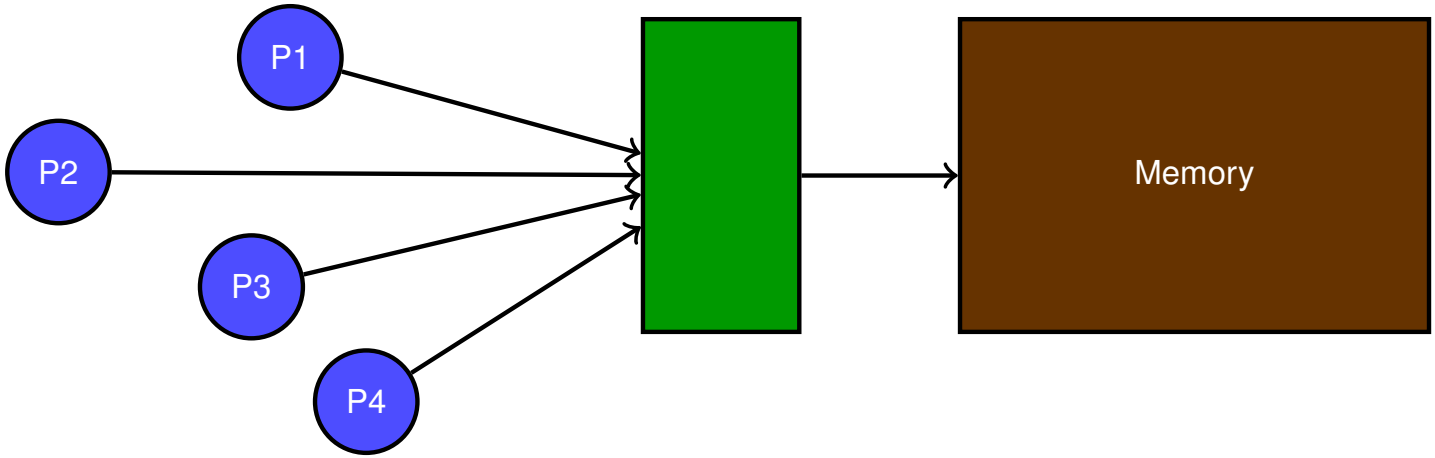
Memory consistency models

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

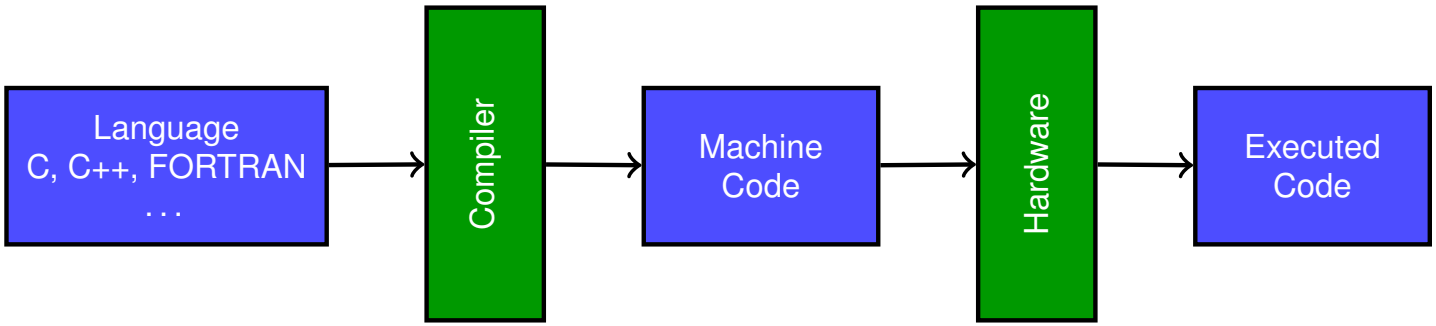
Memory consistency



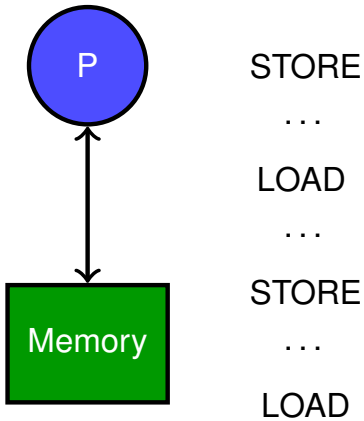
- **Memory consistency model**:
  - Set of rules defining how the **memory system** processes memory operations from **multiple processors**.
  - **Contract** between programmer and system.
  - Determines which **optimizations are valid** on correct programs.

Memory model

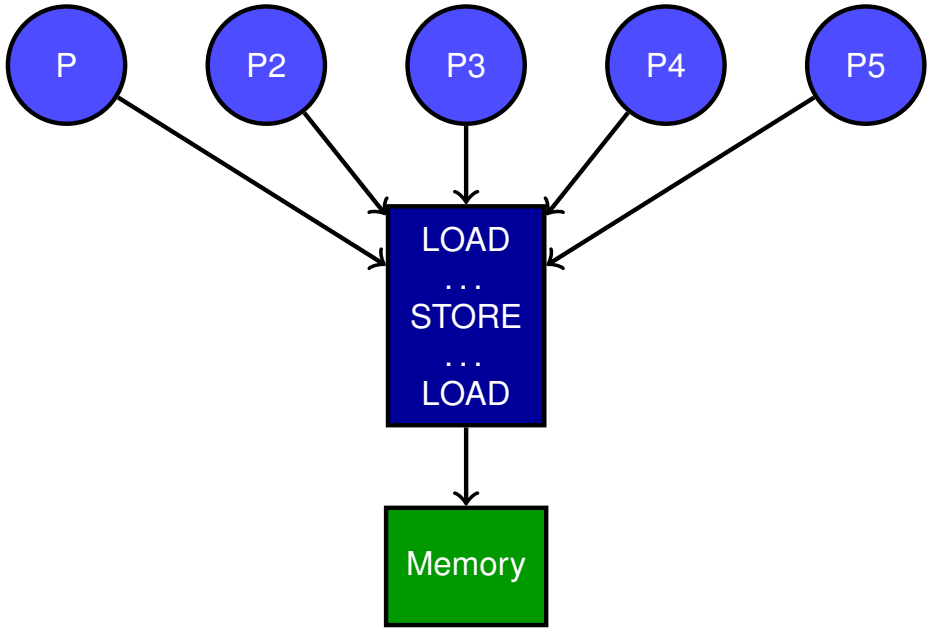
- Interface between program and its transformers.
  - Defines which values can be returned by a read operation.
- The language's memory model has implications for hardware.



Single processor memory model



- **Memory behavior model**:
  - Memory operations happen in **program order**.
    - A read returns the value from the last write in program order.
- Semantics defined by **sequential program order**:
  - **Simple** but **constrained** reasoning.
    - Solve **data and control dependencies**.
  - **Independent** operations may be executed in **parallel**.
  - Optimizations **preserve semantics**.



A multiprocessor system is **sequentially consistent** if the result of any execution is the same that would be obtained if operations from all processors were executed in some sequential order, and operations from each individual processor appear in that sequence in the order established by the program.

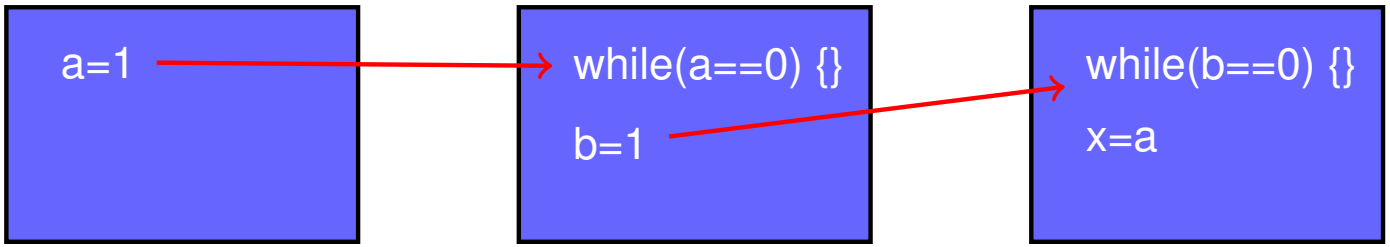
Leslie Lamport, 1979

Sequential Consistency: Constraints

- **Program order**.
  - Memory operations from a program must be made visible to **all processes** in **program order**.

- **Atomicity**.
  - Total execution order between processes must be **consistent** requiring that all operations are **atomic**.
    - All the operations that a processor does after it has seen the new value of a write are not visible to other processes until they have seen the value from that write.

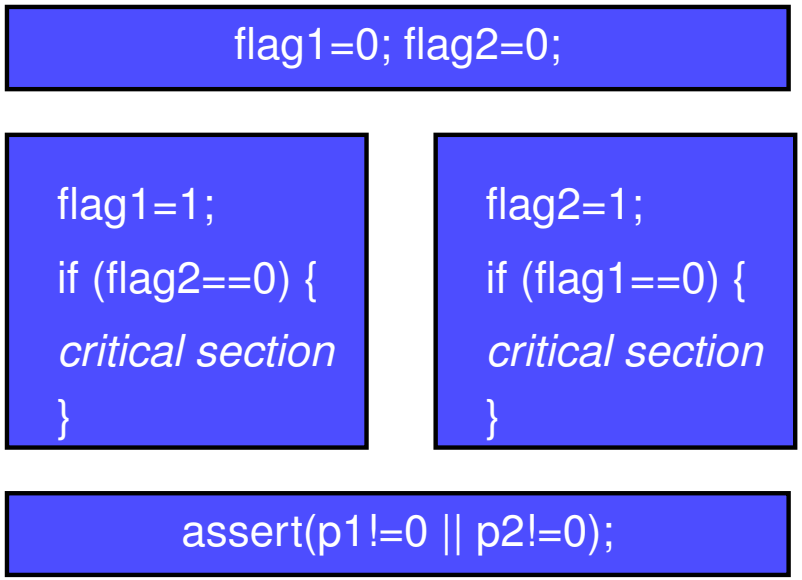
Atomicity



- **Non atomic writes**:
  - Write on **b** could bypass to **while** loop and read from **a** would bypass the write.
    - **X=0**.
- **Atomic writes**:
  - **Sequential consistency is preserved**.

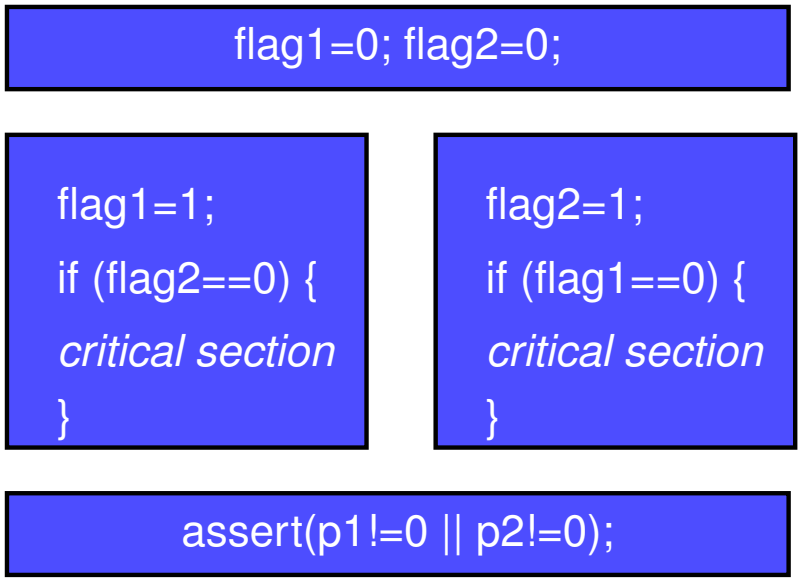
- **Sequential consistency constraints** all memory operations:
  - Write $\rightarrow$ Read.
  - Write $\rightarrow$ Write.
  - Read $\rightarrow$ Read, Read $\rightarrow$ Write.
- **Simple model** to reason about parallel programs.
- But, simple single processor reorderings may **violate sequential consistency** model:
  - **Hardware reordering** to improve performance.
    - Write buffers, overlapped writes, ...
  - **Compiler optimizations** apply transformations with memory operations reordering.
    - Scalar replacement, register allocation, instruction scheduling, ...
  - **Transformations** by programmers, or **refactoring tools** also modify program semantics.

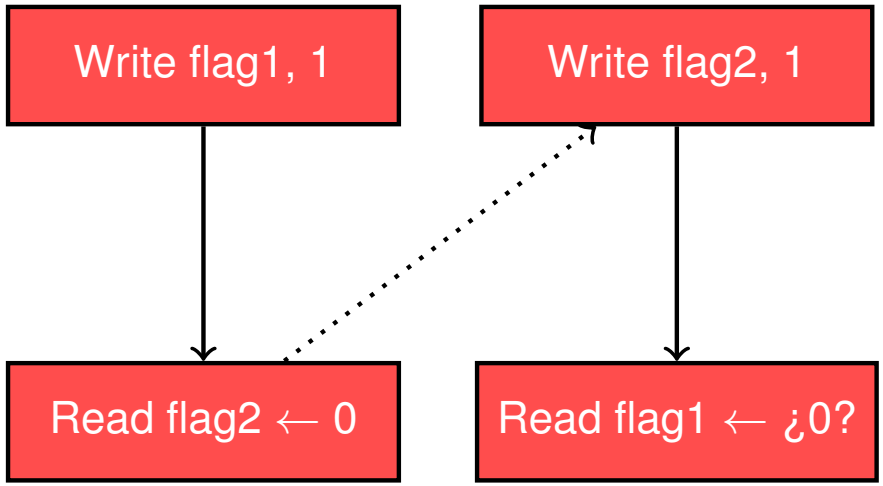
Sequential consistency violation



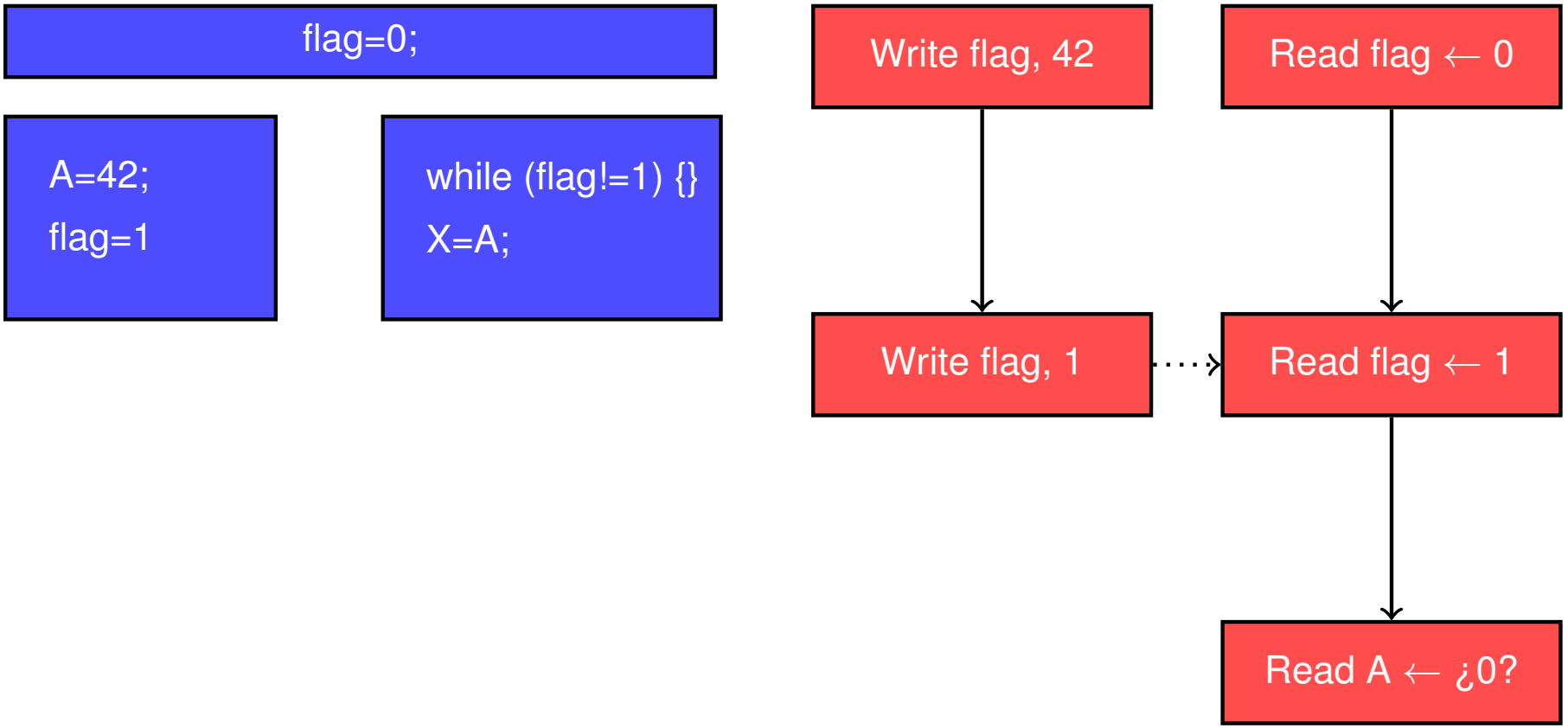
- If caches use a **write buffer**:
  - Writes are **delayed** in buffer.
  - Reads **obtain the old value**.
  - **Dekker Algorithm** is no longer **valid**.
    - **Dekker algorithm** is the first known solution to the mutual exclusion problem.

Program order





Program order



Conditions for sequential consistency

- **Sufficient conditions**:
  - Each process **issues memory operations** in program order.
  - After **issuing a write**, the process that performed the issue **waits for completions** of write **before issuing** another operation.
  - After **issuing a read**, the process that performed the issue **waits for completion** of read and for **completion** of the write of the value being read.
    - Wait for write propagation to all processes.
- **Very demanding conditions.**
  - There might be necessary conditions that are less demanding.

Optimizations

- Models relaxing program execution order.
  - W $\rightarrow$ R.
  - W $\rightarrow$ W.
  - R $\rightarrow$ W, W $\rightarrow$ W.

- **Notation**:
  - X $\rightarrow$ Y
    - Y bypasses X.

Reorderings

Processor	R → R	R → W	W → R	W → W
Alpha	✓	✓	✓	✓
PA-RISC	✓	✓	✓	✓
POWER	✓	✓	✓	✓
SPARC				✓
x86				✓
AMD64				✓
IA64	✓	✓	✓	✓
zSeries				✓

Reads bypass writes (W→R)

- A **read may execute before** a preceding **write**.

- Typical in systems with **write buffer**.
  - Check consistency with buffer.
  - Allow read buffer.

Other models

- $R \rightarrow W$, $W \rightarrow R$.
  - Allow that **writes may arrive into memory** out of program order.

- $R \rightarrow W$, $W \rightarrow R$, $R \rightarrow R$, $W \rightarrow W$.
  - Avoid only data and control dependencies within processor.
  - **Alternatives**:
    - Weak consistency.
    - Release consistency.

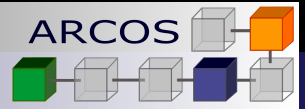
Weak ordering

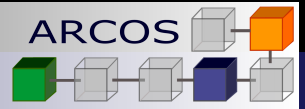
- Divides memory operations into **data operations** and **synchronization operations**.
- **Synchronization operations** act as a **barrier**.
  1. All preceding data operations in program order to a synchronization must complete before synchronization is executed.
  2. All subsequent data operations in program order to a synchronization operation must wait until synchronization ins completed.
  3. Synchronization are performed in program order.
- Hardware implementation of **barrier**.
  - **Processor keeps a counter**:
    - Data operation **issue** $\Rightarrow$ **increment**.
    - Data operation **completed** $\Rightarrow$ **decrement**.

Release/acquire consistency

- **More relaxed** than weak consistency.
- **Synchronization accesses** divided into:
  - **Acquire**.
  - **Release**.

- **Semantics**:
  - **Acquire**
    - Must complete before all subsequent memory accesses.
  - **Release**
    - Must complete all previous memory accesses.
    - Subsequent memory accesses **MAY** initiate.
    - Operations following a **release** and must wait, must be protected with an **acquire**.





4 Use case: Intel

- Consistency model
- Examples
- Model effects

Memory consistency in Intel

- Until 2005 hand not completely clarified its **memory consistency model**.
  - Formalizing the model highly complex.
  - Problems for language implementations (Java, C++, . . .).
- Currently the model is clarified and public.

Initial Intel model

- **i486** and **Pentium**:
  - Operations in program order.
    - **Exception**: Read misses bypass writes in *write buffer* only if all writes are cache hits.
    - It is impossible that a read miss matches with a write.

Atomic operations

- Since **i486**:
  - Read or write 1 byte.
  - Read or write a 16-bit aligned word.
  - Read or write a 32-bit aligned double word.
- Since **Pentium**:
  - Read or write a 64-bit aligned quadword.
  - Non-cached memory access that fits in 32 bit data bus.
- Since **P6**:
  - Non aligned access to data of 16, 32 or 64 bits that fit in a cache line.

Bus blocking (I)

- A processor may issue a **signal to block** the bus.
  - Other elements **cannot access** the bus.

- **Automatic bus blocking**:
  - Instruction **XCHG**.
  - Updating **segment descriptors**, **page directory**, and **page table**.
  - Interrupt acceptance.

Bus blocking (II)

- **Bus software blocking**:
  - Use **LOCK** prefix in:
    - Instructions for bit checking and modification (**BTS**, **BTR**, **BTC**).
    - Exchange instructions (**XADD**, **CMPXCHG**, **CMPXCHG8B**).
    - 1 operand arithmetic instructions (**INC**, **DEC**, **NOT**, **NEG**).
    - 2 operand arithmetic-logic instructions (**ADD**, **ADC**, **SUB**, **SBB**, **AND**, **OR**, **XOR**).

Barrier instructions

- **LFENCE**:
  - Barrier for **load operations**.
  - Every **load preceding** a **LFENCE** is **globally made visible** before any **subsequent load**.
- **SFENCE**:
  - Barrier for **store operations**.
  - Every **store preceding** a **SFENCE** is **globally visible** before any **subsequent store**.
- **MFENCE**:
  - Barrier for **load/store operations**.
  - All **load and store preceding** a **MFENCE** are **globally visible** before any **subsequent load or store**.

Current memory model within processor (I)

- Reads do not bypass other reads (R $\rightarrow$ R).
- Writes do not bypass reads (R $\rightarrow$ W).
- Writes do not bypass writes (W $\rightarrow$ W).
  - There are exceptions for strings and non-temporal moves.
- Reads bypass preceding writes (W $\rightarrow$ R) to different addresses.
- Reads/writes do not bypass I/O operations, locked instructions, or serializing instructions.

Current memory model within processor (II)

- Reads cannot bypass preceding **LFENCE** or **MFENCE**.
- Reads cannot bypass preceding **LFENCE**, **SFENCE**, or **MFENCE**.
- **LFENCE** cannot bypass preceding read.
- **SFENCE** cannot bypass preceding write.
- **MFENCE** cannot bypass preceding read or write.

Multiprocessor memory model

- Every processor is individually compliant with former rules.
- Writes from a processor are observed in the same order by all other processors.
- Writes from a processor are **NOT** ordered with respect to writes from other processors.
- Memory ordering is transitive.
- Two writes are viewed in a consistent order by any other processor distinct from those two processors.
- Lock instructions have a total order.

Example: Write ordering

Processor A	Processor B	Processor C
write A.1	write B.1	write C.1
write A.2	write B.2	write C.2
write A.3	write B.3	write C.3

- Writes from every processor keep order.

Possible order (I)	Possible order (II)
Write A.1	...
Write B.1	Write B.3
Write B.2	Write A.3
Write C.1	Write C.2
Write A.2	Write C.3

- Order for every process is kept.
- No order is guaranteed across processes.

No reordering R→R,W→W

Initial state

X=0, Y=0

Processor 1

MOV [_x], 1
MOV [_y], 1

Processor 2

MOV r1, [_y]
MOV r2, [_x]

State **not allowed**

r1=1 y r2=0

No reordering R→W

Initial state

X=0, Y=0

Processor 1

MOV r1, [_x]
MOV [_y], 1

Processor 2

MOV r2, [_x]
MOV [_x], 1

State **not allowed**

r1=1 y r2=1

Reordering W(a)→R(b)

Initial state

X=0, Y=0

Processor 1

MOV [_x], 1
MOV r1, [_y]

Processor 2

MOV [_y], 1
MOV r2, [_x]

State **allowed**

r1=0 y r2=0

No reordering W→R

Initial state

X=0

Processor 1

MOV [_x], 1
MOV r1, [_x]

State **not allowed**

r1=0

Write visibility from other processor

Initial state

X=0, Y=0

Processor 1

```
MOV [_x], 1
MOV r1, [_x]
MOV r2, [_y]
```

Processor 2

```
MOV [_y], 1
MOV r3, [_y]
MOV r4, [_x]
```

State **allowed**

r2=0 y r4=0

Writes may be perceived in different order by every processor.

Transitive visibility of writes

Initial state

X=0, Y=0

Processor 1

```
MOV [_x], 1
```

Processor 2

```
MOV r1, [_x]
MOV [_y], 1
```

Processor 3

```
MOV r2, [_y]
MOV r3, [_x]
```

State **not allowed**

r1=1 y r2=1 y r3=0

Consistent order of writes for all processors

Initial state

X=0, Y=0

Processor 1	Processor 2	Processor 3	Processor 4
MOV [_x], 1	**MOV** [_y], 1	**MOV** r1, [_x] **MOV** r2, [_y]	**MOV** r3, [_y] **MOV** r4, [_x]

State **not allowed**

r1=1 y r2=0 y r3=1 y r4=0

Locked instructions define total order

Initial state

r1=1, r2=1, X=0, Y=0

Processor 1	Processor 2	Processor 3	Processor 4
XCHG [_X], r1	**XCHG** [_y], r2	**MOV** r3, [_x] **MOV** r4, [_y]	**MOV** r5, [_y] **MOV** r6, [_x]

State **not allowed**

r1=1 y r2=0 y r3=1 y r4=0

Reads not reordered with locks

Initial state

X=0, Y=0, r1=1, r3=1

Processor 1

XCHG [_x], r1
MOV r2, [_y]

Processor 2

XCHG [_y], r3
MOV r4, [_x]

State **not allowed**

r2=0 y r4=0

Writes not reordered with locks

Initial state

X=0, Y=0, r1=1

Processor 1

XCHG [_x], r1
MOV [_y], r1

Processor 2

MOV r2, [_y]
MOV r3, [_x]

State **not allowed**

r2=1 y r3=0

4 Use case: Intel

- Consistency model
- Examples
- Model effects

Consistency models in Intel

- **Sequential consistency**
  - **Load**: **mov reg, [mem]**
  - **Store**: **xchg [mem], reg**
- **Relaxed consistency**
  - **Load**: **mov reg, [mem]**
  - **Store**: **mov [mem], reg**
- **Release/acquire consistency**
  - **Load**: **mov reg, [mem]**
  - **Store**: **mov [mem], reg**

Summary

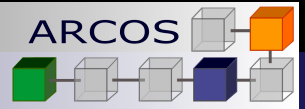
- Consistency memory model determines which optimizations are valid.
- **Sequential consistency** establishes as constraints **atomicity** and **program order**.
- More relaxed models than sequential consistency can be used.
  - **Weak consistency**.
  - **Release/acquire consistency**
- Intel memory model has evolved over last decade.
  - Formalized and publicly available.
  - Establishes what operations are atomic, when bus is blocked, and how barriers are defined.
  - Defines the memory model within processor and between different processors.

References

- **Computer Architecture. A Quantitative Approach.**
  5th Ed.
  Hennessy and Patterson.
  Sections: 5.6

- **Shared memory consistency models: A tutorial.**
  Adve, S. V., and Gharachorloo, K.
  IEEE Computer 29, 12 (December 1996), 66-76.

- **Intel 64 and IA-32 Architectures Software Developer Manuals.**
  Volume 3: Systems Programming Guide.
  8.2: Memory Ordering



Memory consistency models

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid