Charles University

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Inria

Tapestry

Revealing Wait-For Dependencies Between Application Threads

Tomáš Faltín[1,2], Himadri Chhaya-Shailesh[2], Julia Lawall[2], Jean-Pierre Lozi[2]
[1]*Charles University,* [2]*Inria Paris*

Motivation

Synchronization bottlenecks

Multithreaded applications face ***performance issues*** due to ***synchronization*** causing delays and inefficiencies.

Limitations of traditional profiling

Conventional tools focus on function or hardware slowdowns but ***miss*** inter-thread ***dependencies*** impacting performance.

Challenges of busy-waiting

Busy-waiting threads spin in userspace, ***hiding*** actual wait times as continuous ***execution in traces***.

Full picture visualization

Tapestry visualizes ***blocking*** and ***busy-waiting*** dependencies ***together***, revealing deeper insights into multithreaded performance.

Motivation

Synchronization bottlenecks

Multithreaded applications face ***performance issues*** due to ***synchronization*** causing delays and inefficiencies.

Limitations of traditional profiling

Conventional tools focus on function or hardware slowdowns but ***miss*** inter-thread ***dependencies*** impacting performance.

Challenges of busy-waiting

Busy-waiting threads spin in userspace, ***hiding*** actual wait times as continuous ***execution in traces***.

Full picture visualization

Tapestry visualizes ***blocking*** and ***busy-waiting*** dependencies ***together***, revealing deeper insights into multithreaded performance.

Motivation

Synchronization bottlenecks

Multithreaded applications face ***performance issues*** due to ***synchronization*** causing delays and inefficiencies.

Limitations of traditional profiling

Conventional tools focus on function or hardware slowdowns but ***miss*** inter-thread ***dependencies*** impacting performance.

Challenges of busy-waiting

Busy-waiting threads spin in userspace, ***hiding*** actual wait times as continuous ***execution in traces***.

Full picture visualization

Tapestry visualizes ***blocking*** and ***busy-waiting*** dependencies ***together***, revealing deeper insights into multithreaded performance.

Motivation

Synchronization bottlenecks

Multithreaded applications face ***performance issues*** due to ***synchronization*** causing delays and inefficiencies.

Limitations of traditional profiling

Conventional tools focus on function or hardware slowdowns but ***miss*** inter-thread ***dependencies*** impacting performance.

Challenges of busy-waiting

Busy-waiting threads spin in userspace, ***hiding*** actual wait times as continuous ***execution in traces***.

Full picture visualization

Tapestry visualizes ***blocking*** and ***busy-waiting*** dependencies ***together***, revealing deeper insights into multithreaded performance.

Motivation

Synchronization bottlenecks

Multithreaded applications face ***performance issues*** due to ***synchronization*** causing delays and inefficiencies.

Limitations of traditional profiling

Conventional tools focus on function or hardware slowdowns but ***miss*** inter-thread ***dependencies*** impacting performance.

Challenges of busy-waiting

Busy-waiting threads spin in userspace, ***hiding*** actual wait times as continuous ***execution in traces***.

Full picture visualization

Tapestry visualizes ***blocking*** and ***busy-waiting*** dependencies ***together***, revealing deeper insights into multithreaded performance.

Synchronization

Busy-waiting examples

```
/* Barrier */
if (atomic_inc(thr_count) < n_thr)
  while (*barrier_end != true)
    PAUSE;
else *barrier_end = true;
```

```
/* Simple lock */
while (atomic_test_and_set(lock, 0, 1)
       != 0)
  PAUSE;
```

```
/* Ticket lock */
tkt = atomic_inc(max_tkt);
while (*cur_tkt < tkt) PAUSE;
```

Blocking example

```
futex(*addr, WAIT, val, ...)
```

Synchronization

Busy-waiting examples

```
/* Barrier */
if (atomic_inc(thr_count) < n_thr)
  while (*barrier_end != true)
    PAUSE;
else *barrier_end = true;
```

```
/* Simple lock */
while (atomic_test_and_set(lock, 0, 1)
       != 0)
  PAUSE;
```

```
/* Ticket lock */
tkt = atomic_inc(max_tkt);
while (*cur_tkt < tkt) PAUSE;
```

Blocking example

```
futex(*addr, WAIT, val, ...)
```

Wait-for Dependency

Synchronization

Busy-waiting examples

```
/* Barrier */
if (atomic_inc(thr_count) < n_thr)
  while (*barrier_end != true)
    PA
else *
```

```
/* Sim
while
        != 0)
  PAUSE;
```

```
/* Ticket lock */
tkt = atomic_inc(max_tkt);
while (*cur_tkt < tkt) PAUSE;
```

Blocking example

```
futex(*addr, WAIT, val, ...)
```

How to unify the concept of dependencies for synchronization?

Ordinary Inter-thread Wait-for Dependencies

Busy-waiting examples

```
/* Barrier */
if (atomic_inc(thr_count) < n_thr)
  while (*barrier_end != true)
    PAUSE;
else *barrier_end = true;
```

```
/* Simple lock */
while (atomic_test_and_set(lock, 0, 1)
       != 0)
  PAUSE;
```

```
/* Ticket lock */
tkt = atomic_inc(max_tkt);
while (*cur_tkt < tkt) PAUSE;
```

Blocking example

```
futex(*addr, WAIT, val, ...)
```

Identify common patterns

Ordinary Inter-thread Wait-for Dependencies

Busy-waiting examples

```
/* Barrier */
if (atomic_inc(thr_count) < n_thr)
  while (*barrier_end != true)
    PAUSE;
else *barrier_end = true;
```

```
/* Simple lock */
while (atomic_test_and_set(lock, 0, 1)
       != 0)
  PAUSE;
```

```
/* Ticket lock */
tkt = atomic_inc(max_tkt);
while (*cur_tkt < tkt) PAUSE;
```

Blocking example

```
futex(*addr, WAIT, val, ...)
```

The shared variable memory address

A dependency triple:

<addr, ???, ???>

Ordinary Inter-thread Wait-for Dependencies

Busy-waiting examples

```
/* Barrier */
if (atomic_inc(thr_count) < n_thr)
  while (*barrier_end != true)
    PAUSE;
else *barrier_end = true;
```

```
/* Simple lock */
while (atomic_test_and_set(lock, 0, 1)
       != 0)
  PAUSE;
```

```
/* Ticket lock */
tkt = atomic_inc(max_tkt);
while (*cur_tkt < tkt) PAUSE;
```

Blocking example

```
futex(*addr, WAIT, val, ...)
```

blocks if the futex word ***equals to*** the supplied val

A comparison operator

A dependency triple:

<addr, ⊳⊲, ???>

Ordinary Inter-thread Wait-for Dependencies

Busy-waiting examples

```
/* Barrier */
if (atomic_inc(thr_count) < n_thr)
  while (*barrier_end != true)
    PAUSE;
else *barrier_end = true;
```

```
/* Simple lock */
while (atomic_test_and_set(lock, 0, 1)
       != 0)
  PAUSE;
```

```
/* Ticket lock */
tkt = atomic_inc(max_tkt);
while (*cur_tkt < tkt) PAUSE;
```

Blocking example

```
futex(*addr, WAIT, val, ...)
```

blocks if the futex word ***equals to*** the supplied val

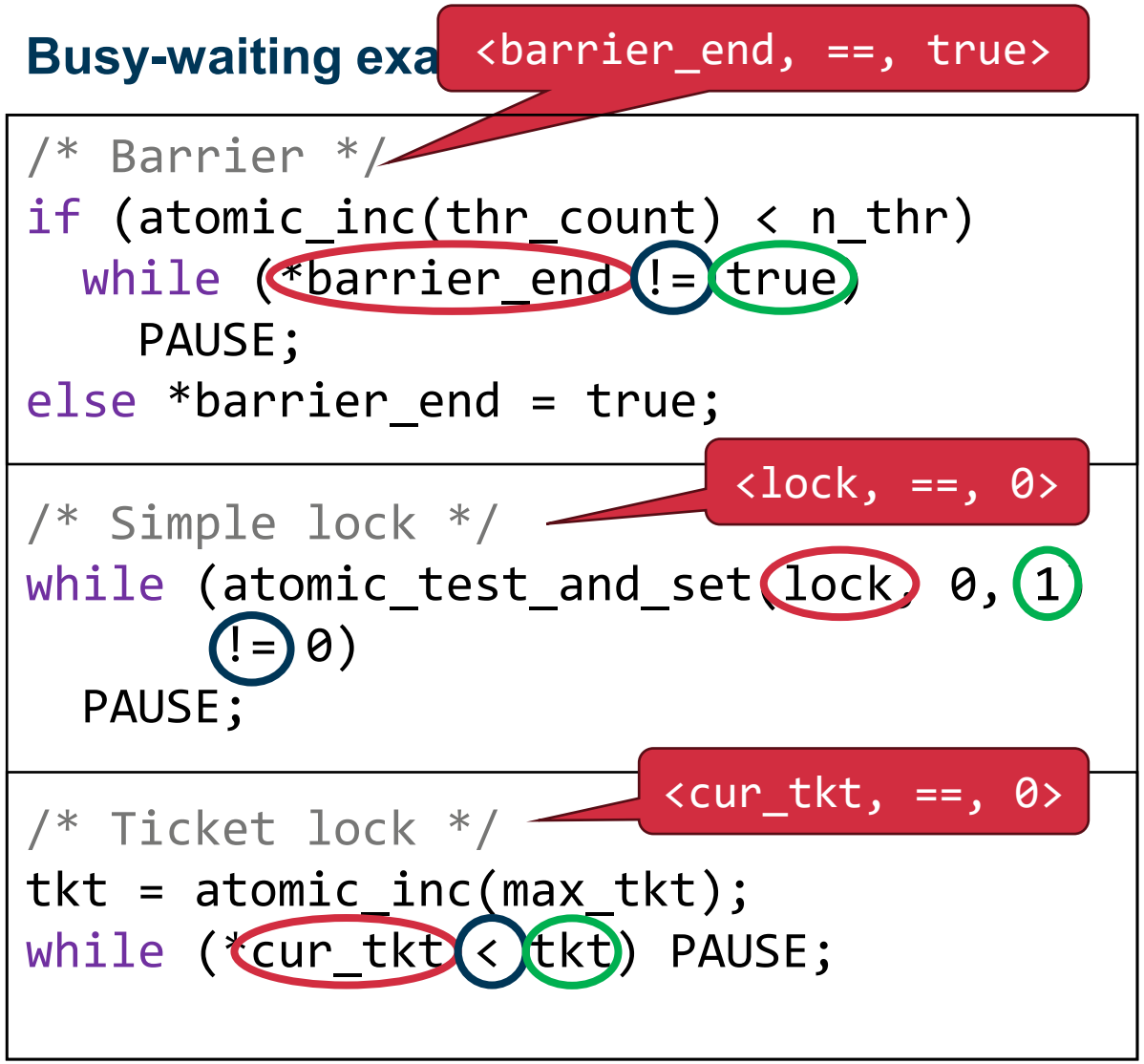
A constant value to compared against

A dependency triple:

<addr, ⊳⊲, val>

Ordinary Inter-thread Wait-for Dependencies

Busy-waiting exa



Blocking example

blocks if the futex word ***equals to*** the supplied val

```
futex(*addr, WAIT, val, ...)
```

<addr, ==, val>

A dependency triple:

<addr, ⊳⊲, val>

Ordinary Inter-thread Wait-for Dependencies

Busy-waiting exa

<barrier_end, ==, true>

```
/* Barrier */
if (atomic_inc(thr_count) < n_thr)
  while (*barrier_end != true)
    PA
else *
```

```
/* Sim
while
      != 0)
  PAUSE;
```

<cur_tkt, ==, 0>

```
/* Ticket lock */
tkt = atomic_inc(max_tkt);
while (*cur_tkt < tkt) PAUSE;
```

Blocking example

blocks if the futex word ***equals to*** the supplied val

```
futex(*addr, WAIT, val, ...)
```

How to capture ordinary inter-thread wait-for dependencies?

A dependency triple:

<addr, ⊳⊲, val>

Ordinary Inter-thread Wait-for Dependencies

Busy-waiting exa

<barrier_end, ==, true>

```
/* Barrier */
if (atomic_inc(thr_count) < n_thr)
  while (*barrier_end != true)
    PA
else *
```

```
/* Sim
while
      != 0)
  PAUSE;
```

<cur_tkt, ==, 0>

```
/* Ticket lock */
tkt = atomic_inc(max_tkt);
while (*cur_tkt < tkt) PAUSE;
```

Blocking example

blocks if the futex word ***equals to*** the supplied val

```
futex(*addr, WAIT, val, ...)
```

Tapestry

A dependency triple:

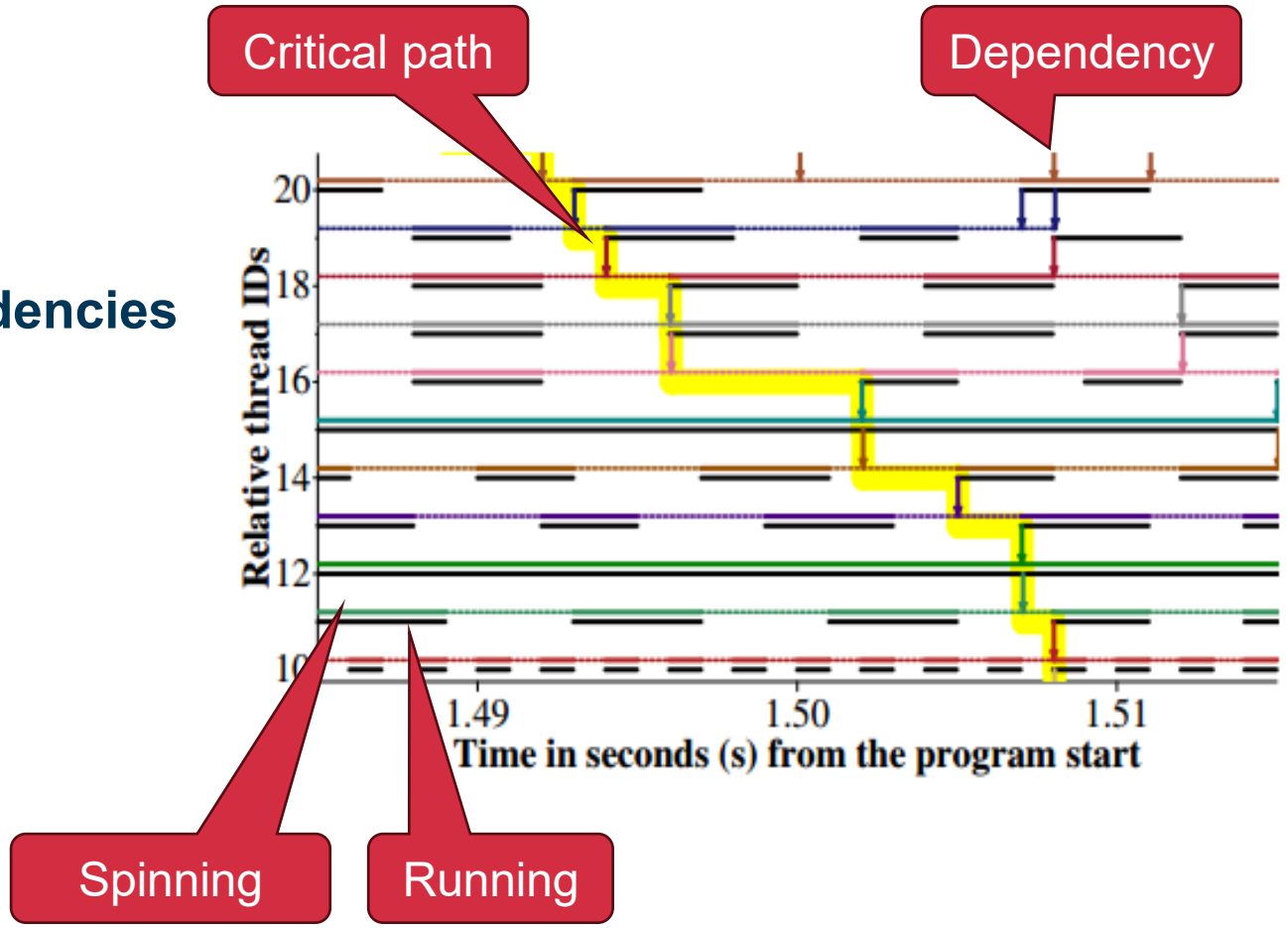
<addr, ⊳⊲, val>

Overview

A Linux tracing framework

- A pipeline: Analyzer → Tracer → Viewer

Captures blocking and busy-waiting **dependencies**
Combines **hardware watchpoints** with **software breakpoints**



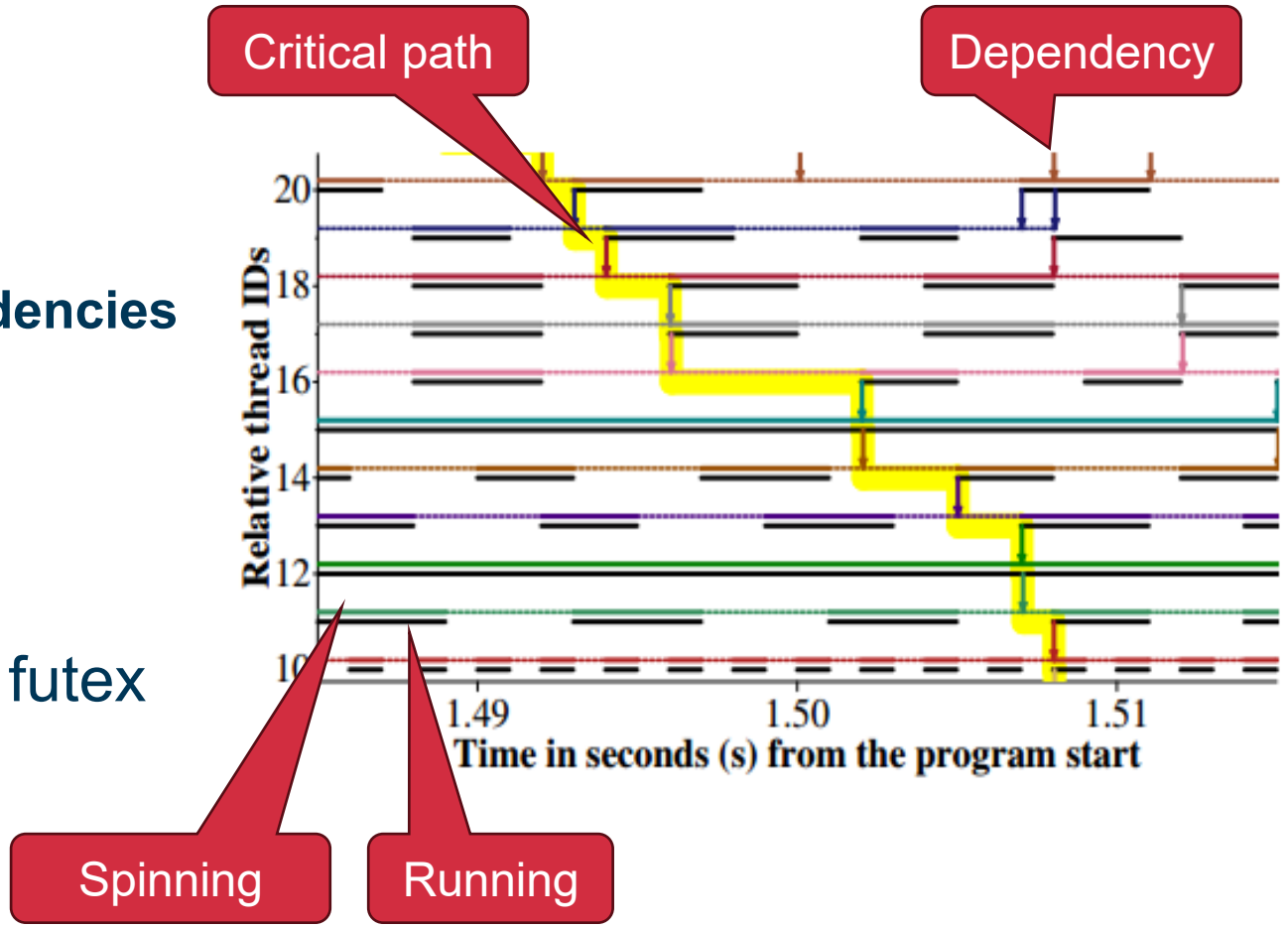
Overview

A Linux tracing framework

- A pipeline: Analyzer → Tracer → Viewer

Captures blocking and busy-waiting **dependencies**
Combines **hardware watchpoints** with **software breakpoints**

Blocking captured by trace-cmd via futex syscalls



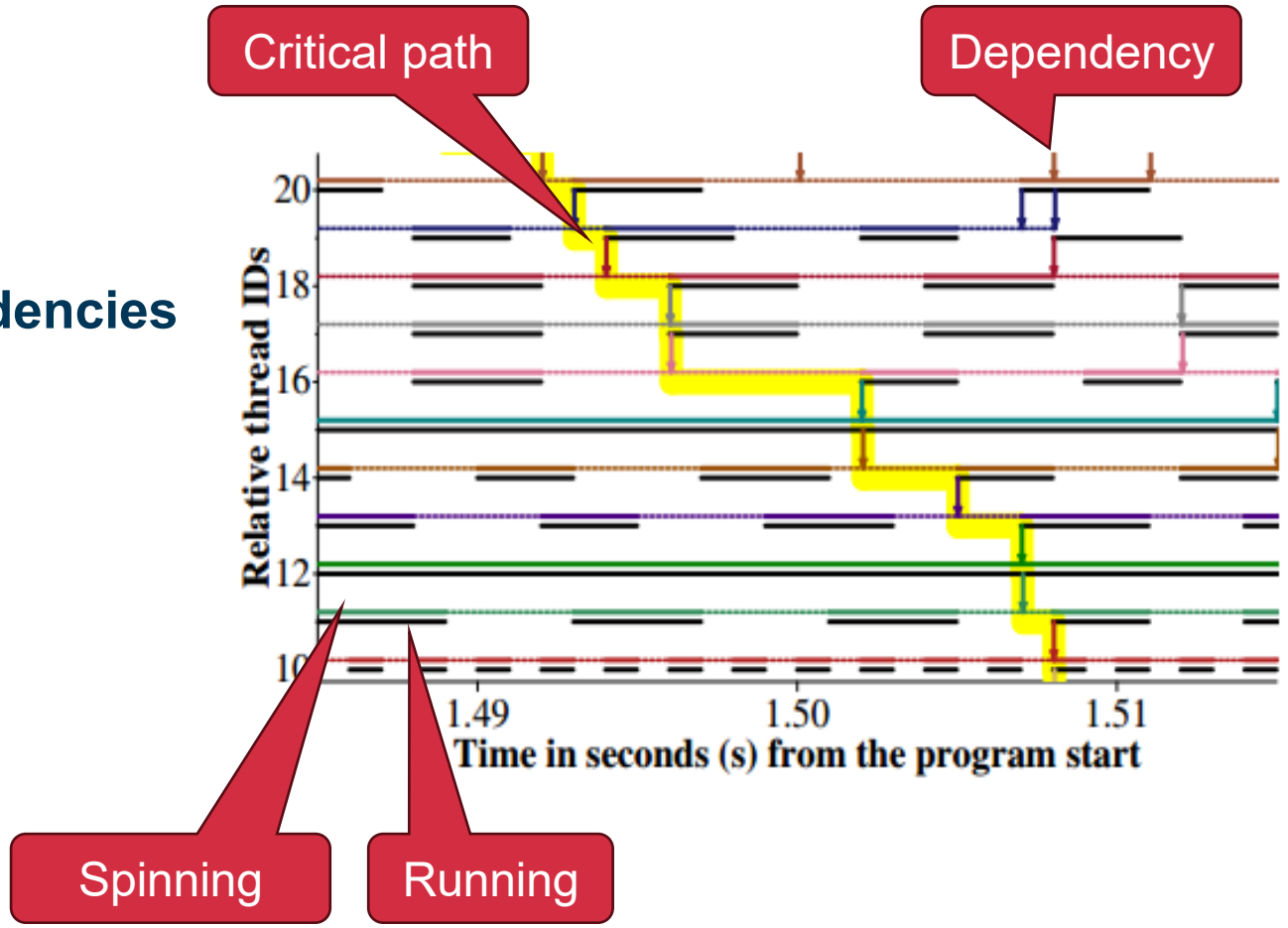
Overview

A Linux tracing framework

- A pipeline: Analyzer → Tracer → Viewer

Captures blocking and busy-waiting **dependencies**
Combines **hardware watchpoints** with **software breakpoints**

How to capture **busy-waiting**?



Architecture: Analyzer → Tracer → Viewer

Finding all busy-wait loops in binaries

Heuristic similar to one proposed by
Jannesari & Tichy, IPDPS 2010

- The loop exit condition depends on memory load
- The conditions' value is not changed by the loop body

```
/* Barrier */
if (atomic_inc(thr_count) < n_thr)
  while (*barrier_end != 0) PAUSE;
else *barrier_end = 1;
```

```
/* Simple lock */
while (atomic_test_and_set(
    lock, 0, 1) != 0)
  PAUSE;
```

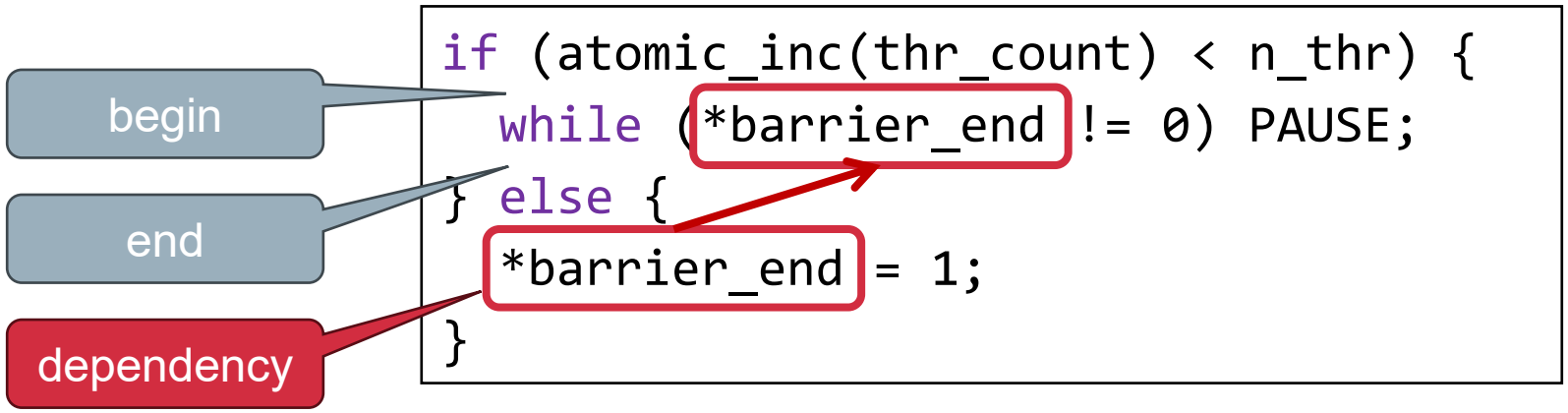
```
/* Ticket lock */
tkt = atomic_inc(max_tkt);
while (*cur_tkt < tkt) PAUSE;
```

Architecture: Analyzer → Tracer → Viewer

Capturing busy-waiting synchronization

a) Capture begin and end of busy-wait loop

b) Capture **dependency**



Capturing Busy-Waiting Synchronization

a) Capture begin and end of busy-wait loop

Statically patch the **beginning** and the **end** of each busy-wait loop to store a **tracepoint**

```
while (*barrier_end != 0) PAUSE;
```

```
spinloop:
  pause
  mov (%rax), %rbx
  cmp %rbx, %rcx
  jne spinloop
```

Capturing Busy-Waiting Synchronization

a) Capture begin and end of busy-wait loop

Statically patch the **beginning** and the **end** of each busy-wait loop to store a **tracepoint**

```
store_time_begin(spinId);
while (*barrier_end != 0) PAUSE;
store_time_end(spinId);
```

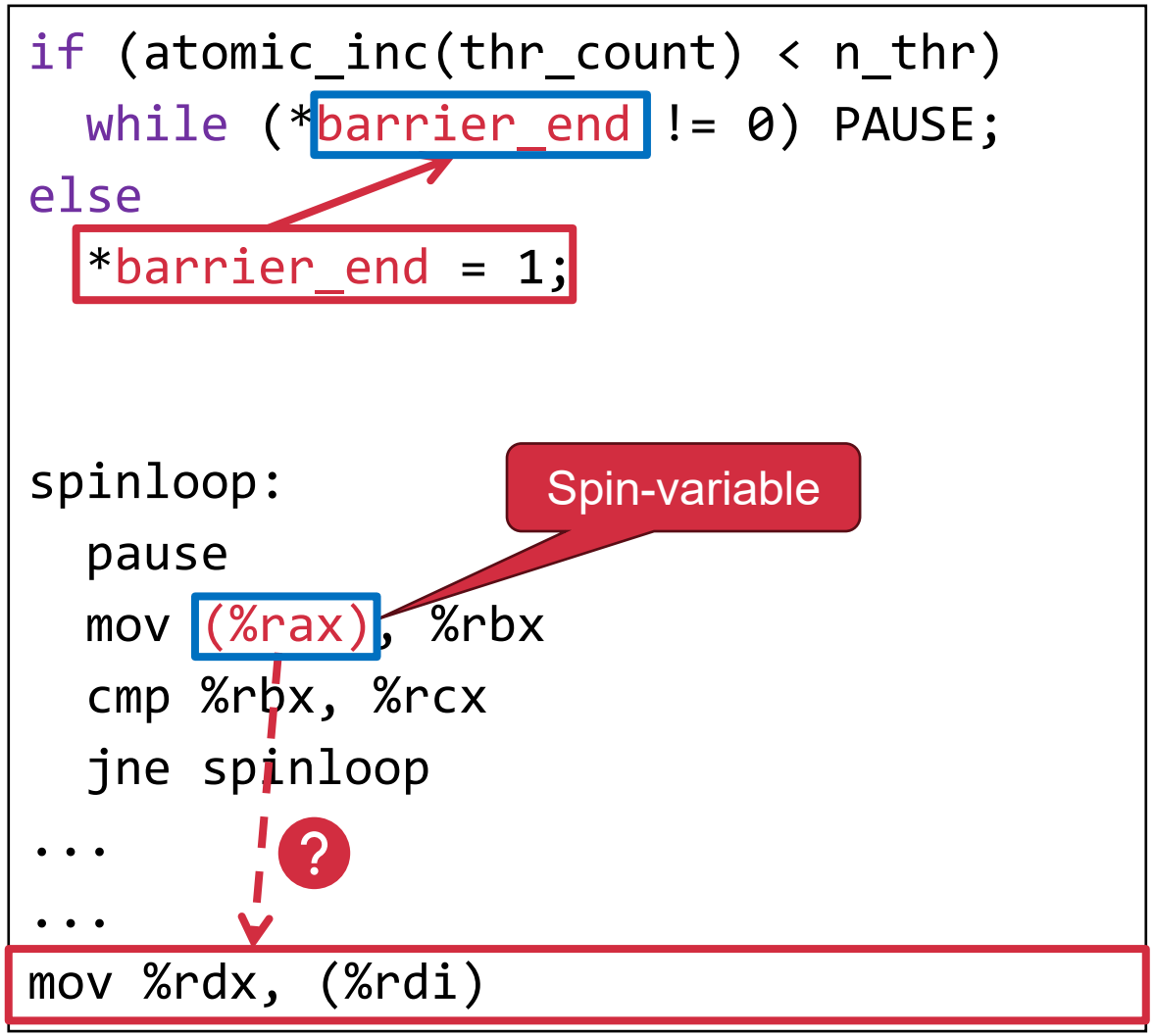
```
call store_time_begin
spinloop:
  pause
  mov (%rax), %rbx
  cmp %rbx, %rcx
  jne spinloop
call store_time_end
```

Capturing Busy-Waiting Synchronization

b) Capturing dependency

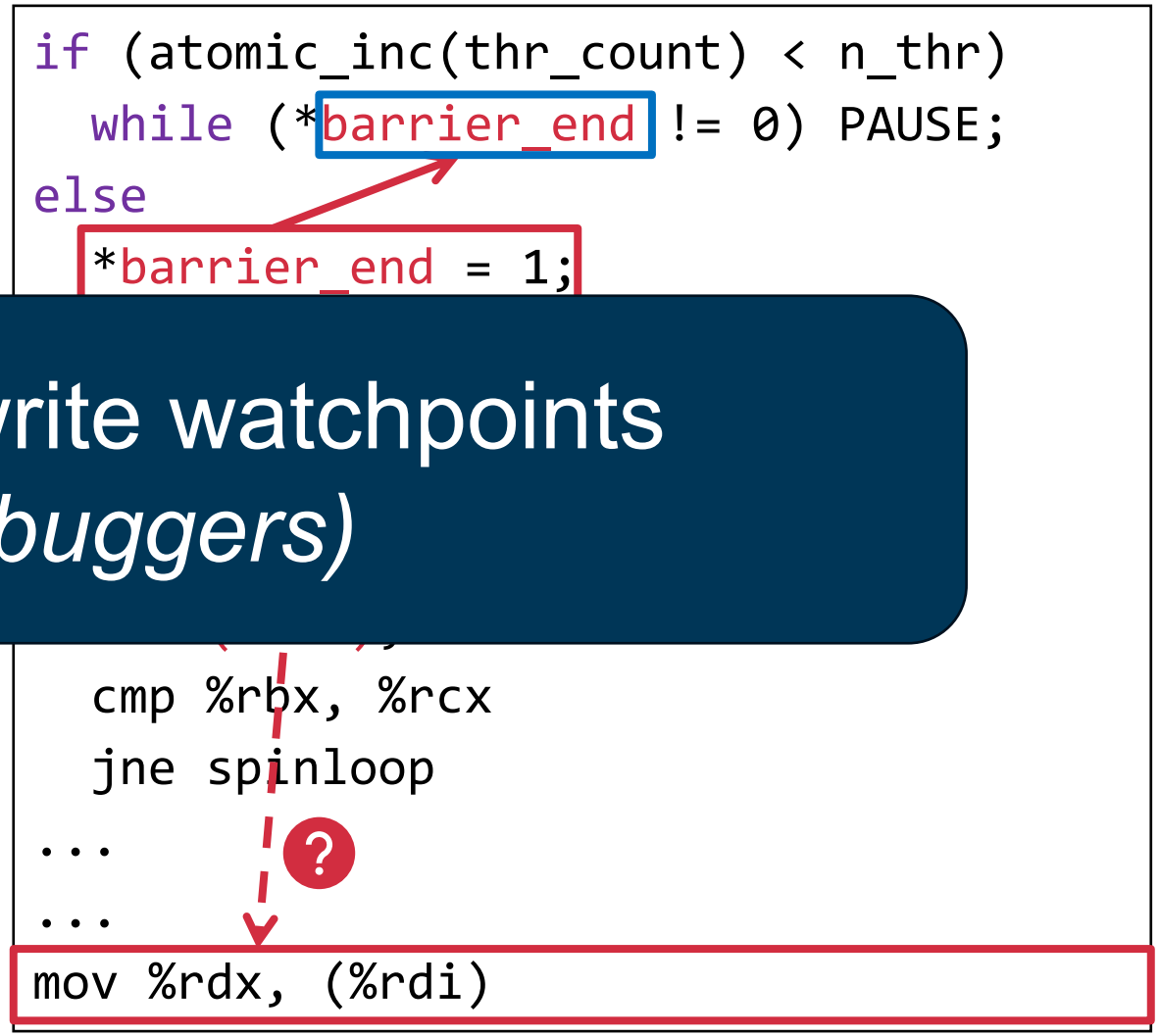
Spin-variable (dependency endpoint) is found using static analysis

How to find **dependency store**?



Capturing Busy-Waiting Synchronization

b) Capturing dependency



Solution: Use write watchpoints
(as in debuggers)

How to find **dependency store**?

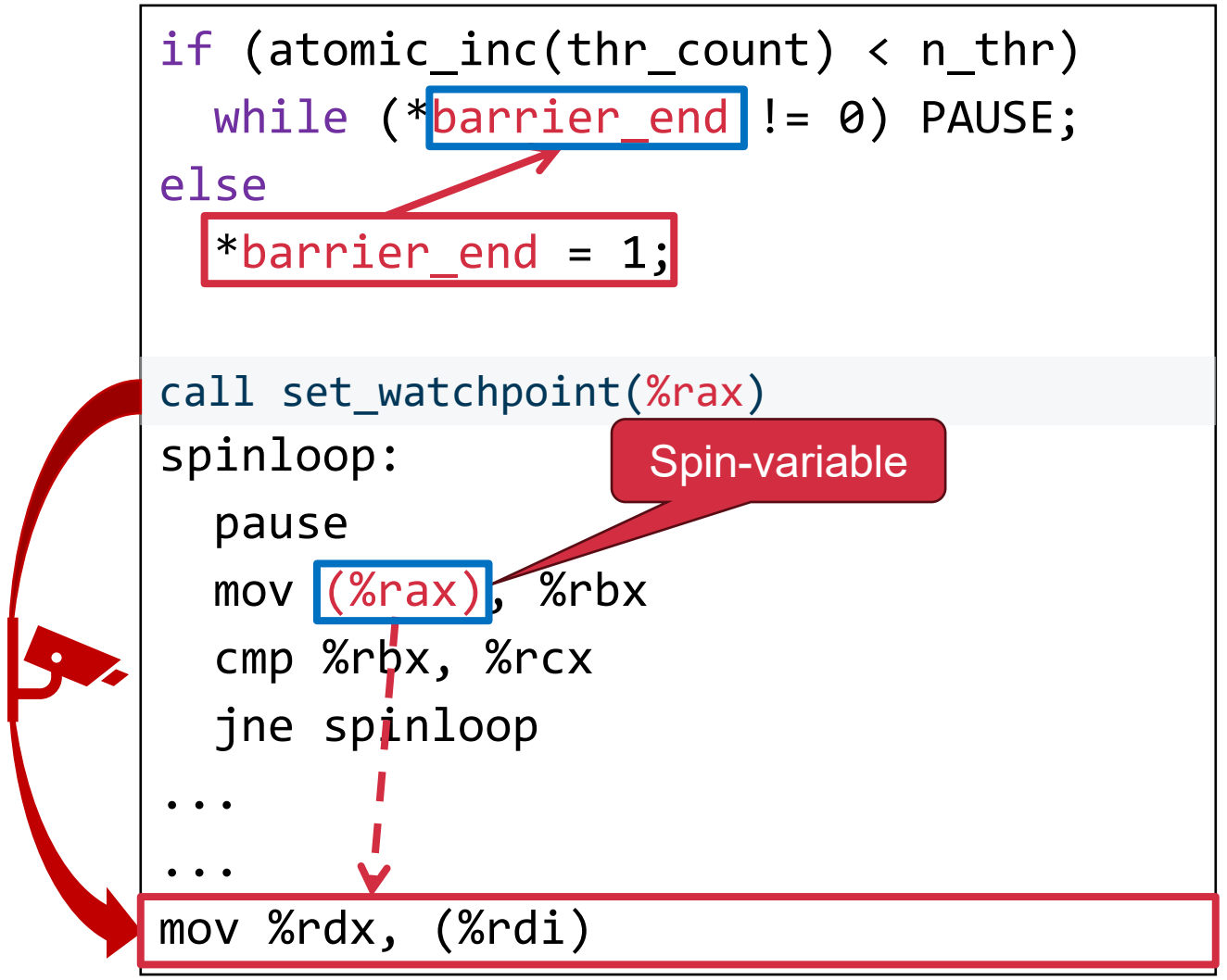
Capturing Busy-Waiting Synchronization

b) Capturing dependency

(i) **Statically inject** code that sets a **write watchpoint** on the dependency variable address

(ii) When the watchpoint is triggered:

- Store the dependency

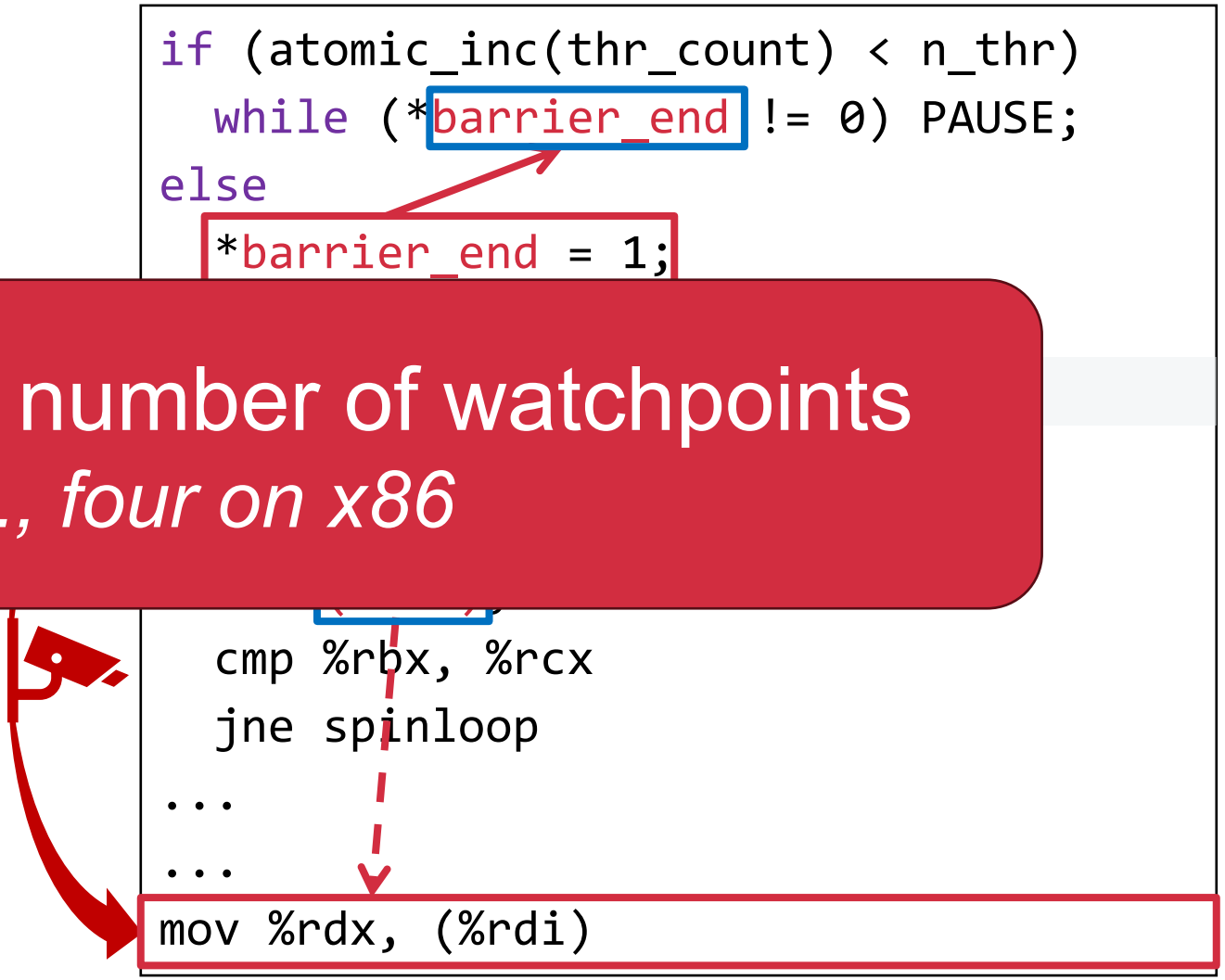


Capturing Busy-Waiting Synchronization

b) Capturing dependency

(i) **Statically inject** code that sets a **write watchp**
addres

Problem: limited number of watchpoints
E.g., four on x86

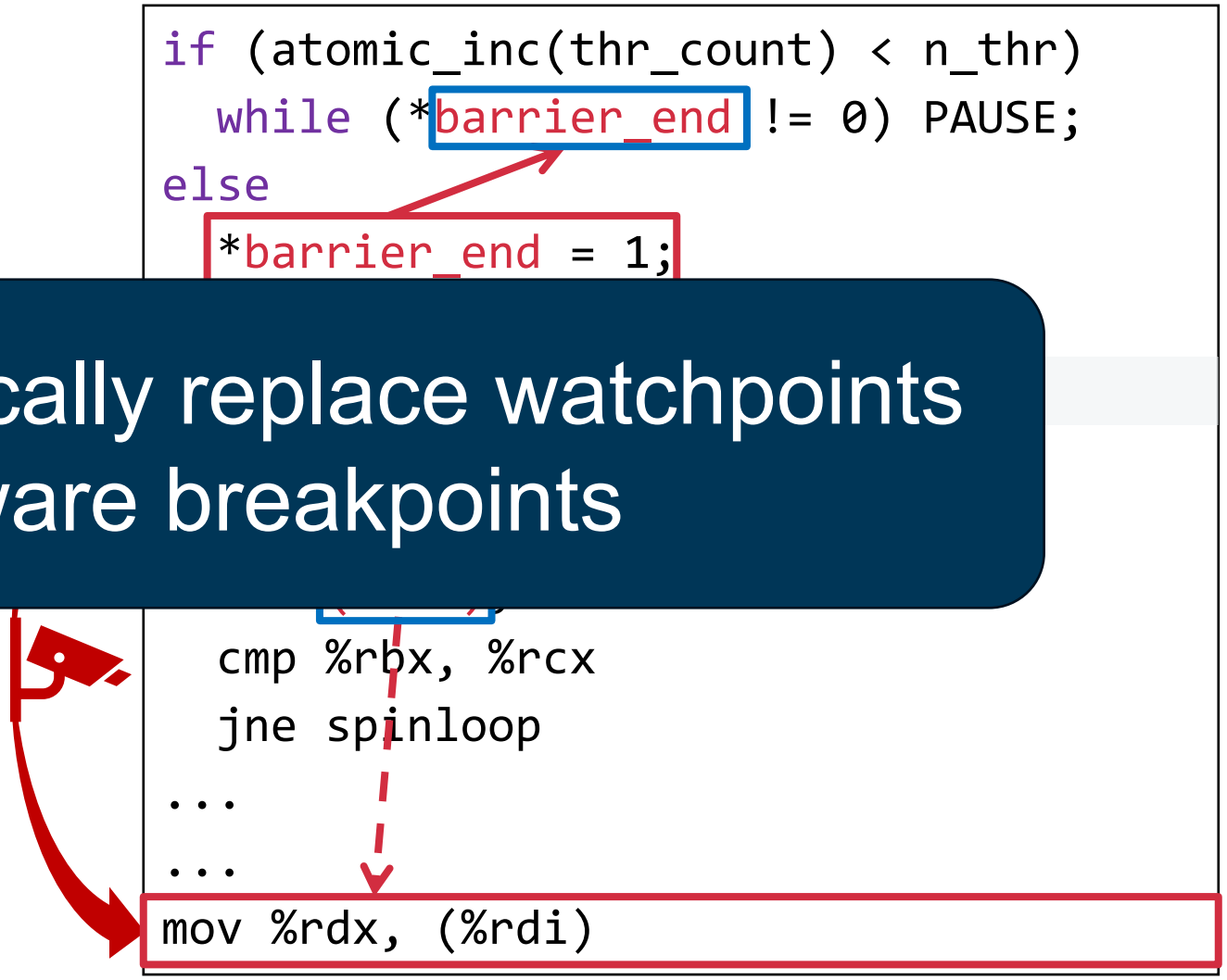


Capturing Busy-Waiting Synchronization

b) Capturing dependency

(i) **Statically inject** code that sets a **write watchp**
addres

Solution: dynamically replace watchpoints with software breakpoints

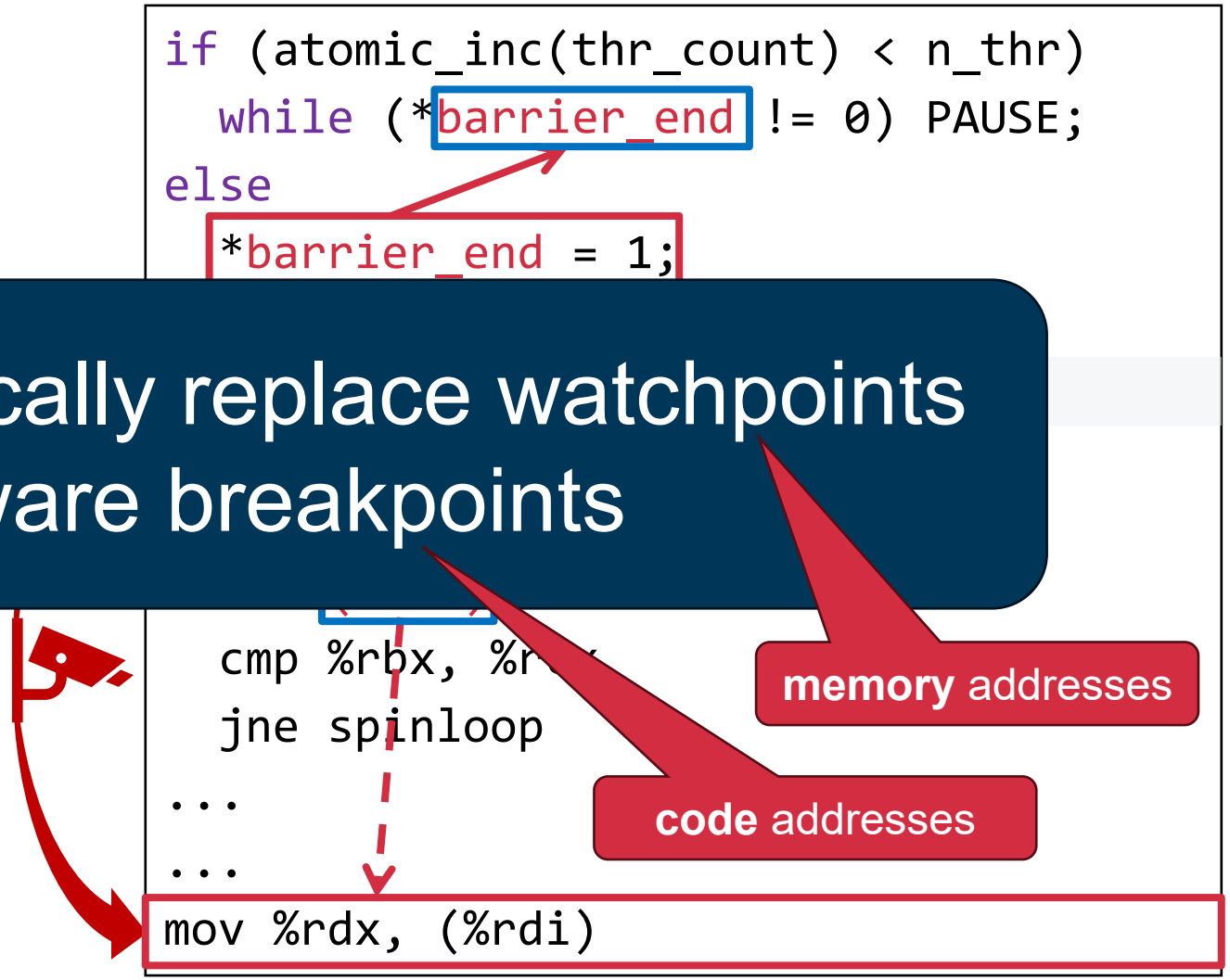


Capturing Busy-Waiting Synchronization

b) Capturing dependency

(i) **Statically inject** code that sets a **write watchp**
addres

Solution: dynamically replace watchpoints with software breakpoints



Capturing Busy-Waiting Synchronization

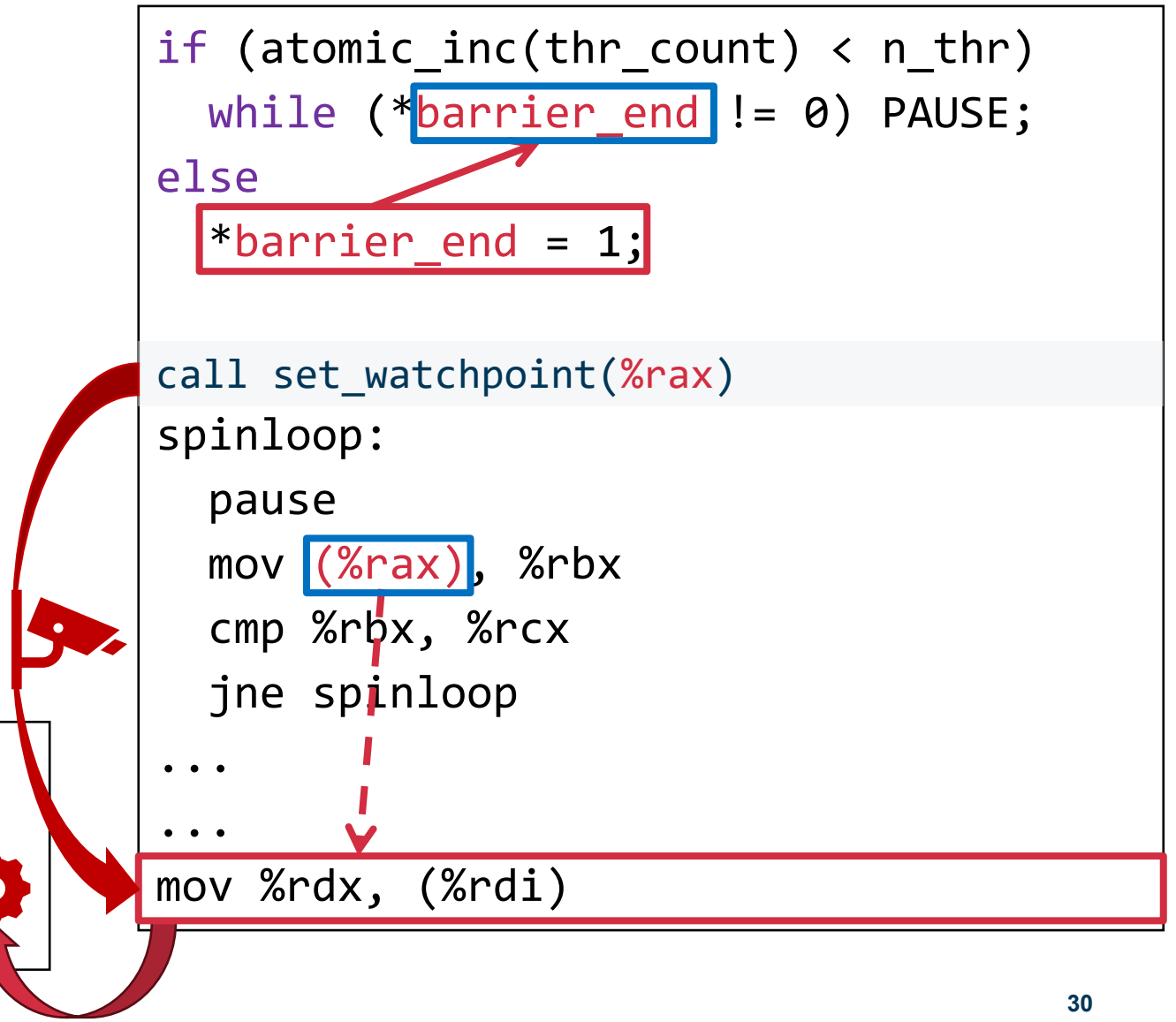
b) Capturing dependency

(i) **Statically inject** code that sets a **write watchpoint** on the dependency variable address

(ii) When the watchpoint is **triggered:**

1. Store the dependency
2. Disable the watchpoint
3. Set a breakpoint on the store code address (%rip) for future dependencies

```
on_trigger(watch_id):
  store_dep(<%rdx,==,0>);
  disable_watch(watch_id);
  set_break(%rip,&store_dep);
```



Architecture: Analyzer → Tracer → **Viewer**

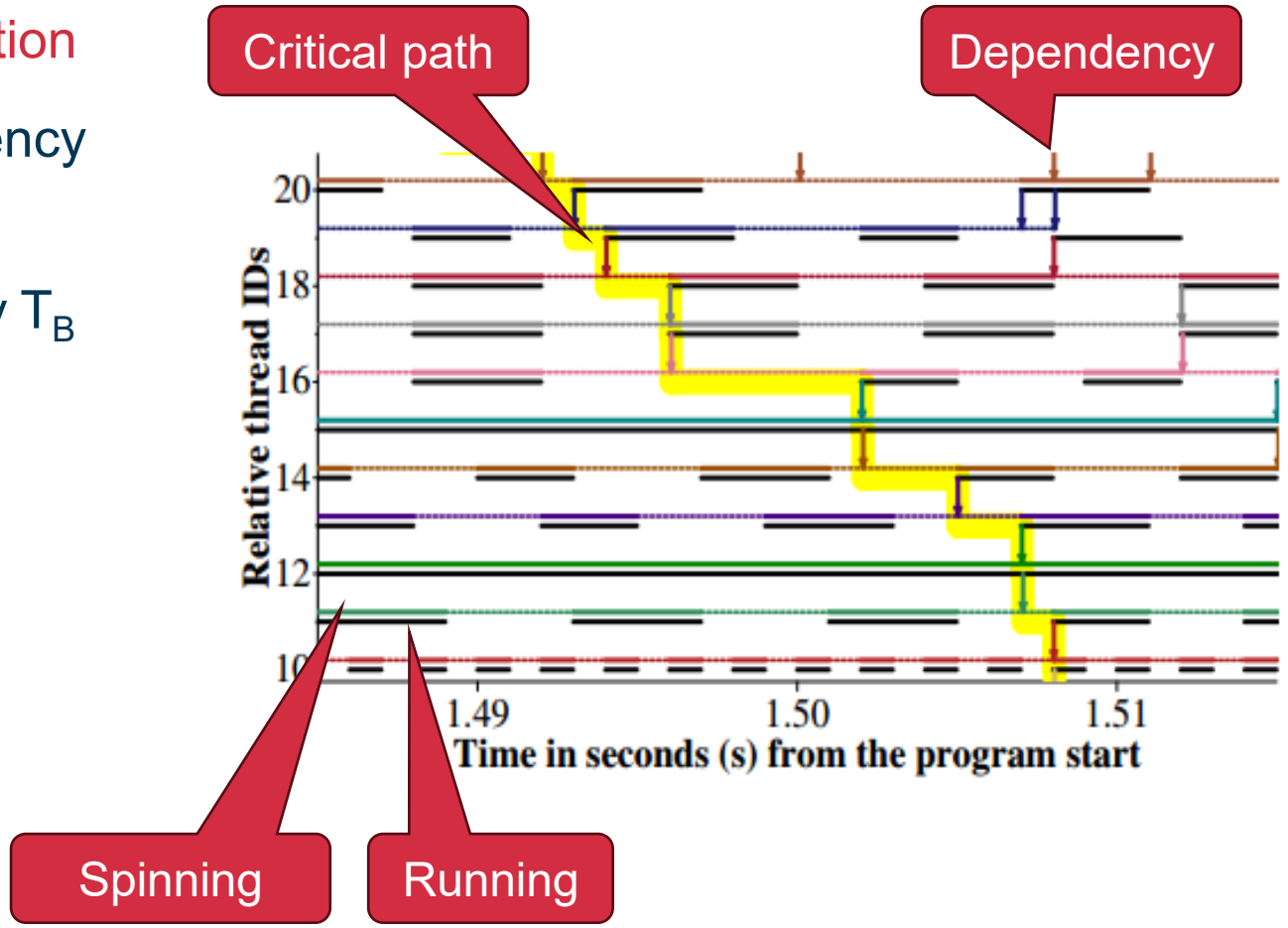
Post-processing of raw traces and visualization

- Connects events with the same dependency

T_A → T_B: A thread T_A resolves a dependency T_B was waiting on

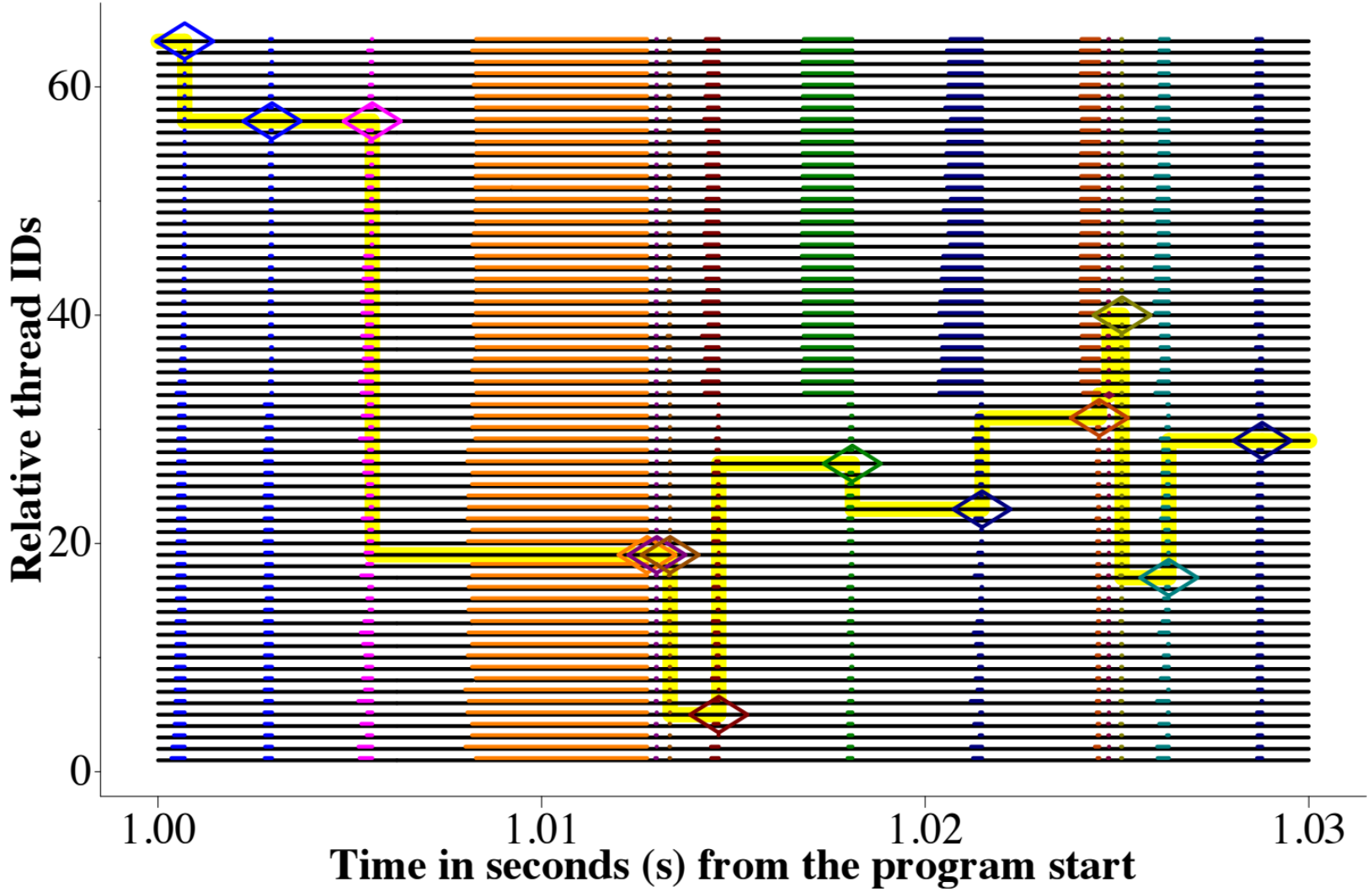
- Forms **a dependency graph**

Critical path is the longest path in the dependency graph



Observing NUMA Effect of Spinning Barrier

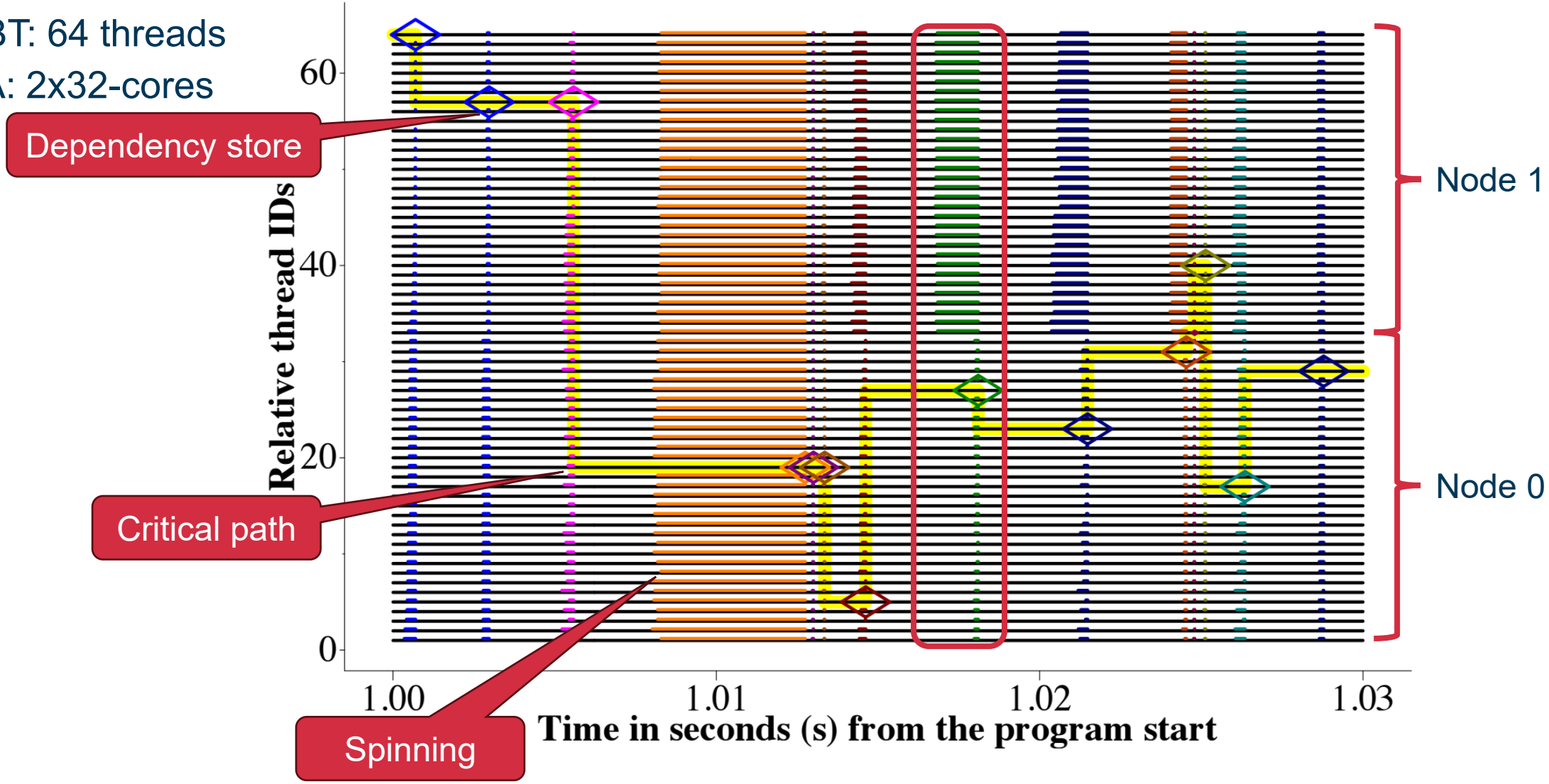
NAS/BT: 64 threads
NUMA: 2x32-cores



Observing NUMA Effect of Spinning Barrier

NAS/BT: 64 threads

NUMA: 2x32-cores



Diagnosing Pathological Busy-Waiting

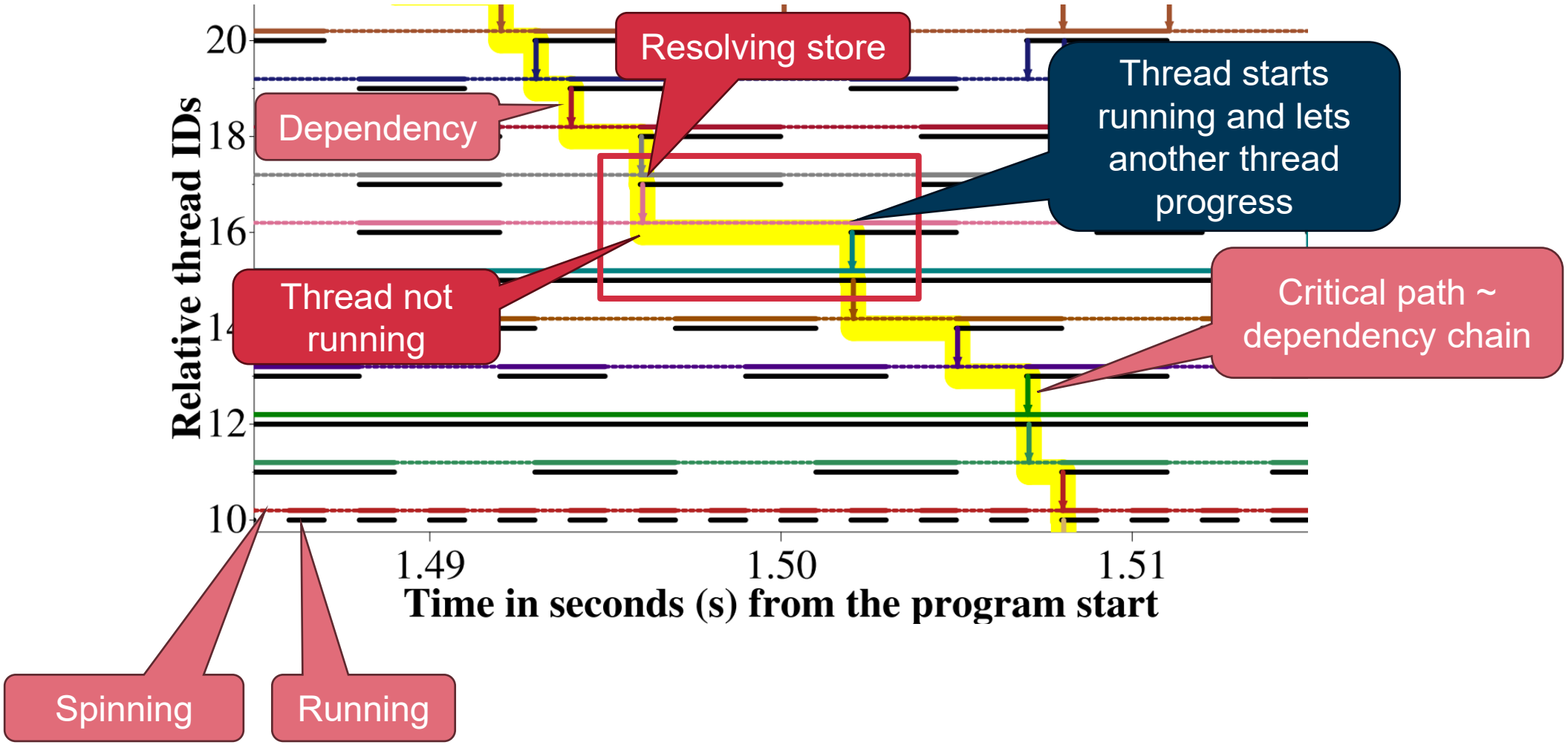
Running two NAS/LU in parallel
OpenMP set to **blocking**

Bench.	Single	Parallel
BT	2.67	5.72
CG	0.09	0.20
EP	0.25	0.52
FT	0.18	0.35
IS	0.02	0.05
LU	2.22	100.09
MG	0.14	0.26
SP	1.58	3.86
UA	3.58	7.23

(a) Single vs. Parallel.

Why is the parallel execution of LU so **slow**?

Diagnosing Pathological Busy-Waiting



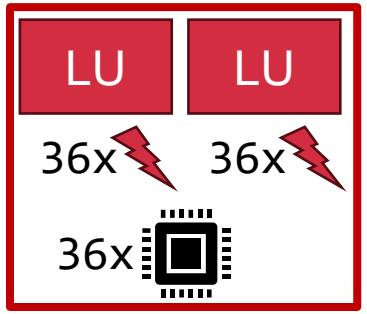
Faster Busy-Waiting in VMs vs. Bare Metal

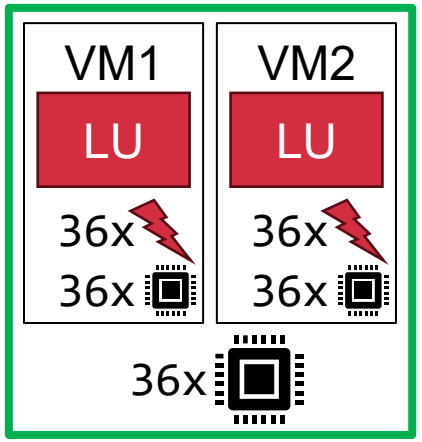
Running two NAS/LU in parallel:

- BM: instances run on bare metal
- VM: each instance runs in a separate VM

OpenMP set to **default**

- **Busy-waiting** for a fixed number of iterations, **then blocking**





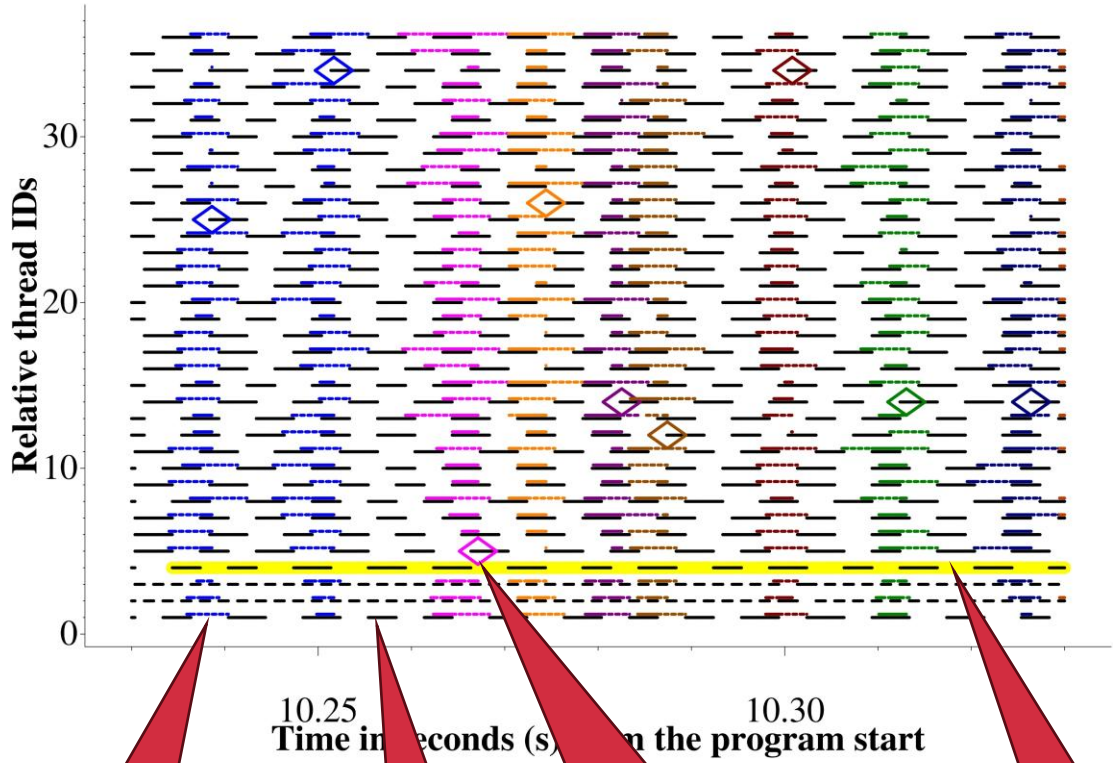
Bench.	BM	VM
BT	13.22	11.94
CG	8.44	5.02
EP	0.53	0.45
FT	0.57	0.54
IS	0.23	0.16
LU	190.75	138.44
MG	2.51	1.6
SP	31.9	22.57
UA	300.2	185.7

(b) Bare Metal (BM) vs. VM.

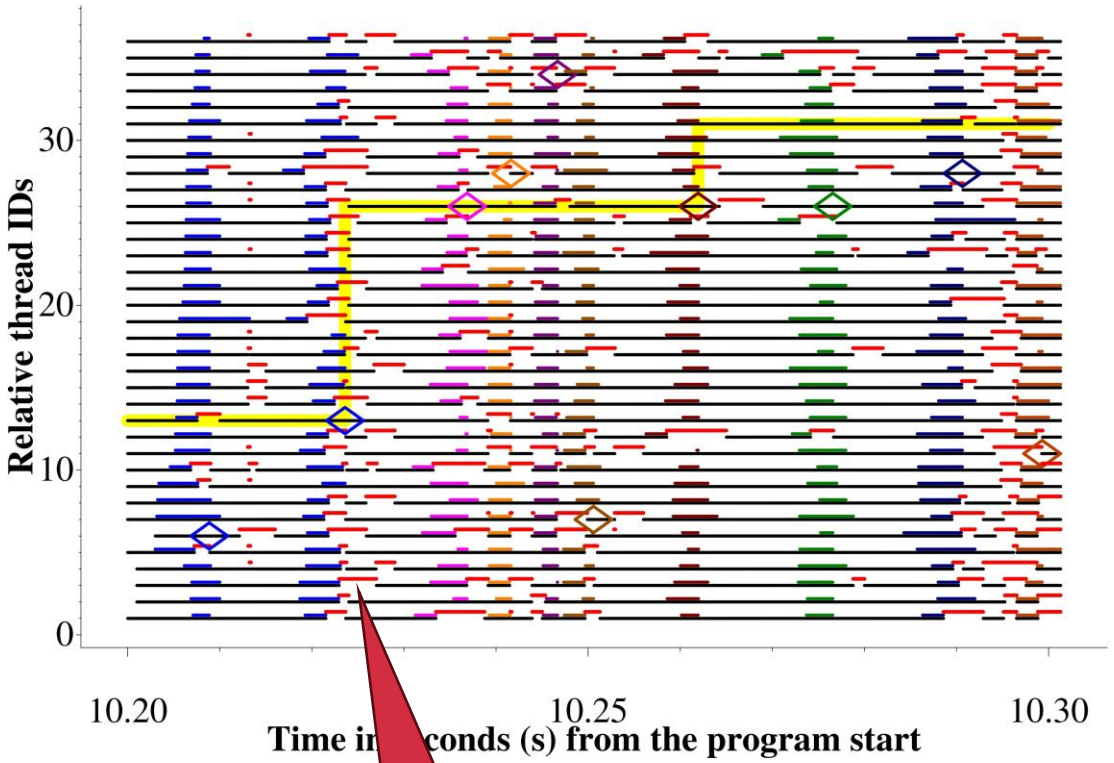
Why is **running in VM faster** than on bare metal?

Faster Busy-Waiting in VMs vs. Bare Metal

Bare metal



VM



Spinning

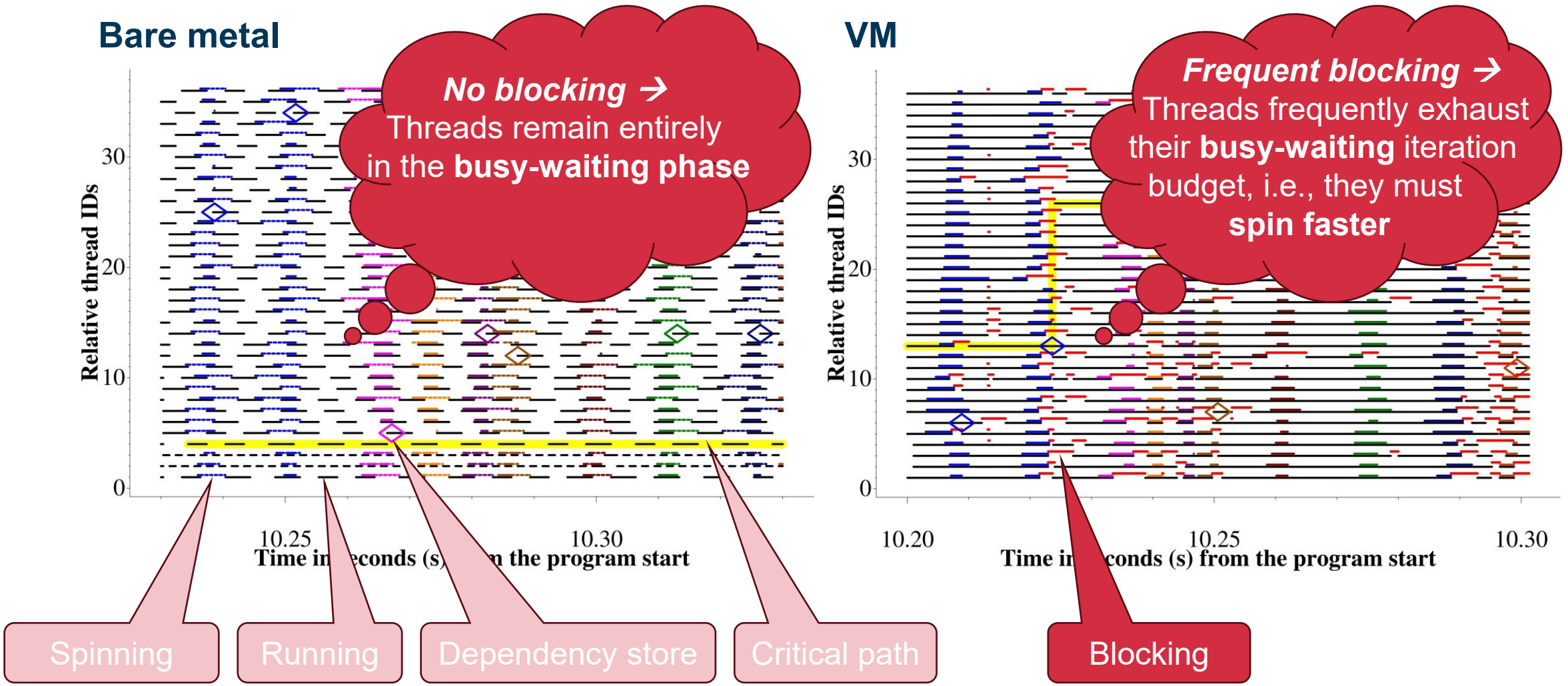
Running

Dependency store

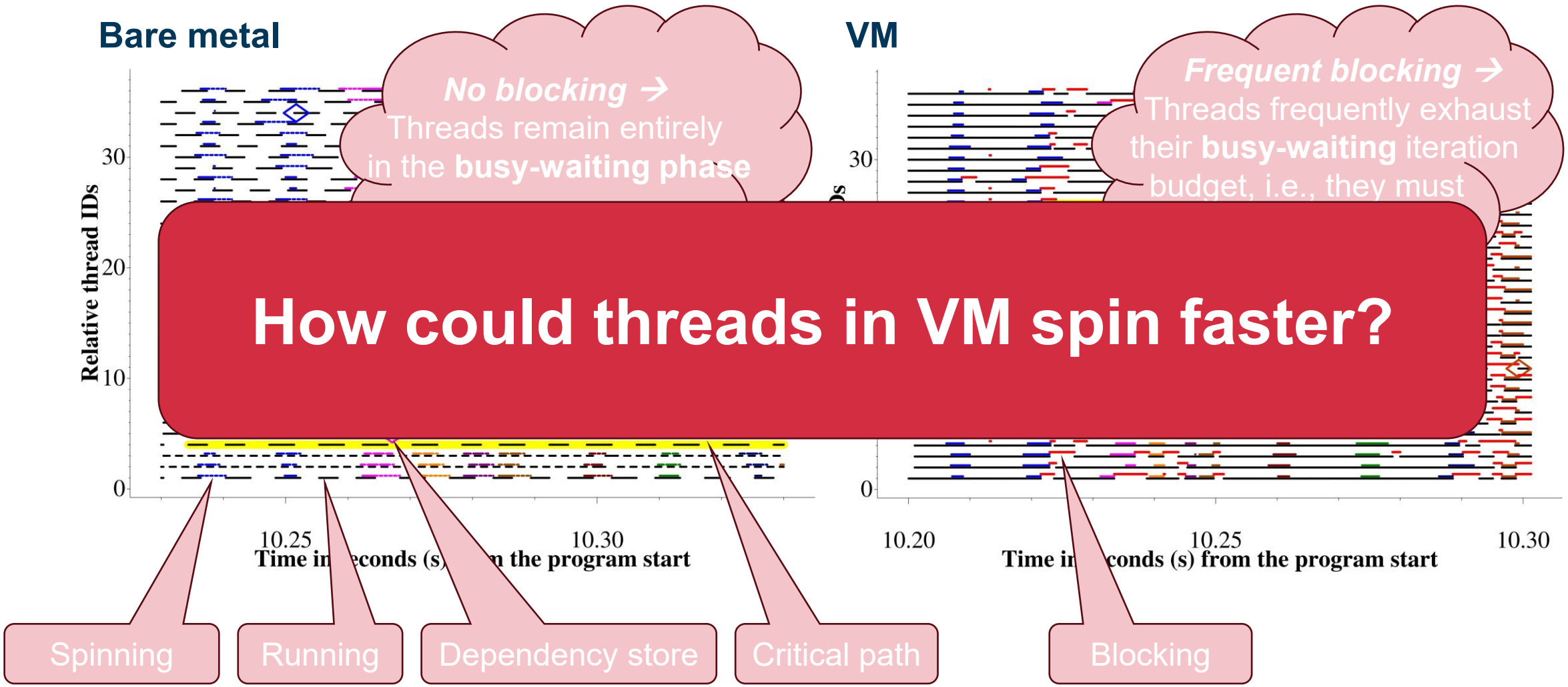
Critical path

Blocking

Faster Busy-Waiting in VMs vs. Bare Metal



Faster Busy-Waiting in VMs vs. Bare Metal



Pause-Loop Exiting (PLE)

A hardware feature of Intel VMX

PLE triggers **VM exits for spinloops** to help the hypervisor **mitigate** guest-kernel **lock holder/waiter preemption issue**

*"PLE VM-execution control is **ignored in userspace**"* [Intel manual, §36.7.3]

```
// userland test:
for (i=0; i < 10^9; i++) PAUSE;
```

Config.		Rt (s)	f (GHz)	10^9 Instr.		10^9 PAUSEs	
				H	*G*	*H*	*G*
H		14.12	3.89	5.0	0	1.0	0
G	*PLE*	6.19	3.89	0	5.0	0	0
G	*no PLE*	14.17	3.89	0	5.0	0	1.0

Table 2. Impact of PLE on a userland busy-wait loop with 10^9 iterations (Rt=runtime, f=CPU frequency, H=host, G=guest).

Pause-Loop Exiting (PLE)

A hardware feature of Intel VMX

PLE triggers **VM exits for spinloops** to help the hypervisor **mitigate** guest-kernel **lock holder/waiter preemption issue**

*"PLE VM-execution control is **ignored in userspace**"* [Intel manual, §36.7.3]

And yet, with PLE enabled, user PAUSE instructions are not even retired on Skylake and Cascade Lake machines

```
// userland test:
for (i=0; i < 10^9; i++) PAUSE;
```

Config.		Rt (s)	f (GHz)	10^9 Instr.		10^9 PAUSEs	
				H	*G*	*H*	*G*
H		14.12	3.89	5.0	0	1.0	0
G	*PLE*	6.19	3.89	0	5.0	0	0
G	*no PLE*	14.17	3.89	0	5.0	0	1.0

Table 2. Impact of PLE on a userland busy-wait loop with 10^9 iterations (Rt=runtime, f=CPU frequency, H=host, G=guest).

Conclusion

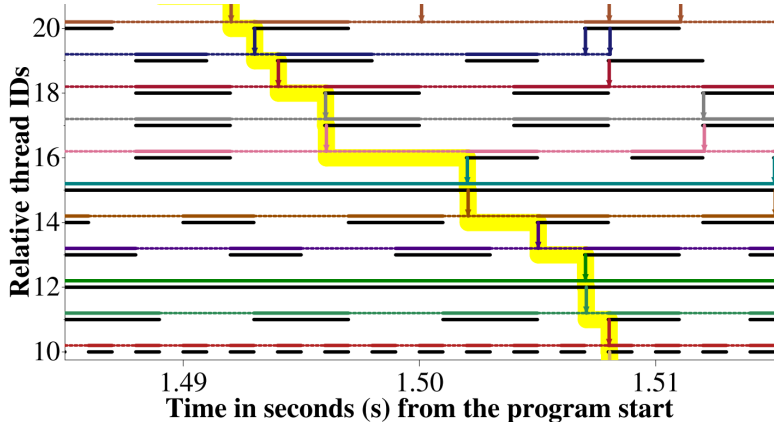
A **dependency model** that unifies the concept of inter-thread wait-for dependencies for blocking and busy-waiting synchronization

Conclusion

A **dependency model** that unifies the concept of inter-thread wait-for dependencies for blocking and busy-waiting synchronization

Tapestry: A **tracing framework** to observe wait-for dependencies

- **Combining** the use of hardware **watchpoints** and software **breakpoints**



Conclusion

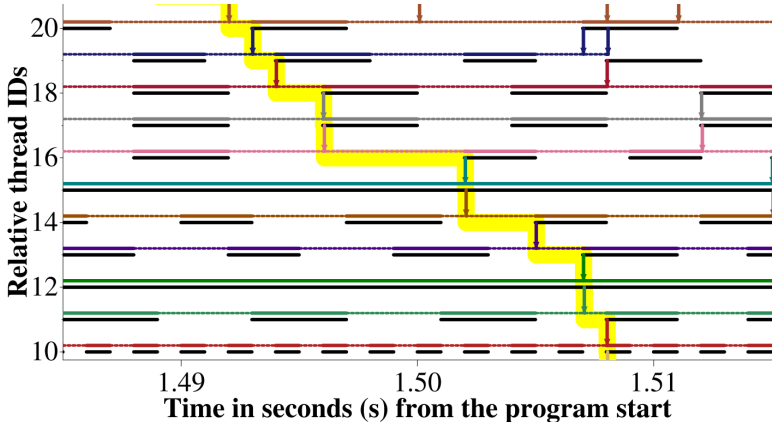
A **dependency model** that unifies the concept of inter-thread wait-for dependencies for blocking and busy-waiting synchronization

Tapestry: A **tracing framework** to observe wait-for dependencies

- **Combining** the use of hardware **watchpoints** and software **breakpoints**

Investigated **three anomalies** using Tapestry

- **NUMA** effect on spin-barriers
- **Pathological busy-waiting**
- Faster busy-waiting in VM vs. bare metal
  - Found **undocumented hardware effect** on some Intel machines



Conclusion

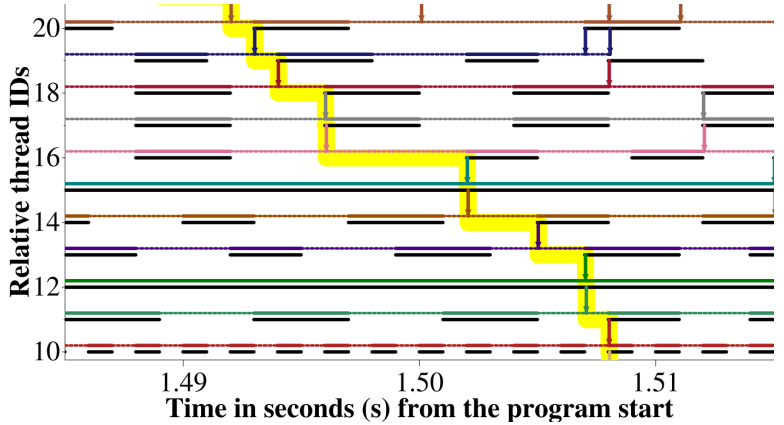
A **dependency model** that unifies the concept of inter-thread wait-for dependencies for blocking and busy-waiting synchronization

Tapestry: A **tracing framework** to observe wait-for dependencies

- **Combining** the use of hardware **watchpoints** and software **breakpoints**

Investigated **three anomalies** using Tapestry

- **NUMA** effect on spin-barriers
- **Pathological busy-waiting**
- Faster busy-waiting in VM vs. bare metal
  - Found **undocumented hardware effect** on some Intel machines



Thank you!