Paradigmas de Computação Paralela

Memory Consistency Models

João Luís Ferreira Sobral
jls@...
17 Nov 2015

Models of memory consistency

- **Outline**
  - Review: threading models
  - Why the need for consistency models?
  - Sequential consistency models
  - Relaxed consistency models
    - Weak consistency
    - Release consistency
  - Case studies
    - OpenMP
    - Java Threads

Models of memory consistency

- **Execution of *multi-threaded* programs in uniprocessors (single core)**
  - The processor executes all threads in the program
    - Thread scheduling policy is not specified
  - Operations executed by each thread are performed in order
  - Thread synchronization with atomic operations (e.g. *locks*)
  - Operations executed by threads are intermixed
    - Non-determinism: the result depends on the order in which threads are scheduled
    - e.g.:

Thread 1	*Thread 2*
…..	*……*
x := 1;	*print(x);*
x := 2;	

Models of memory consistency

- **Execution of *multi-threaded* programs in multi-processors**
  - Each processor executes a thread (considering the number of threads equal to the number of processors)
  - Operations executed by each thread are performed in order
  - Each processor can access to global memory to perform load/store/atomic operations
    - No *caching* of global data (for now!)
  - Possible results are equal to single processor.
  - <u>More realistic considerations:</u>
    - Caching of global data to improve efficiency
      - Requires a protocol for cache cohesion
    - Instruction execution out-of-order
      - Can change the program semantic => a memory consistency model is required!

Models of memory consistency

- **Out-of-order instruction execution**
  - Processors reorder instructions to improve performance
  - Reorder of instructions should respect program dependencies
    - Data– e.g., *loads/stores* should be executed by the required order
    - Control
  - Instruction reorder can be performed by the compiler or by the processor
  - Allowed reorders:
    - *Stores* in different locations

```
store v1, data                store b1, flag
store b1, flag      ←→        store v1, data
```

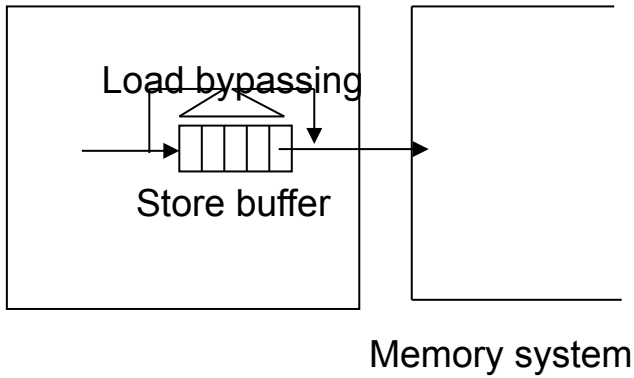
    - *Loads* from different locations

```
load flag, r1                 load data,r2
load data, r2       ←→        load flag, r1
```

    - *Loads* and *stores* from/in different locations

Models of memory consistency

- **Out-of-order instruction execution (cont.)**
  - Example of hardware instruction reorder



  - The *store buffer* saves *store* operations the be sent to memory
  - *loads* have priority through *stores* to keep the processor busy
    - They can bypass *stores* in the buffer
  - *loads* verify if there is a *store* in the same target address
  - Result: *load* and *stores* are not performed in the program order

Models of memory consistency

- **Out-or-order execution in multiprocessors**
  - Canonical model
    - Operations performed by a given processor are done in-order
    - Operations performed by different processors are intermixed
  - If a processor reorders his instruction flow, will be the same result accomplished?

Initially A = Flag = 0

P1
A = 23;
Flag = 1;

P2
while (Flag != 1) {;}
... = A;

  - P1 write data in A and signal it to P2 by changing the Flag value
  - P2 waits until Flag is active, and the reads A
- What happens if P1 exchanges the order of *stores*?

Models of memory consistency

- **Out-or-order execution in multiprocessors (cont.)**
  - Example II

Flag1 = Flag2 = 0

P1	P2
Flag1 = 1;	Flag2 = 1;
If (Flag2 == 0)	If (Flag1 == 0)
critical section	*critical section*

One possible order:

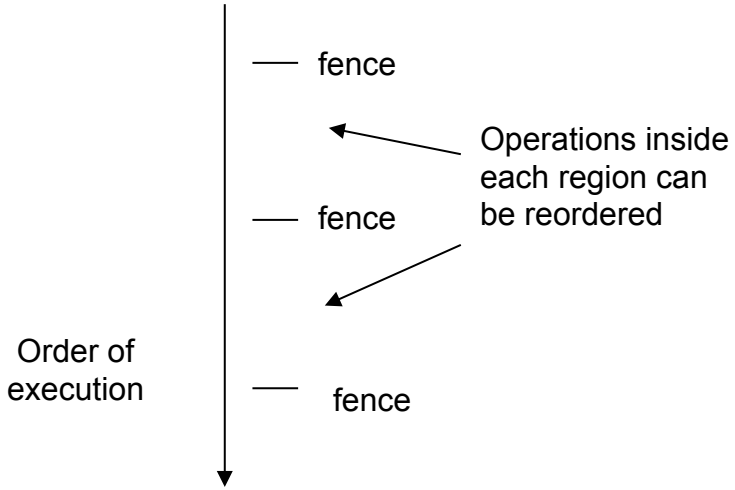
P1	P2
Write Flag1, 1	Write Flag2, 1
Read Flag2 //get 0	Read Flag1 // get 1? => only if load and stores are performed in order

Models of memory consistency

- **Conclusion**
  - Uni-processors can reorder instructions to improve performance, respecting local dependencies
  - This is not valid for multi-processors sharing memory
    - A parallel program can deliver contra-intuitive results
  - What limitations can be imposed to instruction reorder such that:
    - programming can be intuitive
    - do not lose uni-processor performance
  - Solution: memory consistency model supported by the processor:
    - Sequential consistency model (the simplest):
      - Processor can not reorder load/stores in global memory
      - Reduces uniprocessor performance to an inacceptable level

Models of memory consistency

- **Relaxed consistency models – weak consistency**
  - Programmers specify program execution points where operations should be ordered
  - "*fence*" instruction
    - Operations before "fence" should complete before the "fence " execution
    - Operations after the "fence" must wait until de "fence" terminates.
    - "fences" are executed in order
  - "fence" implementation
    - Counter incremented at each fence execution
  - Example:
    - SYNC intruction in PowerPC
    - flush in OpenMP
    - lock/unlock synchronization



Models of memory consistency

- **Relaxed consistency models – weak consistency**
  - Example I - Revised

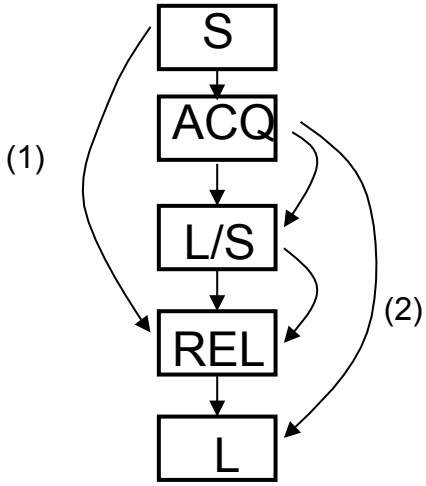
Initially A = Flag = 0

P1	P2
A = 23;	
flush	while (Flag != 1) {;}
Flag = 1;	... = A;

  - P1 write data to A
  - flush waits for the write to A to complete
  - P1 activates the Flag
  - P2 reads Flag==1 when the write in A has completed, even if operations before the flush are completed out-of-order
  - Is a flush required among the two operations in P2?

Models of memory consistency

- **Relaxed consistency models – release consistency**
  - More relaxed model than weak consistency
    - The weak consistency model does not differentiate reads from writes (the fence instruction implies a read/write of all values that have been changed)
    - The release consistency model updates local values (read from memory) when entering in the critical region and write the values on the exit of the region
  - Access for synchronization are divided in:
    - acquisition: operations of type *lock*
      - Should be performed when entering in the critical region
      - Does not wait for the access that precede it (1)
    - release: operations of type *unlock*
      - Should be executed after all writes
      - Accesses after “release” do not need to wait for the release (2)



Models of memory consistency

- **Implementation on actual processors**

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache/Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64	Y			Y				
IA64	Y	Y	Y	Y	Y	Y		Y
(PA–RISC)	Y	Y	Y	Y				
PA–RISC CPUs								
POWER	Y	Y	Y	Y	Y	Y		Y
SPARC RMO	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86	Y	Y		Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries				Y				Y

Models of memory consistency

- **Consistency models**
  - Impose restrictions to the reordering of instructions of a single processor
    - It is not related with memory operations of different processors
  - There are many consistency model
    - There is a consensus around weak/release
  - It is easy to write a program whose behaviour is trapped into a specific consistency model
- **Memory cohesion**
  - Creates the illusion of a single logic localization corresponding to each variable of the program, even if there are multiple localizations for that variable

Models of memory consistency

- **OpenMP model**
  - Based on weak consistency
  - Activities can work on a temporary view of memory (registers, cache, etc)
    - Reads and writes are performed on local memory
  - “fence” instructions
    - #pragma omp flush
    - Enforces the consistency between the temporary view and the memory (for the list of variable in the flush
      - Variables changed locally are written into memory
      - Local copies of variables are discarded to enforce reading for the memory on the next local read.
      - Operation ends when all variables have been updated in memory
    - There is no guarantee of ordering among flush if the list of variables is disjoint
    - Memory/local variable (threadprivate, private) are not affected
  - flush is implicit in:
    - *Barrier*, *set_lock*, *unset_lock*, *atomic*
    - When entering/exiting from *parallel*, *critical* and *ordered*
  - It is not advisable to perform synchronization using variables in memory

Models of memory consistency

- **Java model**

  - The original model was revised in 2004 (Java 5.0) since the original model imposed to much limitation to optimization and do not offer enough guarantees to the programmer

  - Based on a weak consistency model
    - Threads can maintain local copies of data
    - Threads do not hold copies of volatile variables

  - Shared memory among threads
    - Instance variable, static variables, array elements
    - Local variables, parameters and exception handlers are not shared

  - *synchronized*:
    - Invalidates local copies of data at the entrance
    - On exist performs “flush”
    - Implies ordering among successive regions (protected by the same lock)

  - Problem of Java initial model: does not force sequential semantic of other variables relatively to volatiles

    Initially A = Flag = 0
    Flag declared as volatile

    P1
    A = 23;
    Flag = 1;

    P2
    while (Flag != 1) {;}
    ... = A;

    – P1 writes data in A signals P2 through Flag
    – P2 wait until Flag becomes active and then reads A