Analysis of simulation traces for the debug of HW/SW multiprocessor systems

Nicolas Fournel and Frédéric Pétrot

Tima Laboratory
SLS Team
Grenoble - France

Agenda

- **Introduction**
- **Trace Generation**
- **Trace analysis**
  - Analysis Example
  - Other analysis Example
- **Conclusion**

Agenda

- **Introduction**
- Trace Generation
- Trace analysis
  - Analysis Example
  - Other analysis Example
- Conclusion

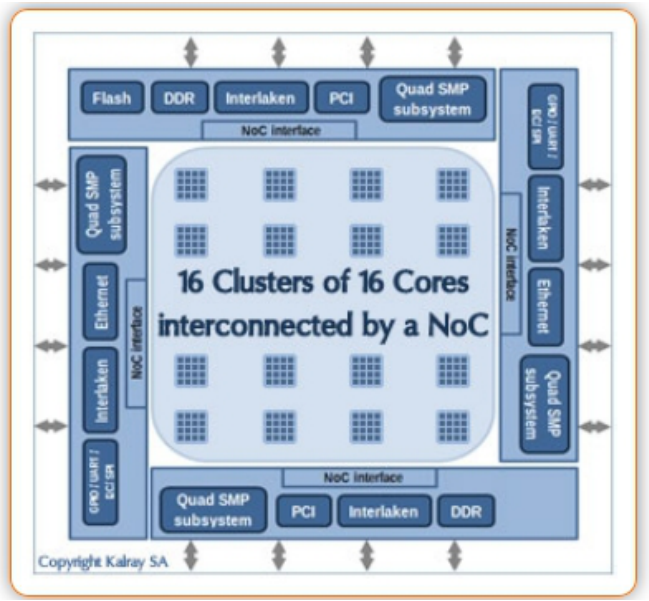
Motivation

Increasing number of processors

Trends for multi/many-core systems:

- ARM big.LITTLE (8 cores)
- Intel SCC (48 cores)
- Tilera Tile-GX8072 (72 cores)
- **Kalray MPPA** (256 cores)

Kalray MPPA256



Increasing hardware parallelism

- DMA, master capable IPs

Debug challenges

Issues

- very difficult to understand parallel behavior
  even worse with high level of parallelism
- usual debuggers cannot freeze the whole system

Possible solutions

- addition of processings to help understand parallel execution
- necessity of catching concurrency effects:
  source of most non-deterministic bugs

Execution traces

Execution traces

- Exhaustive list of events produced by execution of the software
- Different categories of events
  - software events: function calls, lock acquire, ...
  - hardware events: memory access, instruction execution, ...
- Different production schemes
  - full software:
    events probed by software, extraction from the system by software
  - mixed hardware/software:
    events probed by software, extraction using hardware support
  - full hardware: events probed by hardware, extraction using hardware support

Execution traces

Pragmatic solution

- replay of faulty executions
- gives a snapshot of system state
- outsources debug off the system

Relation between events

- Chronological:
  - In SW generation: timestamping of events hard to achieve
  - In HW generation: hardware timestamping
- Causality:
  - SW: Possible but heavy (increases intrusiveness)
  - HW: Impossible

Alternative solution

Simulation generated execution traces

- completely non-intrusive generation of trace
- large amount of information can be probed out
  no limitation in space or bandwidth
- complex information can be retrieved from HW components
  causality chain of events can be maintained in simulation trace generation

Limitations

Depends on:

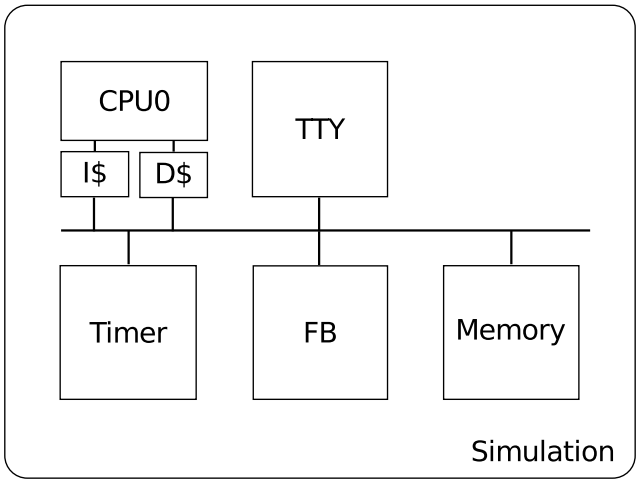
- model availability
- model accuracy

Agenda

- Introduction
- **Trace Generation**
- Trace analysis
  - Analysis Example
  - Other analysis Example
- Conclusion

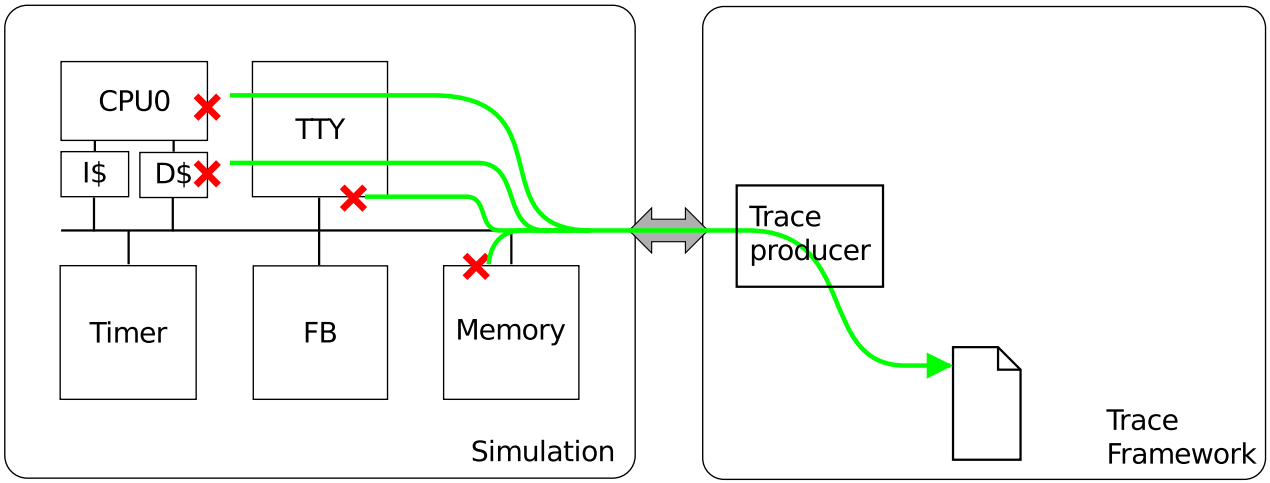
Integration in the workflow

Analysis Software Architecture



Integration in the workflow

Analysis Software Architecture

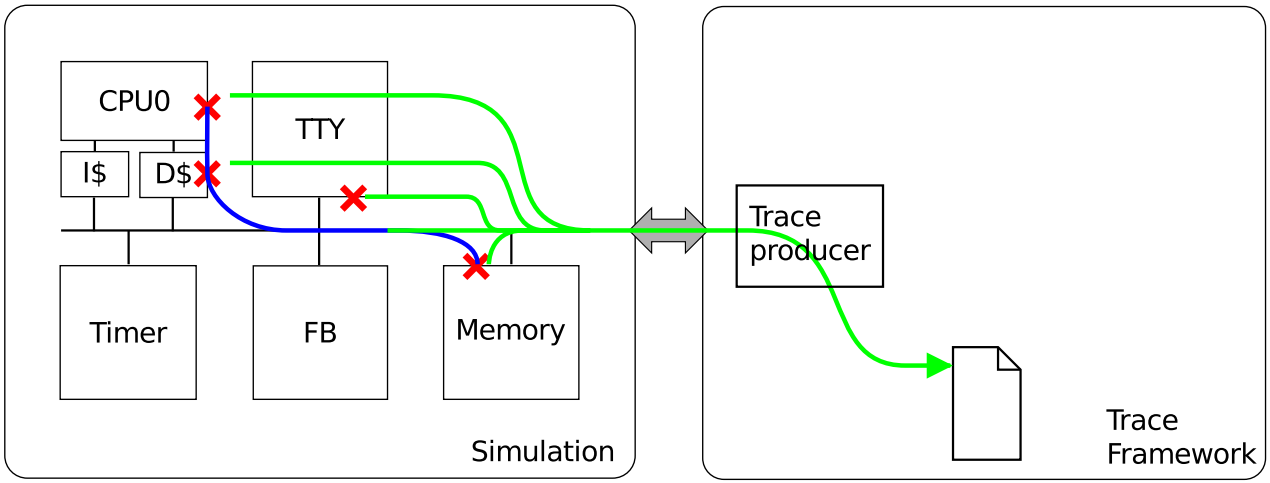


Modification of simulation

- Annotation of models

Integration in the workflow

Analysis Software Architecture

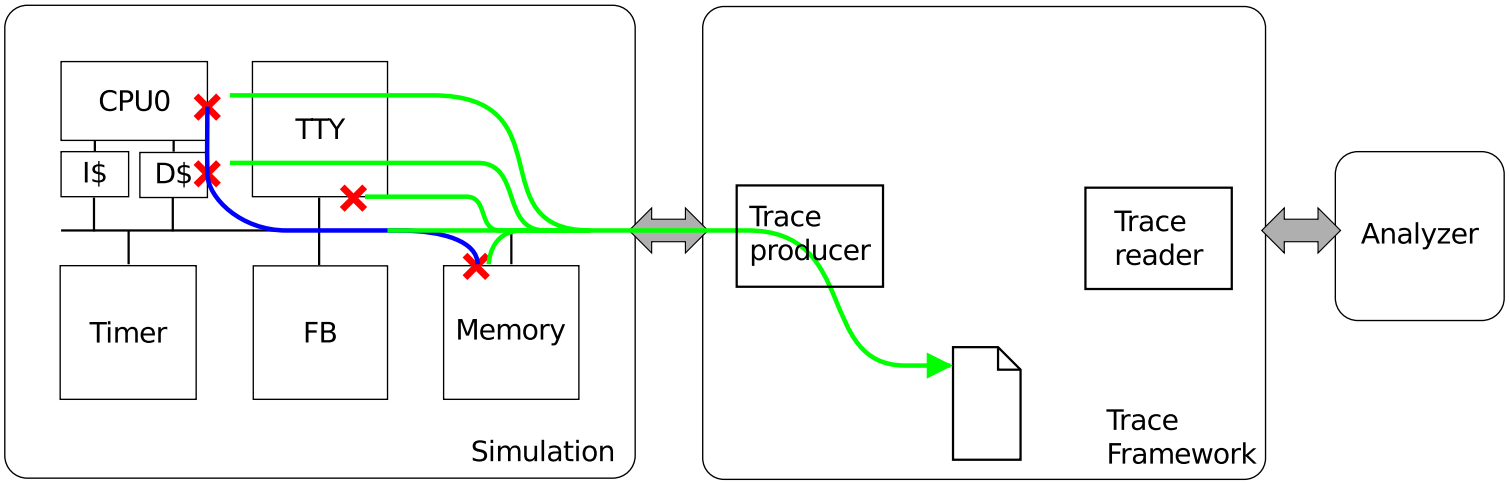


Modification of simulation

- Annotation of models
- Causality chain construction

Integration in the workflow

Analysis Software Architecture

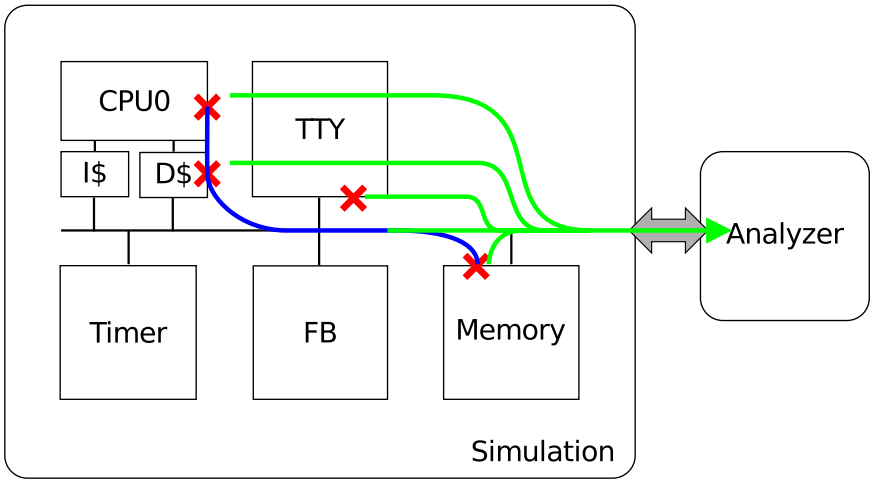


Analysis scheme

- Off-line

Integration in the workflow

Analysis Software Architecture

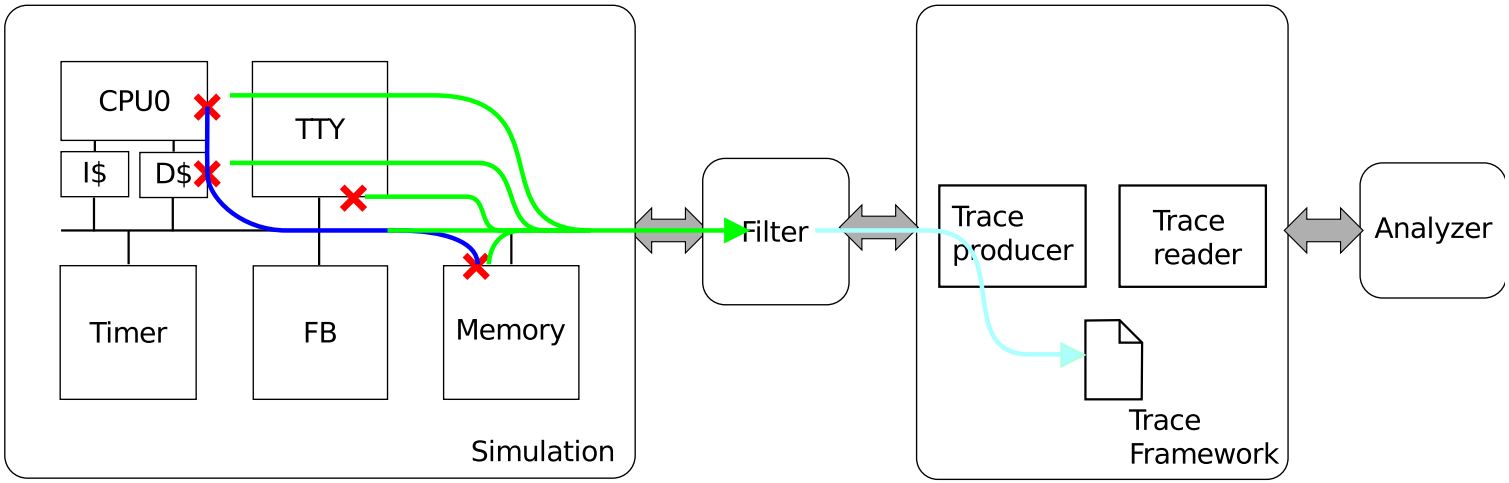


Analysis scheme

- Off-line
- Online

Integration in the workflow

Analysis Software Architecture



Analysis scheme

- Off-line
- Online
- Mixed/filtering

Impact on simulation

Performances (implemented in a CABA simulator: SoClib)

Application	Simulation speed (kCycles / sec.)	Overhead per Processor
MJPEG_1	887	1.9 %
MJPEG_2	672	4.5 %
MJPEG_3	555	6.2 %
MJPEG_4	497	6.8 %
OCEAN_4	229	4.1 %
OCEAN_8	172	2.1 %
OCEAN_16	120	11.9 %
OCEAN_32	67	5.8 %

Size estimation

- quite large, ...
  2.6G for 100 Millions cycles (105 s of CABA simulation)
- online filtering may help a bit

Agenda

- Introduction
- Trace Generation
- **Trace analysis**
  - Analysis Example
  - Other analysis Example
- Conclusion

Analysis context

Possible usage

- enhanced/enriched visualization of the trace
- property analysis/checking
- automatic pattern discovery, data mining, ...

Follow-up: detailed property analysis

- no memory consistency violation
  - for a given memory consistency model
  - for a given execution

Analysis example

Definition of consistency

Manner in which memory accesses behave in shared memory multi-processor systems

Sequential consistency

Multi-processor system behaves exactly as if applications were executed interlaced on a uniprocessor

Intuitive model

- most of the programming models rely on this consistency model
- simplest model to handle (software point of view)
  - memory accesses in program order
  - every access reaches memory
  - FIFO type interconnect

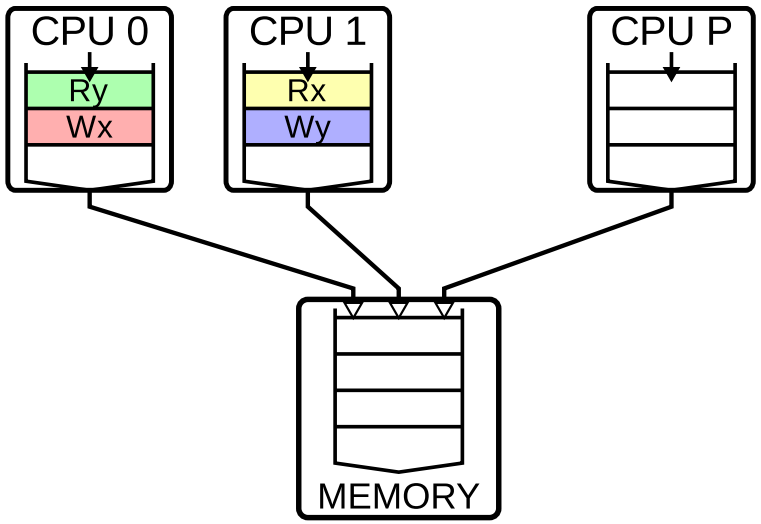
Sequential consistency

Dekker algorithm: Mutual exclusion algorithm

Initially (x == 0) et (y == 0)

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
Mutual exclusion		Mutual exclusion	
end if		**end if**	

Possible Results



Sequential consistency

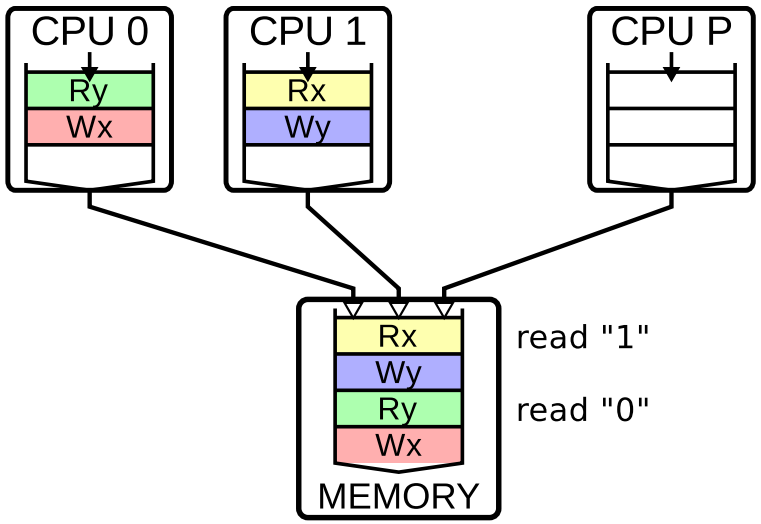
Dekker algorithm: Mutual exclusion algorithm

Initially (x == 0) et (y == 0)

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
Mutual exclusion		Mutual exclusion	
end if		**end if**	

Possible Results

- $(x, y) == (1, 0)$ OK
- $(x, y) == (0, 1)$ OK



Sequential consistency

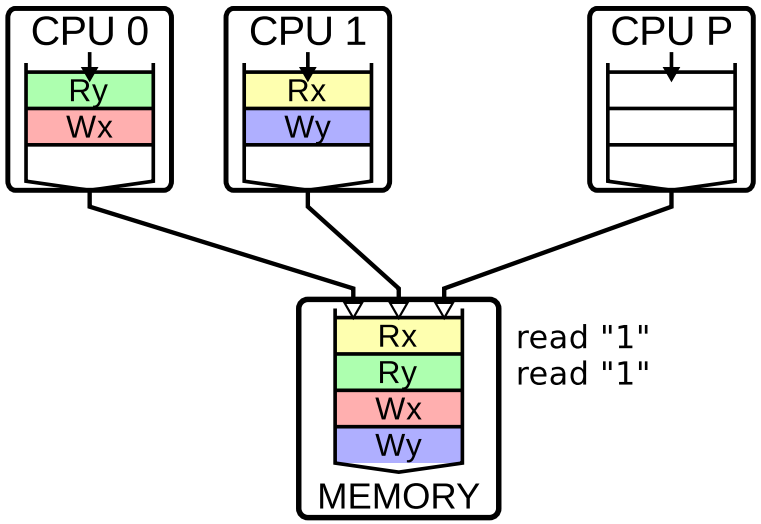
Dekker algorithm: Mutual exclusion algorithm

Initially (x == 0) et (y == 0)

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
Mutual exclusion		Mutual exclusion	
end if		**end if**	

Possible Results

- $(x, y) == (1, 0)$ OK
- $(x, y) == (0, 1)$ OK
- $(x, y) == (1, 1)$ OK



Sequential consistency

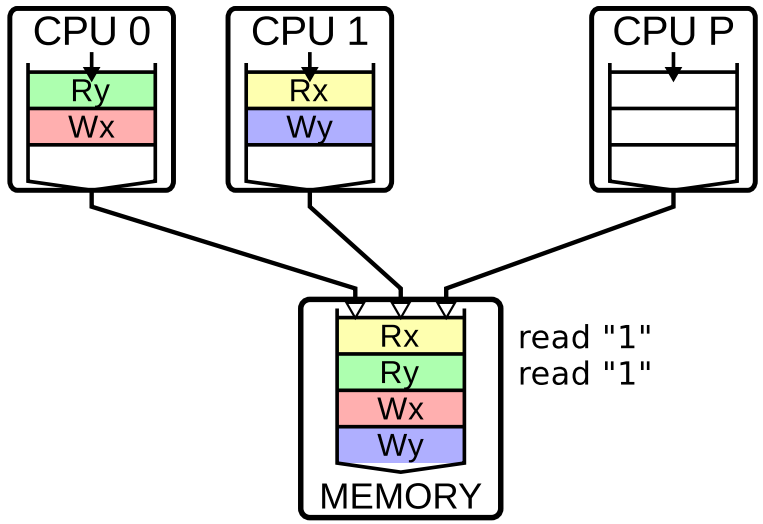
Dekker algorithm: Mutual exclusion algorithm

Initially (x == 0) et (y == 0)

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
Mutual exclusion		Mutual exclusion	
end if		**end if**	

Possible Results

- $(x, y) == (1, 0)$ OK
- $(x, y) == (0, 1)$ OK
- $(x, y) == (1, 1)$ OK
- $(x, y) == (0, 0)$ KO



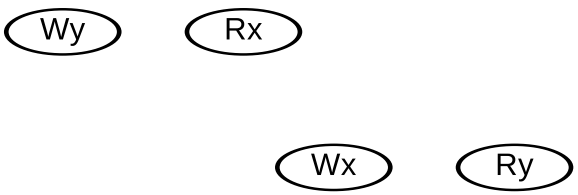
Consistency default detection

Dekker algorithm: Mutual exclusion algorithm

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
end if		**end if**	

Graph formulation

- Node: read, write
- Edge: order constraint



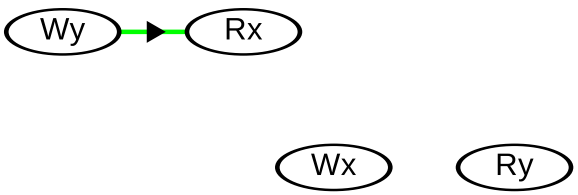
Consistency default detection

Dekker algorithm: Mutual exclusion algorithm

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
end if		**end if**	

Graph formulation

- Node: read, write
- Edge: order constraint



Program order (Sequential consistency)

- W_y before R_x

Consistency default detection

Dekker algorithm: Mutual exclusion algorithm

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
end if		**end if**	

Graph formulation

- Node: read, write
- Edge: order constraint

Program order (Sequential consistency)

- W_y before R_x
- W_x before R_y

Consistency default detection

Dekker algorithm: Mutual exclusion algorithm

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
end if		**end if**	

Graph formulation

- Node: read, write
- Edge: order constraint

Memory order

- x and y: value 0 precedes value 1

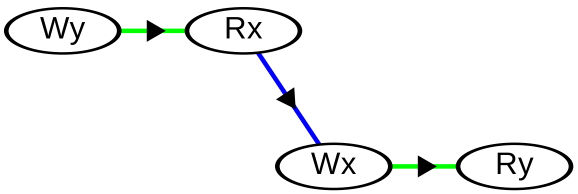
Consistency default detection

Dekker algorithm: Mutual exclusion algorithm

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
end if		**end if**	

Graph formulation

- Node: read, write
- Edge: order constraint



Memory order

- x and y: value 0 precedes value 1
- R_x read 0: R_x precedes W_x

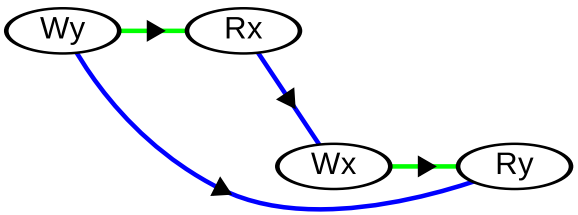
Consistency default detection

Dekker algorithm: Mutual exclusion algorithm

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
end if		**end if**	

Graph formulation

- Node: read, write
- Edge: order constraint



Memory order

- x and y: value 0 precedes value 1
- R_x read 0: R_x precedes W_x
- R_y read 1: W_y precedes R_y

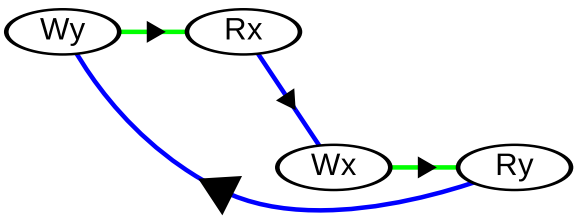
Consistency default detection

Dekker algorithm: Mutual exclusion algorithm

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
end if		**end if**	

Graph formulation

- Node: read, write
- Edge: order constraint



Memory order

- x and y: value 0 precedes value 1
- R_x read 0: R_x precedes W_x
- R_y read 0: R_y precedes W_y

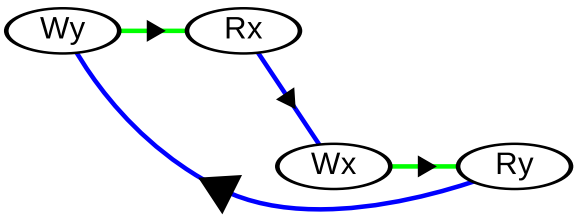
Consistency default detection

Dekker algorithm: Mutual exclusion algorithm

CPU 0		CPU 1	
x = 1	▷ write W_x	y = 1	▷ write W_y
if y == 0 **then**	▷ read R_y	**if** x == 0 **then**	▷ read R_x
end if		**end if**	

Graph formulation

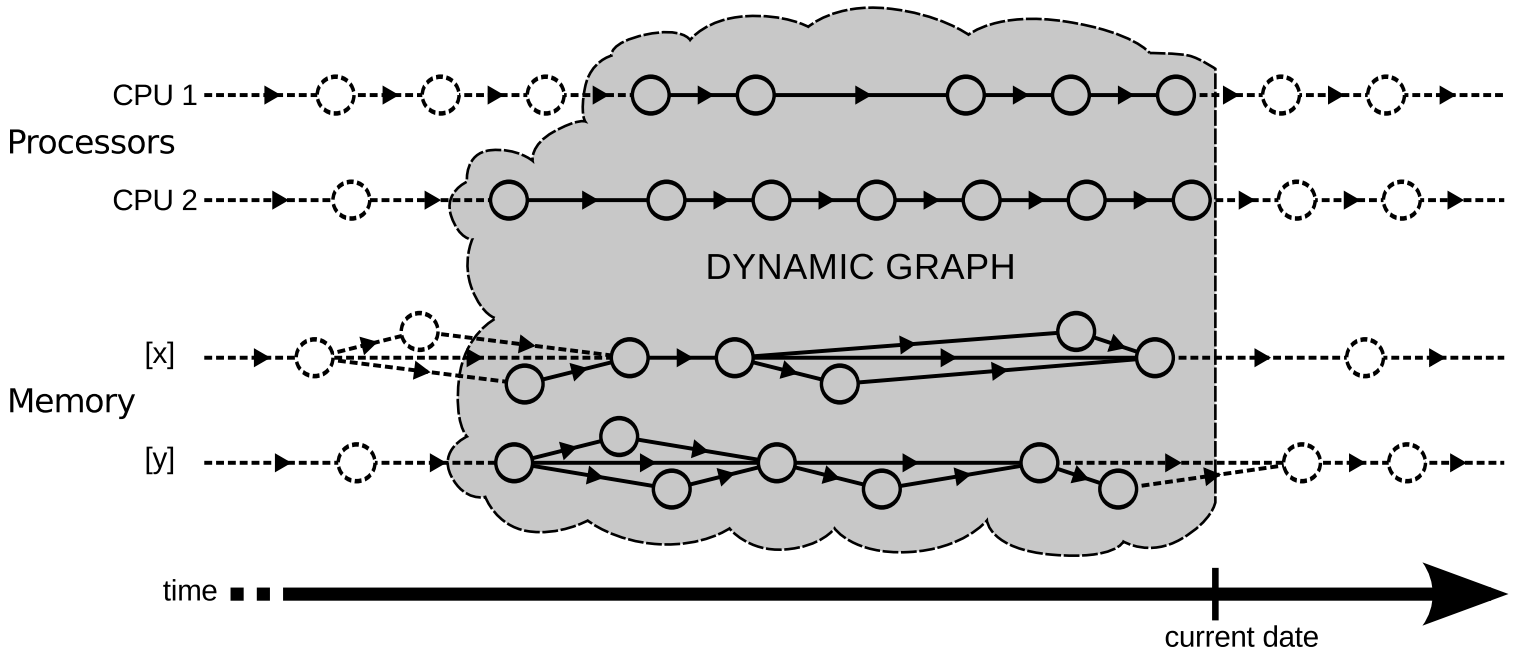
- Node: read, write
- Edge: order constraint



Detection of default

- Cycle <=> Violation

Dynamic graph



- progressive graph creation (while processing trace events)
- each node (access) present in the two sub-graphs (memory and processor)
- Reads returns the value of the last write

Challenges

Goal : avoid graph size and complexity explosion

- Suppressing useless nodes for cycle detection
- Necessary condition: no more predecessors
  - Progressive node suppression
- Execution trace helps in this process:
  - giving overwritten write for each write
  - write accessibility fading

Performances

Reasonable complexity

- linear complexity
- online processing possible

Online analysis cost

Application	Sequential overhead	Simulation speed (kCycles / sec.)	Parallel overhead
MJPEG_1	20.5 %	887	1.9 %
MJPEG_2	28.5 %	672	4.5 %
MJPEG_3	35.7 %	555	6.2 %
MJPEG_4	37.3 %	497	6.8 %
OCEAN_4	21.8 %	229	4.1 %
OCEAN_8	27.2 %	172	2.1 %
OCEAN_16	39.4 %	120	11.9 %
OCEAN_32	43.2 %	67	5.8 %

Software debugging analysis

Large range of possible analysis

- Analysis can highlight bugs not visible on the execution path
  - race condition detection: pure concurrency effects
- Track HW/SW interface implementation
  - Driver state machine checking
- Profile software at instruction granularity
  $\Rightarrow$ metric accounted on the "generating" instruction
  - time and energy performance profiling, ...
  - cache misses, ...

Agenda

- Introduction
- Trace Generation
- Trace analysis
  - Analysis Example
  - Other analysis Example
- **Conclusion**

Conclusion

Simulation Execution Traces for HW/SW debugging

- depends on the quality and accuracy of the models, ...

But

- non-intrusiveness
- opportunity to enrich the information being traced
- reduction of the complexity of computing (for bug detection)
- also applicable on consistency, race conditions, fair profiling, ...