TU WIEN Informatics

E3. Advanced Computer Architecture

Memory Models

Daniel Mueller-Gritschneder (First version slides: Johann Blieberger)

Content

- Interleaving Threads & Interleaving Graph
- Program & Execution Order
- Atomics
- Synchronization and Communication
- Release/Acquire Memory Model
- Blocking Wait
- Non-Blocking Wait
- Performance Comparison
- Summary

E3.1. Sequential Consistency (SC) & Synchronization Problem

SC & Synchronization

Already discussed: **Cache Coherency**

- Cache Coherency Controller establishes a coherent view across the private caches.

Next: **Sequential Consistency (SC*) & Synchronization Problem**

- **Memory Model**: Need mechanisms to ensure that accesses of one processor appear to execute in program order to all other, at least partly.
- **Atomic Operations**: HW-support for synchronization

*Attention: The abbreviation SC stands here for **Sequentially Consistent**
Later in the slides the abbreviation SC will be reused for store conditional

E3. 2Abstract View on Interleaving Threads

Interleavings

Interleavings are all possible intertwinings of sequences of statements from threads.
Example: T1: (a, c), T2: (b, d)
All possible interleavings are:

(a, c, b, d), (a, b, c, d), (a, b, d, c), (b, a, c, d), (b, a, d, c), (b, d, a, c).
The “local” orders a < c and b < d are preserved.

Interleavings graph is a representation of interleavings in form of a graph.

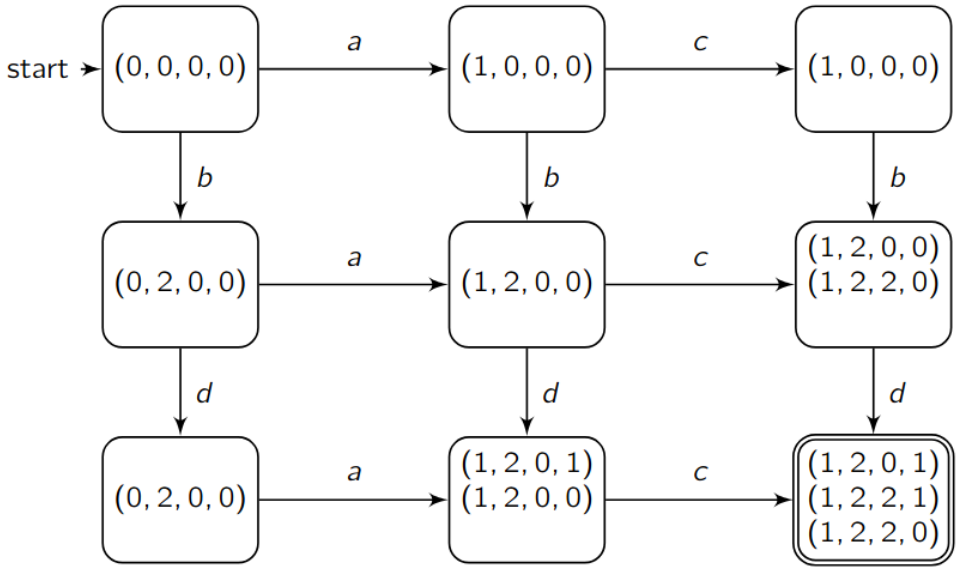
- Each path from the start node to the end node of the graph corresponds to an interleaving.
- The set of all such paths corresponds to the set of all possible interleavings. (Examples follow . . .)

Due to different runtimes, different scheduling strategies, different hardware architectures, the actual execution sequence can match any arbitrary interleaving.

For general considerations (correctness of a program, . . .) one must therefore assume all interleavings as possible.

Introductory Example (with Interleavings Graph)

Thread 1	$(A, B, C, D) = (0, 0, 0, 0)$	**Thread 2**
a: `A := 1;`		b: `B := 2;`
c: `C := B;`		d: `D := A;`



Introductory Example

A **Race Condition** is a situation in which the result of an operation depends on the temporally intertwined execution of certain other operations.

Implicit Assumption:

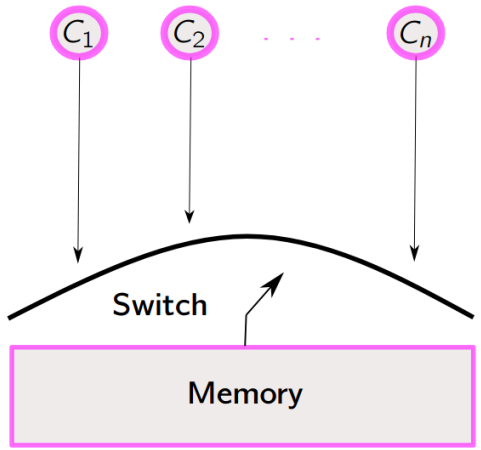
- Assignments occur **atomically**
- Only then are the interleavings correct

Question: Why is the result (1, 2, 0, 0) not possible?

Answer:

- Instructions are executed in each thread in “program order”, i.e. from front to back.
- So, if (1, 2, 0, .) occurs, only instruction `d` is missing.
- Instruction d can only deliver *D*=*A*=1, since `a` was executed before `c`.

Sequential Consistency from the Programmer's Perspective



- A single global **memory**
- Each **core** generates **memory operations in program order**
- At an indeterminate point in time, a **switch** randomly selects a core and executes a memory operation (→ **"memory order"**)
- The switch serializes the memory operations

  Note: This is not what the hardware does! But it can serve as a model for how we want to think about hardware.

E3.3 Atomic Instructions and Variables

Atomics

- Assumption: The assignment of a 16-bit word occurs **non-atomically**, by copying the two 8-bit halves separately.
- Given are two threads T_1 and T_2 and a variable S with the content S=0.
- T_1 shall write the value -1 in two's complement to S
- S := $(1111\ 1111\ 1111\ 1111)_2$.
- T_2 reads the value of S at a different time.
- One should expect that T_2 can only read 0 or -1.

- But the following can happen:
  - First, the 1st half is copied to S.
  - In the second half of S, there are still all 0s.
  - We get: S = $(1111\ 1111\ 0000\ 0000)_2$.
  - Before the second half is copied to S, T_2 reads both halves and gets S=-128.
- **Question:** If T_1 first copies the second half, and then T_2 reads both halves: What value does T_2 get?

Atomics

- We have seen: When multiple threads access common memory cells (variables), it may be necessary to guarantee that operations on variables are executed **atomically**, i.e., **indivisibly**.
- This can only be guaranteed by the **hardware** (CPU).
- All common CPUs offer such **atomic operations** as **instructions**.
  e.g. RISC-V: A- Extension
- Programming languages also know **atomic types** and **operations** on atomic variables
  e.g. Java: `AtomicInteger`
- When multiple threads access the same memory area (variable) simultaneously, this is called a **Data Race**.

. . . we will come back to this later. . .

Example Java Atomic Types

Java offers atomic types, e.g. `AtomicInteger`

This is feature of the programming language

- Java engine ensures that all operations on the variables declared with an atomic type are executed atomically (easy use for the programmer)
- May use atomic instructions from the processor ISA to implement this in a target platform.

Difference between volatile and atomic

- Volatile: all other threads see all accesses to variables (not optimized by compiler)
- Atomic: Additionally, the operations on these variables are atomic.

RISC-V A-Extension

Load-Reserved/Store-Conditional (LR/SC**) Instructions

- `LR.W`: LR loads a word from the address in rs1, places the sign-extended value in rd, and registers a reservation on the memory address..
- `SC.W`: SC writes a word in rs2 to the address in rs1, provided a valid reservation still exists on that address. SC writes zero to rd on success or a nonzero code on failure.

Reservation makes sure no other thread accessed the memory location between LR.W and SC.W.

We will see later how to use this for synchronization – it is also known as Load-Link/Store Conditional (LL/SC**)

Atomic Memory operation (AMO) instructions (Atomic Read-Modify-Write)

- AMO instructions load data value from an address in rs1, put that value into register rd, apply a binary operator to the loaded value and the original value in rs2, and then store the result back to the original address in rs1 **atomically.**
- Example Logic Or Operation: `AMOOR.W`

**Attention: The abbreviation SC stands here for store conditional

E3.4 Synchronization with Atomic Variable

Producer - Consumer

- Task: A piece of data should be safely transferred from one thread to another thread.
- More detailed: Thread T_1 writes to variable D, Thread T_2 shall read the value of variable D.
- Question: When may T_2 read?
- We introduce a flag F, which initially has the value F=0.
- T_1 writes D.
- T_1 sets flag F=1.
- T_2 reads F. If F=0, it continues to read F.
- When T_2 reads F=1, it can read D “safely”.

Producer - Consumer

As source code:

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

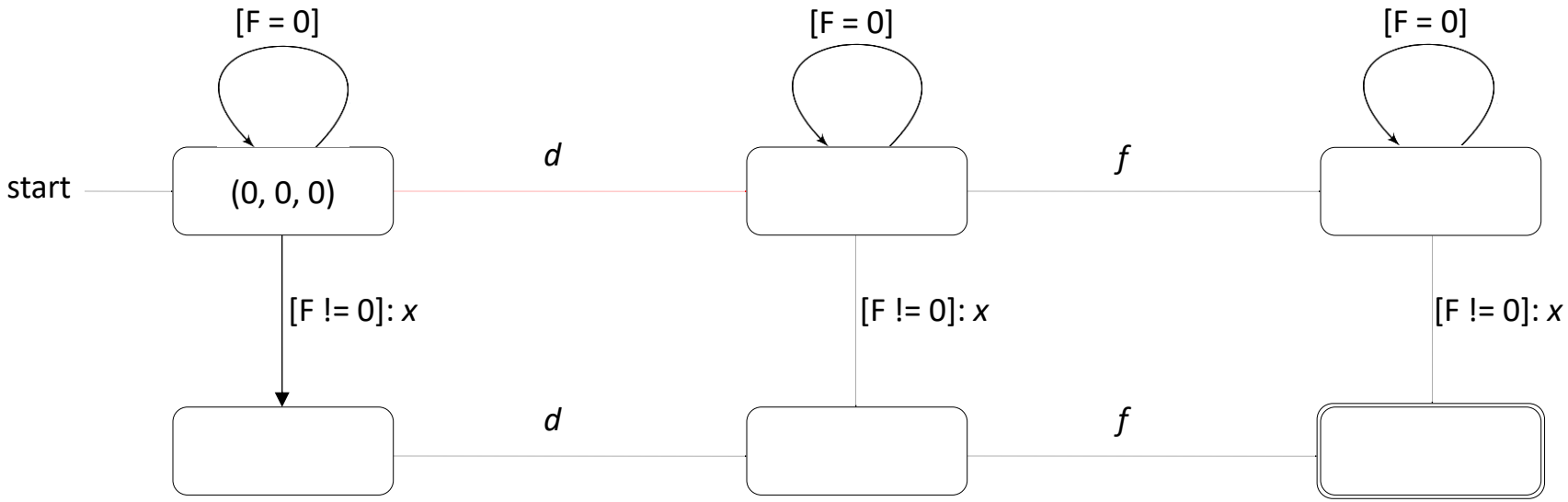
F … atomic!

D … atomic?

X … atomic?

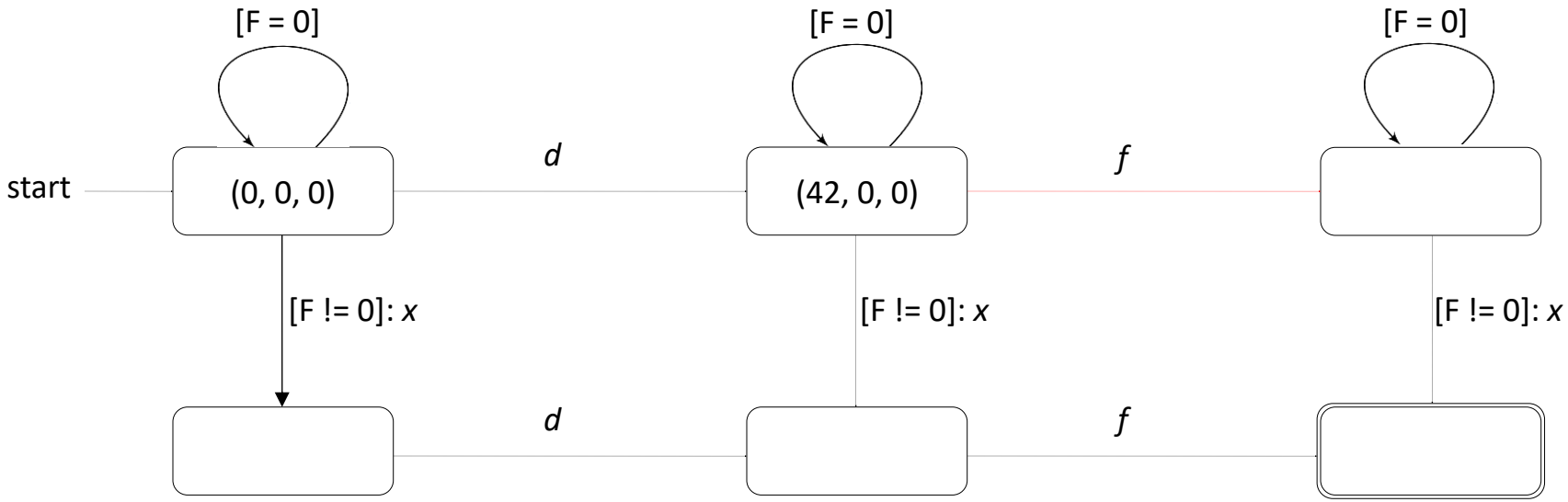
Producer - Consumer

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



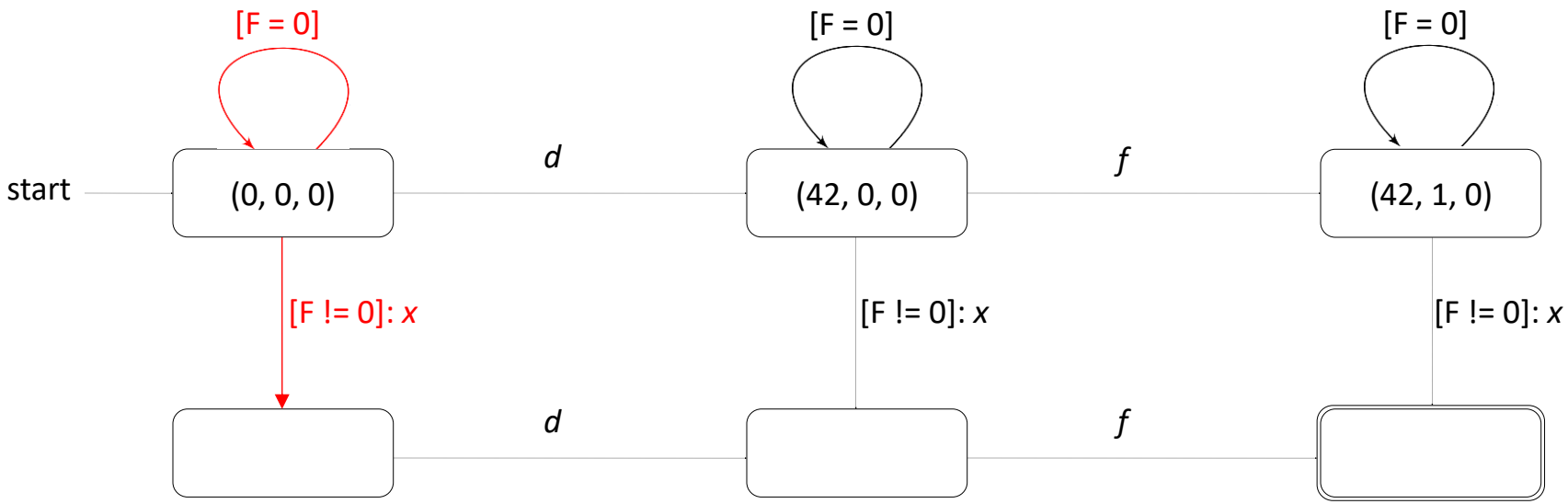
Producer - Consumer

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



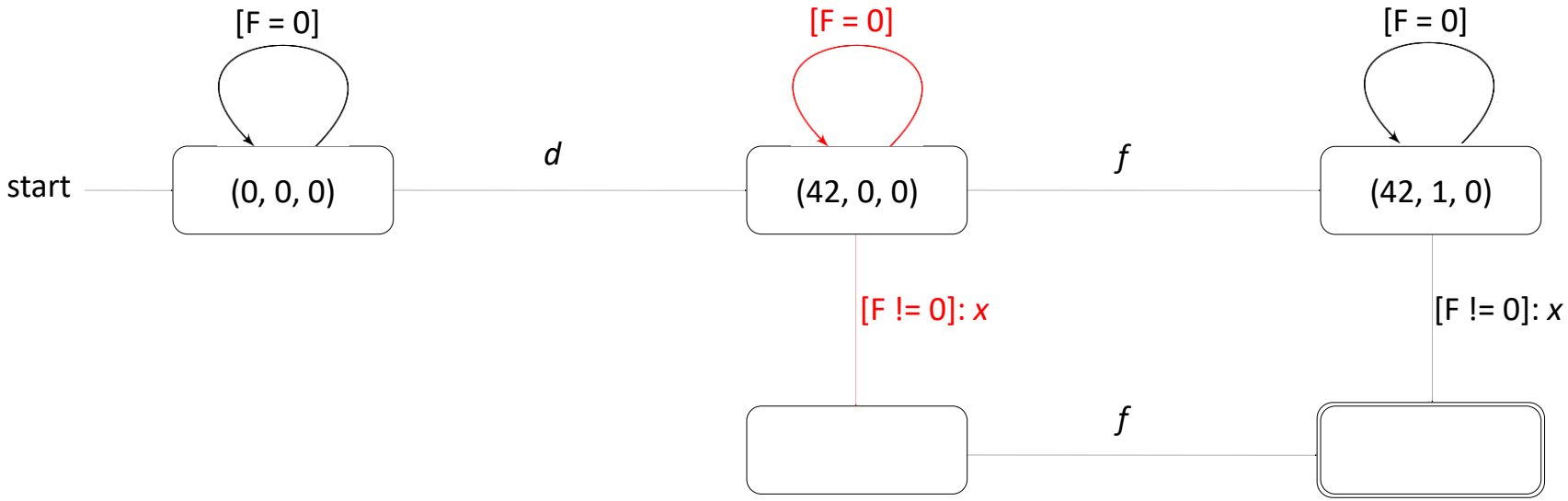
Producer - Consumer

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



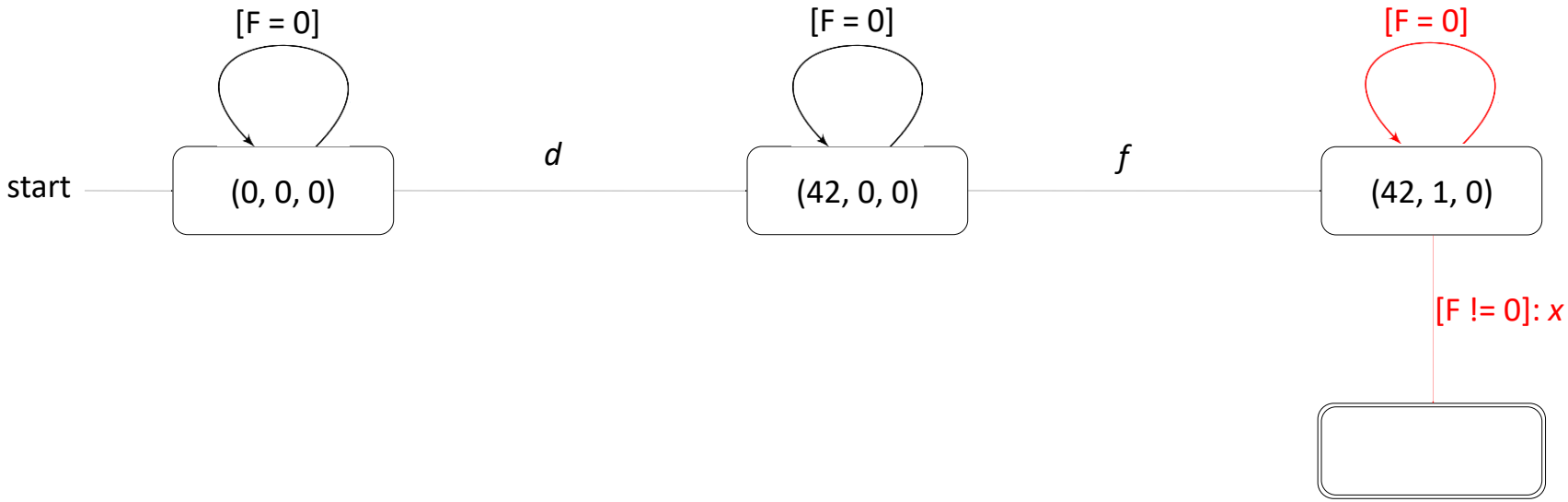
Producer - Consumer

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



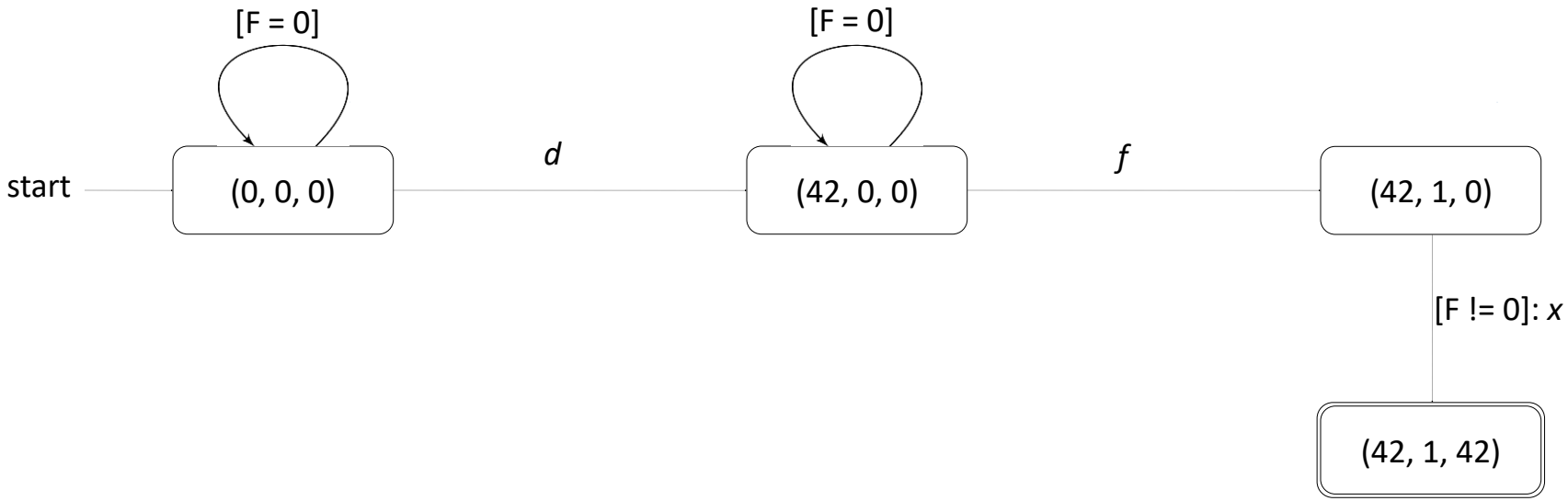
Producer - Consumer

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



Producer - Consumer

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



Producer - Consumer

- With the help of an **atomic** variable (flag F), data can be transferred “ safely” from one thread to another thread.
- **Synchronization**, **communication** between threads
- **Disadvantage:** Thread T_2 is in a **loop** until the flag is set. T_2 unnecessarily consumes computing time and energy.

. . . we will come back to this later. . .

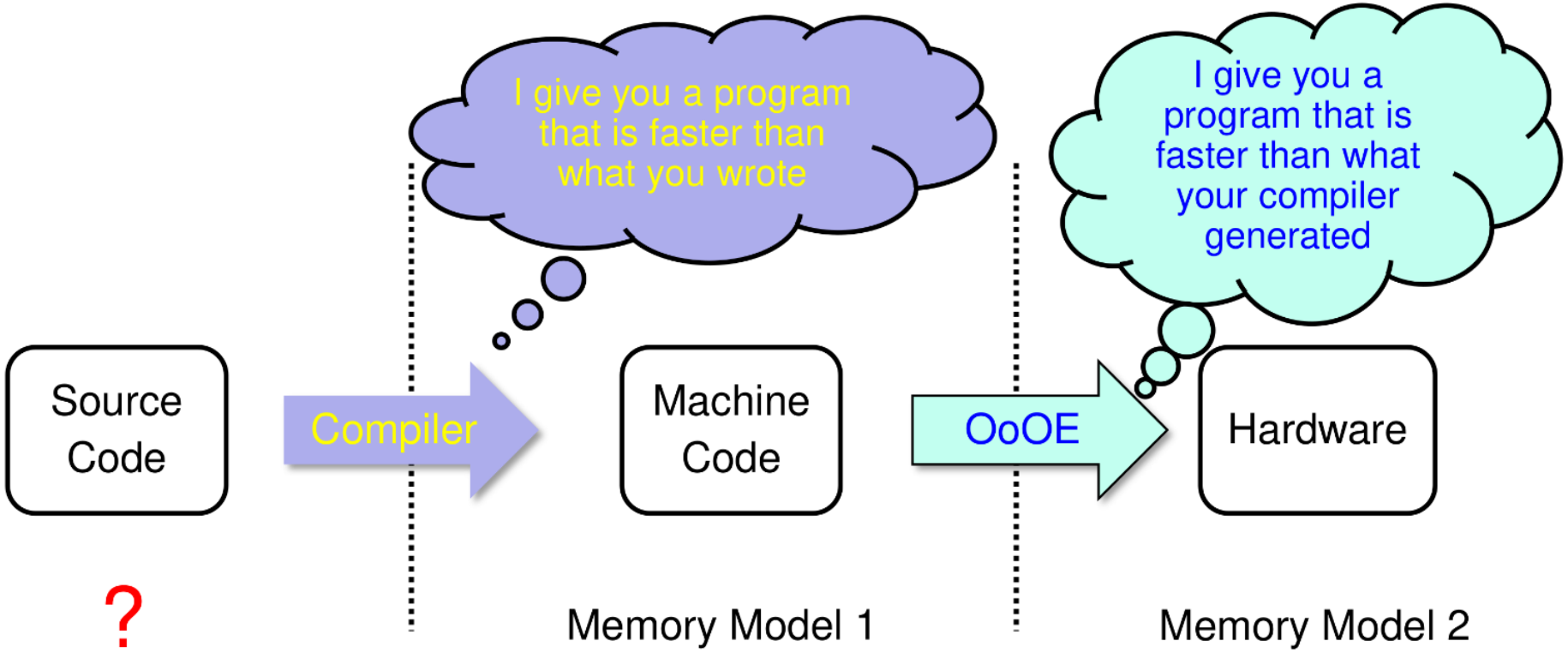
E3.5 Program vs. Execution Order in the Relaxed Memory Model

Program & Execution Order

- In addition to the atomic variables, executing the instructions in **program order** was recognized as a prerequisite for the Sequentially Consistent (SC*) memory model.
- Modern computer architectures do not guarantee executing the instructions in program order!

*Attention: The abbreviation SC stands here for **Sequentially Consistent**
Later in the slides the abbreviation SC will be reused for store conditional

Relaxed Memory Model: Compiler and OoO processor applies optimizations with reordering of instructions as for single-threaded execution



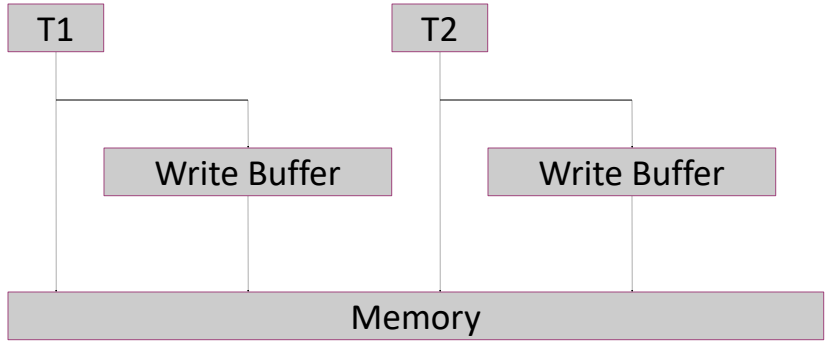
Program Execution Hierarchy

Hierarchy in program execution:

Level	**(Re-)Ordering**
Source Code	Program Order
Compiler	Optimization of the code (Moving and removing instructions)
CPU	Instruction Scheduling, Out-of-Order Execution
Memory	Write Buffer, Caches, ...
Execution	Execution Order

- The result of the computation must be the same before and after reordering for single-core computers, but not for multi-core computers.
- Programmers (Computer Scientists) must **know** and **consider** that
- **Program Order != Execution Order (PO != EO)**
- **Attention**: Instructions from the calling and called subroutine can be “mixed”.

SC Violation – Architecture without Caches



Flag1	0
Flag2	0

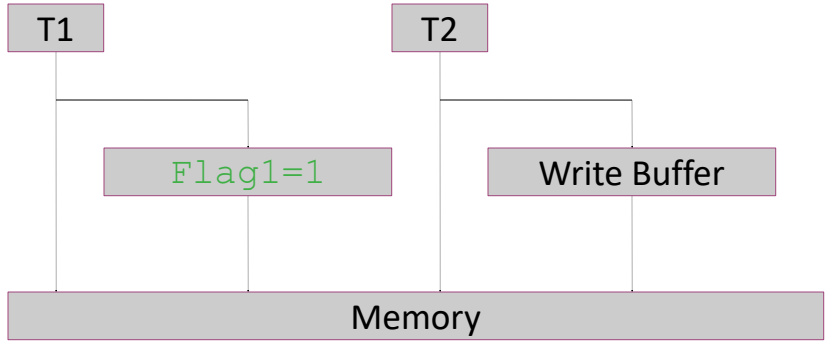
Initial: `Flag1 = Flag2 = 0`

T1:
```
Flag1 := 1;
if Flag2 = 0 then
-- critical
```

T2:
```
Flag2 := 1;
if Flag1 = 0 then
-- critical
```

- CPU-Cores have a **Write-Buffer**
- **Write** operations go into the Write-Buffer
- At an appropriate time, the Write-Buffer is transferred to memory
- **Advantage:** no waiting time until the written data actually arrives in memory
- **Attention: Read** operations can overtake write operations in the Write-Buffer

SC Violation – Architecture without Caches



Flag1	0
Flag2	0

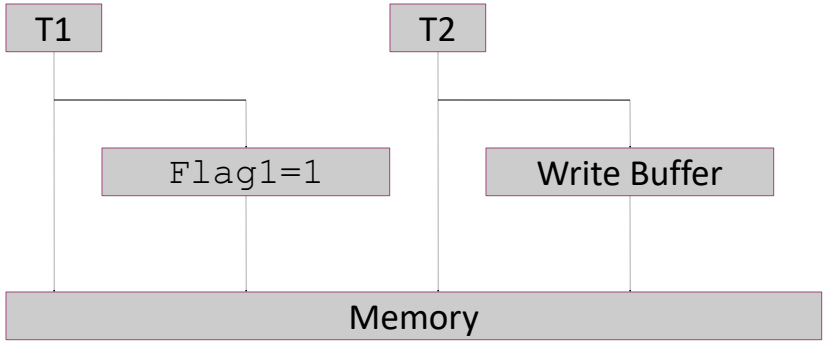
Initial: `Flag1 = Flag2 = 0`

T1:
```
Flag1 := 1;
if Flag2 = 0 then
-- critical
```

T2:
```
Flag2 := 1;
if Flag1 = 0 then
-- critical
```

- T1 write operation `Flag1 := 1` goes into the Write-Buffer
- There it cannot be seen by T2
- In memory, `Flag1` still has value 0

SC Violation – Architecture without Caches



Flag1	0
Flag2	0

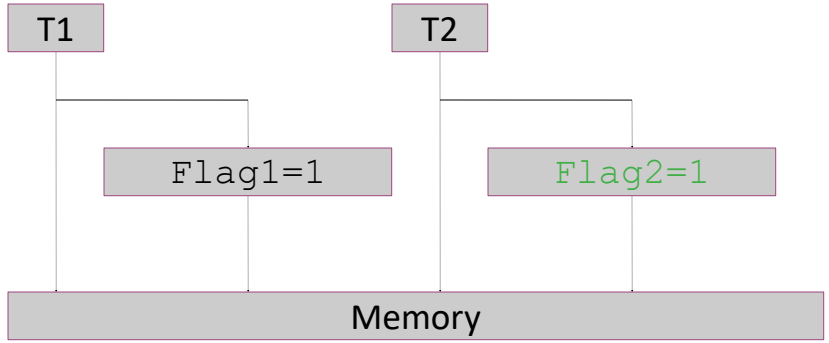
Initial: `Flag1 = Flag2 = 0`

T1:
```
Flag1 := 1;
if Flag2 = 0 then
-- critical
```

T2:
```
Flag2 := 1;
if Flag1 = 0 then
-- critical
```

- T1 read operation of `Flag2` overtakes the write operation in Write-Buffer
- Read operation arrives in memory before write operation
- This is the **1st** operation in memory order

SC Violation – Architecture without Caches



Flag1	0
Flag2	0

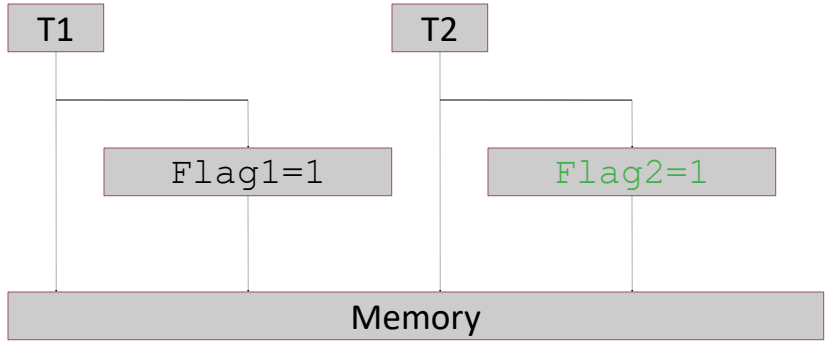
Initial: `Flag1 = Flag2 = 0`

T1:
```
Flag1 := 1;
if Flag2 = 0 then
-- critical
```

T2:
```
Flag2 := 1;
if Flag1 = 0 then
-- critical
```

- T2 write operation of `Flag2` goes into Write-Buffer

SC Violation – Architecture without Caches



Flag1	0
Flag2	0

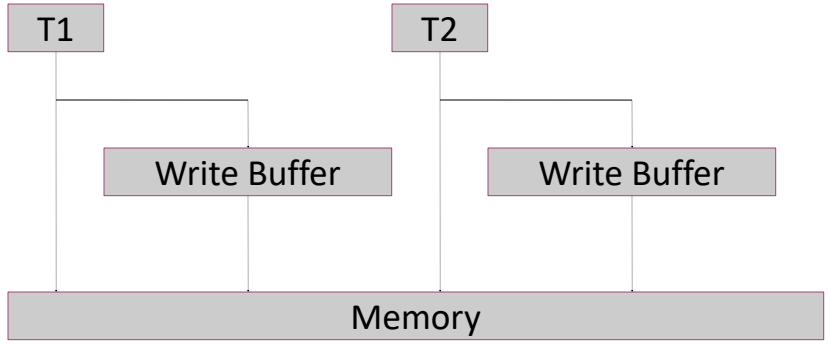
Initial: `Flag1 = Flag2 = 0`

T1:
```
Flag1 := 1;
if Flag2 = 0 then
-- critical
```

T2:
```
Flag2 := 1;
if Flag1 = 0 then
-- critical
```

- T2 read operation of `Flag1` overtakes the Write-Buffer
- This is the **2nd** operation in memory order

SC Violation – Architecture without Caches



`Flag1`	`0`
`Flag2`	`0`

Initial: `Flag1 = Flag2 = 0`

T1:
```
Flag1 := 1;
if Flag2 = 0 then
-- critical
```

T2:
```
Flag2 := 1;
if Flag1 = 0 then
-- critical
```

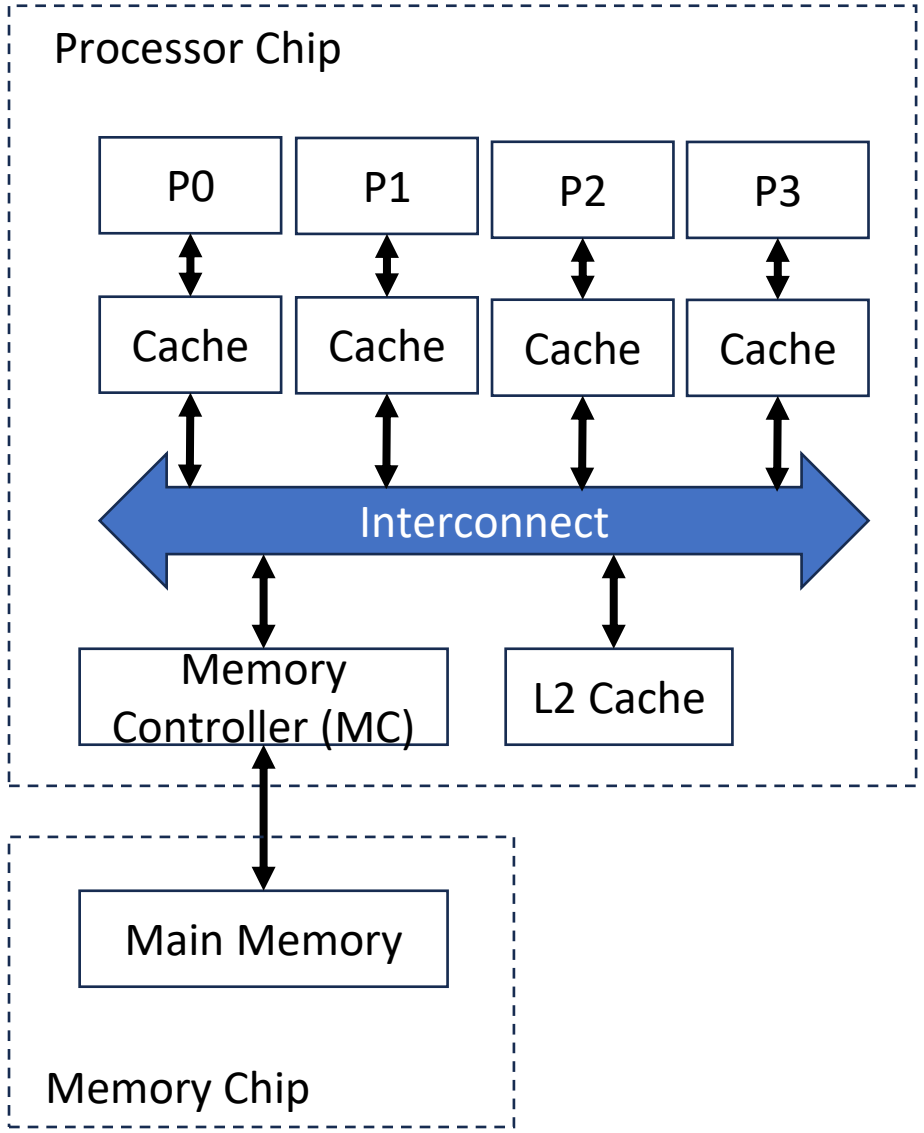
- Due to the Write-Buffer, the SC order (`Write(Flag1), Read(Flag2), Write(Flag2)` and `Read(Flag1)`) is different from the memory order (`Read(Flag2), Read(Flag1), Write(Flag1)` and `Write(Flag2)`)
- SC is violated
- Is there another possible memory order in this example?

SC Violation – Architecture with Private Caches

- SC requires that memory operations are executed atomically or instantaneously propagating changes to multiple cache copies is inevitably a **non-atomic** operation (cache coherency protocol may have delays)

Write-Atomicity:

- **Write** operations must happen immediately; if one core can observe the result of a write operation, then **all** cores can
- **Read** operations are **delayed** until all cache copies have confirmed the receipt of the last write operation.



PO != EO

- What are the implications of PO != EO?
- Back to our introductory example:

(*A, B, C, D*) = (0, 0, 0, 0)

Thread 1	**Thread 2**
`a: A := 1;` `c: C := B;`	`b: B := 2;` `d: D := A;`

- Considering each thread by itself (what compilers & CPUs do), the instructions a and c or b and d can be swapped, because the result remains the same.

(*A, B, C, D*) = (0, 0, 0, 0)

Thread 1	**Thread 2**
`a: A := 1;` `c: C := B;`	`b: B := 2;` `d: D := A;`

- **Now the result** (A, B, C, D) = (1, 2, 0, 0) **is possible**

Template

- Why is the result (*A, B, C, D*) = (1, 2, 0, 0) bad?
- It contradicts common sense!
- It may be that two events *X* and *Y* are seen by one thread in the order *X* < *Y* (*X* before *Y*), but by another thread in the order *Y* < *X* (*Y* before *X*).
- (Example follows shortly)
- **Violation of temporal relativity!**
- **Violation of causality!**
- → **Relaxed Memory Model** . . .
- . . . has problems. We will see what those are shortly . . .

Violation of Temporal Relativity - Example

T_1	T_2	T_3	T_4
`a: A := 1;`	`b: B := 1;`	`c: C := A;` `d: D := B;`	`e: E := A;` `f: F:= B;`

- Initially A=B=C=D=E=F=0.
- **Relaxed (no RAW,WAW,WAR in threads)**: a, b, c, d, e, f can be executed in any order.
- Order (A, B, C, D, E, F) = (0, 0, 0, 0, 0, 0) $\xrightarrow{d}$ (0, 0, 0, $\underline{0}$, 0, 0) $\xrightarrow{b}$ (0, $\underline{1}$, 0, 0, 0, 0) $\xrightarrow{f}$ (0, 1, 0, 0, 0, $\underline{1}$) $\xrightarrow{e}$ (0, 1, 0, 0, $\underline{0}$, 1) $\xrightarrow{a}$ ($\underline{1}$, 1, 0, 0, 0, 1) $\xrightarrow{c}$ (1, 1, $\underline{1}$, 0, 0, 1).
- T_3 sees C=1 and D=0, therefore he concludes that $a < b$.
- T_4 sees E=0 and F=1, therefore he concludes that $b < a$.
- **Main cause**: f is executed before e and d before c.

Producer – Consumer (Relaxed Version)

- Does our approach work in the Relaxed Memory Model?

(D, F, X) = (0, 0, 0)

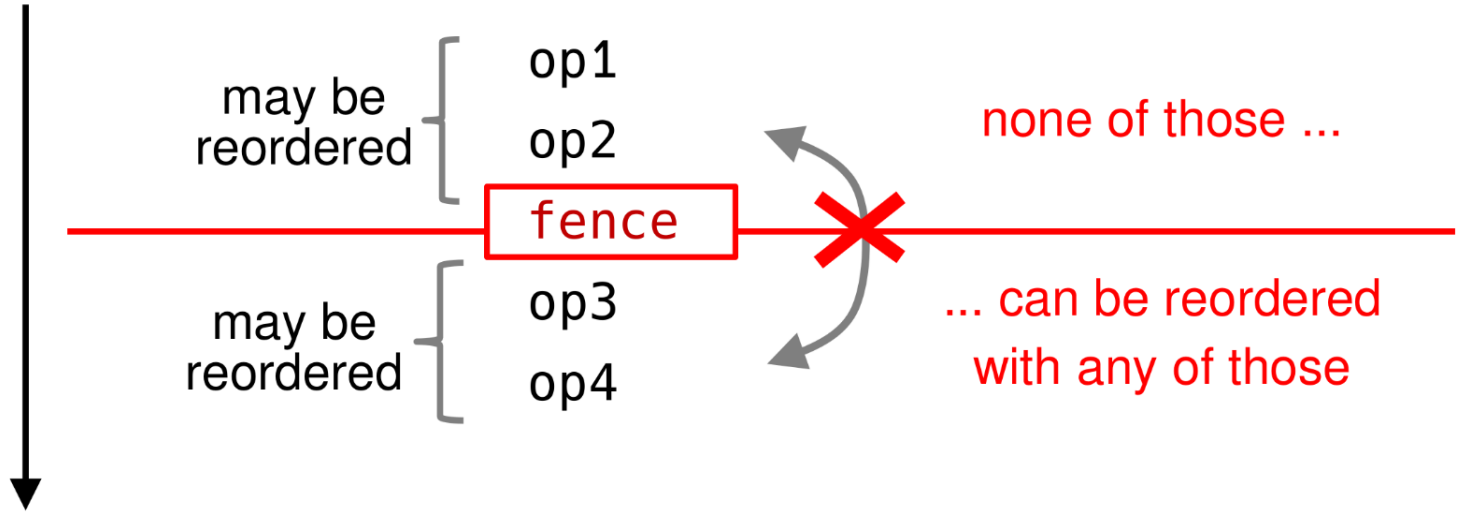
T_1	T_2
`d: D := 42;` `f: F := 1;`	`if: if F=0 then goto if;` `x: X := D;`

- **No!**
- Counterexample: f is executed before d.
- Order: (D, F, X) = (0, 0, 0) $\xrightarrow{f}$ (0, 1, 0), $\xrightarrow{if}$ (0, 1, 0) $\xrightarrow{x}$ (0, 1, 0) $\xrightarrow{d}$ (42, 1, 0).
- Data was not correctly transferred (X= 0 != 42 = D).
- **The simplest form of communication and synchronization does not work in the Relaxed Memory Model!**

E3.6 Release/Acquire Memory Model

Release/Acquire Memory Model

- How can we integrate the new memory models so that sensible work is possible? We need additional hardware tools.
- Modern computer architectures offer so-called
- **Memory-Fences** (Memory Barriers).
- Moving instructions across Memory-Fences is prohibited.



Release/Acquire Memory Model

- Programming languages must offer adequate language features so that Memory-Fences can be utilized.
- **Release** and **Acquire** operations for **atomic variables**.
- Temporal relativity can be ensured for atomic variables, but not for conventional variables.
- Responsible for this: Programmer / Computer Scientist.

Note: Automatic placement of Memory-Fences is not possible (undecidable problem)!
Equivalent to the Halting Problem
(There is no program that can automatically detect infinite loops in programs.)

Sequential Consistency on Modern Computers

Hardware tools can also ensure SC.

Advantages:

- Programmers do not have to worry.
- Programs are easier to write and debug.
- The correctness of such programs is easier to prove.

Disadvantages:

- The HW instructions for SC are very expensive (slow).
- The performance advantages of modern architectures are not utilized.

Memory Models

- The hardware must define memory models (how exactly does instruction scheduling, caching, etc., happen)
- Programming languages must specify which instruction reorderings the hardware is allowed to perform (interface to HW).
- Programming languages must describe the interactions between threads that take place over memory, and how shared data can be defined and used (interface to the programmer).

Memory Model Operations

Store: `atomic_store(atomic_var, value, memory_order):`

Stores *`value`* into the atomic variable *`atomic_var`*, where the Memory Order is specified as: *SC, Release*, or *Relaxed*.

Load: `atomic_load(atomic_var, memory_order):`

Returns the value of the atomic variable *`atomic_var`*, where the Memory Order is specified as: *SC, Acquire,* or *Relaxed.*

Exchange: `atomic_exchange(atomic_var1, atomic_var2, memory_order):`

Exchanges the values of the two atomic variables *`atomic_var1`* and *`atomic_var1`*, where the Memory Order is specified as: *SC, Release_Acquire,* or *Relaxed*

Release/Acquire Memory Model

We still need to clarify what effects Release and Acquire have.

Release-Operation: Sets a Memory Fence so that no Load and Store operations that stand in program order **before** the Release operation can be moved behind the Release operation.

Acquire-Operation: Sets a Memory Fence so that no Load and Store operations that stand in program order **after** the Acquire operation can be moved before the Acquire operation.

Relaxed-Operation: Sets no Memory Fences.

The programmer must know what s/he is doing!

Producer – Consumer (Release-Acquire Version)

$(D, F, X) = (0, 0, 0)$

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

- Because of the Fence in f, d cannot be moved behind f.
- Because of the Fence in if, x cannot be moved before if.
- Therefore, **communication** and **synchronization** work in the RA memory model!
- Responsible for the correct use of SC, RA, Relaxed: Programmer/Computer Scientist.

Programming Languages & Weak Memory Models

Java: first attempt at a memory model was incorrect;
from 5.0: SC.

C++: weak memory model
(SC, Release-Acquire, Relaxed, …); from C++11 on.

C: weak memory model
(SC, Release-Acquire, Relaxed, …); from C11 on.

Processors & Weak Memory Models

Intel x86/64: SC

ARM: weak memory model

RISC-V: weak memory model

Release/Acquire Memory Model

- Can the RA memory model guarantee SC?
- **No!**
- RA orders the instructions only (thread-)locally, SC is a global property.
- That's why SC is slower than RA.
- Why do we need Relaxed Memory Order?
- Sometimes nothing bad can happen; then one can use Relaxed instead of Release or instead of Acquire or even instead of SC.
- Relaxed brings **performance gains** over RA.

(… more on the topic of performance later ...)

Relaxed Examples

- The following examples are intended to show which RA operations can possibly be relaxed.
- For simplicity, let's assume all variables starting with A, such as A, A1, A2, . . . , are atomic and all other variables are non-atomic.
- There should be no further instructions before and after the given instructions that can be reordered.

Relaxed Examples

T_1	T_2

`z: Z := 15;`
`y1: Y := 11;`
`a: atomic_store(A, 1, Release);`
	`if: if atomic_load(A, Acquire)!=1 then goto if;`
	`y2: Y := Z;`

- Initially A = 0
- Can one of the RA be relaxed?
- **No!**
- Reason: *Z* is read in `y2` (T_2) but written in `z` (T_1).
- `z` therefore cannot be moved behind `a`, `y2` not before `if`.
- `y1` also cannot be moved behind a, otherwise the value that *Y* received in `y2` could be overwritten.

Relaxed Examples

T_1	T_2

`y: Y := 42;`
`a1: atomic_store(A1, 0, Release);`
`a2: atomic_store(A2, 1, Release);`
	`if: if (atomic_load(A1, Acquire)!=0 or`
	`atomic_load(A2, Acquire)!=1) then goto if;`
	`x: X := Y;`

- Initially *A1* = 1, *A2* = 0
- The `if` loop is only exited if *A1* = 0 **and** *A2* = 1.
- `a1` can be relaxed because the order in which the two stores occur is irrelevant.
- `a2` cannot be relaxed because the last store in program order needs a Release.
- Alternatively: relax `a2`, not `a1`?

Relaxed Examples

T_1	T_2

`y: Y := 42;`
`a1: atomic_store(A1, 0, Relaxed);`
`a2: atomic_store(A2, 1, Release);`
	`if: if (atomic_load(A1, Acquire)!=0 or`
	`atomic_load(A2, Relaxed)!=1) then goto if;`
	`x: X := Y;`

- The second Load can be relaxed; the first cannot …
- … provided that the semantics of the programming language guarantee that the condition of an If statement is evaluated from left to right.
- Otherwise, a compiler could arbitrarily reorder subexpressions of the Boolean expression.

Relaxed Examples

T_1	T_2

`y: Y := 42;`
`a: atomic_store(A, 1, Release);`
	`x: X := atomic_load(A, Acquire);`
	`if: if X != 1 then goto x;`
	`z: Z := X+1;`

- Initial A=0
- `a` can be relaxed because Y is not read in T_2.

Relaxed Examples

T_1	T_2

`y: Y := 42;`
`a: atomic_store(A, 1, Relaxed);`
	`x: X := atomic_load(A, Acquire);`
	`if: if X != 1 then goto x;`
	`z: Z := X+1;`

- `x` can be relaxed because if is data-dependent on `x` and `z` is data-dependent on `x`.
- Because of these data dependencies, `if` and `z` cannot be moved before `x`.
- (`z` may be moved before `if`.)

Relaxed Examples

T_1	T_2

`y: Y := 42;`
`a: atomic_store(A, 1, Relaxed);`
	`x: X := atomic_load(A, Relaxed);`
	`if: if X != 1 then goto x;`
	`z: Z := X+1;`

- `x` can be relaxed because if is data-dependent on `x` and `z` is data-dependent on `x`.
- Because of these data dependencies, `if` and `z` cannot be moved before `x`.
- (`z` may be moved before `if`.)
- If there are no other reasons against it, *A* may also be non-atomic

Memory Models – Spectrum of Common Architectures

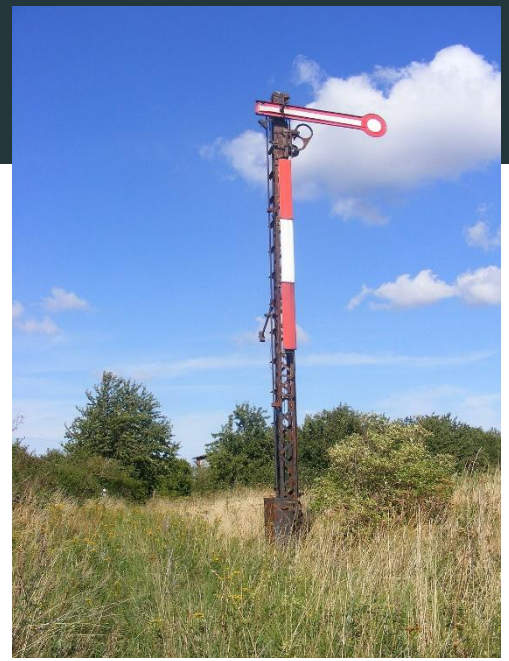
adapted from https://preshing.com/20120930/weak-vs-strong-memory-models/

E3.7 Blocking Wait

Blocking Wait

- Disadvantage of the previous type of communication/synchronization: a thread is in a **loop** until data can be read.
- Wastes unnecessary computing time and energy. Alternative:
- **Blocking Wait.**

Semaphore



Von Global Fish - Eigenes Werk, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=4637960

Counter, initialized to 1

Two operations:

Lock: Decrease counter by 1.
If counter >= 0, thread may continue execution.
If counter < 0, enqueue thread in a waiting queue & stop execution.

Unlock: Increase counter by 1.
If counter > 0, thread may continue execution.
If counter <= 0, release 1st thread from waiting queue & start execution.

Race condition!

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

Cf. lecture on the topic of “Operating Systems”

Memory Fences may be required to prevent code from the Critical Section (between Lock and Unlock) from “wandering out”.

Semaphore

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

Higher Mechanisms

- Disadvantage of Lock/Unlock: Correctness difficult to verify.
- Is there an Unlock for every Lock?
- Higher mechanisms in programming languages:
- Monitors
- Synchronized Objects (Java)
- Protected Objects (Ada)
- …

Example

- A central (global) Integer variable *Z* is to be increased by threads by a value.
- The “simultaneous” access should be prevented by a semaphore *S*.

Example

T
1: Lock(S); 2: Z_local := Z; 3: Z_local := Z_local + …; 4: Z := Z_local; 5: Unlock(S);

Advantages:

- Easy to understand.
- No waste of computing time and energy.
- Sequentially consistent.

Disadvantages:

- If a thread crashes between Lock and Unlock, no other thread that calls Semaphore-Lock can make progress.
- Thread dispatching (starting and stopping of threads) takes a lot of time.
- Read-Modify-Write operations are slow.

E3.8 Non-Blocking Wait

Non-Blocking Wait

- Instead of synchronization via semaphore or similar, direct use of Read-Modify-Write operations.
- Optimistic approach.
- → **Non-Blocking Wait**

Read-Modify-Write Operation

- Function `RMW(V, old_value, new_value)`
- Returns `true`, if the atomic variable *V* still has the old value; *V* receives the new value simultaneously.
- Returns `false`, otherwise. Implicitly `old_value` is set to `new_value`.

Example

T (*Z* atomic)
`1: Z_local := Z;` `2: if not RMW(Z, Z_local, Z_local + …) then goto 1;`

Advantages:

- If few threads access the central variable simultaneously, very efficient.
- If a thread crashes, other threads can still make progress.
- Sequentially consistent **and** Release/Acquire memory model possible.

Disadvantages:

- Difficult to understand for more complex algorithms.
- Even more difficult to understand for more complex algorithms together with Release/Acquire memory order.
- Suffers from ABA problem (to be explained on the next slides)

The ABA problem

The ABA problem occurs when multiple threads (or processes) accessing shared data interleave. Here is a sequence of events that illustrates the ABA problem:

1. Process *P1* reads value *A* from some shared memory location,
2. *P1* is preempted, allowing process *P2* to run,
3. *P2* writes value *B* to the shared memory location,
4. *P2* writes value *A* to the shared memory location,
5. *P2* is preempted, allowing process *P1* to run,
6. *P1* reads value *A* from the shared memory location,
7. *P1* determines that the shared memory value has not changed and continues.
   -> Thus an RWM operation may succeed, although it actually should not.

Although *P1* can continue executing, it is possible that the behavior will not be correct due to the “hidden” modification in shared memory.

Solving the ABA Problem with CAS

The ABA problem can be solved via CAS operation by **counting** the number of accesses to shared data.

Disadvantages:

- Counter has to be integrated into shared data word (which complicates accessing the actual data or may require modifying pointer values) or
- Use additional word for the counter (which requires double word or multi word CAS that are not provided by all CPUs)

Load-Link/Store-Conditional (LL/SC) Operation

- Alternative to CAS operation
- Equivalent to CAS
- LL/SC is sometimes called load-reserved/store-conditional (LR/SC)

Implementing LL/SC

- Function `LL(address)` loads value stored at address
- Function `SC(address, value)` stores value at address provided that there was no interfering store to address. Returns `true` if successful, `false` otherwise.

- **LL**: store address at cache line
- Any modification to any portion of the cache line (via conditional or ordinary store) cause the store-conditional (**SC**) to fail

- LL/SC operations are supported by DEC Alpha, PowerPC, MIPS, ARM, RISC-V, . . .

Load-Link/Store-Conditional (LL/SC) Operation

Advantages:

- Insensitive to ABA problem
- Instruction set: needs two words instead of three needed by CAS

Disadvantages:

- Sometimes fails if context switch occurs between LL and SC operation
- Sometimes fails if a second LL/SC occurs
- No nesting of LL/SC operations

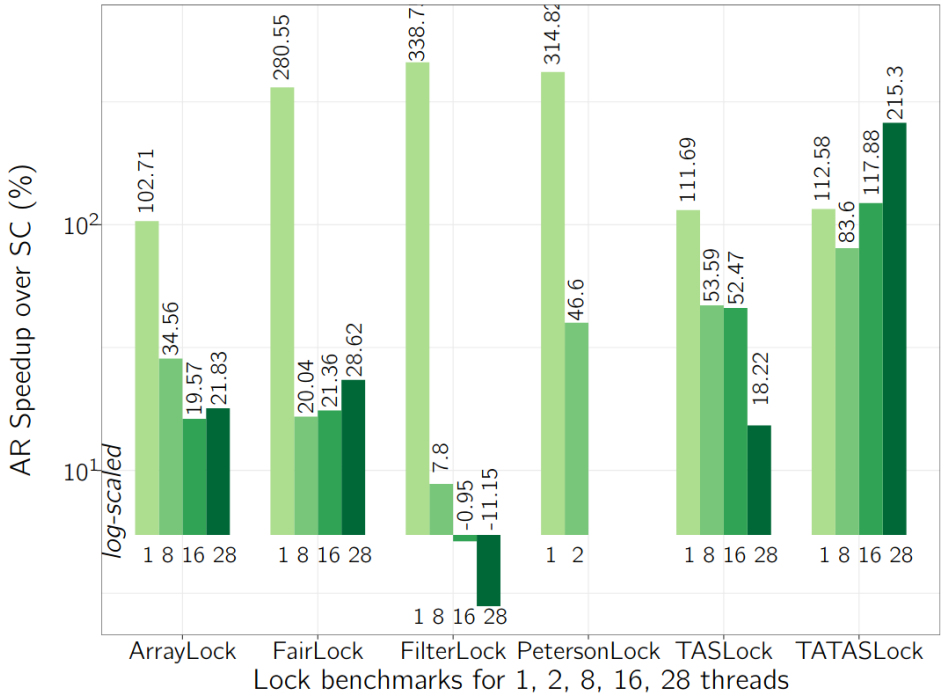
E3.9 Performance Comparison

Performance Comparison

Blocking vs. Non-Blocking (SC):

- Intel x86
- 1 thread writes to a queue, 1 thread reads from a queue
- Non-Blocking more than 100 times faster than Blocking

Performance Gain Intel X86 through SC-AR-Relaxation

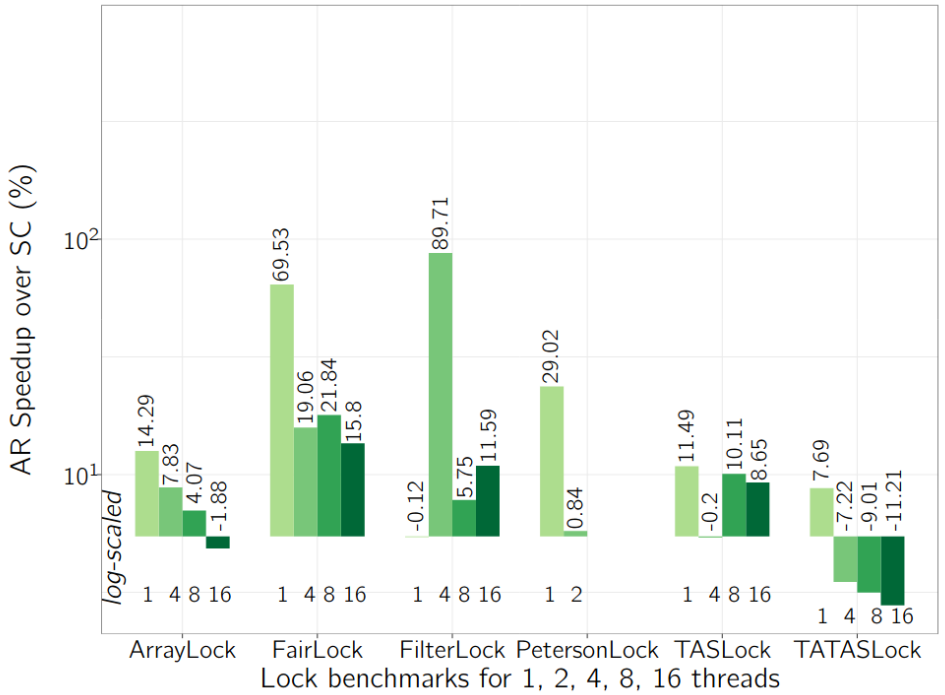


Experiment:

- N Threads
- 10M Lock/Unlock operations in a loop, no operation between Lock and Unlock
- 28 Cores, 2-socket system (Intel Xeon E5-2697 v3 @ 2.60 GHz)

from: S. Yang, S. Jeong, B. Min, Y. Kim, B. Burgstaller, J. Blieberger, Design-space evaluation for non-blocking synchronization in Ada: lock elision of protected objects, concurrent objects, and low-level atomics, Journal of System Architecture, Volume 110, 2020, 101764, ISSN 1383-7621, https://doi.org/10.1016/j.sysarc.2020.101764 .

Performance Gain ARM v8 through SC-AR-Relaxation



Experiment:

- N Threads
- 10M Lock/Unlock operations in a loop, no operation between Lock and Unlock
- 16 Cores, 4-socket system (AWS Graviton ARM v8)

from: S. Yang, S. Jeong, B. Min, Y. Kim, B. Burgstaller, J. Blieberger, Design-space evaluation for non-blocking synchronization in Ada: lock elision of protected objects, concurrent objects, and low-level atomics, Journal of System Architecture, Volume 110, 2020, 101764, ISSN 1383-7621, https://doi.org/10.1016/j.sysarc.2020.101764 .

E3.10 Summary

Summary (1/2)

Memory Models:

- **Sequentially Consistent Memory Model**: corresponds to common sense.
- Due to various hardware and software optimizations for performance improvement, SC is no longer given.
- → weak memory model: **Relaxed Memory Model**.
- Practical compromise between programmers' intuition and performance:
  Release/Acquire Memory Model

Summary (2/2)

Programming Multi-Threaded Applications:

- If performance is not a major concern, prefer Blocking.
- If Blocking, prefer higher language features (no semaphores!)
- If performance is important, initially prefer SC (easier to understand!)
- If SC is too slow, switch to RA (Relaxing very important, but difficult!)
- SC → RA-Relaxation can also bring performance gains for SC-HW!
- Prefer to use pre-made libraries with data structures or algorithms (already well tested!)
- Freely accessible, open-source libraries are better!

Thank you for your attention!