A Case of System-level HW/SW Co-design and Co-verification of a Commodity Multi-Processor System with Custom Hardware

Sungpack Hong*, Tayo Oguntebi, Jared Casper,
Nathan Bronson*, Christos Kozyrakis, Kunle Olukotun

Pervasive Parallelism Lab
Stanford University

* This work was done while the authors were at Stanford.

Trends in hardware design

- Moore’s Law still in effect
  - But no more free lunch…performance is no longer free
- Parallelism
  - Due to limited clock frequency
  - Many CPUs rather than a single super-fast one.
- Heterogeneity
  - Due to limited power budget
  - Specialized HW for a specific task

Questions Raised…

- This trend leads us to a system with …
    - Multiple CPUs
    - Custom HW units
    - All working concurrently for *a single program*
        - More than simple time-sharing
- Questions
    - What does such a system look like?
    - Do we have proper design/verification methodology for such a system?
    - If not, what are the issues?
- This is a case-study presentation to explore these questions rather than to answer them.

Our system as a case study

- External hardware acceleration of software transactional memory on commodity CPUs
  - … wait, what?
- Several x86 CPUs
  - Two sockets, each AMD quad core
  - All running a single multi-threaded application
- Plus a custom HW
  - A FPGA, attached *coherently* to CPUs
  - Accelerating a special software library, called Software Transactional Memory (STM)
- All working in parallel
- Okay then, what is STM?

Backgrounds: Transactional Memory

- Parallel programming is hard
  - data races..
  - Related issues: dead-lock, live-lock, …
- Transactional Memory (TM)
  - A proposal to simplify parallel programming
  - The programmer simply declares critical regions in the program as *transactions* and puts all shared reads/writes inside
  - The runtime system (a.k.a. transactional memory) detects all the runtime data races
  - The runtime roll-backs conflicting transactions and thus guarantees serialize-ability of the program execution.

TM Example: Programming

```
void money_transfer(int account[], int from,
                    int to, int amount) {

  BEGIN_TX();

    int from_before = READ(account[from]);
    int to_before   = READ(account[to]);

    int from_after  = from_before - amount;
    int to_after    = to_before + amount;

    WRITE (account[from], from_after);
    WRITE (account[to], to_after);

  END_TX();
}
```

Begin transaction

Read shared variables

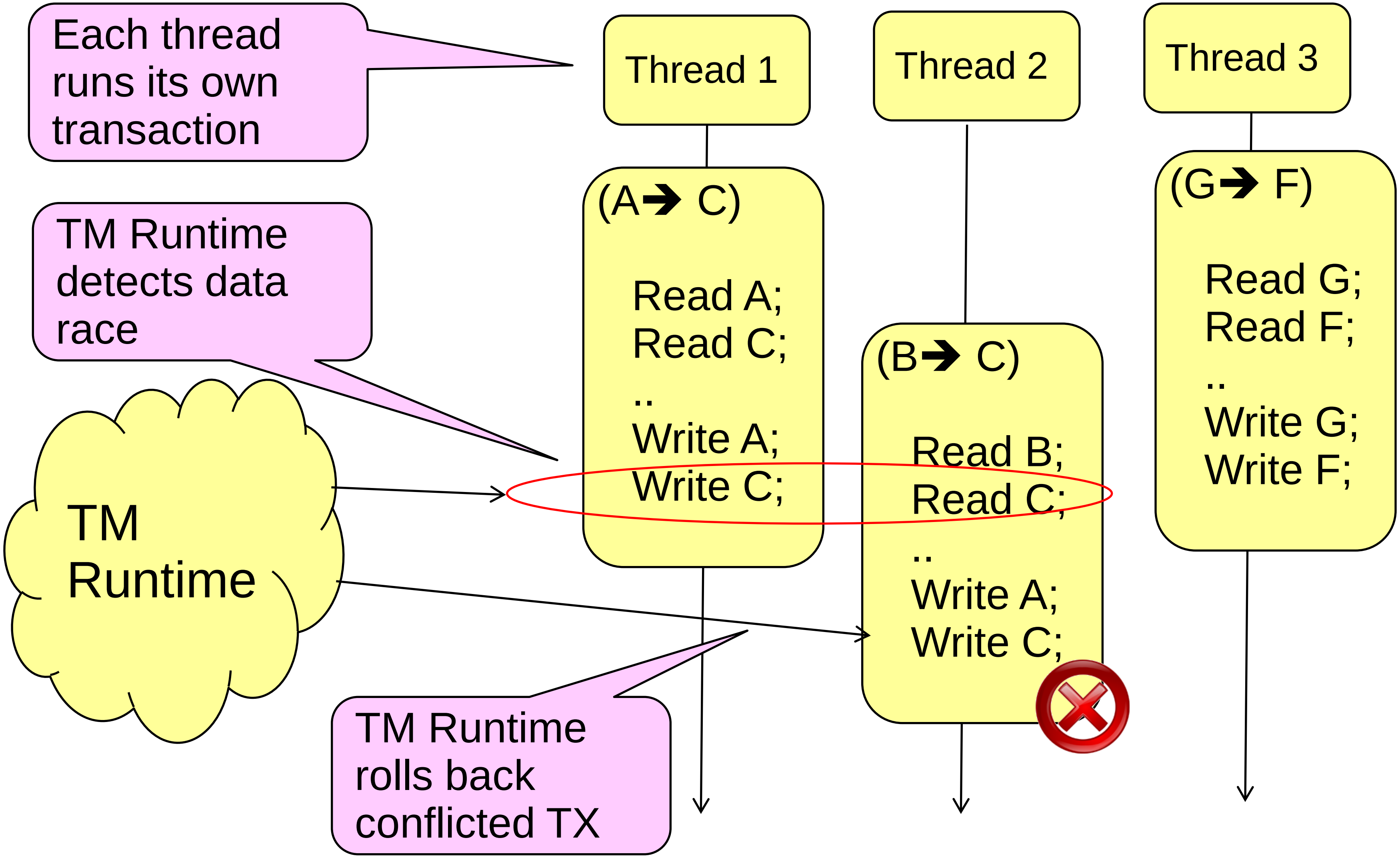
Local computation

Write shared variable computation

End transaction

Each transaction is guaranteed to be “atomic”

TM Example: Runtime



Each thread runs its own transaction

Thread 1

(A➔ C)

Read A;
Read C;
..
Write A;
Write C;

Thread 2

(B➔ C)

Read B;
Read C;
..
Write A;
Write C;

Thread 3

(G➔ F)

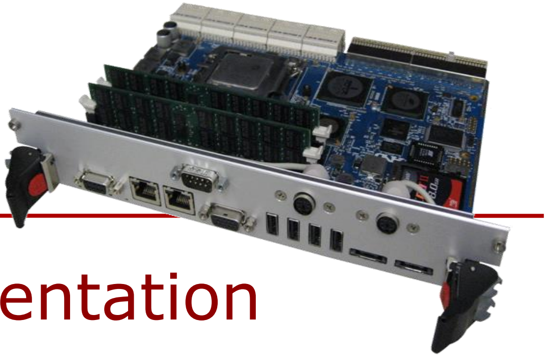
Read G;
Read F;
..
Write G;
Write F;

TM Runtime detects data race

TM Runtime

TM Runtime rolls back conflicted TX

Back to our case

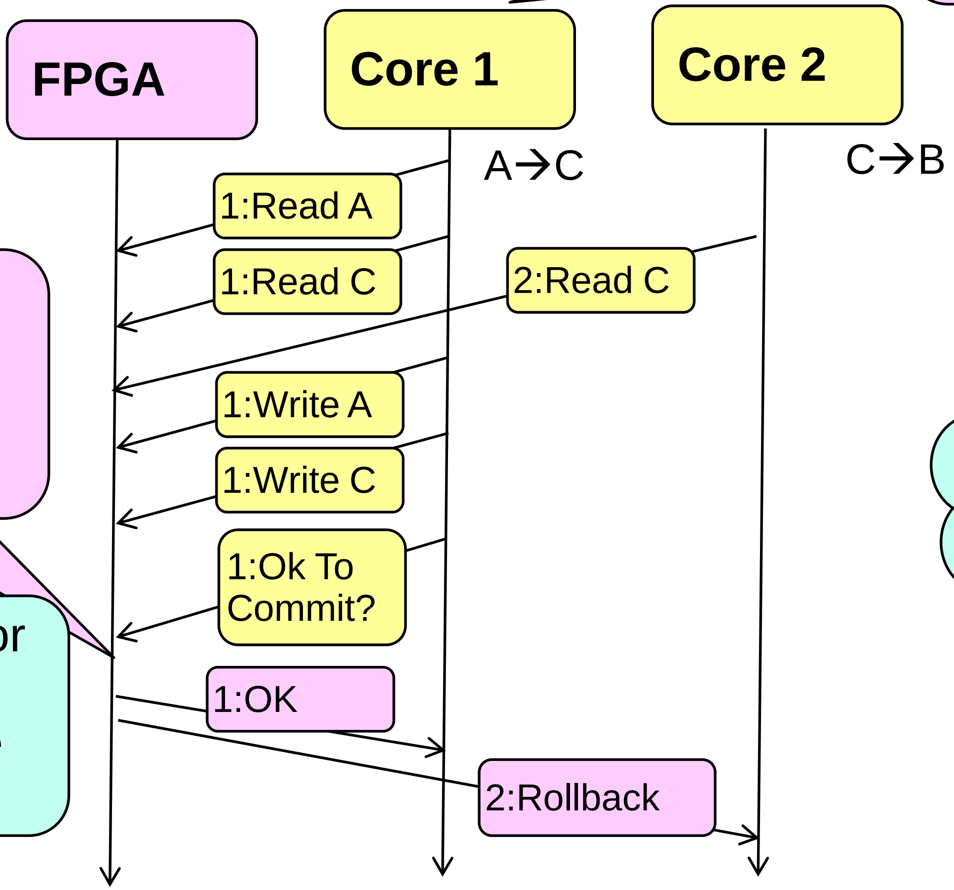


- Approaches in TM runtime implementation
  - STM: all in SW ➔ lots of overhead, slow
  - HTM: all in HW ➔ Requires CPU core modification
    - … but you’d rather avoid changing a commodity core’s RTL
- Our approach
  - Part in HW, rest in SW (a hybrid approach)
  - External custom HW
    - Sits outside cores (on FPGA) via memory bus
    - No core modification required
    - External communication takes some time but we know how to mitigate this!

Our idea in a nutshell

Each core sends a message to FPGA, for each read/write.

+ Some ideas for latency hiding on the SW-side

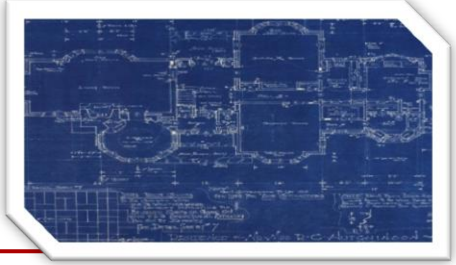


FPGA detects violation and sends notification

+ Some ideas for fast violation detection on the HW-side

"Wow, this is so cool. Let's build this thing already!"

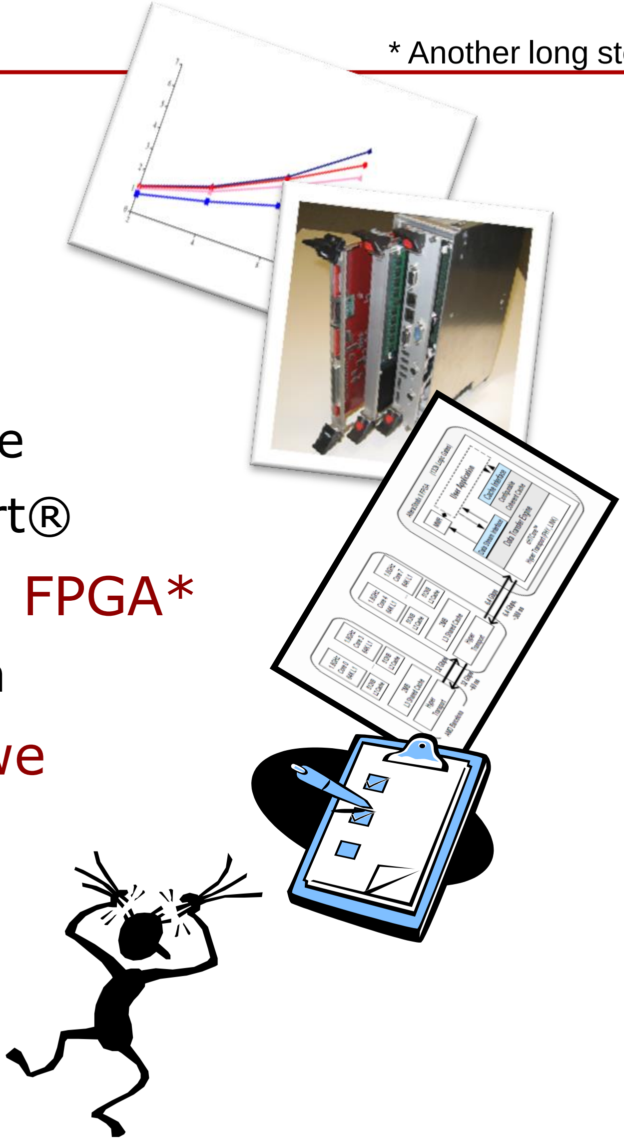
Our initial *co-design*



- How to design a closely-coupled system?
  - New HW
  - New software library (STM) that uses the new HW
- Let’s *simulate* it!
  - Design HW/SW interface (i.e. communication protocol) with a cycle-based x86 ISS (Instruction-Set simulator)
  - Custom HW ➔ pure virtual model
  - Develop the SW on top of simulation
- What about HW design?
  - Do RTL design separately
  - Find RTL bugs with unit test
  - High-level protocol is already validated with simulation

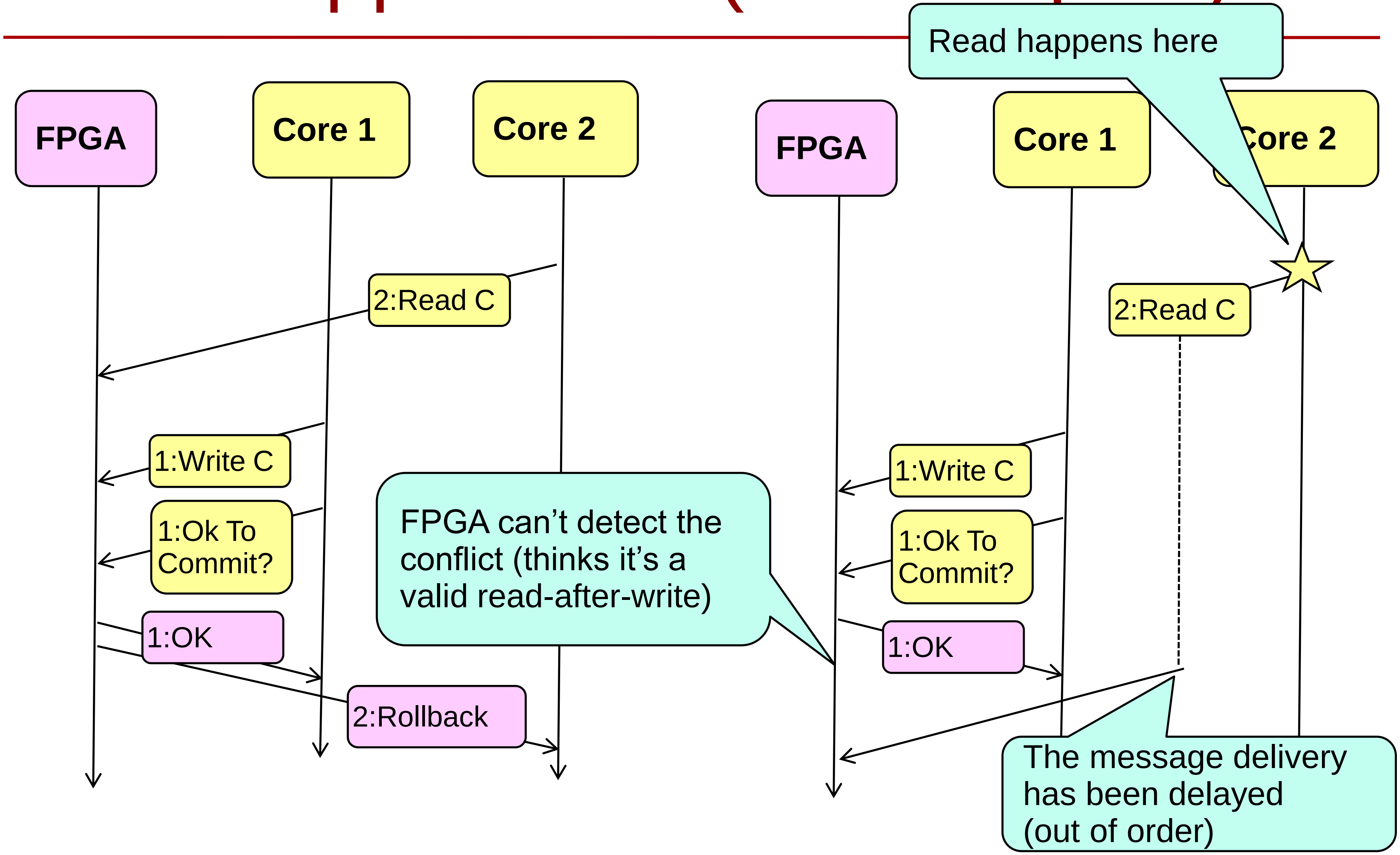
And there we go …

- Our simulation was successful
  - Our protocol works and is faster than conventional STM
- We obtained a HW framework
  - Two sockets, each with AMD quad-core
  - An FPGA connected via HyperTransport®
- We implemented a coherent cache on FPGA*
  - It was a (re-usable) part of our design
- We implemented the custom HW as we designed
  - All the unit tests are passed
- So we ran the whole system ….
  - … And it didn't work
  - Transactions were not atomic at all



* Another long story

What happened..? (In retrospect)



Read happens here

FPGA

Core 1

Core 2

2:Read C

1:Write C

1:Ok To Commit?

1:OK

2:Rollback

FPGA can't detect the conflict (thinks it's a valid read-after-write)

FPGA

Core 1

Core 2

2:Read C

1:Write C

1:Ok To Commit?

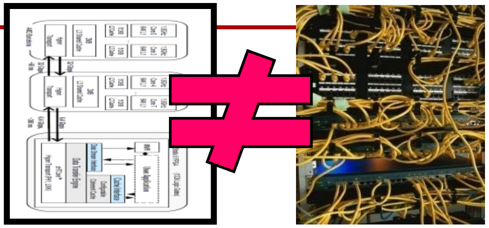
1:OK

The message delivery has been delayed (out of order)

What we designed

What actually happened

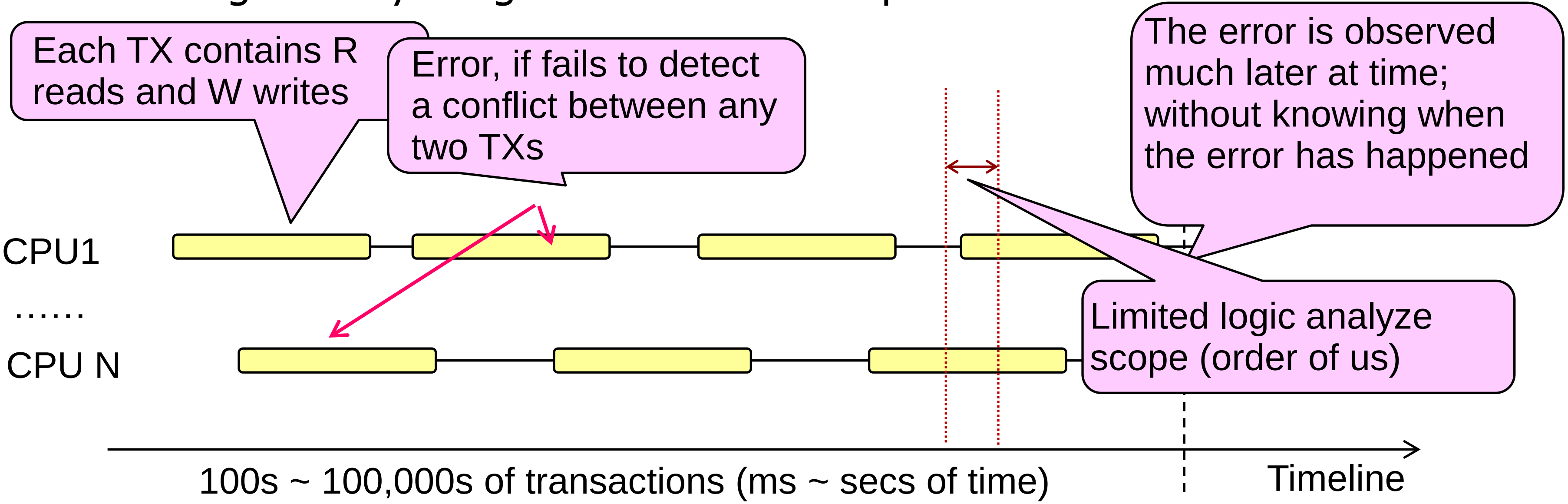
How could we have missed that?



- Problems with our simulator
  - We used a detailed x86 ISS
    - All instructions included and some cache protocols.
  - But the simulated interconnect was far from that of real HW system…
    - HyperTransport + external pin-out + FPGA …
  - The simulation was *in-order* and *deterministic*
    - no latency variance
- Problems with unit testing
  - Cannot generate the complicated error sequence!
  - Requires a lot of *interaction with software*

A Futile Resistance

- “Hey, we already have the FPGA implementation. Let’s just debug it (with a logic analyzer).”
- Problem
  - The ‘time span’ of a typical error is very long
  - It is not clear when and how the problem happens
    - i.e. Error not detectable by a simple trigger
  - Logic analyzer gives a limited scope in time



Each TX contains R reads and W writes

Error, if fails to detect a conflict between any two TXs

The error is observed much later at time; without knowing when the error has happened

Limited logic analyze scope (order of us)

100s ~ 100,000s of transactions (ms ~ secs of time)

What do we need?

- Verification of a concurrent system
  - Interleaving of parallel executions
  - Out-of-order message delivery
  - Many different interleaving in a short time (i.e. fast execution)
- Resemblance to the actual system
  - Actual HW (RTL) + Actual SW debugging preferred
  - Minimum modification for verification
- Crucial features for verification
  - Deterministic replay – the exact same interleaving should be generated at will
  - A better mechanism for bug finding than waveform view
    - Easier log analysis, at least

Comparisons of Available Tools

Method	Pros	Cons
Prototyping	• Target HW + SW • Fast execution	• Limited visibility • No deterministic replay
Full RTL sim. (CPU + interconnection + Custom HW)	• Target HW + SW • Deterministic replay	• All RTL not available • Too slow • No variation of interleaving
Binary instrumentation (i.e. PIN-based simulation)	• Target SW • Fast execution	• No HW debugging • No deterministic replay
Instruction-set sim. + RTL sim (or virtual HW)	• Target SW • Deterministic replay	• No variation of interleaving
SW Model + network sim. (Bus Functional Model) + RTL sim. (or virtual HW)	• Faster than ISS	• SW modification • Variation of interleaving? • Deterministic replay?

✗

?

[Option 1] Modify x86 ISS
- Connect ISS with network sim (BFM) + RTL sim
- Add various interleavings?

[Option 2] Modify Target SW
- Connect SW with BFM + RTL sim
- Add various interleavings
- Add deterministic replay

✓

- **Easier to do**
(you know a lot more about SW than simulator)
- **Faster to run**

ISS-based approach (illustration)

```
void foo(…){
  BEGIN_TX();
  int … = READ(…);
  local_compute();
  WRITE (…);
  END_TX();
}
```

User Program

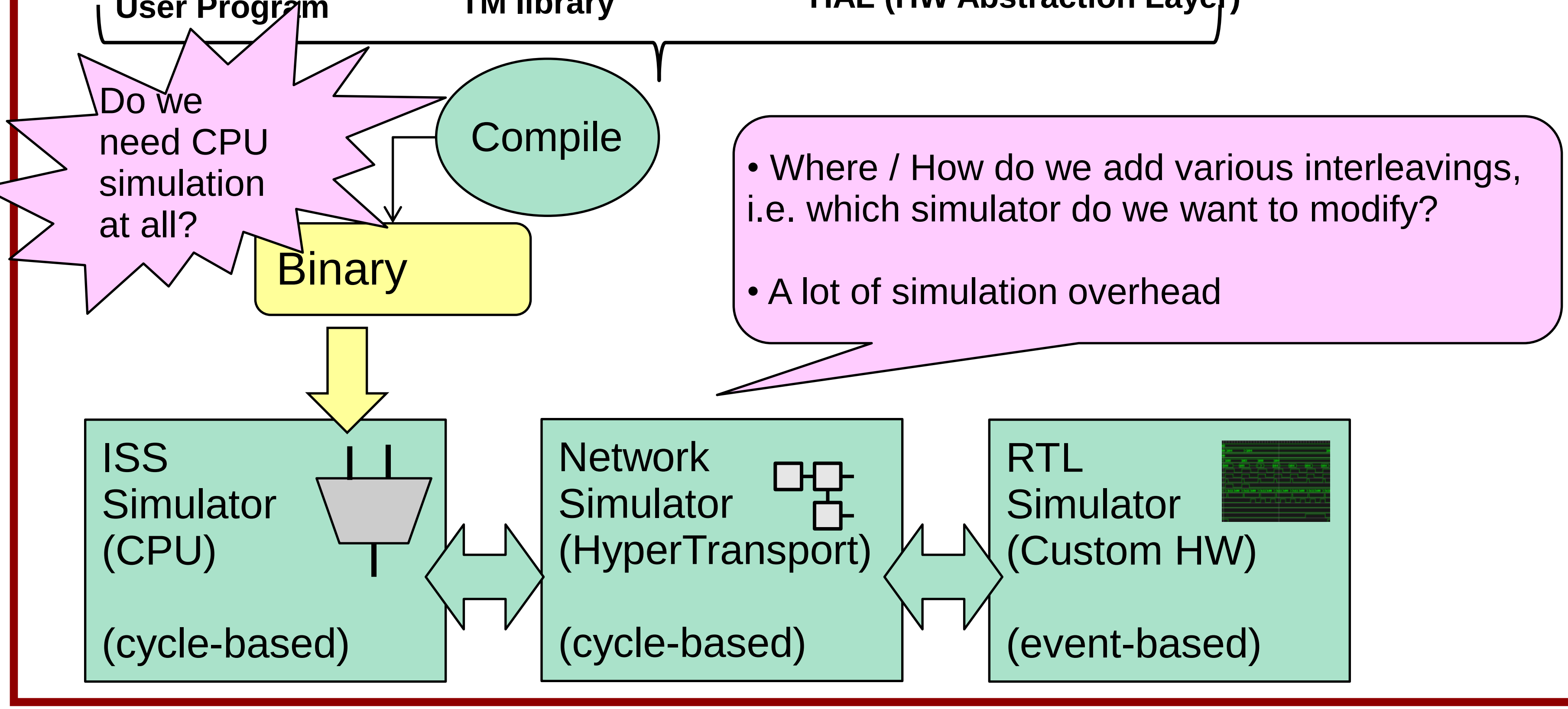
```
void READ(…){
 HW_check_status();
 …
 some_processing();

 HW_send_msg(); … }
```

TM library

```
inline
int HW_check_status(…)
 return
  *FPGA_ADDR & bitmask;
}
```

HAL (HW Abstraction Layer)



Do we need CPU simulation at all?

Compile

Binary

- Where / How do we add various interleavings, i.e. which simulator do we want to modify?
- A lot of simulation overhead

ISS Simulator (CPU)

(cycle-based)

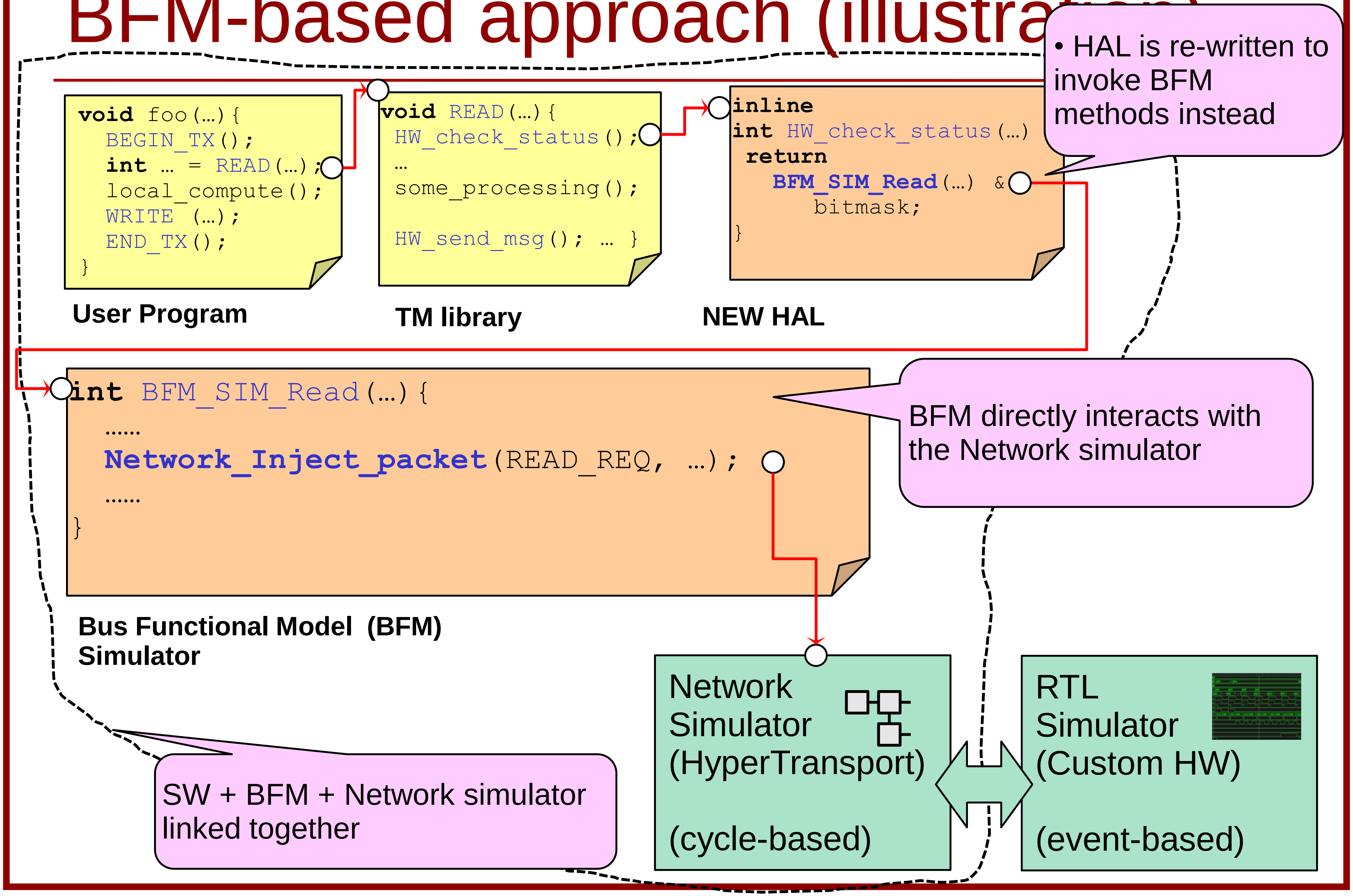
Network Simulator (HyperTransport)

(cycle-based)

RTL Simulator (Custom HW)

(event-based)

BFM-based approach (illustration)



- HAL is re-written to invoke BFM methods instead

```
void foo(…){
  BEGIN_TX();
  int … = READ(…);
  local_compute();
  WRITE (…);
  END_TX();
}
```

User Program

```
void READ(…){
 HW_check_status();
 …
 some_processing();

 HW_send_msg(); … }
```

TM library

```
inline
int HW_check_status(…)
 return
   BFM_SIM_Read(…) &
      bitmask;
}
```

NEW HAL

```
int BFM_SIM_Read(…){
  ……
  Network_Inject_packet(READ_REQ, …);
  ……
}
```

Bus Functional Model (BFM) Simulator

BFM directly interacts with the Network simulator

Network Simulator (HyperTransport)

(cycle-based)

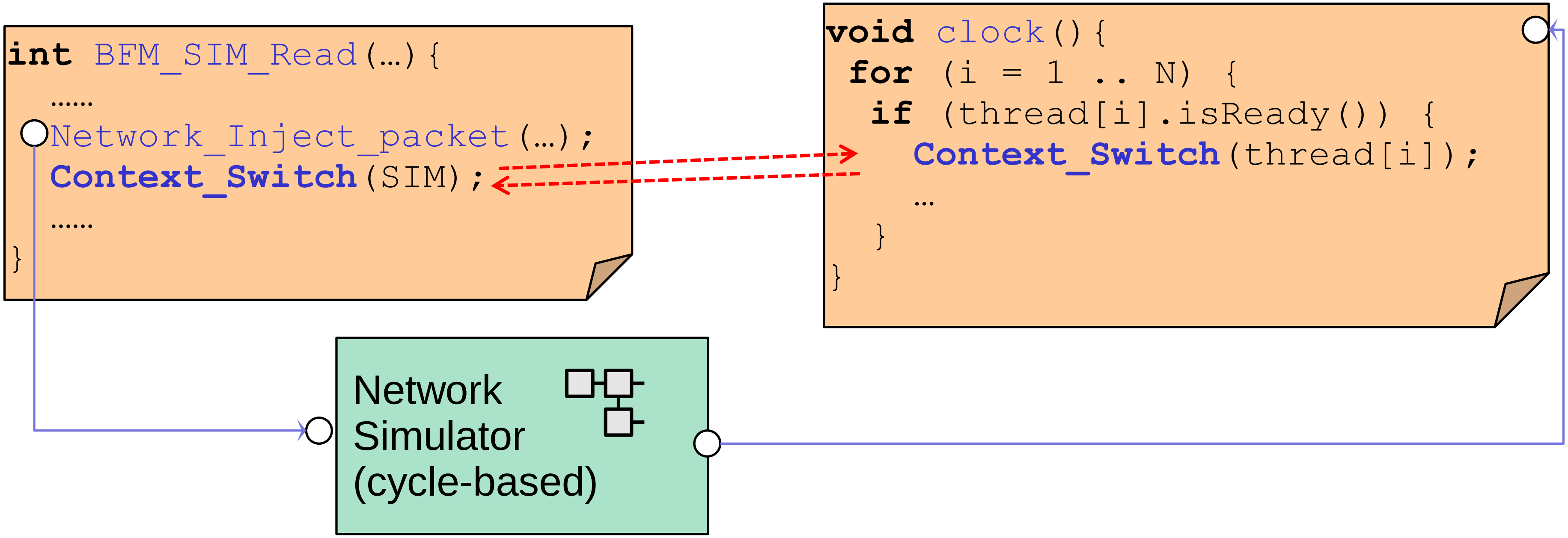
RTL Simulator (Custom HW)

(event-based)

SW + BFM + Network simulator linked together

Our BFM Simulator

- Deterministic Concurrency Control
  - BFM itself is single-threaded
  - BFM uses light-weight threads (i.e. fibers) to implement user threads in the applications
    - Contexts switch happens at network packet injection

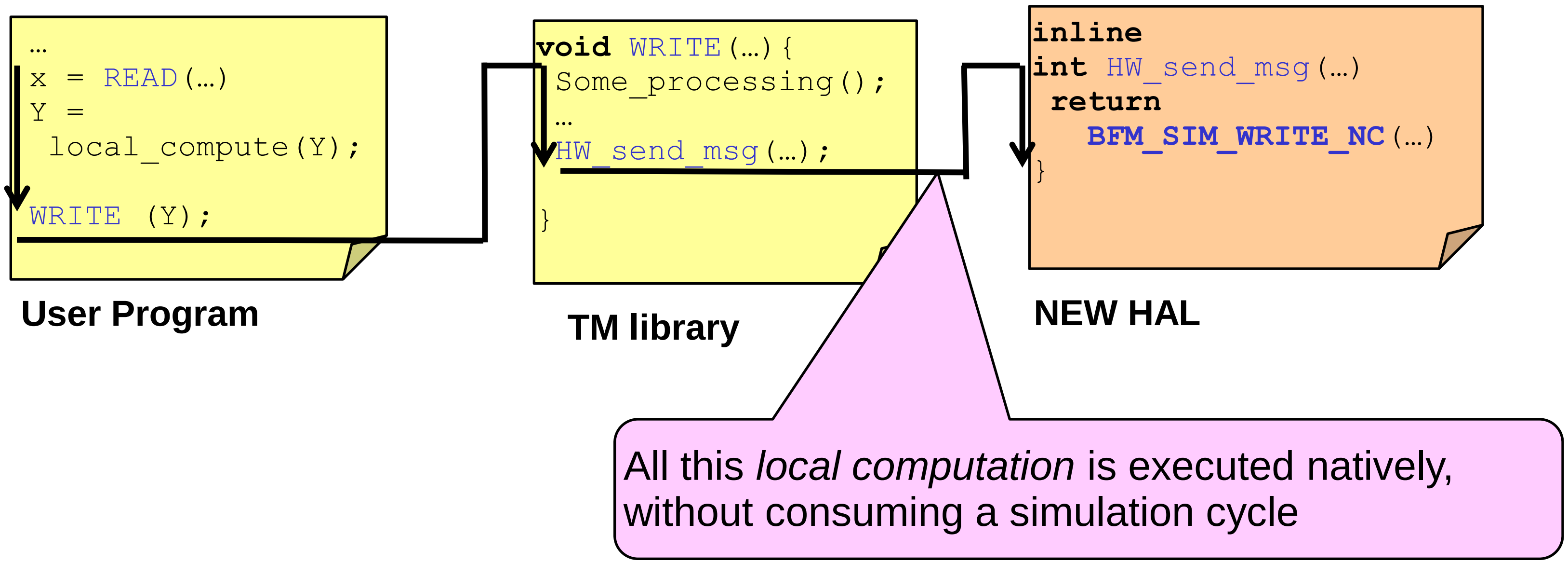


```
int BFM_SIM_Read(…){
   ……
  Network_Inject_packet(…);
  Context_Switch(SIM);
   ……
}
```

```
void clock(){
 for (i = 1 .. N) {
  if (thread[i].isReady()) {
    Context_Switch(thread[i]);
    …
  }
}
```

Our BFM Simulator

- Fast execution
  - All the *local computations* are natively executed
  - No CPU simulation at all
    - We only need software interacting with HW simulation
    - Do not waste simulation cycles for computation



```
…
x = READ(…)
Y =
 local_compute(Y);

WRITE (Y);
```

User Program

```
void WRITE(…){
 Some_processing();
 …
HW_send_msg(…);

}
```

TM library

```
inline
int HW_send_msg(…)
 return
   BFM_SIM_WRITE_NC(…)
}
```

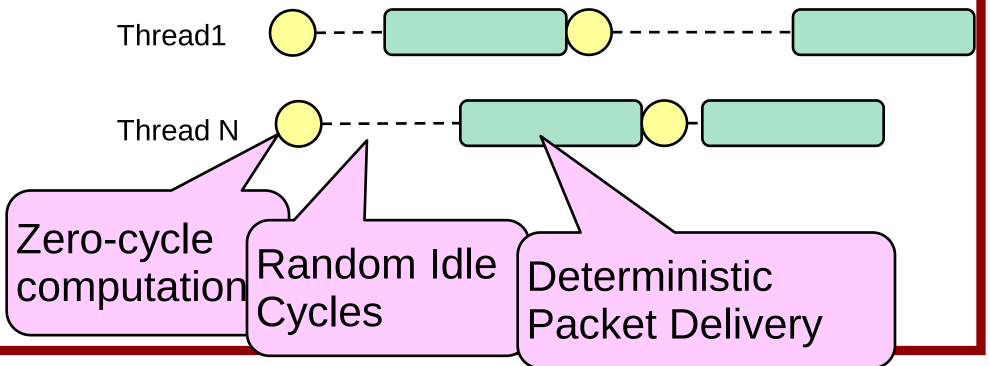
NEW HAL

All this *local computation* is executed natively, without consuming a simulation cycle

Our BFM Simulator

- Variable interleaving of concurrent executions
  + Deterministic replay
  - All the local computation happens at a cycle
  - Actual packet delivery time is deterministic
- ➔ Insert random idle cycles before packet injection
  - Not meant to compensate for computation time
  - But inserts deterministic variation in concurrent executions
- Interleaving is dependent solely on random seed
  - Deterministic re-play ➔ use the same random seed

```
int BFM_SIM_Read(…){
  BFM_Idle_Cycles(get_random());
  Context_Switch(SIM);
  …
  Network_Inject_packet(READ_REQ, …);
  Context_Switch(SIM);
}
```



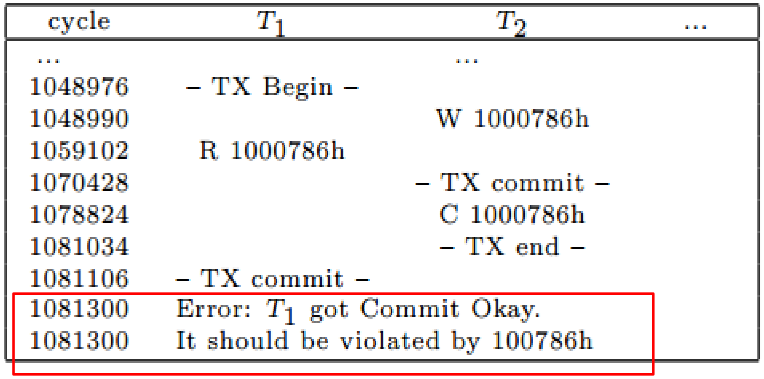
Zero-cycle computation

Random Idle Cycles

Deterministic Packet Delivery

Our BFM Simulator

- Convenient error analysis
  - Logging at high-level
    - at packet level, or
    - at HAL level



cycle	T_1	T_2	…
…		…	
1048976	– TX Begin –		
1048990		W 1000786h	
1059102	R 1000786h		
1070428		– TX commit –	
1078824		C 1000786h	
1081034		– TX end –	
1081106	– TX commit –		
1081300	Error: T_1 got Commit Okay.		
1081300	It should be violated by 100786h		

  - Automatic error detection
    - Simulation = shared-memory, single threaded, deterministic execution
      - Each user thread can *see* what other threads are doing
    - Further modify STM

➔ Maintain a shadow data-structure that checks conflicts on-line (only works for simulation)

Worked well for our case

- Fast simulation enabled many different interleaving of concurrent executions in short time

Small test-bench execution time

Environment	Execution Time
Prototype HW (1.8Ghz x86 + FPGA)	~ 10 ms
BFM + RTL sim	~ 15 mins
BFM + Virtual HW	~ 100 ms
ISS + Virtual HW	~ 100 mins

What we actually used

•For comparisons.
•BFM uses much less simulation cycle

- With this environment, we actually designed and debugged
  - The Custom HW (RTL)
  - The new SW (STM library)
  - And the new communication protocol (system)
  - All together

Generalization and Pitfalls

- Key insights
  - SW modification is easier than simulator modification
  - Local computation can be natively executed
  - Only global communication is simulated via Network simulation
- Caveat: Ease of SW modification
  - Assumes that you can identify HW interface easily
  - Assumes that you can distinguish local computation and global communication (i.e. shared data access)
  - Usually true
    - Parallel SW designed with HAL and critical sections
    - But you should check your SW…
- Not suitable for performance estimation

Requests for CAD Researchers

- Our approach was still ad hoc …
  - Is there a more systematic solution?

- Part-wise selection of details of simulation

  (e.g) Native SW execution (for local computation)

  + Detailed HW simulation (for custom HW design)

  + Detailed network simulation

- Randomizing variance of concurrent executions
  - Should be deterministically re-playable

Summary

- Co-design and Co-verification for post Moore’s law era
  - Parallelism and Heterogeneity
  - Potential concurrency issues at design time
- Required Features
  - Variable interleaving of concurrent executions
  - Deterministic Replay
  - Fast execution time + sufficient of visibility
- In our case study
  - We used SW Model + BFM (interconnection) + RTL sim
  - SW modification was easier than ISS improvement
  - Hope there can be a generalized solution

Questions?