switch
  - restrict VM mapping so synonyms map to same cache set

Memory Consistency (MC)

- MC defines the correct behaviour of shared-memory subsystems, *regardless of how they are implemented*
- Programmers know what to expect, implementors know what to provide



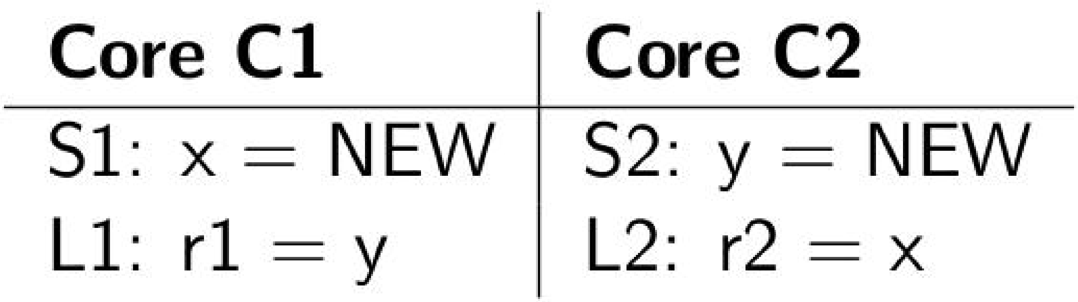
Core C1	Core C2
S1: store data = NEW	
S2: store flag = SET	L1: load r1 = flag
	B1: if (r1 $\neq$ SET) goto L1
	L2: load r2 = data

- What is the value of r2?

Reordering of Memory Accesses

- Reordering occurs when two memory R/W operations:
  - Are committed by a core in order
  - Are seen by other cores in a different order
- Mainly for performance reasons
  - Out-of-order execution/retirement
  - Speculation (e.g., branch prediction)
  - Delayed/combined stores
- Four possible reorderings
  - Store-store reordering
  - Load-load reordering
  - Store-load reordering
  - Load-store reordering

An example



Core C1	Core C2
S1: x = NEW	S2: y = NEW
L1: r1 = y	L2: r2 = x

- Multiple reorderings are possible:
  - (r1, r2) = (0, NEW) [S1, L1, S2, L2]
  - (r1, r2) = (NEW, 0) [S2, L2, S1, L1]
  - (r1, r2) = (NEW, NEW) [S1, S2, L1, L2]
  - (r1, r2) = (0, 0) [L1, L2, S1, S2]

Allowed by most real hardware architectures (also x86!)

Program and Memory Orders

- A program order $\prec_p$ is a per-core total order that captures the order in which each core logically executes memory operations
- A memory order $\prec_m$ is a system-wide total order that captures the order in which memory logically serializes memory operations from all cores
- Memory consistency can be defined imposing constraints on how $\prec_p$ and $\prec_m$ relate to each other

Sequential Consistency

- Let *L(a)* and *S(a)* be a Load and a Store to address *a*, respectively
- A *Sequentially Consistent* execution requires that:
  - All cores insert their loads and stores into $\prec_m$ respecting their program order regardless of whether they are to the same or different address (i.e., a = b or a ≠ b):
    - If $L(a) \prec_p L(b) \Rightarrow L(a) \prec_m L(b)$ (Load/Load)
    - If $L(a) \prec_p S(b) \Rightarrow L(a) \prec_m S(b)$ (Load/Store)
    - If $S(a) \prec_p S(b) \Rightarrow S(a) \prec_m S(b)$ (Store/Store)
    - If $S(a) \prec_p L(b) \Rightarrow S(a) \prec_m L(b)$ (Store/Load)
  - Every load gets its value from the last store before it (as seen in memory order) to the same address:
    - value of $L(a) =$ value of $\max_{\prec m} \{S(a) | S(a) \prec_m L(a)\}$ where $\max_{\prec m}$ is the latest in memory order

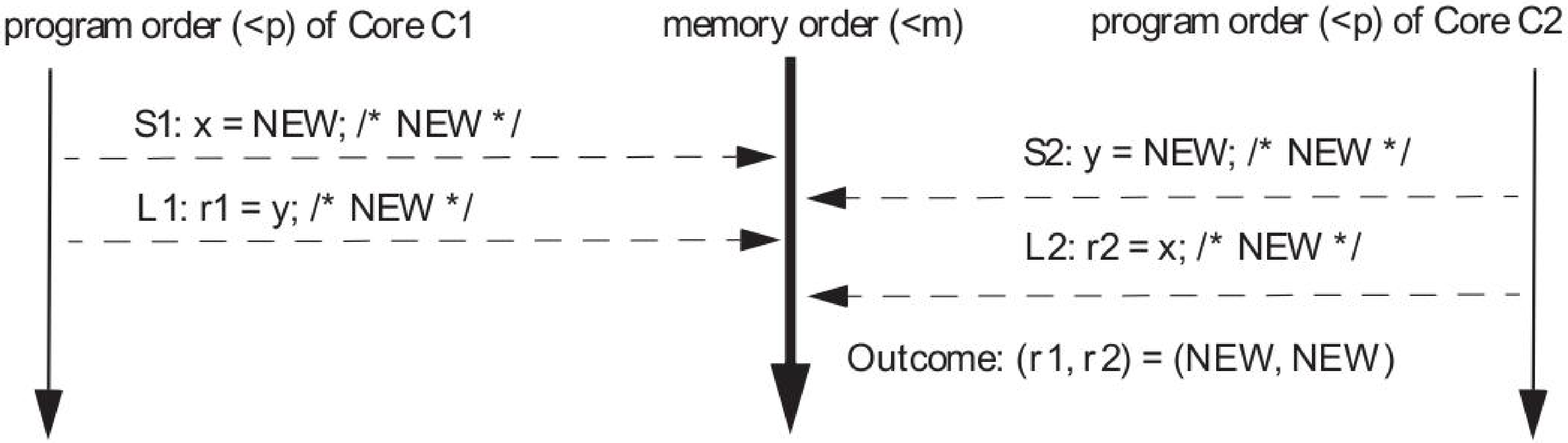
Sequential Consistency in Practice

Core C1	Core C2
S1: x = NEW	S2: y = NEW
L1: r1 = y	L2: r2 = x

- We can globally reorder the execution in four different ways
- Only three of them are sequentially consistent

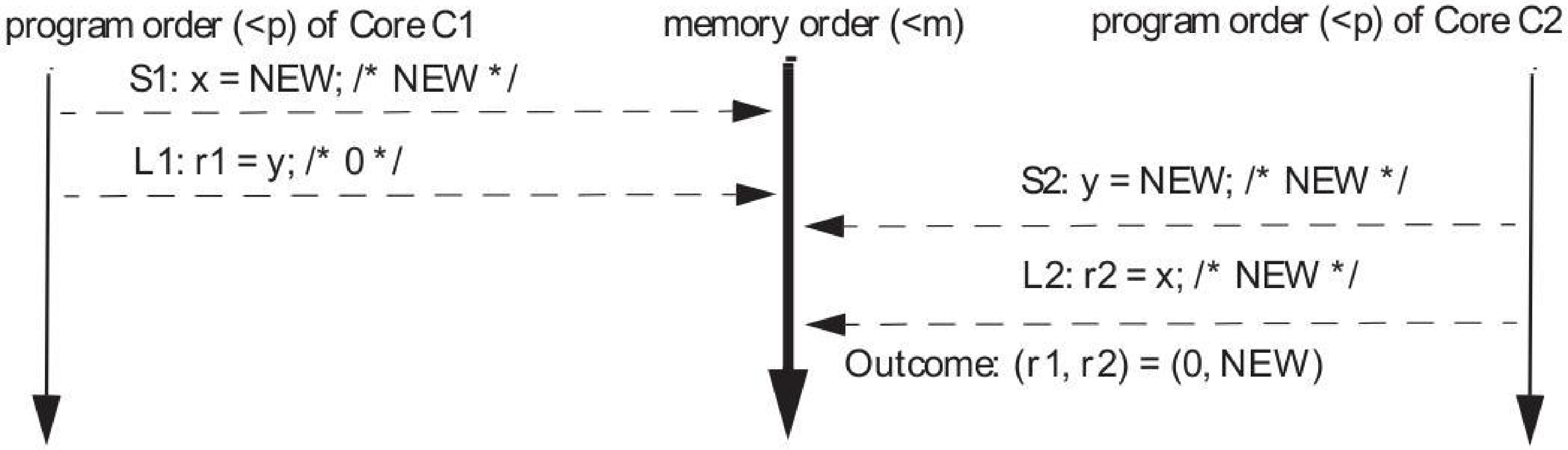
Sequential Consistency in Practice

- This is a SC execution



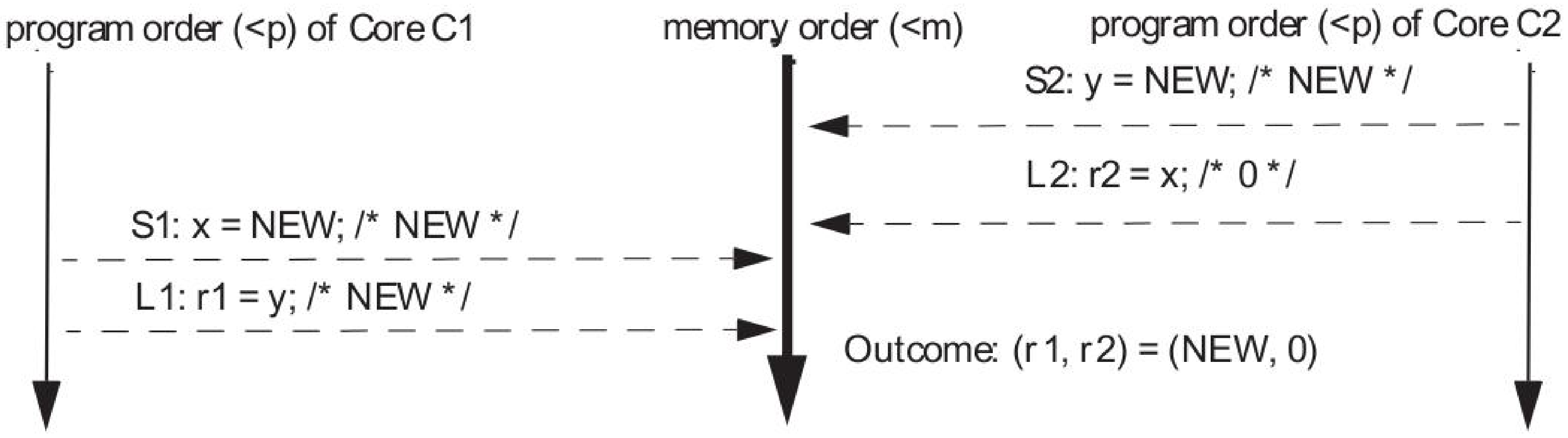
Sequential Consistency in Practice

- This is a SC execution



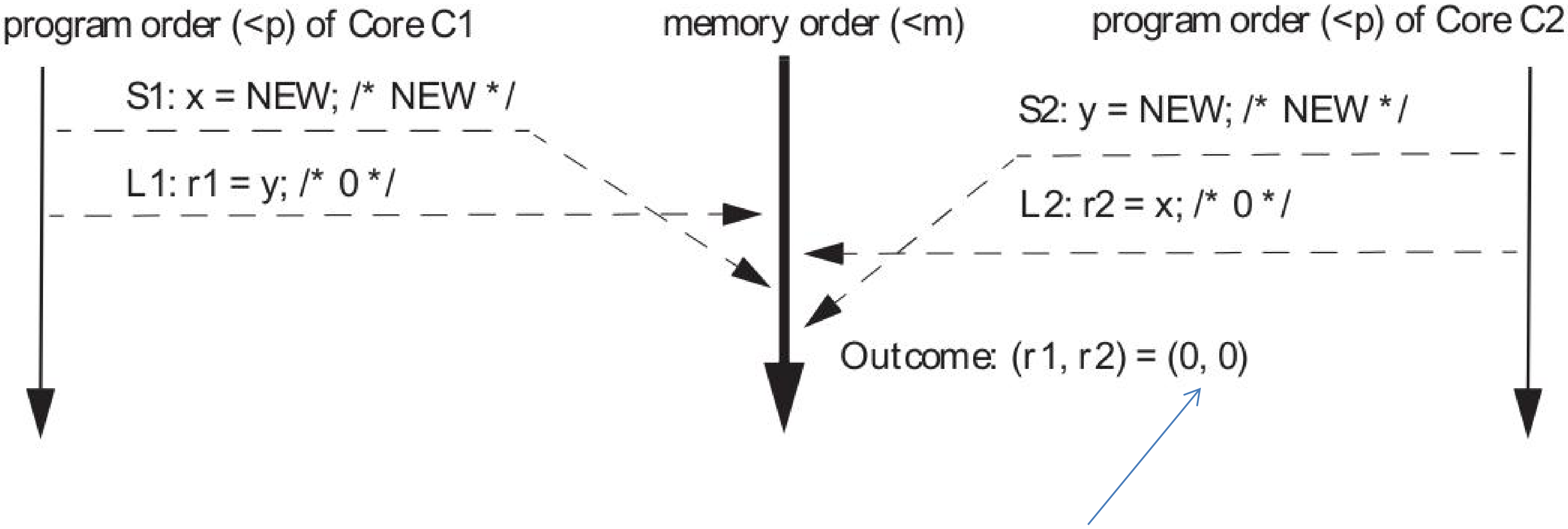
Sequential Consistency in Practice

- This is a SC execution



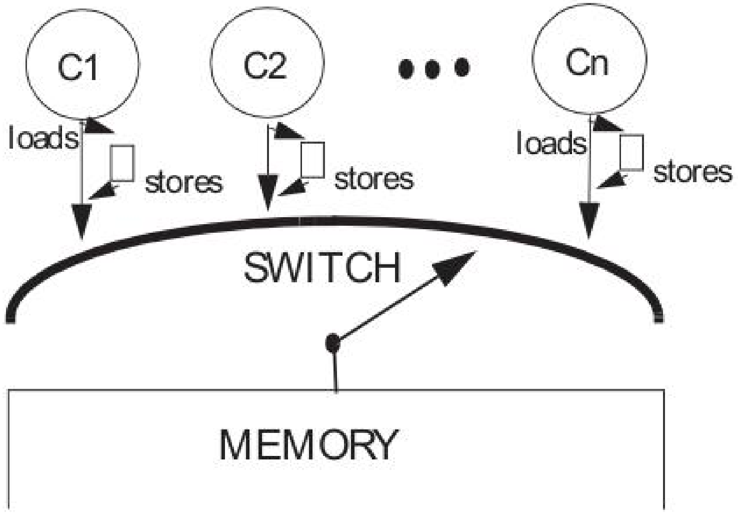
Sequential Consistency in Practice

- This is not a SC execution



Why did we say that this outcome
is allowed on common hardware architectures?

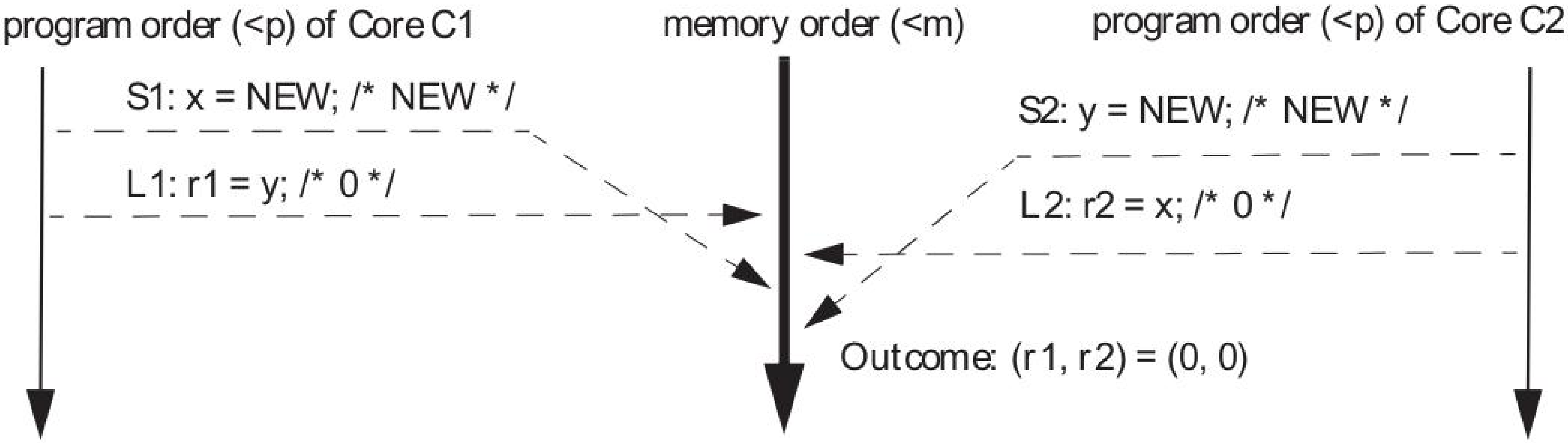
Weaker Consistency: Total Store Order



- A FIFO *store buffer* is used to hold committed stores until the memory subsystem can process it
- When a load is issued by a core, the store buffer is looked up for a matching store
  - if found, the load is served by the store buffer (*forwarding*)
  - otherwise it is served by the memory subsystem (*bypassing*)

Sequential Consistency in Practice

- This is a valid TSO execution



- A programmer might want to avoid the result (r1, r2) = (0, 0)

Memory Reordering in the Real World

Type	Alpha	ARMv7	POWER	SPARC PSO	x86	AMD64	IA-64
LOAD/LOAD	√	√	√				√
LOAD/STORE	√	√	√				√
STORE/STORE	√	√	√	√			√
STORE/LOAD	√	√	√	√	√	√	√
ATOMIC/LOAD	√	√	√				√
ATOMIC/STORE	√	√	√	√			√
Dependent LOADs	√						
Incoherent I-cache	√	√	√	√	√		√

Memory Fences

- Let *X(a)* be either a load or a store operation to *a*
- *Memory fences* force the memory order of load/store operations:
  - If $X(a) \prec_p FENCE \Rightarrow X(a) \prec_m FENCE$
  - If $FENCE \prec_p X(a) \Rightarrow FENCE \prec_m X(a)$
  - If $FENCE \prec_p FENCE \Rightarrow FENCE \prec_m FENCE$
- With fences it is possible to implement SC over TSO (with a significant penalty)

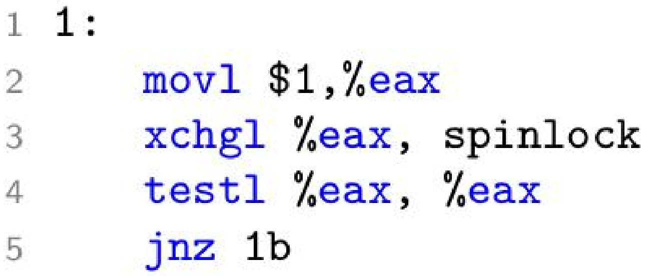
x86 Fences

- MFENCE: Full barrier
  - If $X(a) \prec p$ MFENCE $\prec_p X(b) \Rightarrow X(a) \prec_m$ MFENCE $\prec_m X(b)$
- SFENCE: Store/Store barrier
  - If $S(a) \prec_p$ SFENCE $\prec_p S(b) \Rightarrow S(a) \prec_m$ SFENCE $\prec_m S(b)$
- LFENCE: Load/Load and Load/Store barrier
  - If $L(a) \prec_p$ LFENCE $\prec_p X(b) \Rightarrow L(a) \prec_m$ LFENCE $\prec_m X(b)$

- Both MFENCE and SFENCE drain the store buffer

Atomic Operations

- The following is a naïve implementation of a spinlock on x86
  - It repeatedly tries to replace the content of a spinlock variable

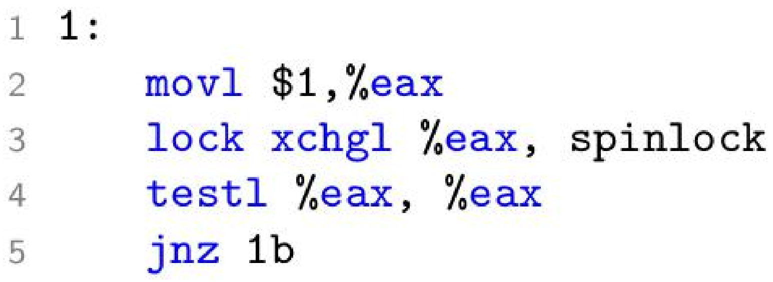


```
1:
    movl $1,%eax
    xchgl %eax, spinlock
    testl %eax, %eax
    jnz 1b
```

- Is it correct?

Atomic Operations

- The following is a naïve implementation of a spinlock on x86
  - It repeatedly tries to replace the content of a spinlock variable



```
1:
        movl $1,%eax
        lock xchgl %eax, spinlock
        testl %eax, %eax
        jnz 1b
```

- Is it correct?

Read-Modify-Write Instructions

- RMW is a class of instructions which implement *atomic operations*
- They allow to read memory and modify its content in an *apparently instantaneous* fashion
- Some important instructions:
  - Compare and Swap (CAS)
  - Fetch and Add (FAA)
  - Test and Set (TAS)
  - Load-Link/Store-Conditional (LL/SC)
- The updated value can be a new value, or a function of the previous value

x86 `lock` prefix

- Atomicity affects both memory consistency and cache coherency
  - Reordering can put an arbitrary operation X between R and W
  - A read and/or write request can appear in between a GetS and a GetM/Upg transaction for the same block
- The goal of the `lock` prefix is to ensure the atomicity of RMW instructions
- Therefore, the execution of a lock'd RMW instruction entails:
  1. Draining the store buffer so as to prevent reordering effects due to forwarding/bypassing
  2. Loading the cache block using a GetM operation so as to limit the effect of concurrent transactions
  3. Retaining the bus so as to lock the state of the cache block until the RMW completes

`lock` and MC

- “R” and “W” must appear consecutively in $\prec_m$ even in TSO
  - Suppose a load part (“R”) could bypass an earlier store...
  - ...then, the store part (“W”) should be moved as well...
  - ...but stores are not allowed to bypass each other in TSO...
  - ...so a load associated with a RMW instruction cannot bypass!
- Actual draining of the store buffer is typically performed on x86

> Locked instructions can be used to synchronize data written by one processor and read by another processor. For the P6 family processors, locked operations serialize all outstanding load and store operations (that is, wait for them to complete).

`lock` and CC

- MESI protocols guarantee that if a block is held exclusively by a certain core, no one else has it
- By adding a simple flag, controllers can perform multiple operations atomically on an exclusive block
- Incoming coherence requests for the exclusive, locked block are not serviced until after the store

> For the P6 and more recent processor families, if the area of memory being locked during a LOCK operation is cached in the processor that is performing the LOCK operation [...] the processor may not assert the LOCK# signal on the bus. Instead, it will modify the memory location internally and allow its cache coherency mechanism to ensure that the operation is carried out atomically. This operation is called “cache locking.”

Correctness vs Efficiency: which comes first?

- Software designers want everything to be *correct*, then fast
  - No reordering of instructions
  - Strong cache coherency
- Hardware designers want everything to be *fast*, then correct
  - Different programs requires different notion of correctness
  - The expected correctness can always be achieved in software

Security Aspects

- A speculative processor can let an attacker exploit micro-architectural effects by using *phantom instructions* even though the pipeline is flushed
- Phantom instructions leave effects in the micro-architectural state for a transient period
- These effects can be observed
- This is the rationale behind *Spectre/Meltdown* attacks
  - It affects Intel, AMD and ARM processors
- Side channel: *memory which allows to read other memory content*

Meltdown Primer

possible byte values

```
uint8_t *probe_array = new uint8_t[256 * 4096];
// Make sure probe_array is not cached
uint8_t kernel_memory = *(uint8_t*)(kernel_address);
uint64_t final_kernel_memory = kernel_memory * 4096;
uint8_t dummy = probe_array[final_kernel_memory];
// handle (eventual) SEGFAULT
// determine which of 256 slots in probe_array is cached
```

Intel OOO Execution

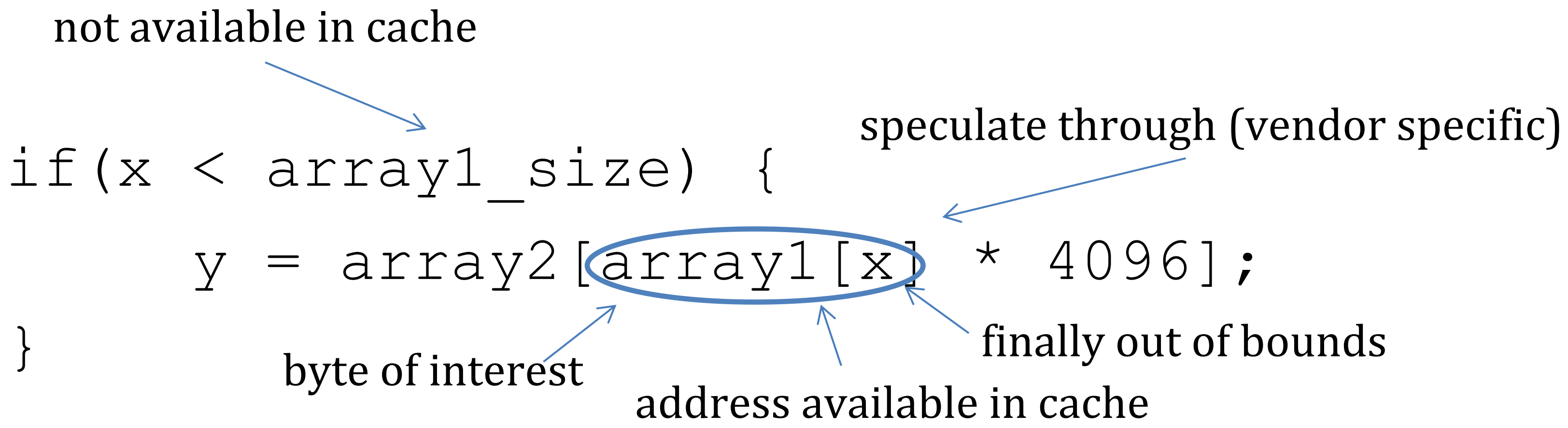
Avoid prefetching
& different lines

- Simply measuring the latency to access `probe_array` at the end of this execution path allows to know the value of the kernel memory byte

Spectre Primer

```
if(x < array1_size) {
     y = array2[array1[x] * 4096];
}
```

Spectre Primer



not available in cache

speculate through (vendor specific)

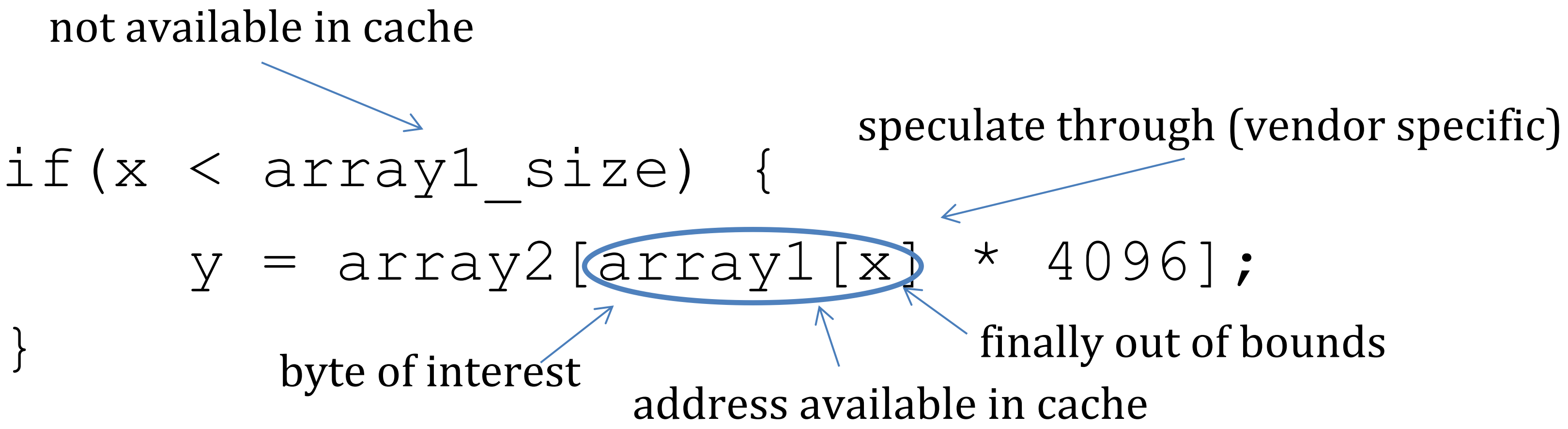
```
if(x < array1_size) {
     y = array2[array1[x] * 4096];
}
```

byte of interest

address available in cache

finally out of bounds

Spectre Primer



```
if(x < array1_size) {
      y = array2[array1[x] * 4096];
}
```

not available in cache

speculate through (vendor specific)

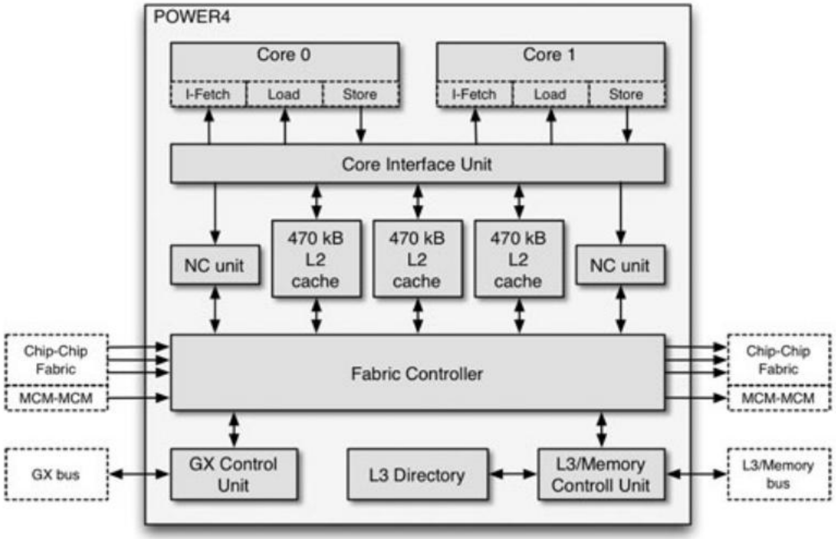
byte of interest

address available in cache

finally out of bounds

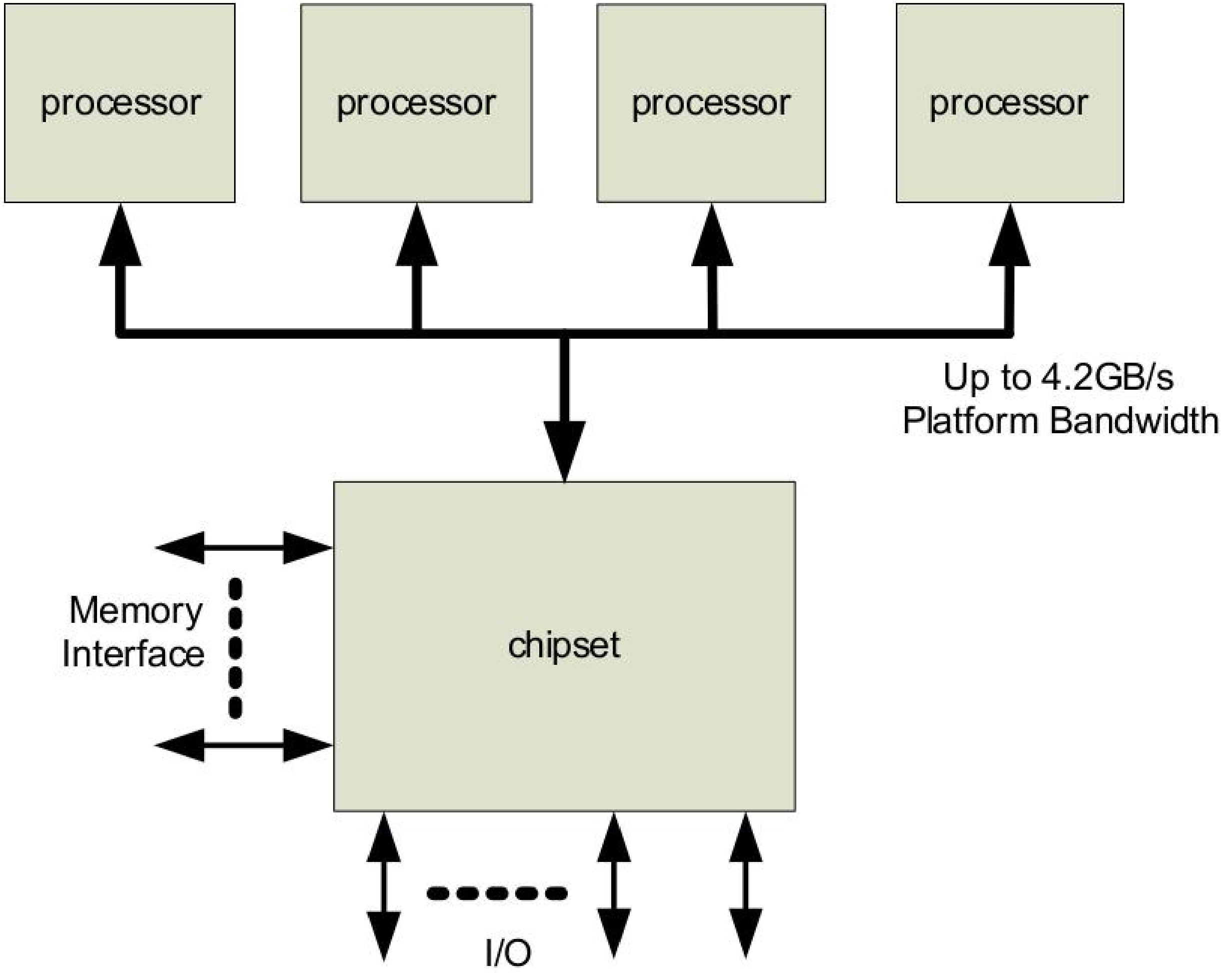
- It is then possible to inspect the cache state of `array2` to see what was the speculatively accessed value of `array1[x]`.

Inter-Core Connection



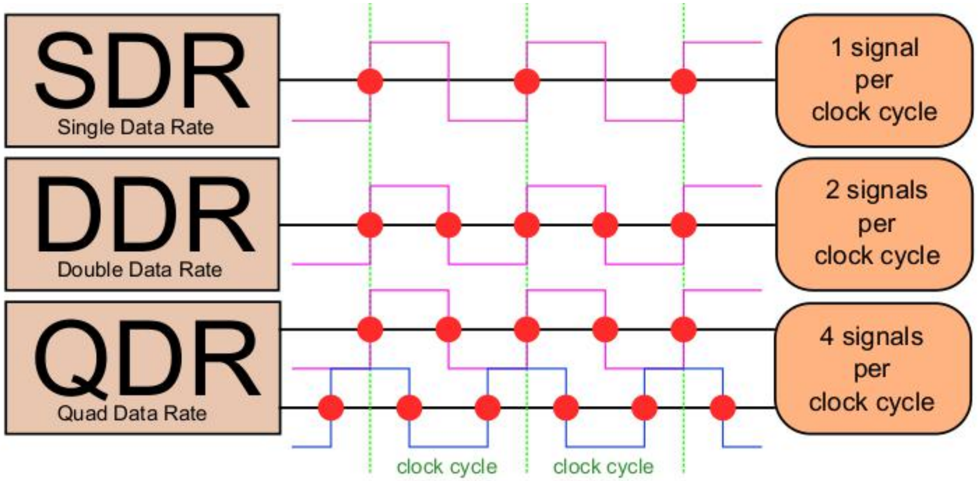
- What is the importance of inter-core connection?

Front-Side Bus (up to 2004)

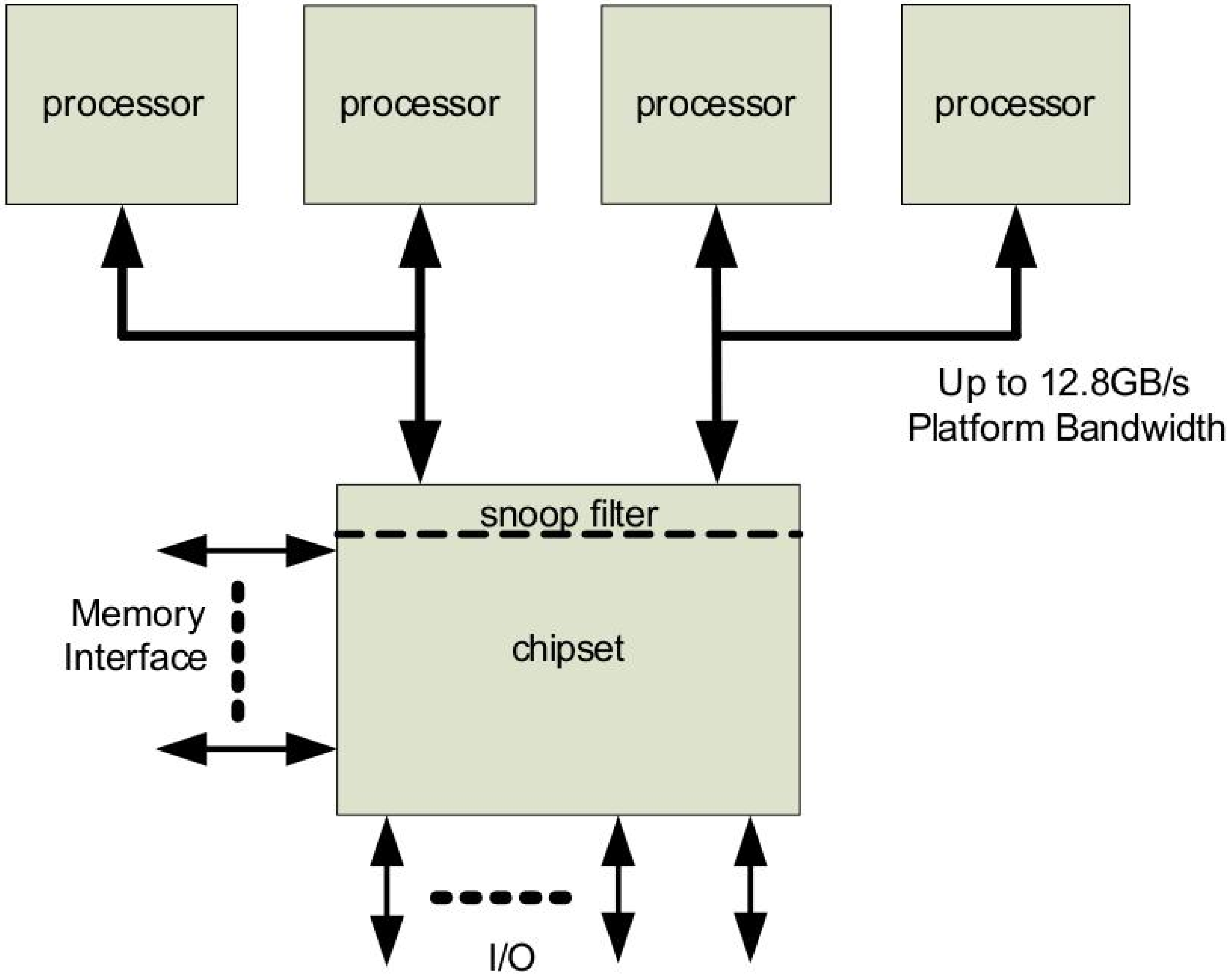


Front-Side Bus (up to 2004)

- All traffic is sent across a single shared bi-directional bus
- Common width: 64 bits, 128 bits — multiple data bytes at a time
- To increase data throughput, data has been clocked in up to 4x the bus clock
  - *double-pumped* or *quad-pumped* bus



Dual Independent Buses (2005)

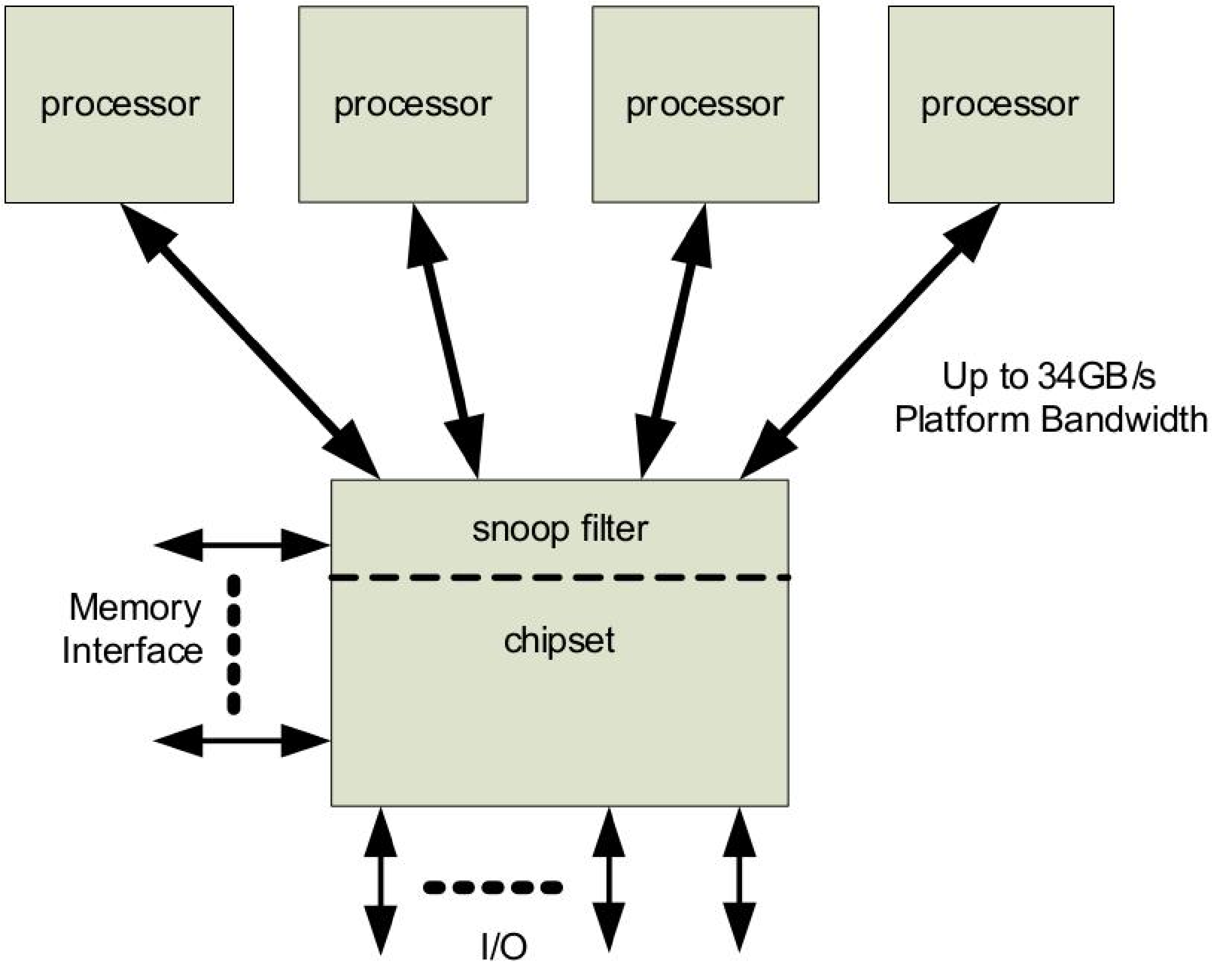


Dual Independent Buses (2005)

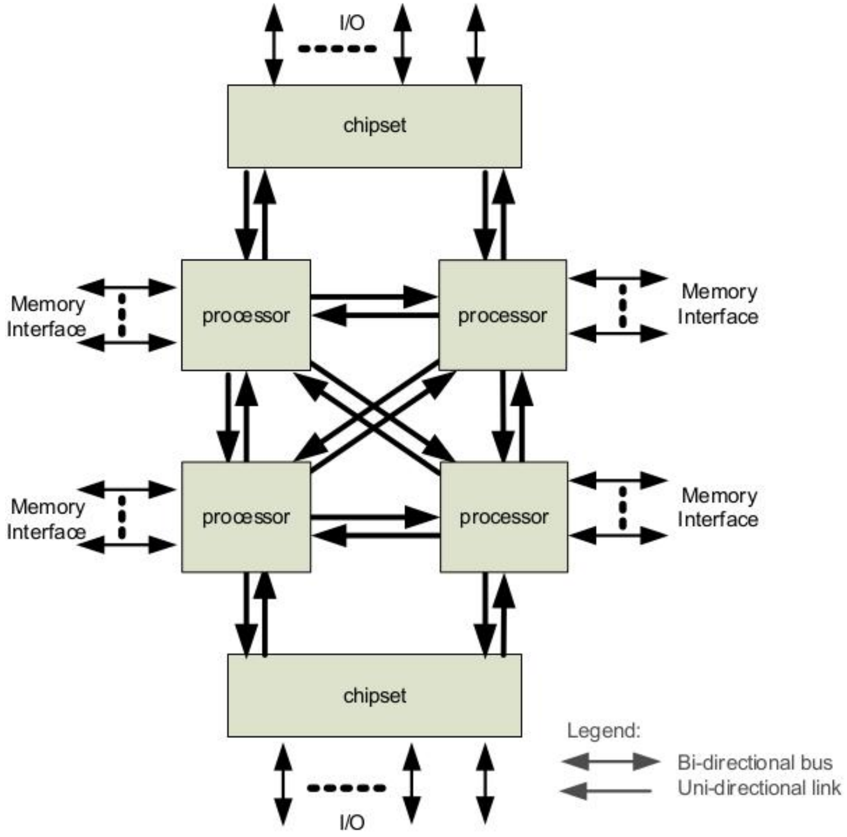
- The single bus is split into two separate buffers
- This doubles the available bandwidth, in principles

- All snoop traffic had to be broadcast on both buses
- This would reduce the effective bandwidth
  - Snoop filters are introduced in the chipset
  - They are used as a cache of snoop messages

Dedicated High-Speed Interconnects (2007)

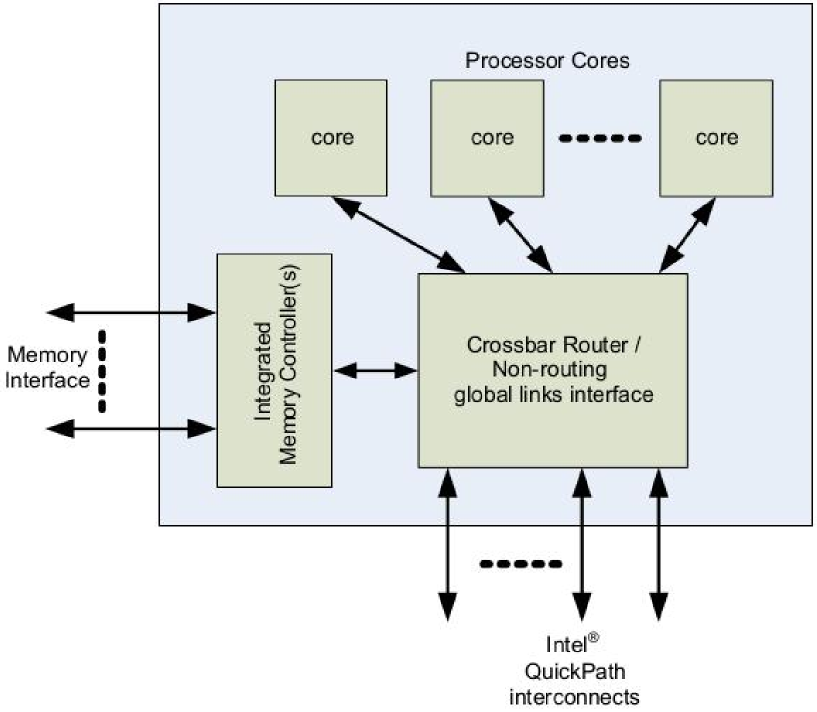


QuickPath Interconnect (2009)

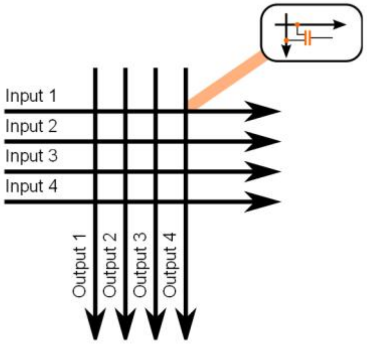


- Migration to a distributed shared memory architecture chipset
- Inter-CPU communication based on high-speed uni-directional point-to-point links
- Data can be sent across multiple lanes
- Transfers are packetized: data is broken into multiple transfers

QPI and Multicores

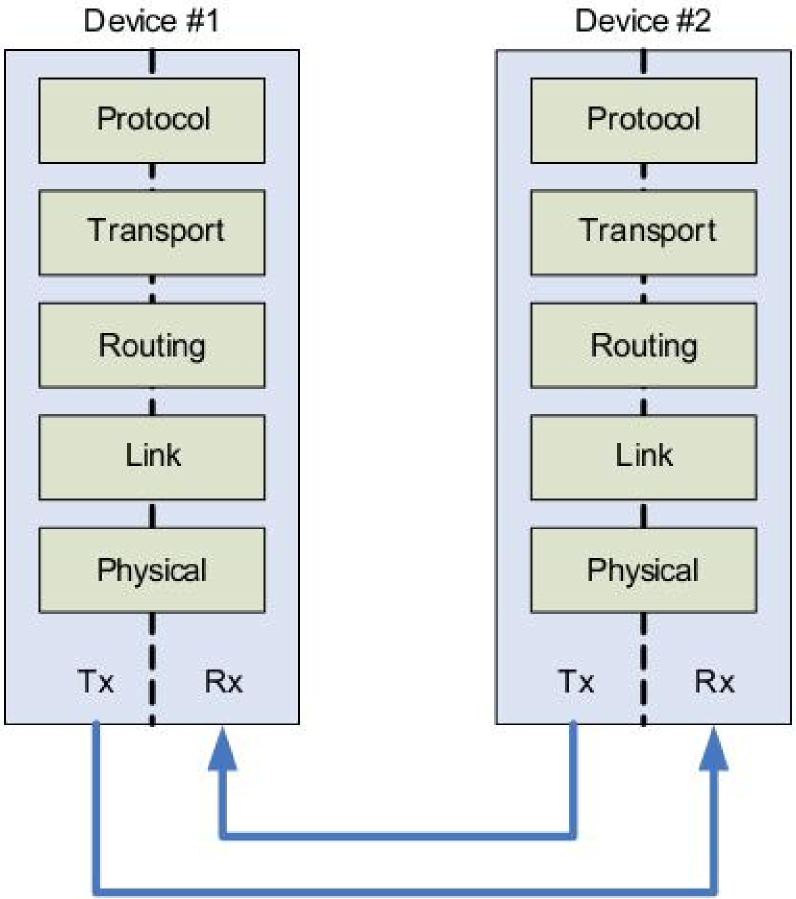


- The connection between a core and QPI is realized using a crossbar router:



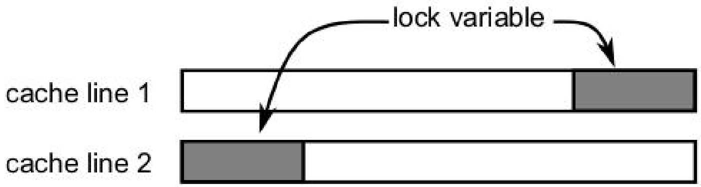
QPI Layers

- Each link is made of 20 signal pairs and a forwarded clock
- Each port has a link pair with two uni-directional link
- Traffic is supported simultaneously in both directions



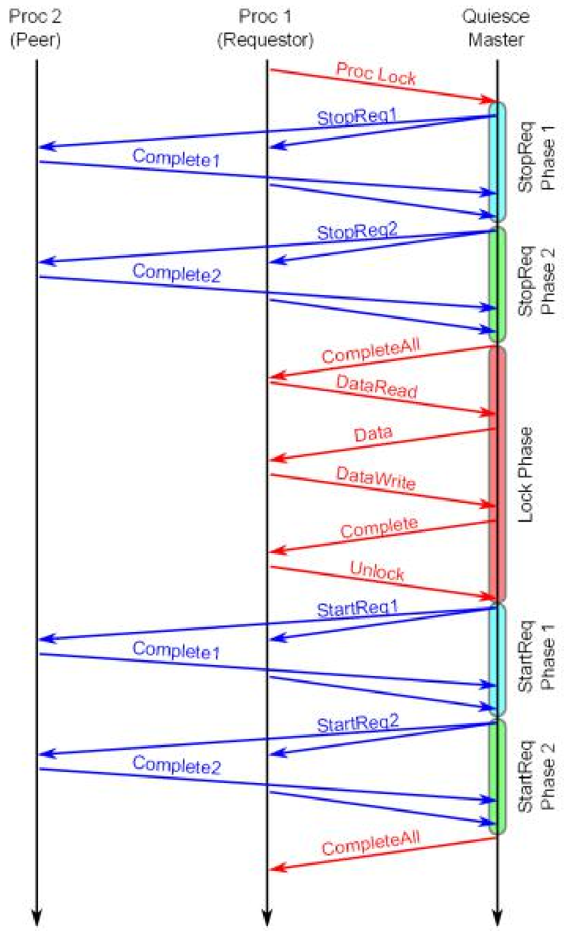
Once again on the `lock` prefix: split locks

For the P6 and more recent processor families, if the area of memory being locked during a LOCK operation is cached in the processor that is performing the LOCK operation as write-back memory and is completely contained in a cache line, the processor may not assert the LOCK# signal on the bus. Instead, it will modify the memory location internally and allow it's cache coherency mechanism to ensure that the operation is carried out atomically. This operation is called "cache locking." The cache coherency mechanism automatically prevents two or more processors that have cached the same area of memory from simultaneously modifying data in that area.



```
1:
      movl $1,%eax
      lock xchgl %eax, spinlock
      testl %eax, %eax
      jnz 1b
```

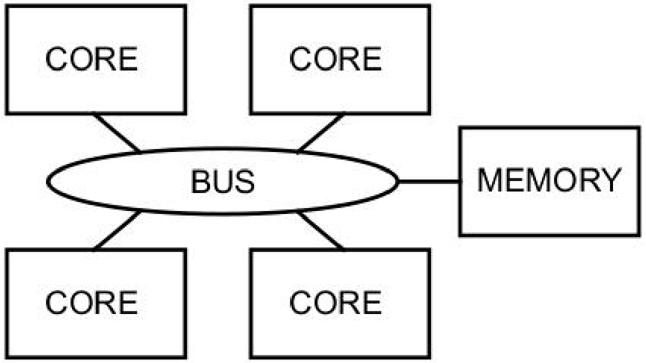
Once again on the `lock` prefix: split locks

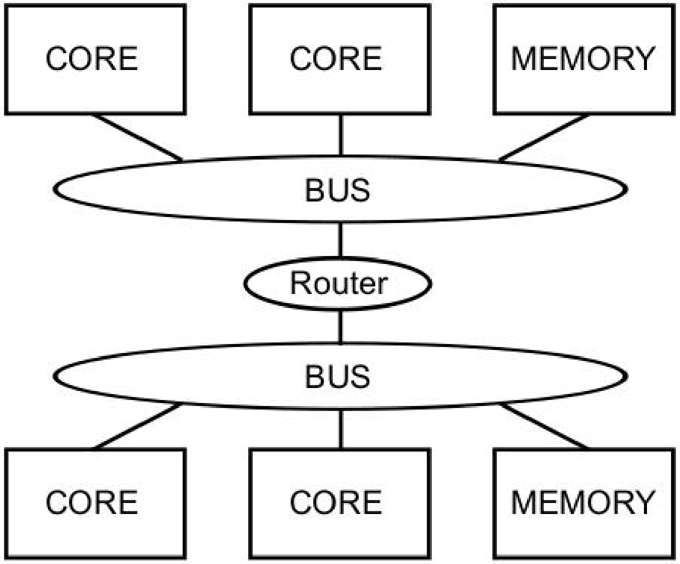


- If two processors defer snoops on both cache lines, the system might deadlock
- They instead compete to acquire the bus lock, which serializes the two transactions
- QPI designates a Quiesce Master for the entire system (an arbiter)
- A processor issuing a LOCK# signal asks the QM to force stopping all transactions
- This entails DMA transactions as well
- Draining all transactions from the system has severe performance implications
- Imagine this sledge diagram embedded into a loop!

UMA vs NUMA

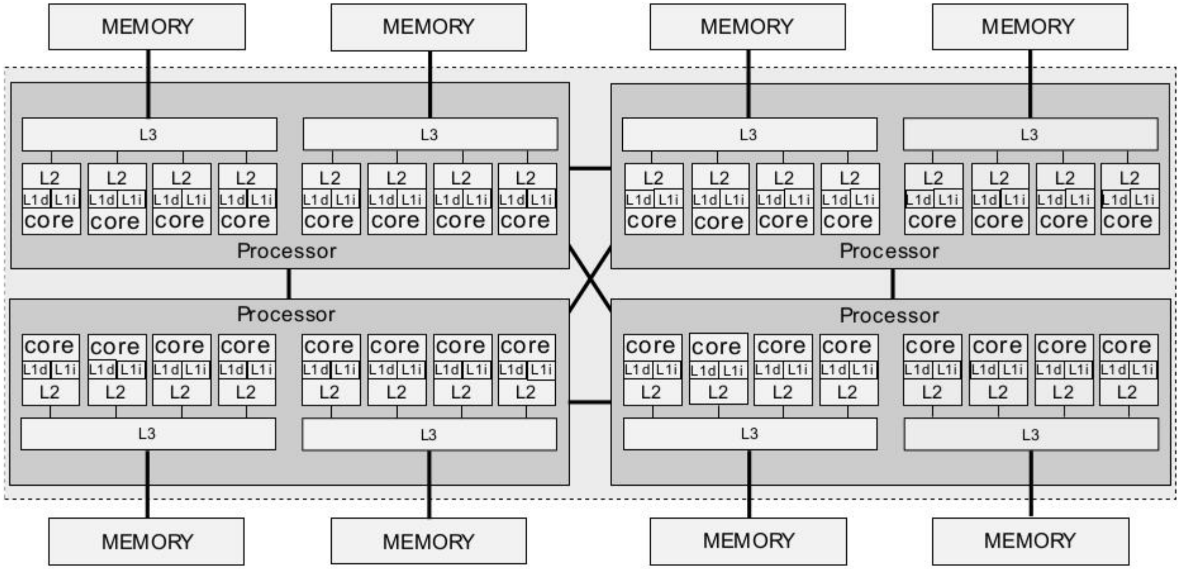
- In Symmetric Multiprocessing (SMP) Systems, a single memory controller is shared among all CPUs (Uniform Memory Access—UMA)
- To scale more, Non-Uniform Memory Architectures (NUMA) implement multiple buses and memory controllers





Non-Uniform Memory Access

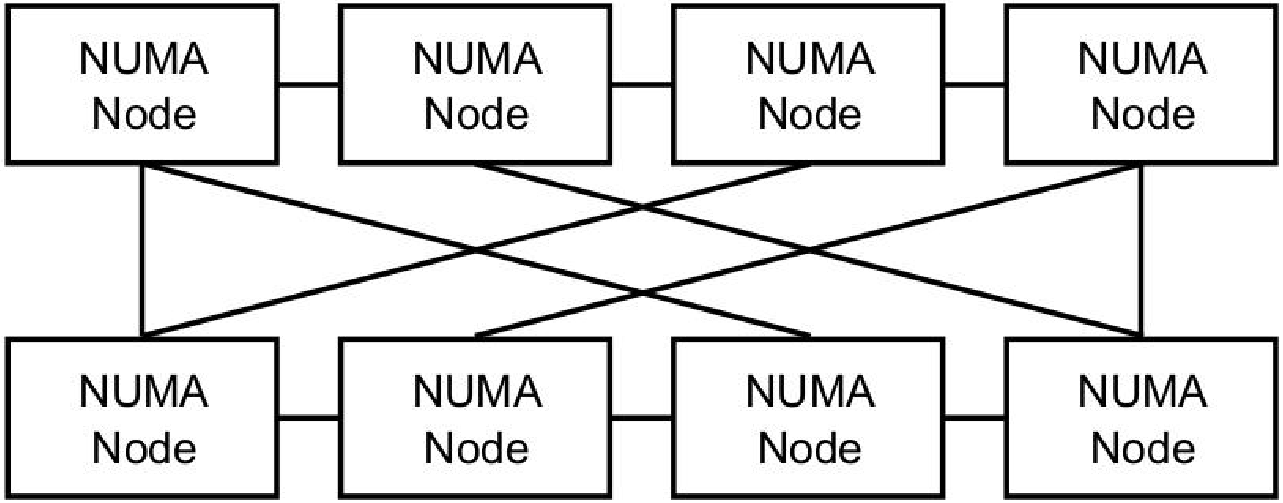
- Each CPU has its own local memory which is accessed faster
- Shared memory is the union of local memories
- The latency to access remote memory depends on the 'distance'



[NUMA organization with 4 AMD Opteron 6128 (2010)]

Non-Uniform Memory Access

- A processor (made of multiple cores) and the memory local to it form a *NUMA node*
- There are commodity systems which are not fully meshed: remote nodes can be only accessed with multiple hops
- The effect of a hop on commodity systems has been shown to produce a performance degradation of even 100%—but it can be even higher with increased load on the interconnect



Linux and NUMA

- Linux is NUMA-aware
  - Support started in 2004 (Linux 2.6)
- Two optimization areas (we will come back later on both):
  - Thread scheduling
  - Memory allocation
- User space support:
  - `libnuma`

`libnuma`

- This library offers an abstracted interface
- It is the preferred way to interact with a NUMA-aware kernel
- Requires `#include<numa.h>` and linking with `–lnuma`
- Some symbols are exposed through `numaif.h`

- `numactl` is a command line tool to run processes with a specific NUMA policy without changing the code, or to gather information on the NUMA system

`numactl`

- `numactl --cpubin=0 --membind=0,1 program`
  - Run program on CPUs of node 0 and allocate memory from nodes 0 and 1
- `numactl --preferred=1 numactl --show`
  - Allocate memory preferably from node 1 and show the resulting state
- `numactl --interleave=all program`
  - Run program with memory interleaved over all available nodes
- `numactl --offset=1G --length=1G --membind=1 --file /dev/shm/A --touch`
  - Bind the second gigabyte in the tempfs file `/dev/shm/A` to node 1
- `numactl --localalloc /dev/shm/file`
  - Reset the policy for the shared memory file /dev/shm/file
- `numactl --hardware`
  - Print an overview of the available nodes