UNIVERSITY OF CAMBRIDGE

Accelerate Cycle-Level Multi-Core RISC-V Simulation with Binary Translation

Xuan Guo, Robert Mullins

Department of Computer Science and Technology

Motivation

- We want to evaluate processor designs with meaningful workloads
  - Not just microbenchmarks
  - Existing simulators are too slow for the task
- Last year we looked at TLB simulation:
  - Fast TLB Simulation for RISC-V Systems @ CARRV 2019
  - We based the work on top of QEMU
  - For TLB design, we don’t really need cycle accuracy
  - The assumption does not hold for cache simulation!

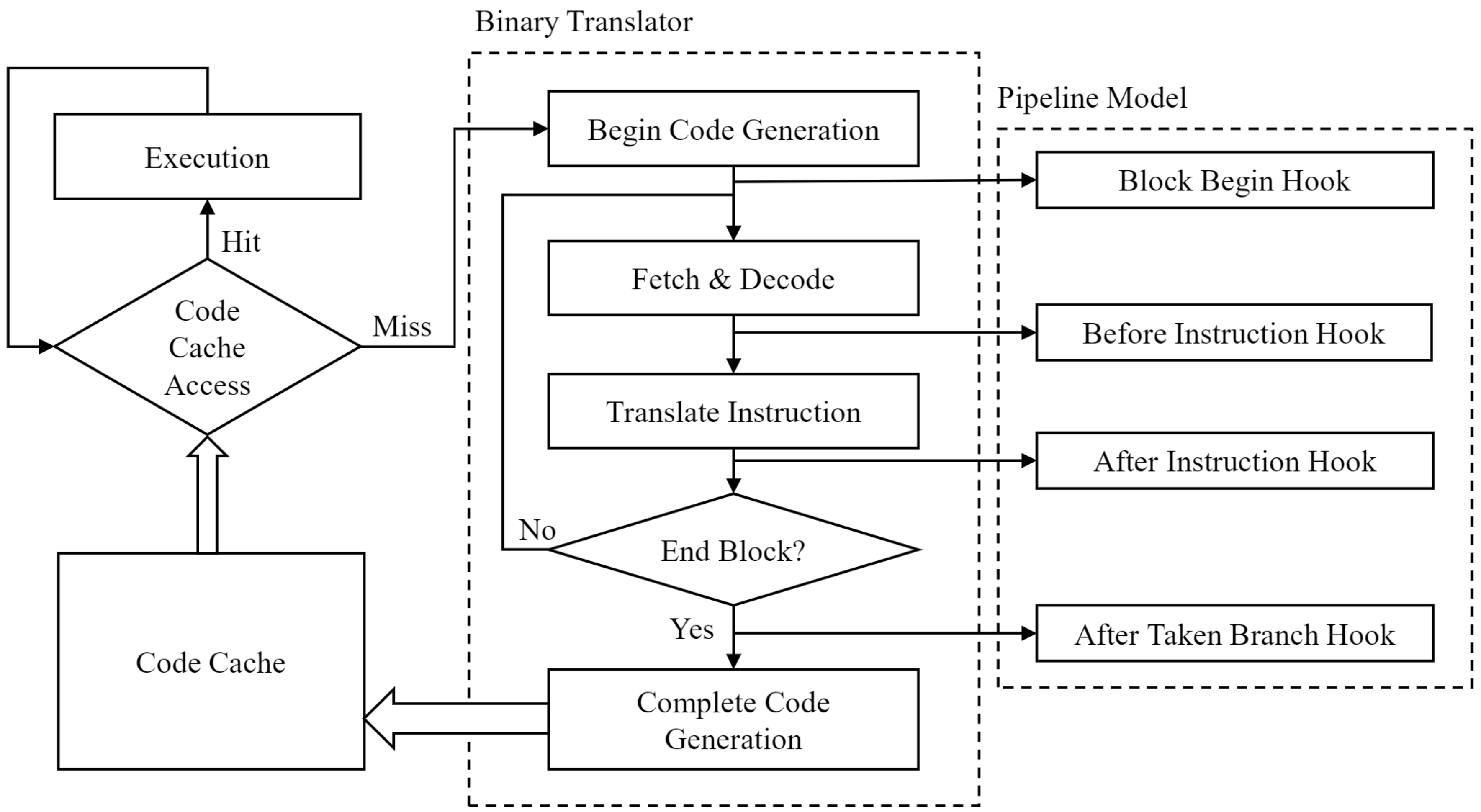
Design Goals

- Full-system capable
  - With the presence of an operating system
- Cycle-level simulation
- Ability to model multicore interaction
  - Include cache coherency and shared caches
- Fast!

R2VM

- **R**ust **R**ISC-V **V**irtual **M**achine

Design


Binary Translator
Pipeline Model
Execution
Hit
Code Cache Access
Miss
Code Cache
Begin Code Generation
Fetch & Decode
Translate Instruction
End Block?
No
Yes
Complete Code Generation
Block Begin Hook
Before Instruction Hook
After Instruction Hook
After Taken Branch Hook

Prior Art

- Igor Böhm, Björn Franke, and Nigel Topham. 2010. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator.

From Single-Core to Multi-Core

- We have an accurate single-core cycle-level simulator
- We instantiate multiple copies of it in parallel
- Assume each single-core simulator is thread safe already
- What could go wrong?

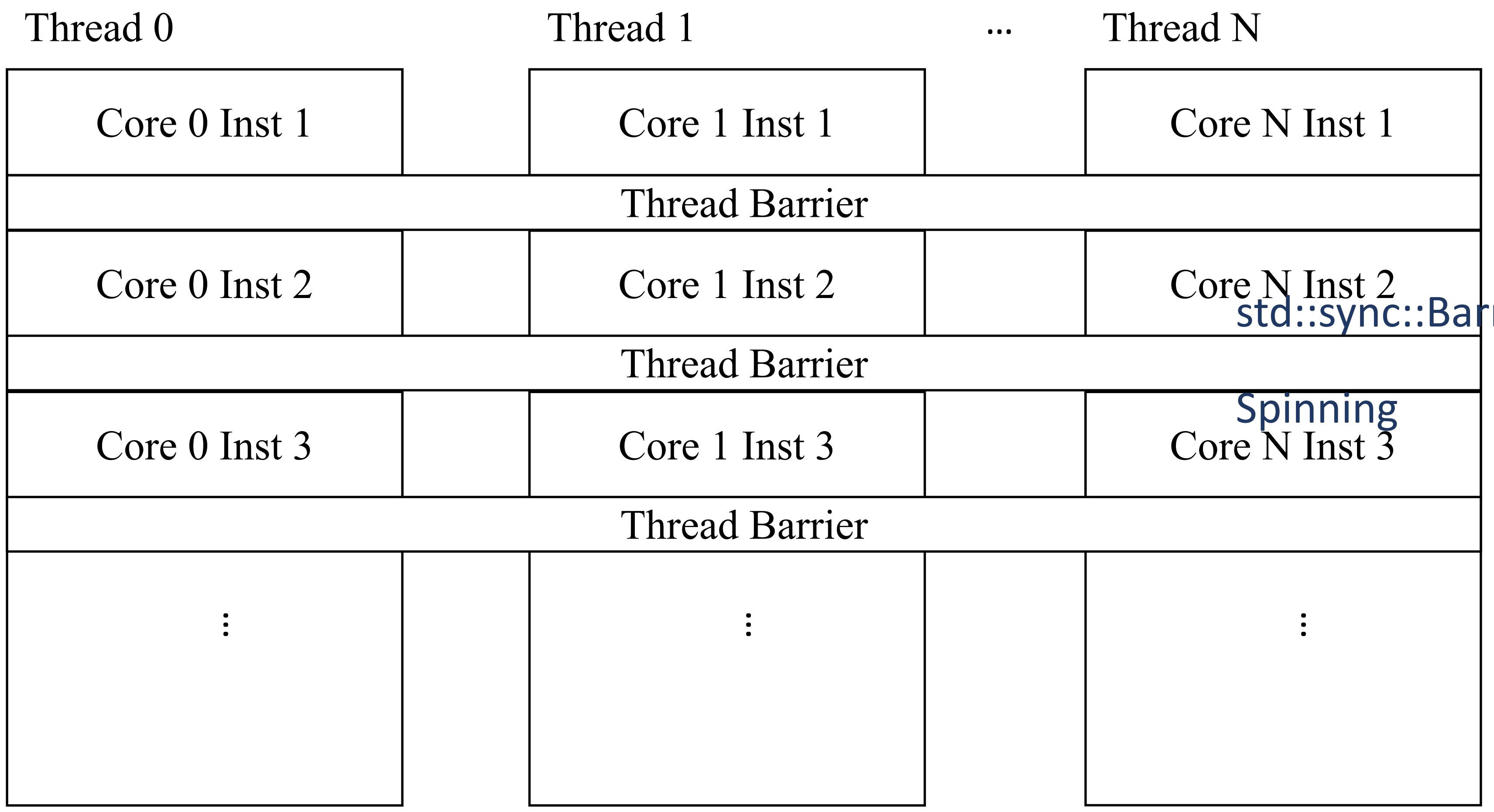
Multi-Core Interaction

- Prone to distortion from the host
  - OS scheduler
  - Length of JITed code
  - Multithreading
- Cannot model interaction within the guest
  - Single-writer-multiple-reader cache coherency
  - Micro-contention
  - Etc

Lockstep Execution

- Need to keep simulated cores in sync
- So we need to have them run in lockstep
- Hard with binary translation

A Failed Attempt



std::sync::Barrier	100k/s
Spinning	1M/s

Lockstep Execution

- Need to keep simulated cores in sync
- So we need to have them run in lockstep
- Hard with binary translation
- Thread barriers are slow and do not scale.

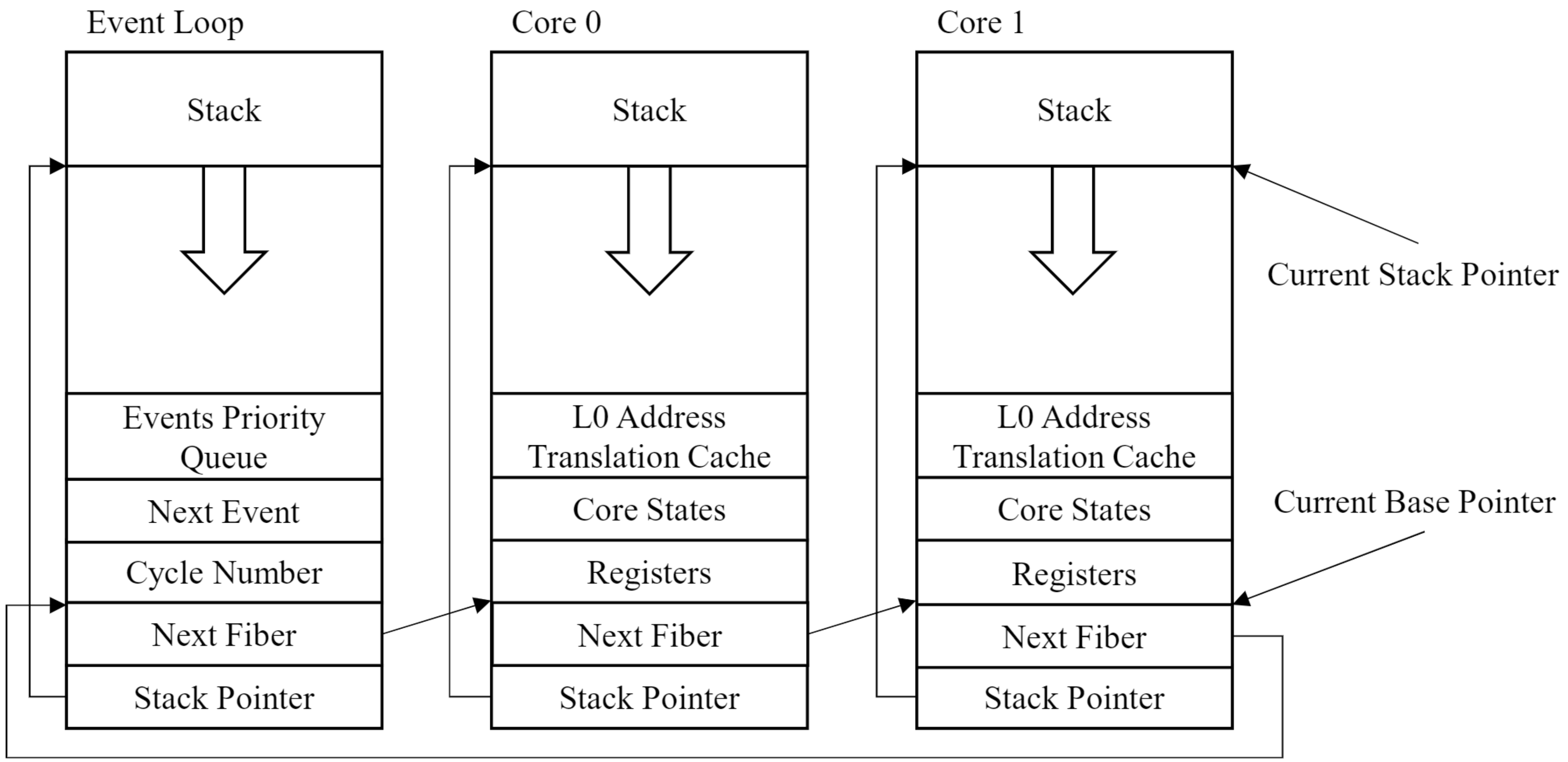
Fiber/Coroutine

- Yield control within a function
- We use stackful fibers
  - Boost::Coroutine is stackful
  - Goroutines are stackful
  - Most modern languages use stackless

Fiber

- How is it implemented (traditional approach):
  - Get the current fiber from TLS
  - Save registers of current fiber
  - Switch to the next fiber and set TLS
  - Switch the stack to the new fiber’s
  - Restore registers from the new fiber
  - Restore execution
- 50M yields/second

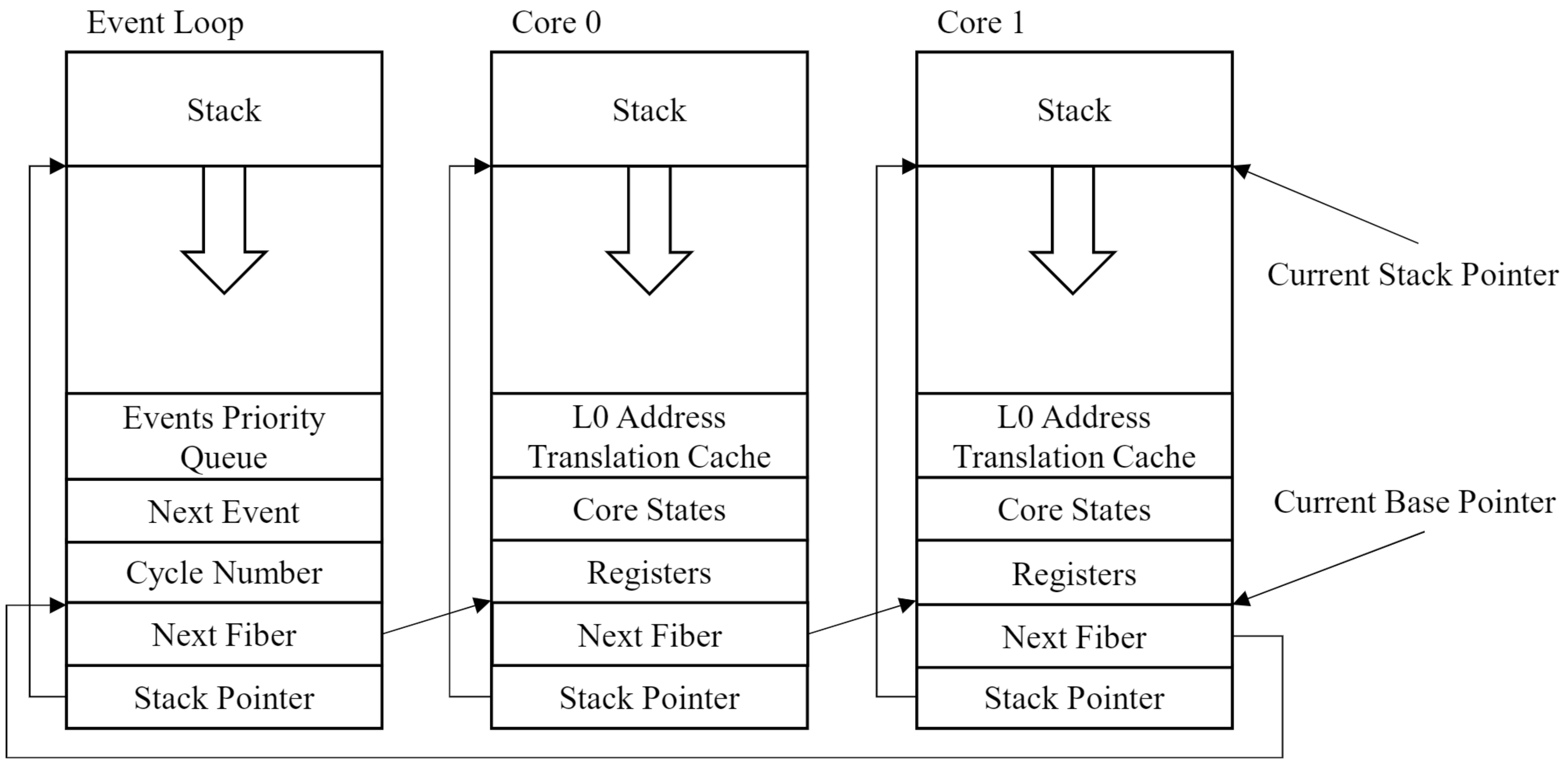
Fiber


Event Loop
Core 0
Core 1
Stack
Stack
Stack
Current Stack Pointer
Events Priority Queue
Next Event
Cycle Number
Next Fiber
Stack Pointer
L0 Address Translation Cache
Core States
Registers
Next Fiber
Stack Pointer
L0 Address Translation Cache
Core States
Registers
Next Fiber
Stack Pointer
Current Base Pointer

Fiber

- How is it implemented (traditional approach):
  - Get the current fiber from TLS
  - ~~Save registers of current fiber~~
  - Switch to the next fiber and set TLS
  - Switch the stack to the new fiber’s
  - ~~Restore registers from the new fiber~~
  - Restore execution
- 50M yields/second

Fiber



Event Loop
Core 0
Core 1
Stack
Stack
Stack
Current Stack Pointer
Events Priority Queue
L0 Address Translation Cache
L0 Address Translation Cache
Current Base Pointer
Next Event
Core States
Core States
Cycle Number
Registers
Registers
Next Fiber
Next Fiber
Next Fiber
Stack Pointer
Stack Pointer
Stack Pointer

Fiber

- How is it implemented (traditional approach):
  - ~~Get the current fiber from TLS~~
  - ~~Save registers of current fiber~~
  - ~~Switch to the next fiber and set TLS~~
  - Switch the stack to the new fiber’s
  - ~~Restore registers from the new fiber~~
  - Restore execution
- 50M yields/second

Fiber

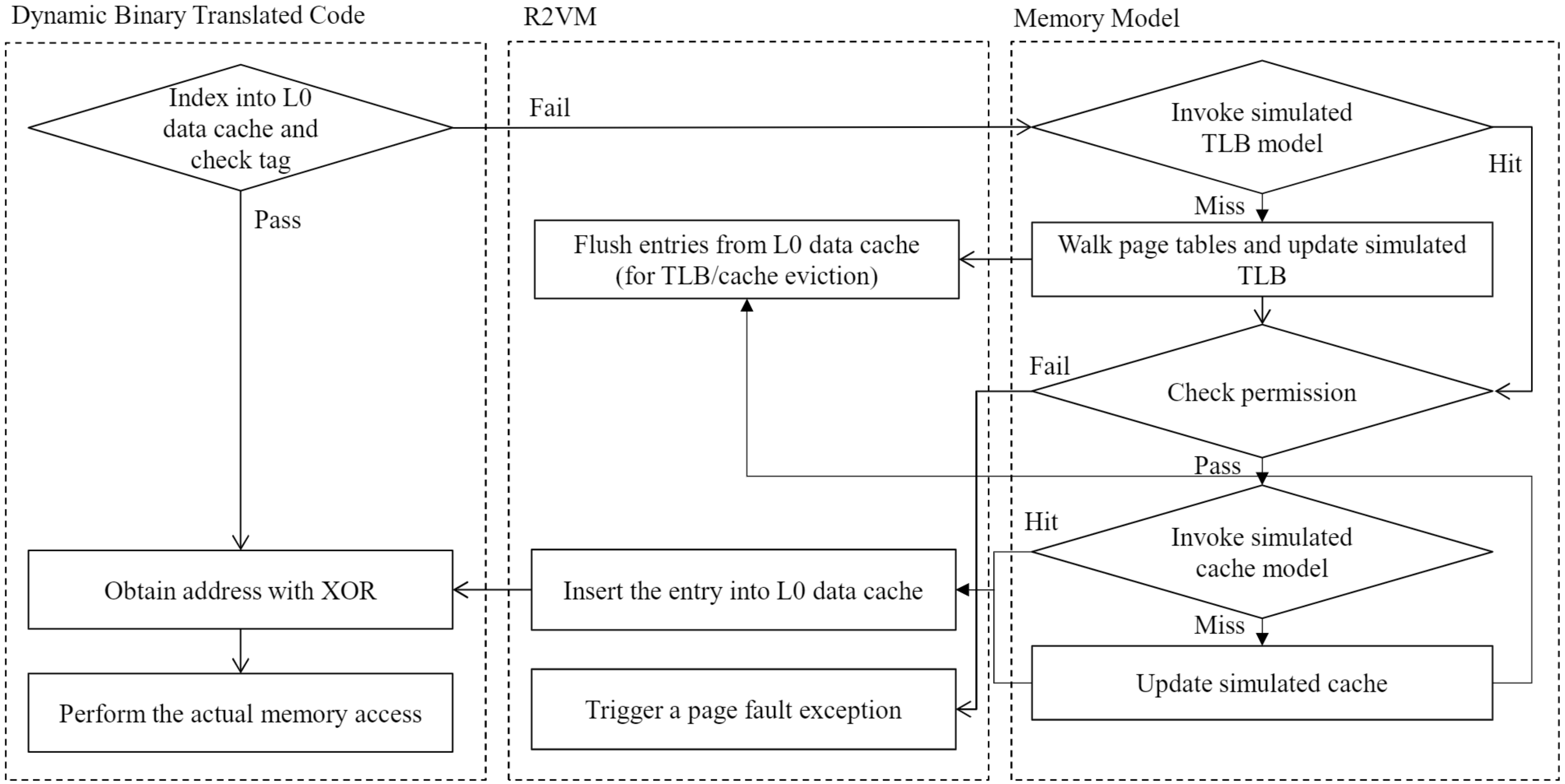
- fiber_yield_raw:

```
        mov [rbp - 32], rsp ; Save current stack pointer
        mov rbp, [rbp - 16] ; Move to next fiber
        mov rsp, [rbp - 32] ; Restore stack pointer
        ret
```

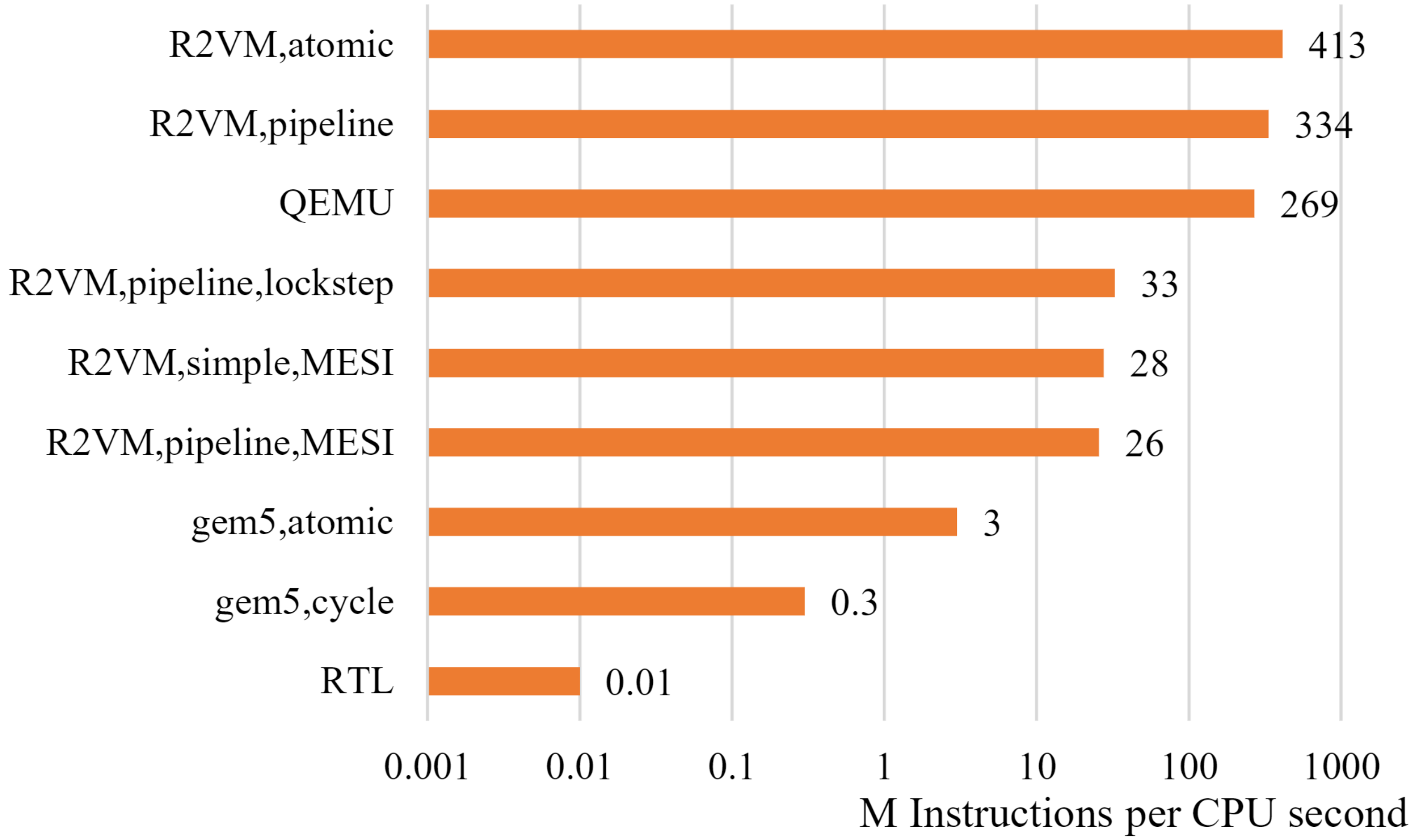
- 80-90M yields/second

Memory Simulation

Memory Access Flow


Dynamic Binary Translated Code
R2VM
Memory Model
Index into L0 data cache and check tag
Fail
Pass
Invoke simulated TLB model
Hit
Miss
Flush entries from L0 data cache (for TLB/cache eviction)
Walk page tables and update simulated TLB
Fail
Check permission
Pass
Hit
Invoke simulated cache model
Miss
Obtain address with XOR
Insert the entry into L0 data cache
Update simulated cache
Perform the actual memory access
Trigger a page fault exception

Performance



Open Source

- https://github.com/nbdd0121/r2vm
- MIT/Apache-2.0 Dual Licensed
    - Not GPL!