Ensuring reproducible parallel LT TLM models simulation with SCale SystemC kernel

Tanguy SASSOLAS,
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

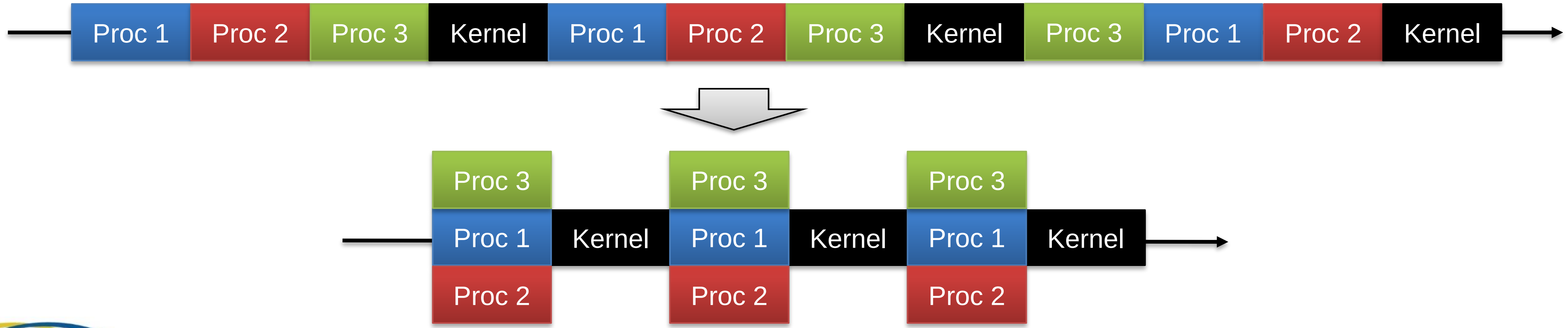
Joint work with Gabriel BUSNOT, Nicolas VENTROUX and Matthieu MOY

Copyright Permission

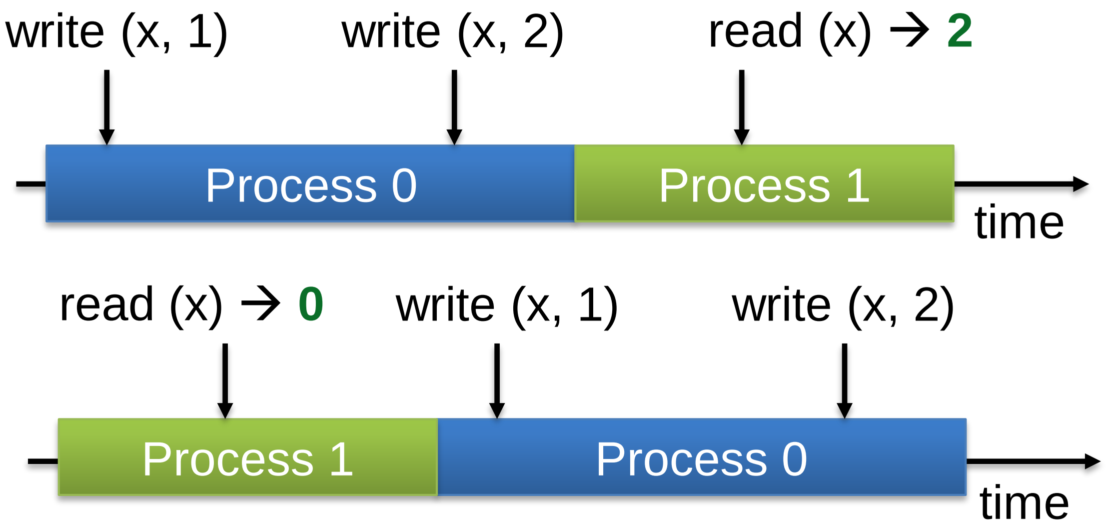
- A non-exclusive, irrevocable, royalty-free copyright permission is granted by **CEA** to use this material in developing all future revisions and editions of the resulting draft and approved Accellera Systems Initiative **SystemC** standard, and in derivative works based on the standard.

SCale 2.0 in a nutshell

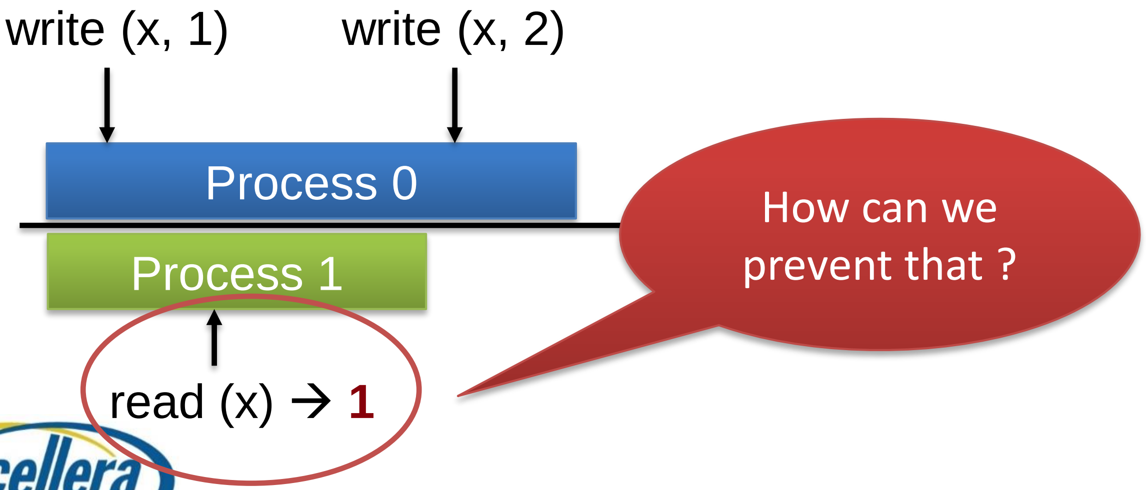
- An extended SystemC kernel for parallel simulation of TLM models
  - Allocates `SC_THREAD`s on several host CPUs aka workers executing in parallel
  - Provides API to ensure atomic evaluation between `wait` statements
  - Allows reproducible execution aka replay



Parallel atomic process evaluation problem



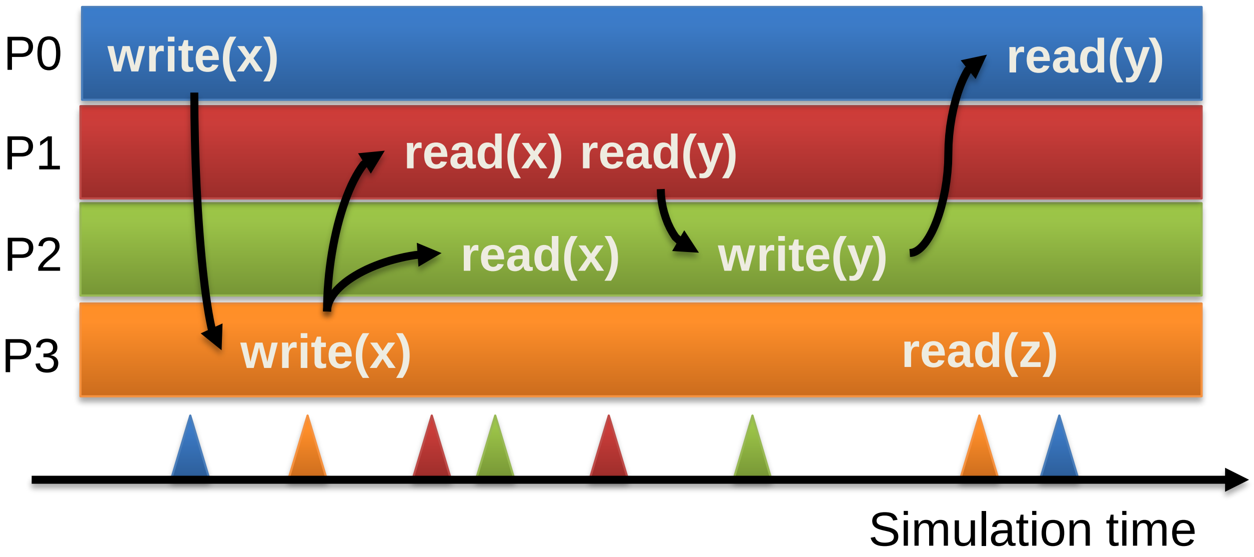
2 valid sequential evaluations yielding different results !



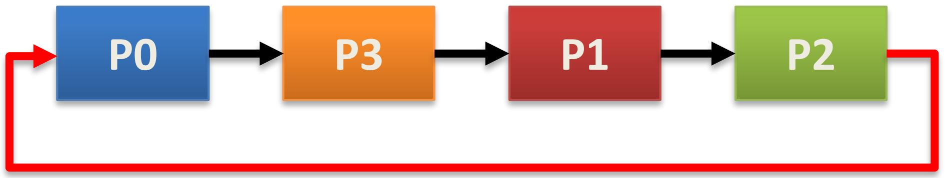
Parallel simulation yielding to a non reproducible read due to process atomicity violation !

Conflicts : Violation of atomic evaluation

- Occurs when **no equivalent sequential schedule** is found
- A sequential schedule always exhibit a total order between process R→W, W→R and W→W access patterns
- Finding conflicts is equivalent to identifying process access dependency loop
  → build dependency graph



Equivalent to sequential execution of:



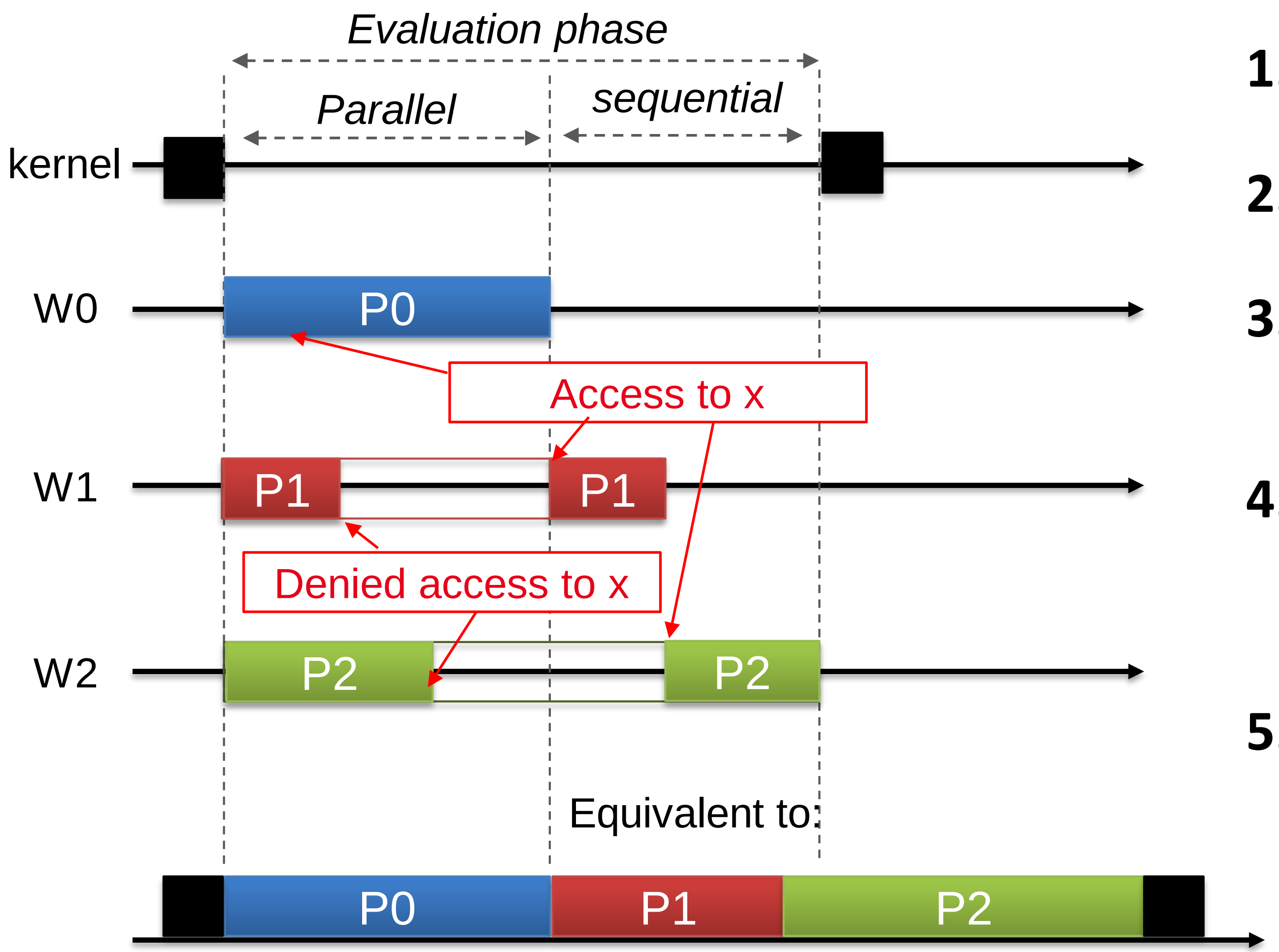
No equivalent sequential schedule

So what do we need to prevent this ?

How to ensure atomic process evaluation ?

- To identify if atomicity is violated we need knowledge on accesses
  - →Pervasive **access monitoring**
  - →We need to no what and **when (←requires costly barriers)**
- Not all accesses are problematic + cannot stop // on every access for perf.
  - →We need to **filter problematic accesses**
  - →Identify **shared resources**
- In the general case: no mean to know shared resources before evaluation (e.g. CPU models memory access patterns)
  - →Need **dynamic** analysis
  - →Need **rollback** if shared resources identified to late

SCale 2.0 Rationale



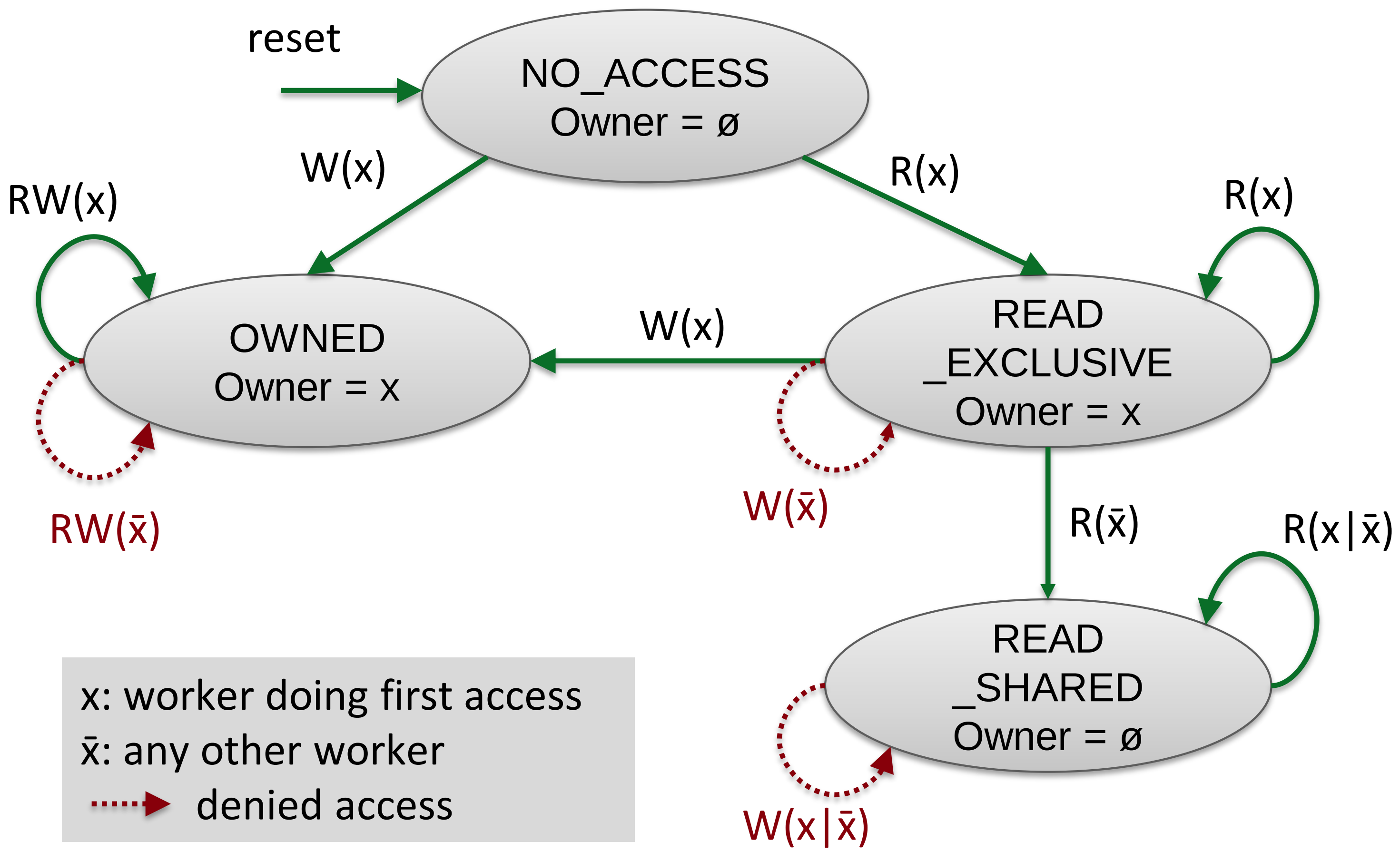
1. **Execute SC processes in parallel** (using worker threads)
2. **Monitor all memory accesses** performed (and log them)
3. **Postpone processes trying to access a shared variable** to a sequential evaluation phase to avoid atomicity violation
4. **Determine if an address is shared** thx to an FSM-based heuristic
   - 3 & 4 ensure that no atomicity violation can occur during parallel evaluation
5. **Assert that no conflicts occurred** after a SystemC evaluation phase by analyzing the log
   - **Rollback** to a previous state if need be
   - Restart the execution while ensuring dependencies are consistent from previous run

`mem_instr` example

- All memory access instrumented with `mem_instr` function:
- Suspends worker accessing any address identified as shared
- Builds *worker dependency graph*

```
// SC_THREAD simulating a CPU
void cpu_process(){
    while(!terminate){
        auto instr = get_next_instr();
        if(is_mem_access(instr)){
            mem_instr( // HERE
                access_type(instr),
                access_phy_addr(instr),
                access_bytes(instr));
        }
        sim_instr(instr); //<- perform the
        access
    }
}
```

Address classification FSM



Upon denied access :

Processes are yielded and will resume their execution in a sequential evaluation phase

Notice that :
no R→W, W→R or W→W dependency can occur between 2 processes during parallel evaluation

We call this **ZDG**
(Zero-dependency Guarantee)

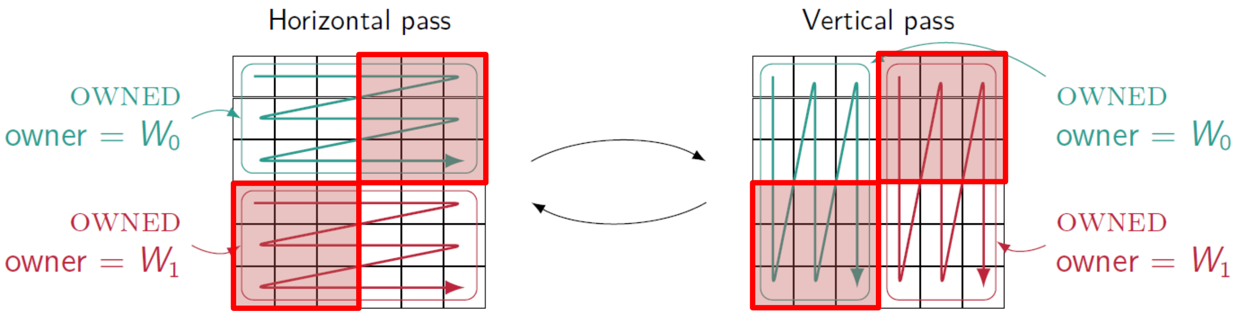
ZDG advantages during parallel phase

- Instrumentation performance :
  - [Instrumentation + memory access] no longer need to be atomic
    - True as long as instrumentation comes first
    - **Removes costly barriers** during instrumentation
  - Memory accesses during the parallel phase can be recorded in parallel
    - As they never depend on each other, their **order is not important**

- Conflict checking performance :
  - If no worker is unscheduled during the parallel phase, then no dependencies exist : the evaluation is valid without further analysis

Efficient FSM forgetting

- FSM state is generally maintained from one evaluation to the next
  - Needed as FSM state changes use costly CAS access
- Access pattern to resources can change during execution
  - Need to forget previous classification to **avoid over-pessimistic unscheduling**



Multithreaded Deriche image filter

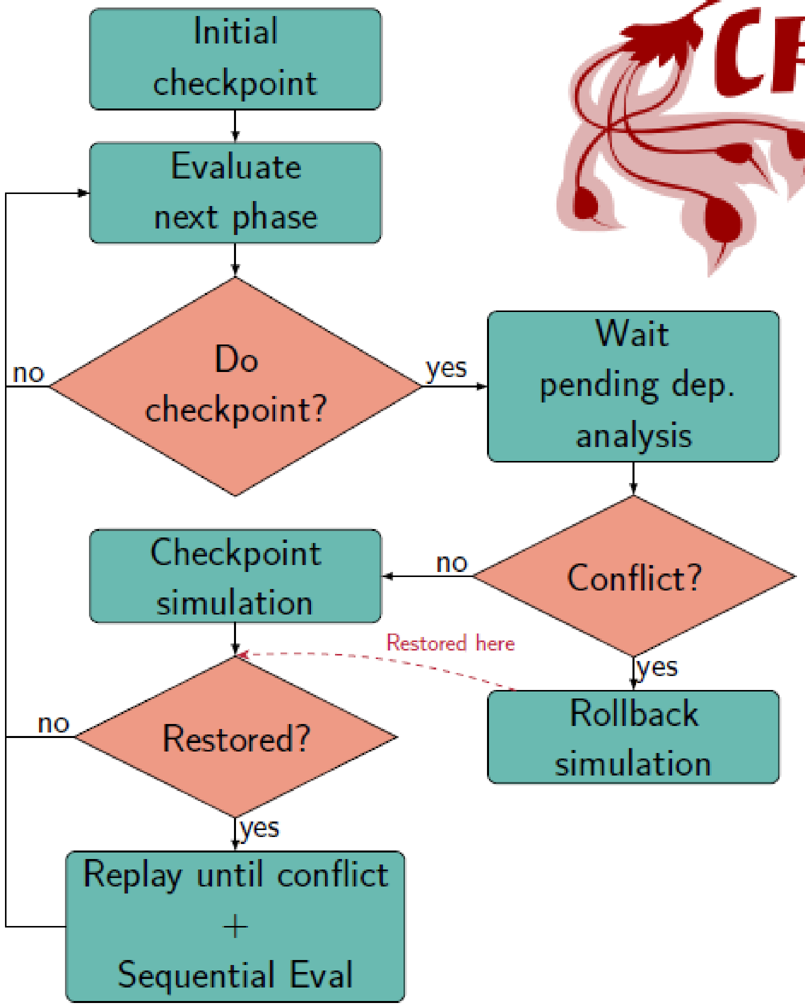
- Reset heuristic : when an unscheduling occured during last evaluation
- O(1) Generation-counter-based reset

Conflicts & Rollback

- Upon completion of a sequential evaluation phase
  - Start an **asynchronous conflict check** to assert no dependency loop exist
  - Start immediately next evaluation phase
  - Collect dependency analysis results and store observed valid process order for *replay*

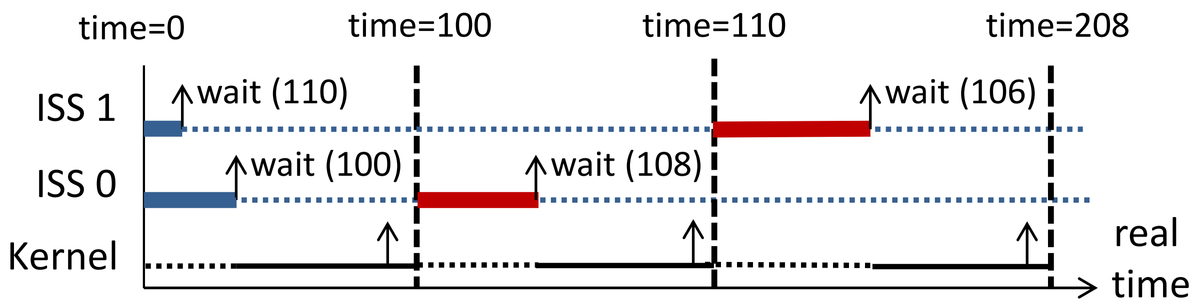
- Periodic check-pointing of simulation state <u>when valid</u>

- When a conflict is found :
  - Rollback to previous valid state
  - *Replay* up to problematic eval phase
  - Execute problematic phase sequentially

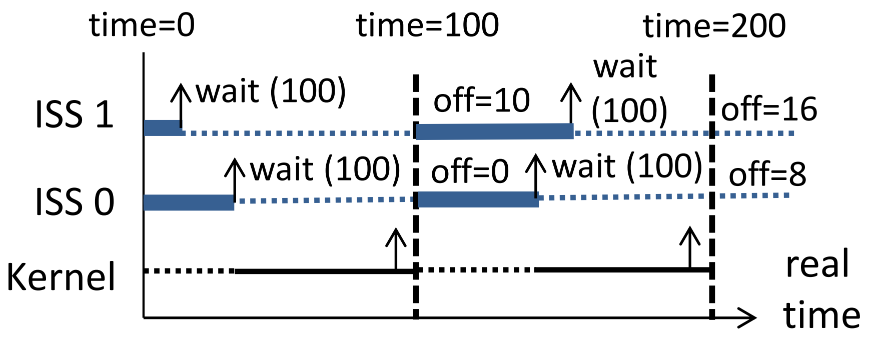


Ensuring good workload

- Need to have processes starting at the same cycles



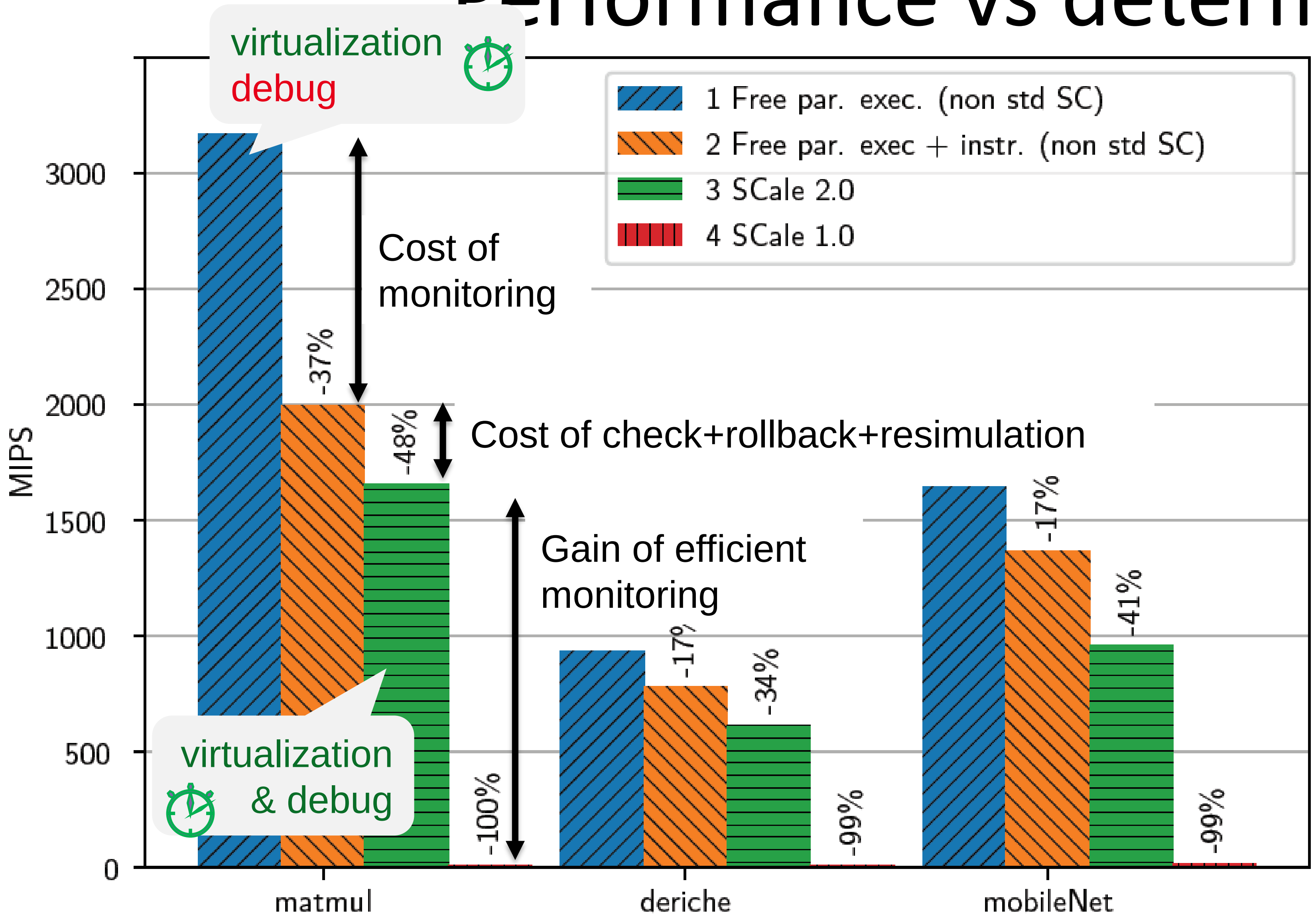
☹ Sequential evaluation
(trashing timing effect)



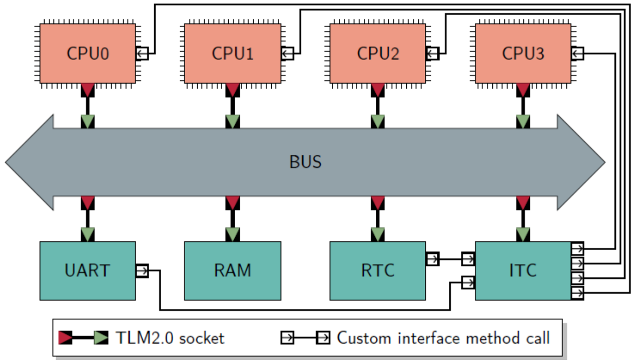
☺ Parallel evaluation*

* with the use of sc_global_quantum_sync()

Performance vs determinism

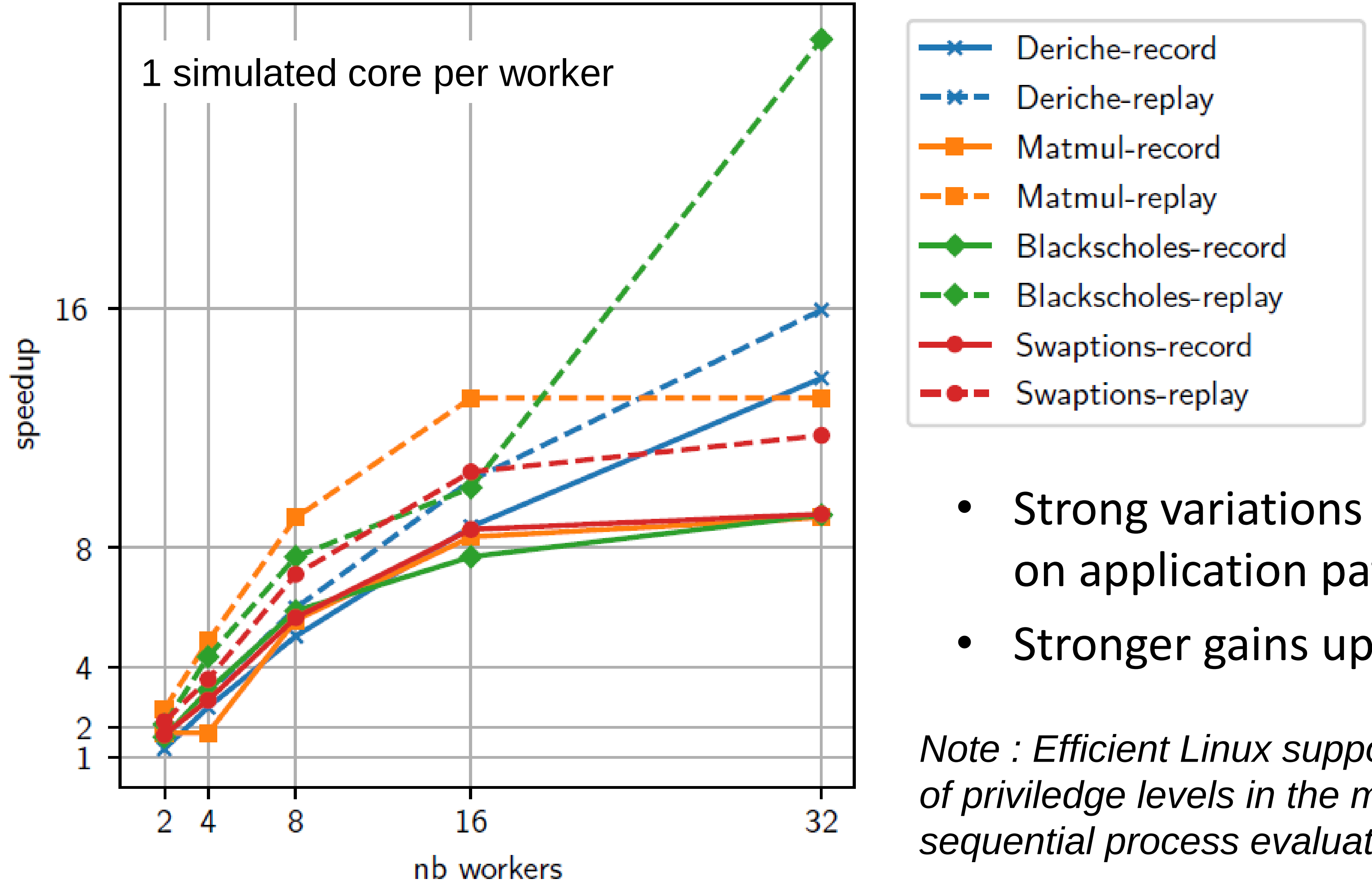


- 32 simulated processors using 32 workers



- QEMU Instruction Set Simulator
  - load/store simulated in SystemC
- Quantum : 30,000 cy
- Baremetal applications

Linux performance



- Always provides an acceleration
- Recording run: ×9 to ×13 (32 workers)
- *Replay* run: ×11.5 to **×24** (32 workers)

- Strong variations in gains depending on application pattern
- Stronger gains upon *replay*

Note : Efficient Linux support requires the monitoring of priviledge levels in the modelled CPUs to enforce sequential process evaluation

Conclusion

- SCale provides means to ensure atomic evaluation and replay in parallel simulation
  - Efficient monitoring that still halves the undeterministic execution speed
  - Requires rollback support ☹
  - Successfully provides acceleration if few resources are actually shared (up to x24 in replay)

- Monitoring of shared resources access is necessary
  - Requires designer knowledge and annotations

- Future work
  - Provide source model analyzer to help designer annotate their models
  - Study the impact in performance of more complex memory hierarchy with several levels of sharing
  - Refine analysis of problematic access patterns in Linux guest