Getting Started with LITMUS^RT

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

— July 2016 —

Max
Planck
Institute
for
Software Systems

Björn B. Brandenburg
Manohar Vanga
Mahircan Gül

Agenda

1 **What? Why? How?**
The first decade of LITMUS^RT

2 **Major Features**
What sets LITMUS^RT apart?

3 **Key Concepts**
What you need to know to get started

Max Planck Institute for Software Systems

Getting Stated with LITMUS^RT

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

What? Why? How?

The first decade of LITMUS^RT

— Part 1 —

What is LITMUS^{RT}?

A real-time extension of the Linux kernel.

What is LITMUS^{RT}?

Linux kernel patch

+

user-space interface

+

tracing infrastructure

What is LITMUS^{RT}?

Linux kernel patch
- RT schedulers
- RT synchronization
- [cache & GPU]

+

user-space interface
- C API
- device files
- scripts & tools

+

tracing infrastructure
- overheads
- schedules
- kernel debug log

What is LITMUS^{RT}?

Releases
2007.1
2007.2
2007.3
2008.1
2008.2
2008.3
2010.1
2010.2
2011.1
2012.1
2012.2
2012.3
2013.1
2014.1
2014.2
2015.1
2016.1

Linux kernel patch
- RT schedulers
- RT synchronization
- [cache & GPU]

+

user-space interface
- C API
- device files
- scripts & tools

+

tracing infrastructure
- overheads
- schedules
- kernel debug log

Mission

Enable *practical* multiprocessor real-time *systems* research under *realistic conditions*.

Mission

Enable *practical* multiprocessor real-time *systems* research under *realistic conditions*.

practical and *realistic*:

Efficiently…

➡ *enable apples-to-apples comparison with existing systems (esp. Linux)*

…support real applications…

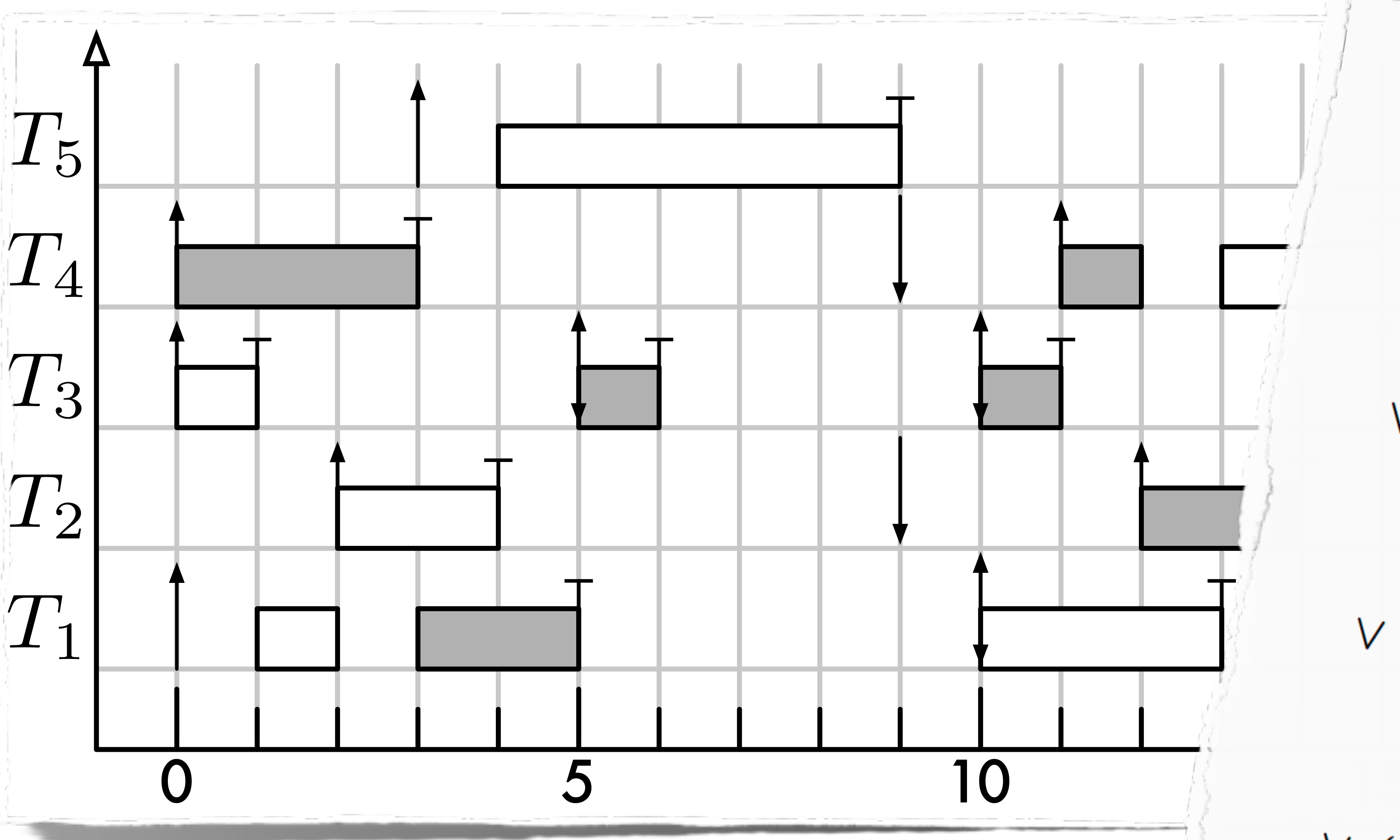
➡ *I/O, synchronization, legacy code*

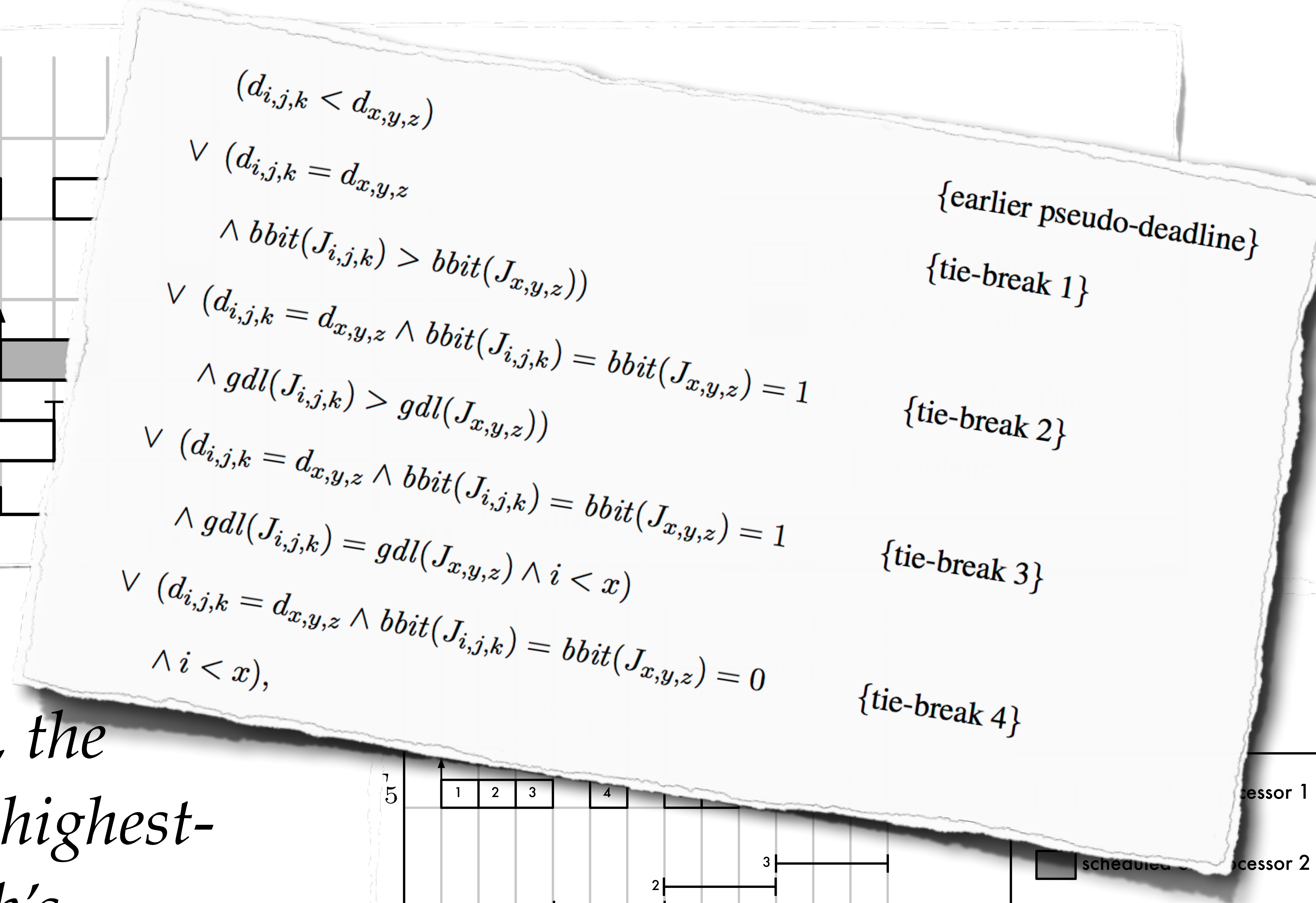
…on real multicore hardware…

➡ *Realistic overheads on commodity platforms.*

…in a real OS.

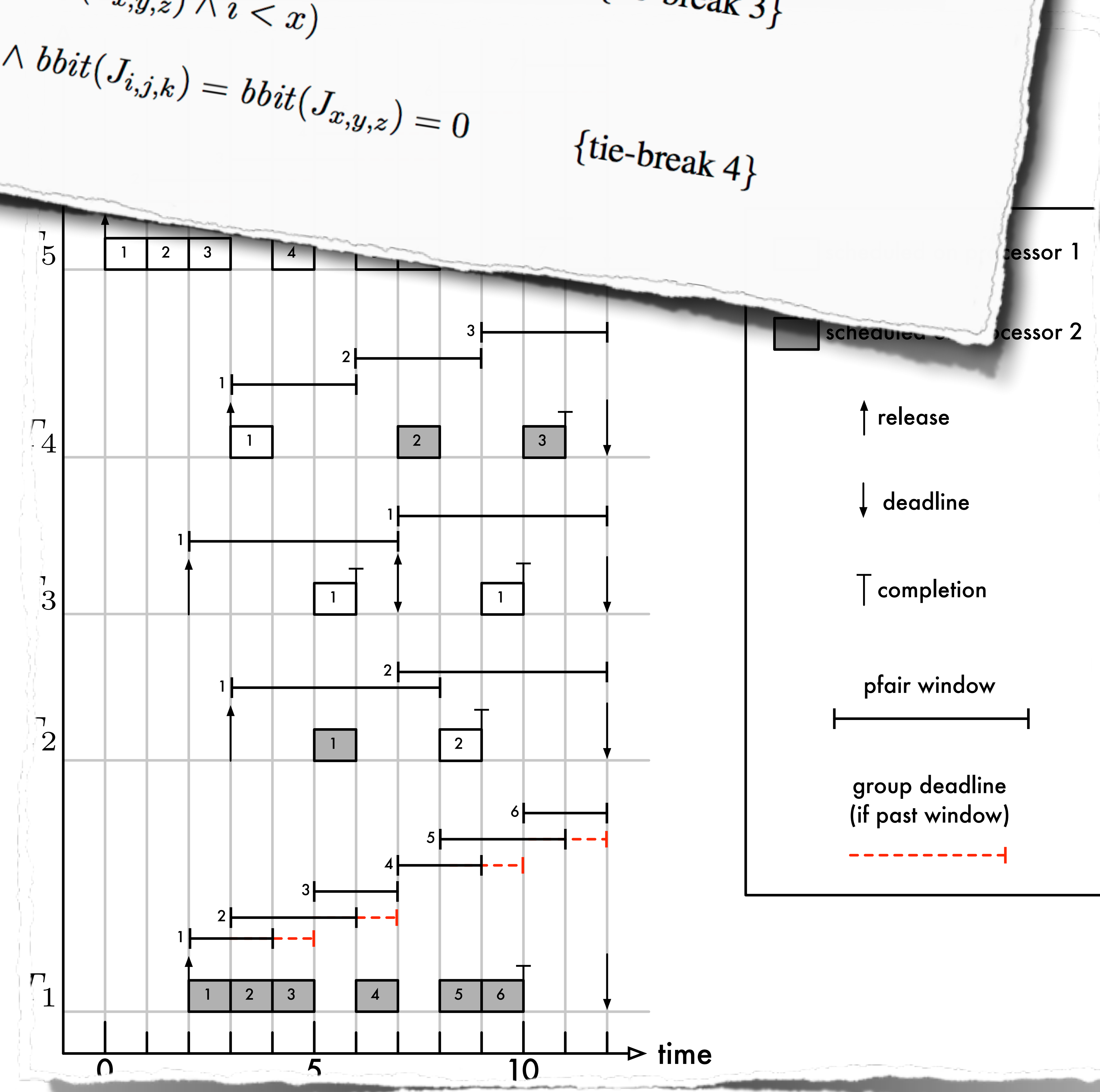
➡ *Realistic implementation constraints and challenges.*





$$
\begin{aligned}
&(d_{i,j,k} < d_{x,y,z}) && \{\text{earlier pseudo-deadline}\}\\
&\lor\ (d_{i,j,k} = d_{x,y,z} \\
&\quad \land\ bbit(J_{i,j,k}) > bbit(J_{x,y,z})) && \{\text{tie-break 1}\}\\
&\lor\ (d_{i,j,k} = d_{x,y,z} \land bbit(J_{i,j,k}) = bbit(J_{x,y,z}) = 1 \\
&\quad \land\ gdl(J_{i,j,k}) > gdl(J_{x,y,z})) && \{\text{tie-break 2}\}\\
&\lor\ (d_{i,j,k} = d_{x,y,z} \land bbit(J_{i,j,k}) = bbit(J_{x,y,z}) = 1 \\
&\quad \land\ gdl(J_{i,j,k}) = gdl(J_{x,y,z}) \land i < x) && \{\text{tie-break 3}\}\\
&\lor\ (d_{i,j,k} = d_{x,y,z} \land bbit(J_{i,j,k}) = bbit(J_{x,y,z}) = 0 \\
&\quad \land\ i < x), && \{\text{tie-break 4}\}
\end{aligned}
$$

*"At **any point in time**, the system schedules the **m** highest-priority jobs, where a job's current priority is given by…"*



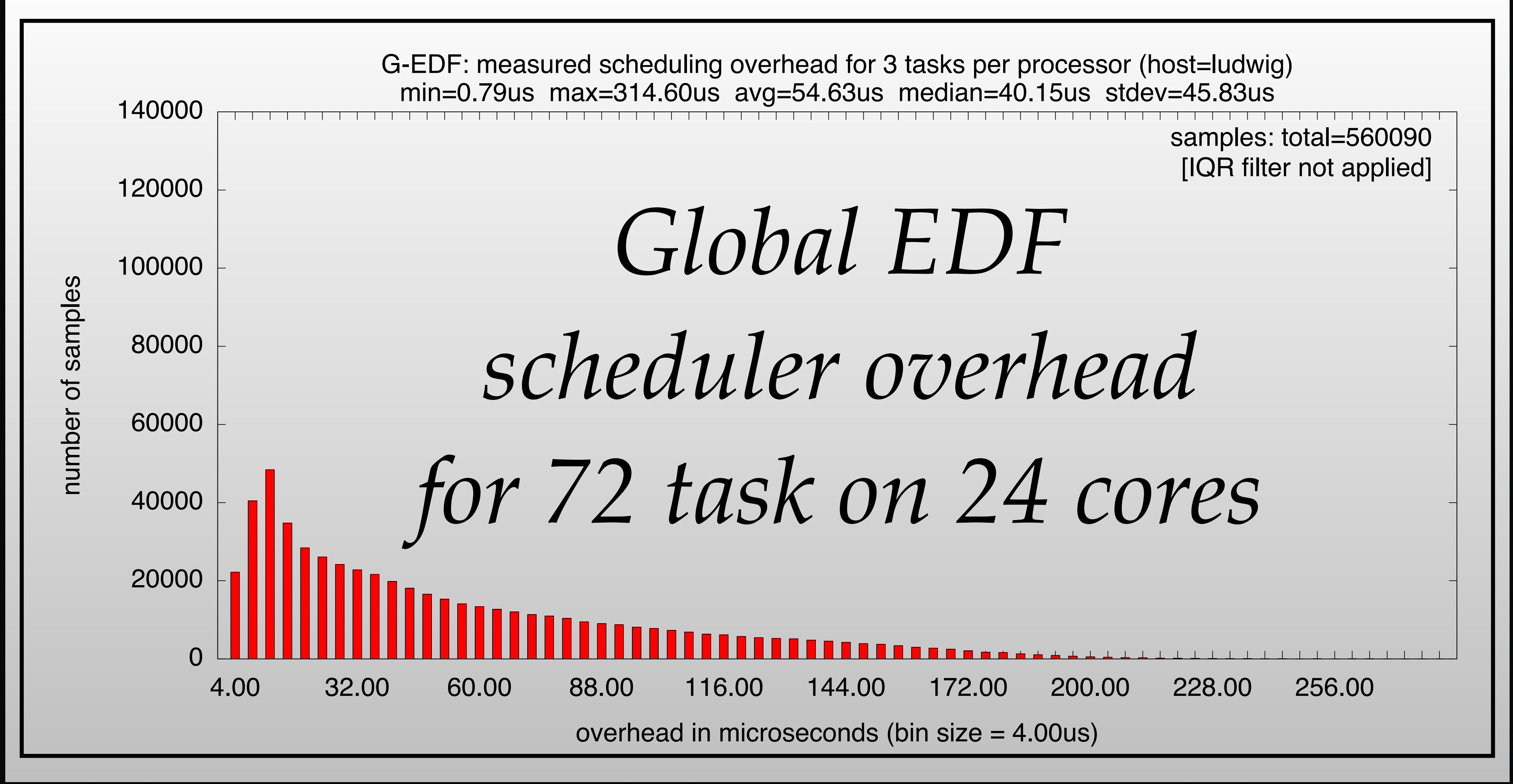
Going from this…

```c
 * assumptions on the state of the current task since it may be called for a
 * number of reasons. The reasons include a scheduler_tick() determined that it
 * was necessary, because sys_exit_np() was called, because some Linux
 * subsystem determined so, or even (in the worst case) because there is a bug
 * hidden somewhere. Thus, we must take extreme care to determine what the
 * current state is.
 *
 * The CPU could currently be scheduling a task (or not), be linked (or not).
 *
 * The following assertions for the scheduled task could hold:
 *
 *      - !is_running(scheduled)        // the job blocks
 *      - scheduled->timeslice == 0     // the job completed (forcefully)
 *      - get_rt_flag() == RT_F_SLEEP   // the job completed (by syscall)
 *      - linked != scheduled           // we need to reschedule (for any reason)
 *      - is_np(scheduled)              // rescheduling must be delayed,
 *                                         sys_exit_np must be requested
 *
 * Any of these can occur together.
 */
static struct task_struct* gsnedf_schedule(struct task_struct * prev)
{
	cpu_entry_t* entry = &__get_cpu_var(gsnedf_cpu_entries);
	int out_of_time, sleep, preempt, np, exists, blocks;
	struct task_struct* next = NULL;

#ifdef CONFIG_RELEASE_MASTER
	/* Bail out early if we are the release master.
	 * The release master never schedules any real-time tasks.
	 */
	if (unlikely(gsnedf.release_master == entry->cpu)) {
		sched_state_task_picked();
		return NULL;
	}
#endif

	raw_spin_lock(&gsnedf_lock);

	/* sanity checking */
	BUG_ON(entry->scheduled && entry->scheduled != prev);
	BUG_ON(entry->scheduled && !is_realtime(prev));
	BUG_ON(is_realtime(prev) && !entry->scheduled);

	/* (0) Determine state */
	exists      = entry->scheduled != NULL;
	blocks      = exists && !is_running(entry->scheduled);
	out_of_time = exists && budget_enforced(entry->scheduled)
		&& budget_exhausted(entry->scheduled);
	np          = exists && is_np(entry->scheduled);
	sleep       = exists && get_rt_flags(entry->scheduled) == RT_F_SLEEP;
	preempt     = entry->scheduled != entry->linked;

#ifdef WANT_ALL_SCHED_EVENTS
	TRACE_TASK(prev, "invoked gsnedf_schedule.\n");
#endif

	if (exists)
		TRACE_TASK(prev,
			   "blocks:%d out_of_time:%d np:%d sleep:%d preempt:%d "
			   "state:%d sig:%d\n",
			   blocks, out_of_time, np, sleep, preempt,
			   prev->state, signal_pending(prev));
	if (entry->linked && preempt)
		TRACE_TASK(prev, "will be preempted by %s/%d\n",
			   entry->linked->comm, entry->linked->pid);
-11-:----F1  sched_gsn_edf.c    42% (419,0)    Git-wip-job-counts   (C/l Abbrev)--4:43PM----
```

… to this!

LITMUS RT

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems



Why *You* Should Be Using LITMUS^{RT}

If you are doing kernel-level work anyway…

- ➡ Get a ***head-start*** — simplified kernel interfaces, debugging infrastructure, user-space interface, tracing infrastructure
- ➡ As a ***baseline*** — compare with schedulers in LITMUS^{RT}

If you are developing real-time applications…

- ➡ Get a predictable execution environment with "***textbook algorithms***" matching the literature
- ➡ Isolate processes with ***reservation-based scheduling***!
- ➡ Understand ***kernel overheads*** with just a few commands!

If your primary focus is theory and analysis…

- ➡ To understand the impact of ***overheads***.
- ➡ To ***demonstrate practicality*** of proposed approaches.

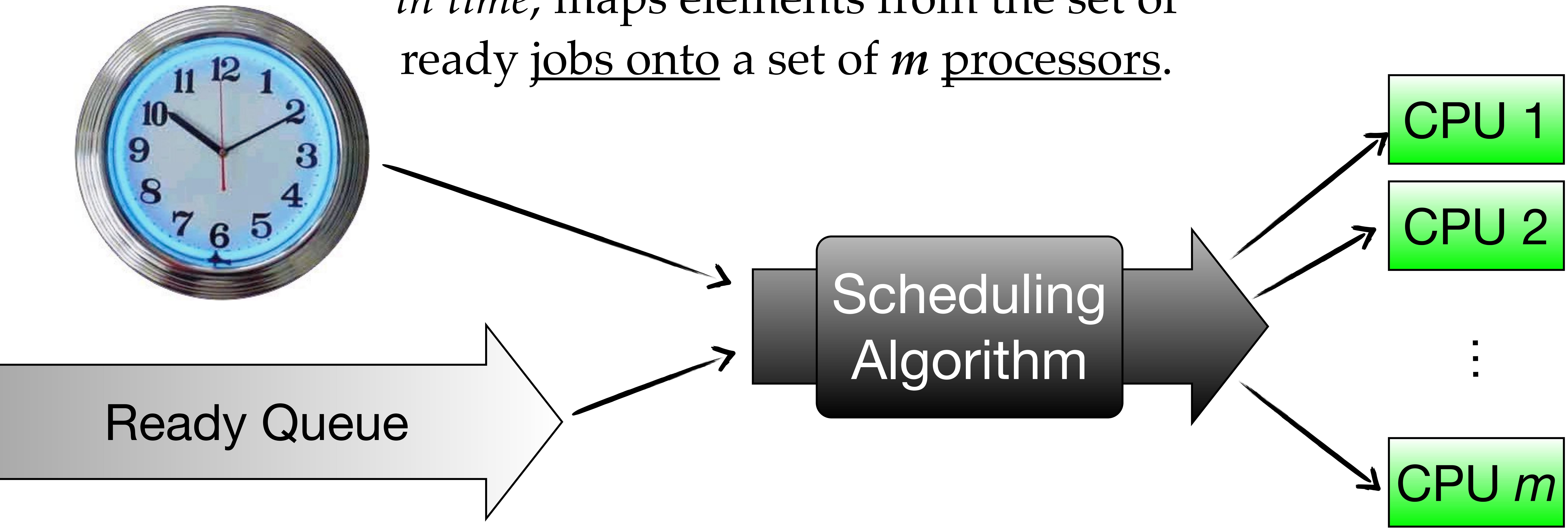
Theory vs. Practice

Why is implementing "textbook" schedulers difficult?

Besides the usual kernel fun:
restricted environment, special APIs, difficult to debug, …

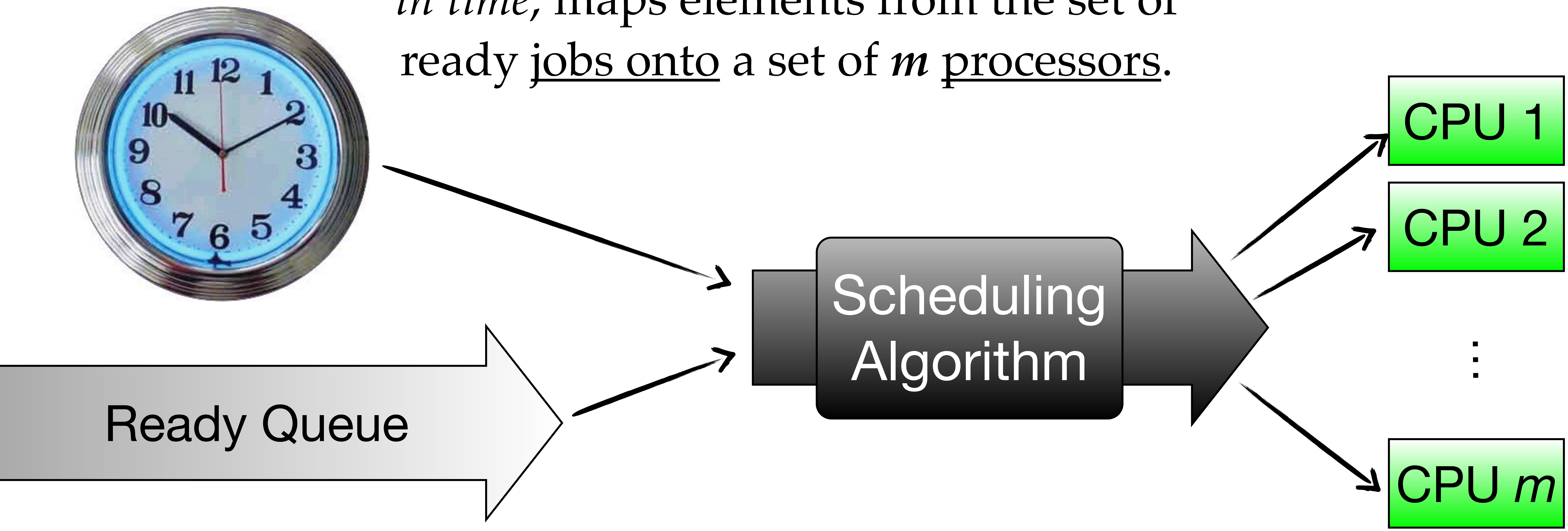
Scheduling in Theory

Scheduler: a <u>function</u> that, *at each point in time*, maps elements from the set of ready <u>jobs onto</u> a set of ***m*** <u>processors</u>.



Scheduling in Theory

Scheduler: a <u>function</u> that, *at each point in time*, maps elements from the set of ready <u>jobs onto</u> a set of ***m*** <u>processors</u>.



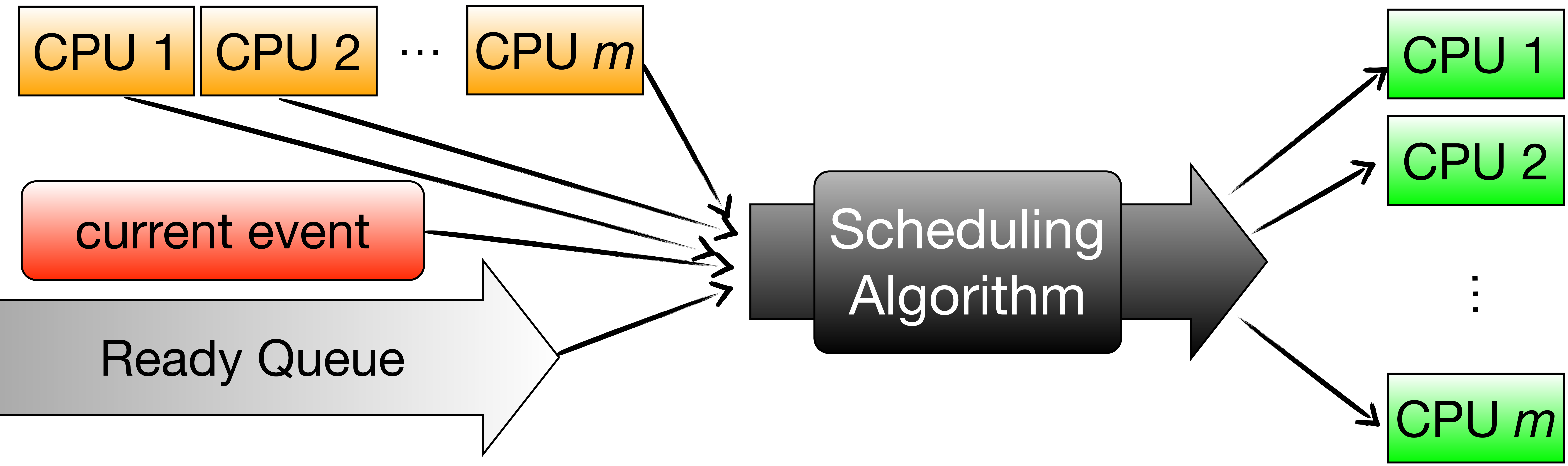
Global policies based on global state

➡*E.g., "At any point in time, the m highest-priority…"*

Sequential policies, assuming **total order** of events.

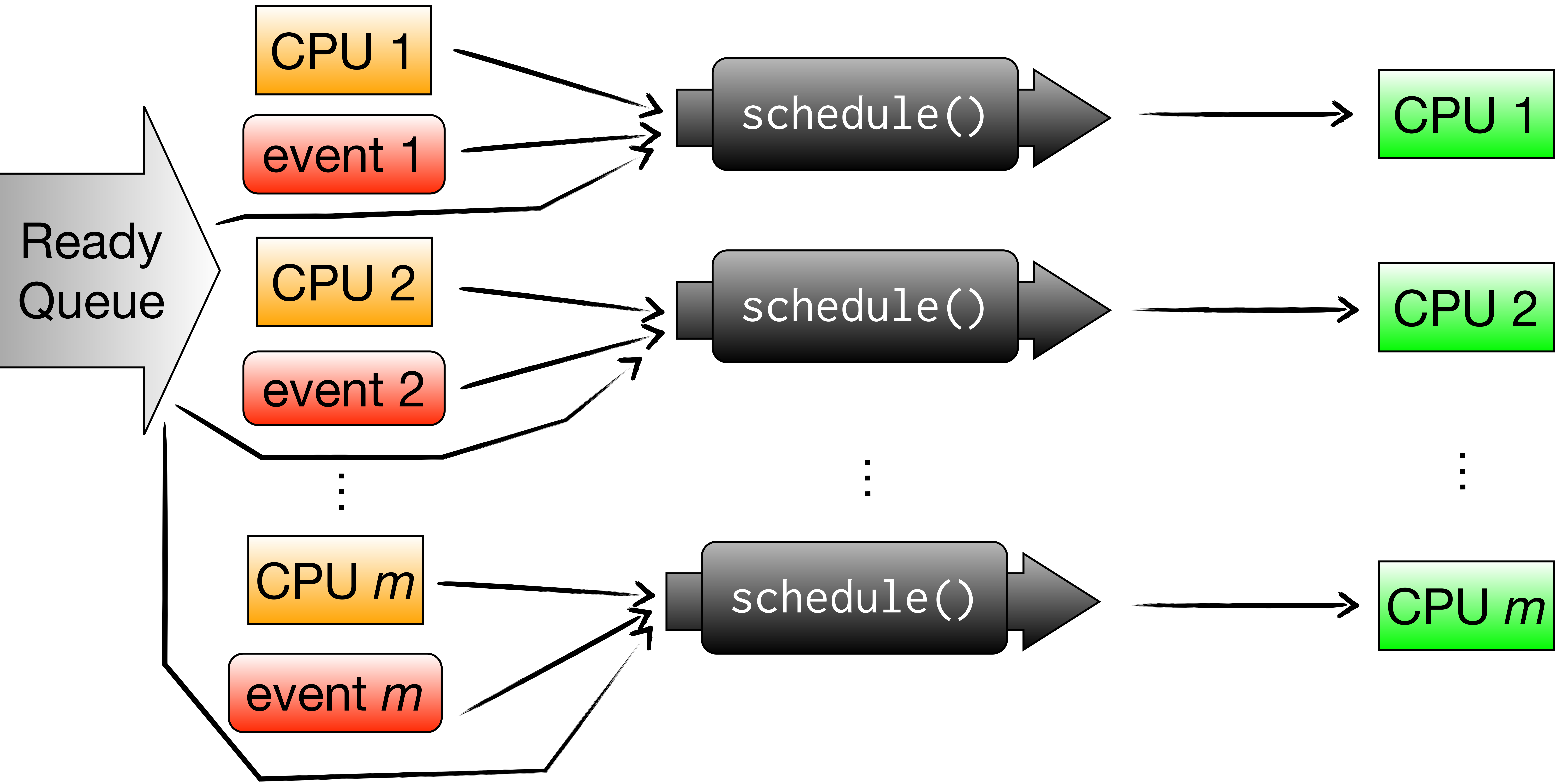
➡*E.g., "If a job arrives at time t…"*

Scheduling in Theory

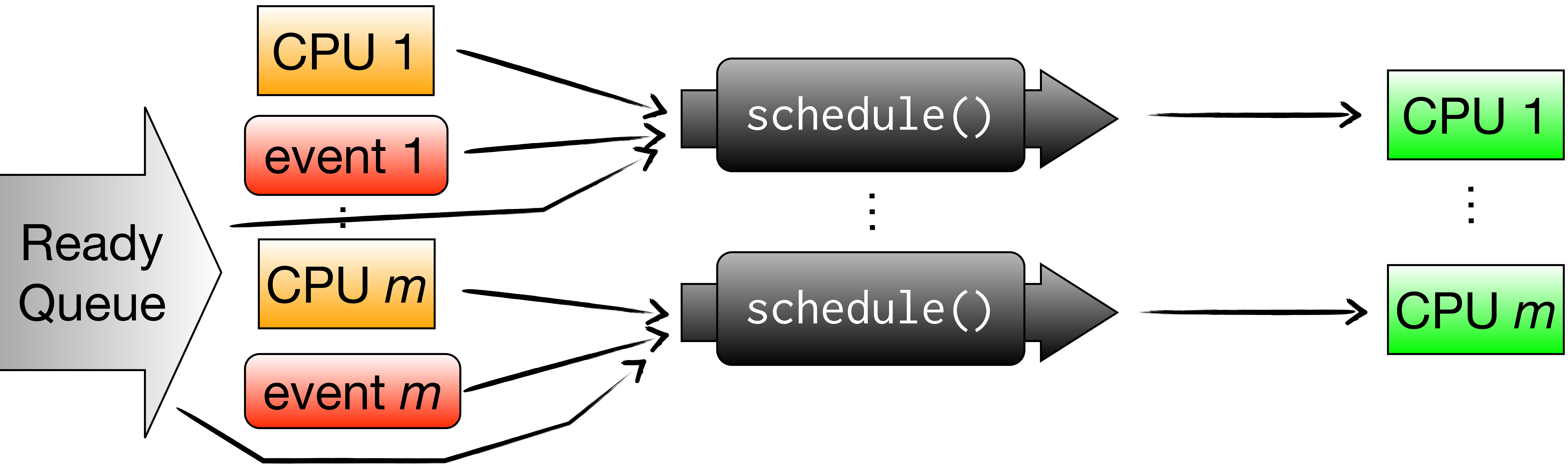


Practical scheduler: job assignment changes only in response to well-defined scheduling events (or at well-known points in time).

Scheduling in Practice



Scheduling in Practice

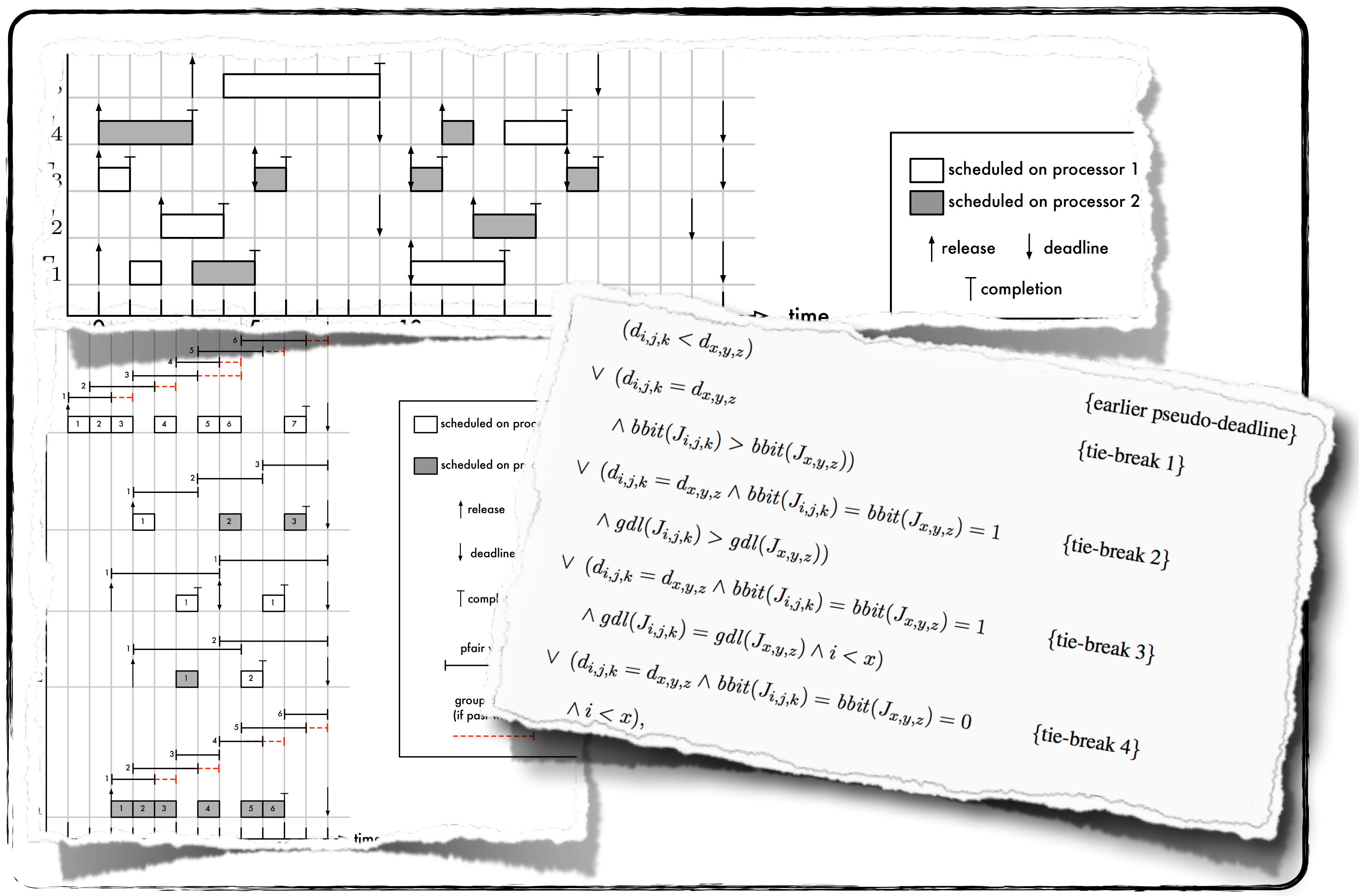


Each processor schedules only itself locally.

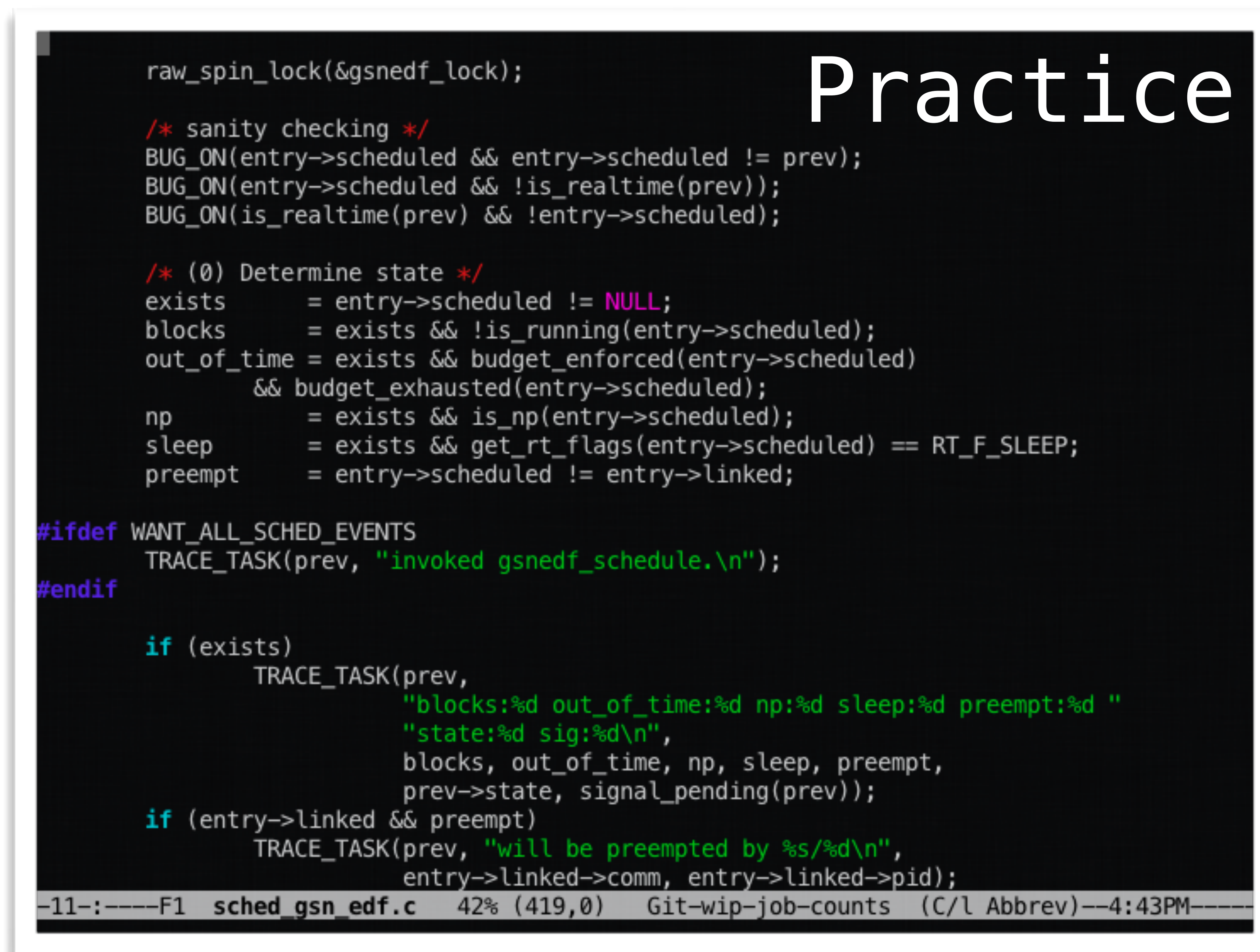
- Multiprocessor schedulers are *parallel* algorithms.
- *Concurrent*, **unpredictable** scheduling events!
- *New events* occur while making decision!
- No *globally consistent atomic snapshot* for free!

Original Purpose of LITMUS^RT

Theory



Develop efficient implementations.




```
        raw_spin_lock(&gsnedf_lock);

        /* sanity checking */
        BUG_ON(entry->scheduled && entry->scheduled != prev);
        BUG_ON(entry->scheduled && !is_realtime(prev));
        BUG_ON(is_realtime(prev) && !entry->scheduled);

        /* (0) Determine state */
        exists      = entry->scheduled != NULL;
        blocks      = exists && !is_running(entry->scheduled);
        out_of_time = exists && budget_enforced(entry->scheduled)
                && budget_exhausted(entry->scheduled);
        np          = exists && is_np(entry->scheduled);
        sleep       = exists && get_rt_flags(entry->scheduled) == RT_F_SLEEP;
        preempt     = entry->scheduled != entry->linked;

#ifdef WANT_ALL_SCHED_EVENTS
        TRACE_TASK(prev, "invoked gsnedf_schedule.\n");
#endif

        if (exists)
                TRACE_TASK(prev,
                           "blocks:%d out_of_time:%d np:%d sleep:%d preempt:%d "
                           "state:%d sig:%d\n",
                           blocks, out_of_time, np, sleep, preempt,
                           prev->state, signal_pending(prev));
        if (entry->linked && preempt)
                TRACE_TASK(prev, "will be preempted by %s/%d\n",
                           entry->linked->comm, entry->linked->pid);
-11-:----F1  sched_gsn_edf.c    42% (419,0)    Git-wip-job-counts   (C/l Abbrev)--4:43PM----
```

Inform: what works well and what doesn't?

History — The first Ten Years

<u>Releases</u>
[RTSS'06]
2007.1
2007.2
2007.3
2008.1
2008.2
2008.3
2010.1
2010.2
2011.1
2012.1
2012.2
2012.3
2013.1
2014.1
2014.2
2015.1
2016.1

Calandrino et al. (2006)
[*not publicly released*]

THE UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL
[2006–2011]

Max Planck Institute for Software Systems
[2011–]

Project initiated by **Jim Anderson** (UNC); *first prototype* implemented by John Calandrino, Hennadiy Leontyev, Aaron Block, and Uma Devi.

Graciously supported over the years by: NSF, ARO, AFOSR, AFRL, and Intel, Sun, IBM, AT&T, and Northrop Grumman Corps.
Thanks!

History — The first Ten Years

<u>Releases</u>

[RTSS'06]
2007.1
2007.2
2007.3
2008.1
2008.2
2008.3
2010.1
2010.2
2011.1
2012.1
2012.2
2012.3
2013.1
2014.1
2014.2
2015.1
2016.1

Calandrino et al. (2006)
[*not publicly released*]

THE UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL
[2006–2011]

Max Planck Institute for Software Systems
[2011–]

<u>Continuously maintained</u>

- ➡ reimplemented for 2007.1
- ➡ 17 major releases spanning 40 major kernel versions (Linux 2.6.20 — 4.1)

<u>Impact</u>

- ➡ used in 50+ papers, and 7 PhD & 3 MSc theses
- ➡ several hundred citations
- ➡ used in South & North America, Europe, and Asia

Goals and Non-Goals

Goal: Make life easier for real-time *systems* researchers

➡ LITMUS^{RT} always was, and remains, primarily a research vehicle

➡ encourage systems research by making it more approachable

Goal: Be sufficiently feature complete & stable to be practical

➡ no point in evaluating systems that can't **run real workloads**

Non-Goal: POSIX compliance

➡ We provide our own APIs — POSIX is old and limiting.

Non-Goal: API stability

➡ We rarely break interfaces, but do it without hesitation if needed.

Non-Goal: Upstream inclusion

➡ LITMUS^{RT} is neither intended nor suited to be merged into Linux.

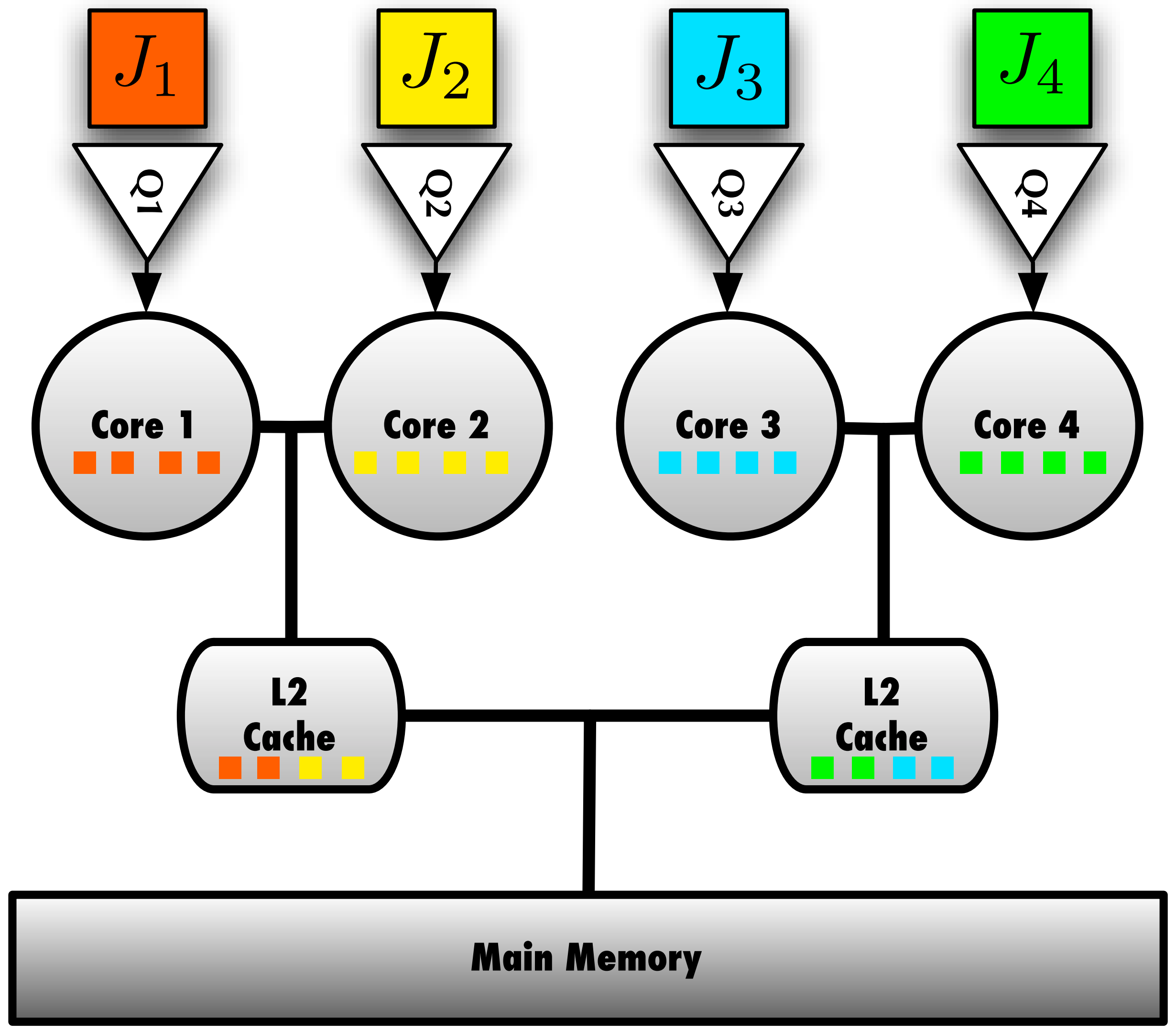
Major Features

What sets LITMUS^RT apart?

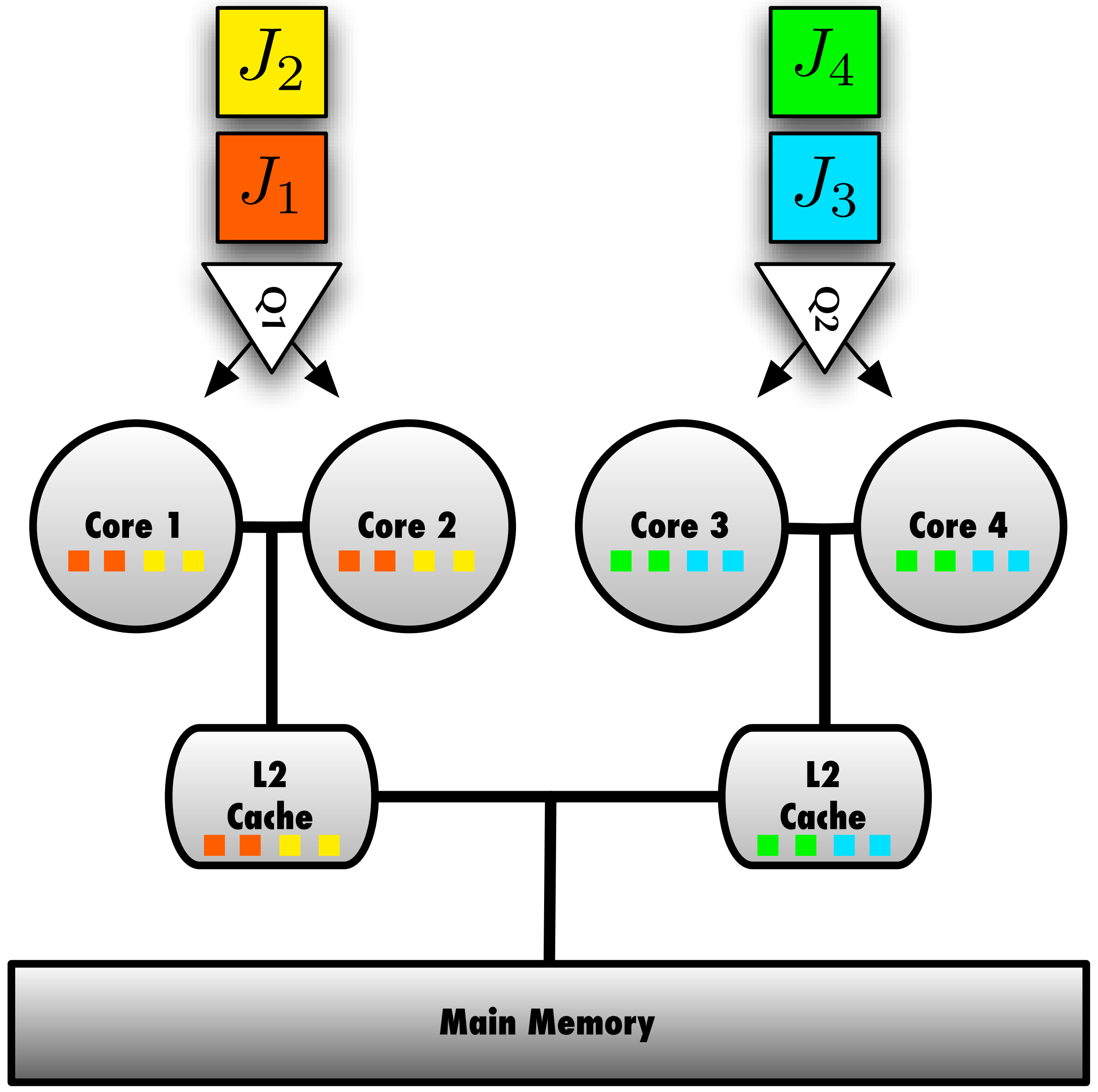
— Part 2 —

Partitioned vs. Clustered vs. Global

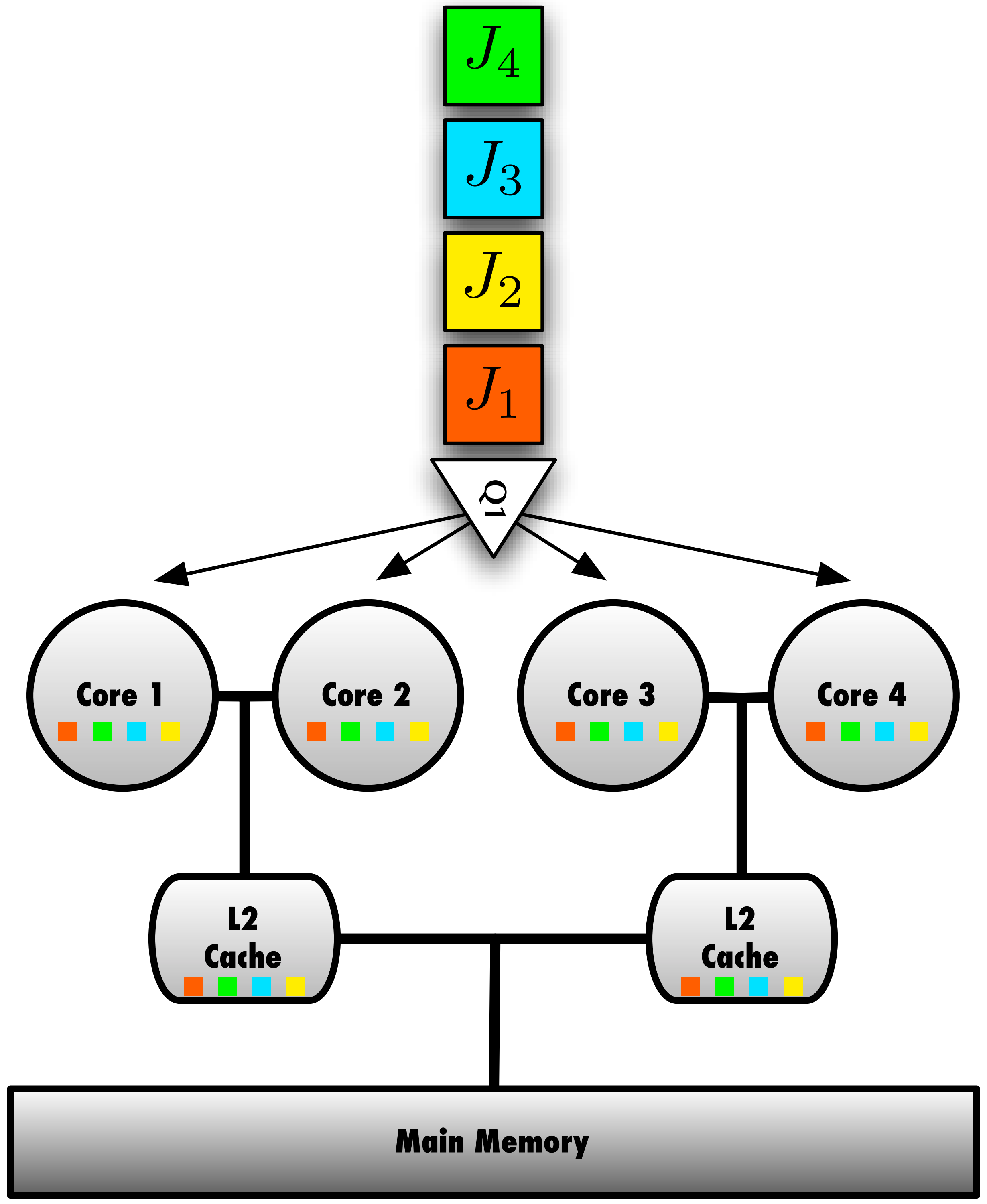
real-time multiprocessor scheduling approaches



partitioned scheduling



clustered scheduling



global scheduling

Predictable Real-Time Schedulers

Matching the literature!

Global EDF

Pfair (PD^2)

Clustered EDF

Partitioned EDF

Partitioned Fixed-Priority (FP)

Partitioned Reservation-Based
polling + table-driven

maintained in mainline LITMUS^RT^

Predictable Real-Time Schedulers

Matching the literature!

Global EDF

Pfair (PD²)

Clustered EDF

Partitioned EDF

Partitioned Fixed-Priority (FP)

Partitioned Reservation-Based
polling + table-driven

maintained in mainline LITMUS^RT^

Global & Clustered Adaptive EDF

Global FIFO

Global FP

RUN

slot shifting

QPS

MC²

Global Message-Passing EDF &FP

Strong Laminar APA FP

EDF-HSB

EDF-WM

NPS-F

EDF-fm

EDF-C=D

…

Sporadic Servers

CBS

CASH

soft-polling

slack sharing

external branches & patches / paper-specific prototypes

Jump-Start Your Research

Bottom line:

➡ The scheduler that you need might already be available.

(Almost) never start from scratch:

➡ If you need to implement a new scheduler, there likely exists a good starting point (e.g., of similar structure).

Plenty of baselines:

➡ At the very least, LITMUS^{RT} can provide you with interesting baselines to compare against.

Predictable Locking Protocols

Matching the literature!

SRP MPCP-VS

FMLP+

DPCP

PCP

DFLP

MPCP

non-preemptive spin locks

maintained in mainline LITMUS^{RT}

MC-IPC

MBWI

Global OMLP

RNLP

OMIP

…

Clustered OMLP

k-exclusion locks

external branches & patches / paper-specific prototypes

Lightweight Overhead Tracing

feather trace

minimal static trace points

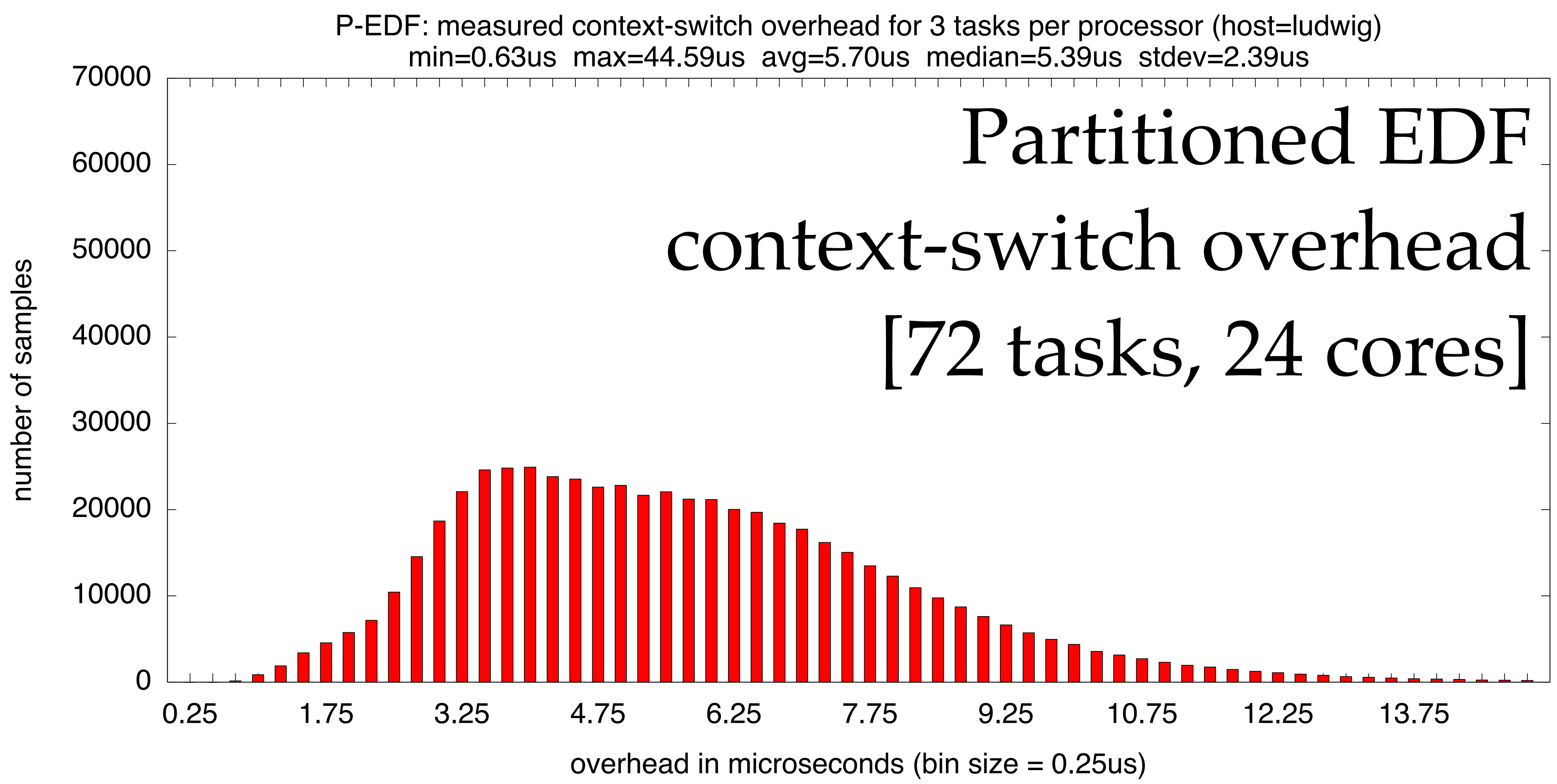
+

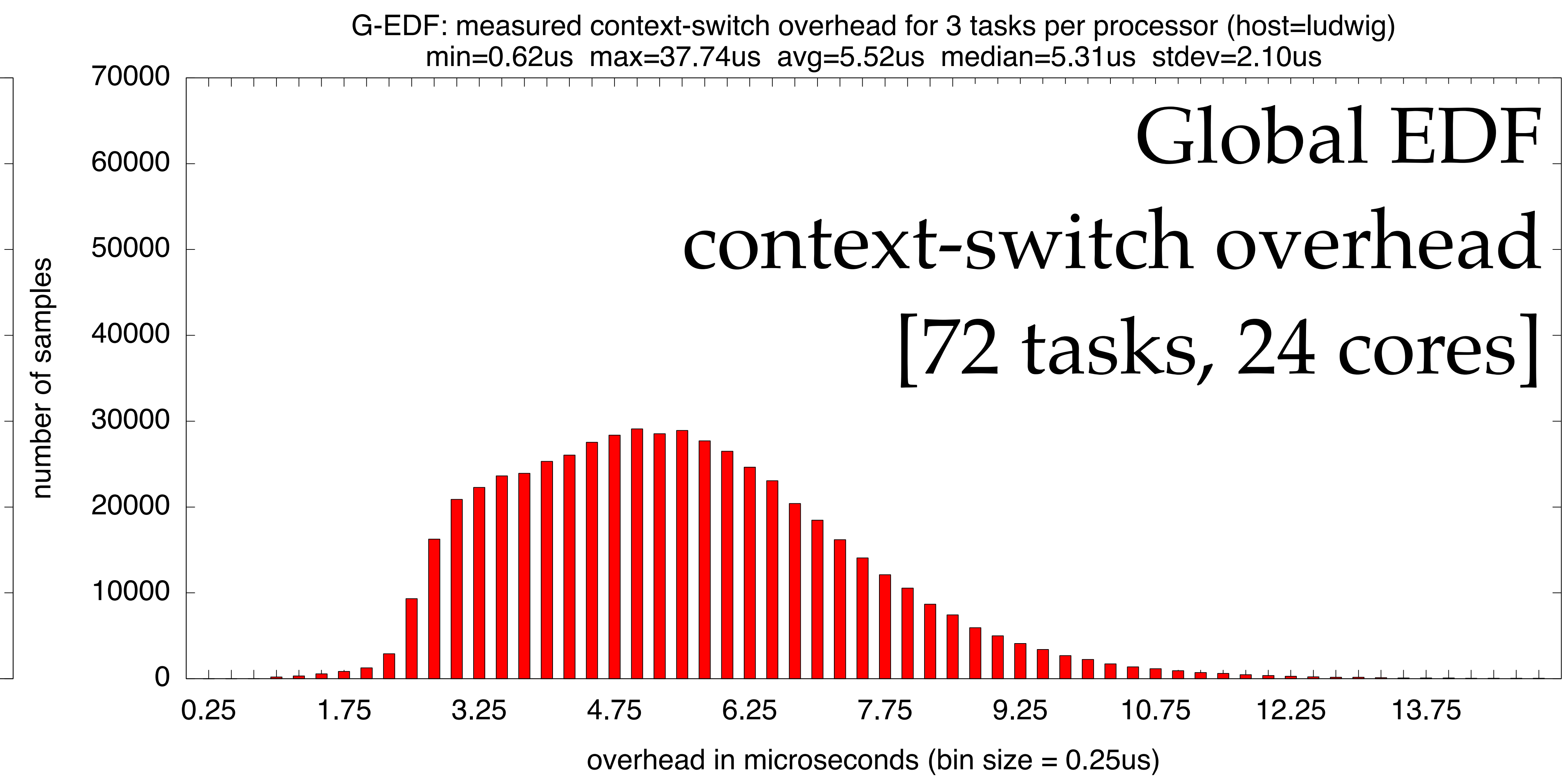
binary rewriting (`jmp` ↔ `nop`)

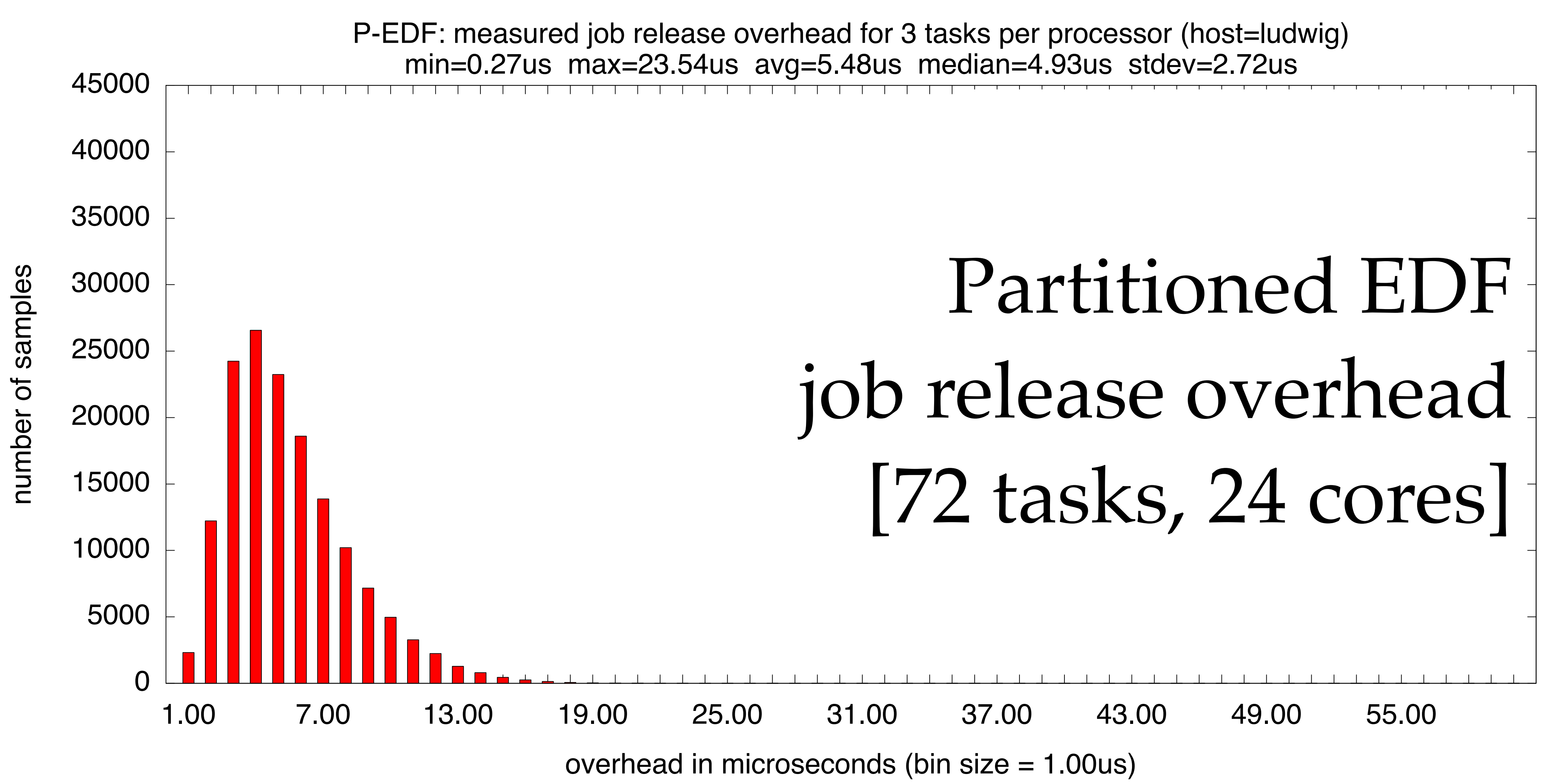
+

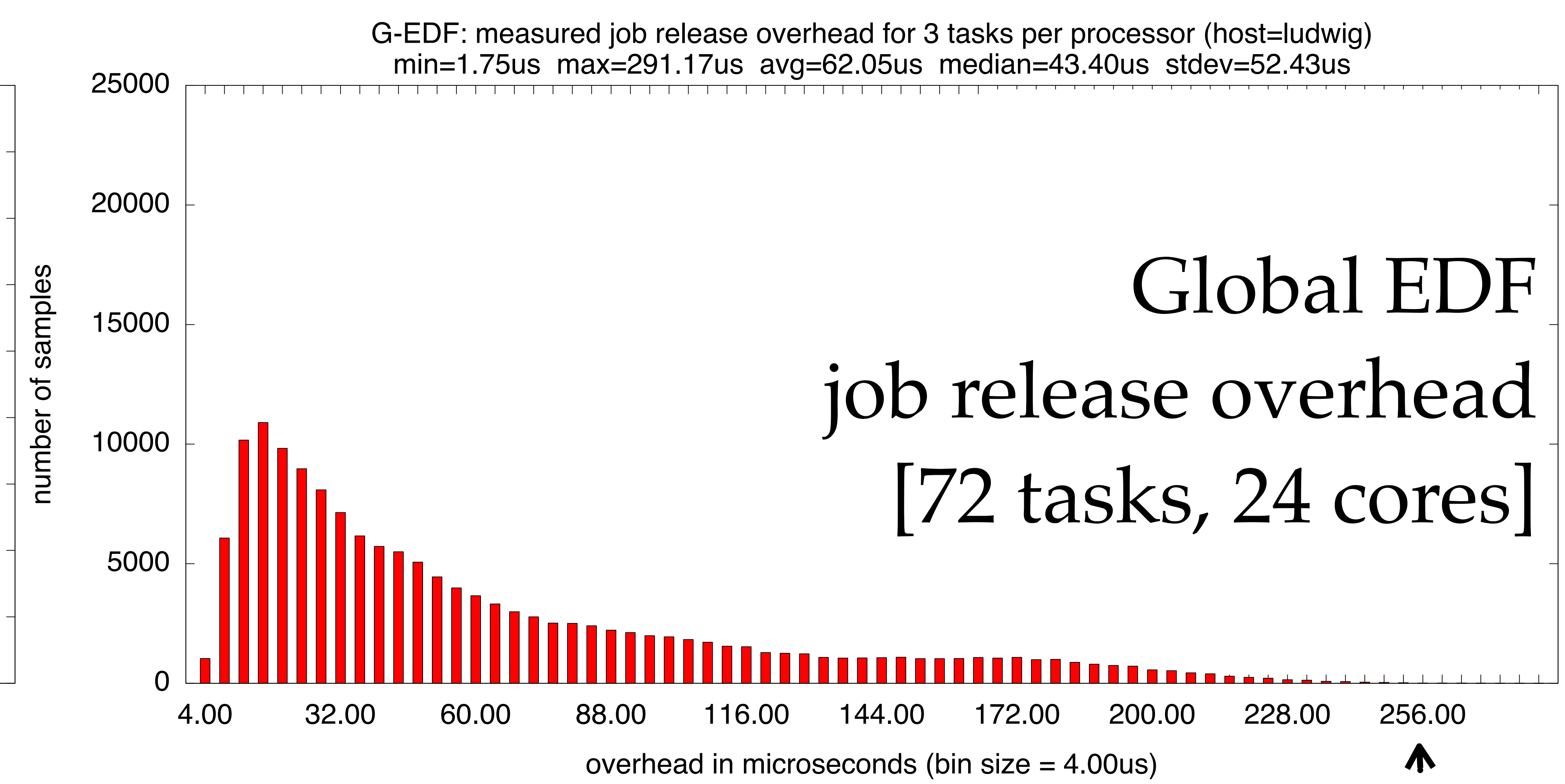
per-processor, wait-free buffers

Evaluate Your Workload *with Realistic Overheads*







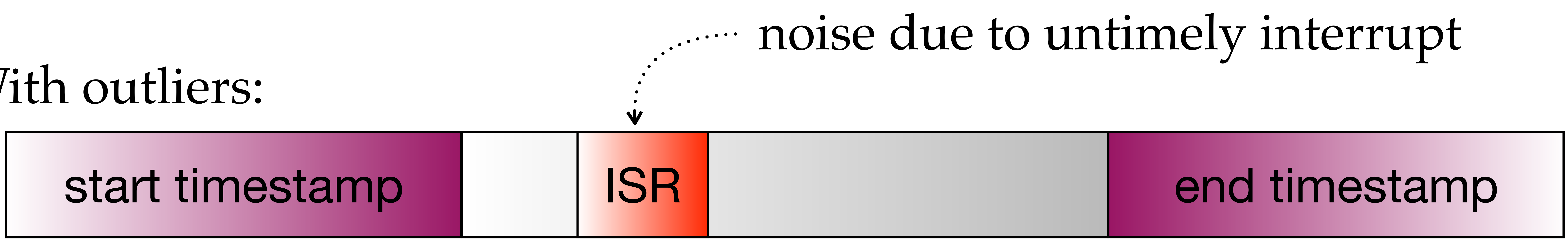


Note the scale!

Automatic Interrupt Filtering

Overhead tracing, ideally:

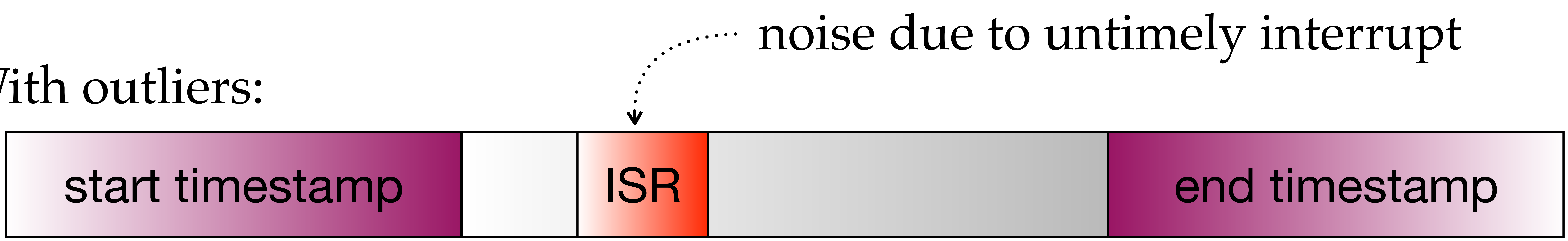
With outliers:



Automatic Interrupt Filtering

Overhead tracing, ideally:

With outliers:



How to cope?

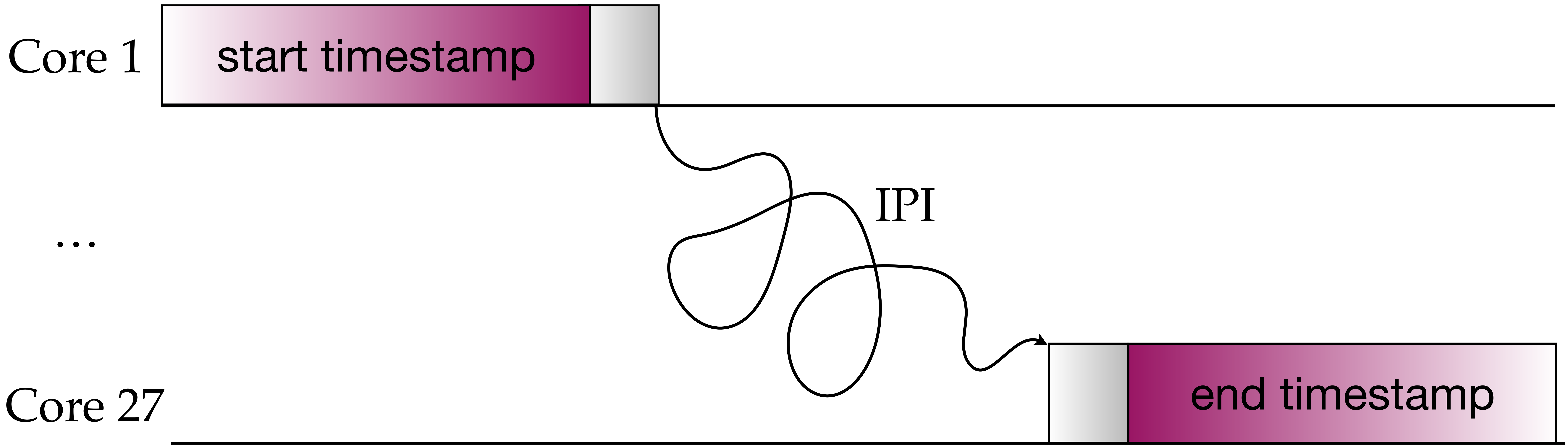
- ➡ can't just turn off interrupts
- ➡ Used statistical filters…
  - …but *which* filter?
  - … what if there are *true* outliers?

Since LITMUS^{RT} 2012.2:

- ➡ ISRs increment counter
- ➡ timestamps include counter snapshots & flag
- ➡ interrupted samples ***discarded automatically***

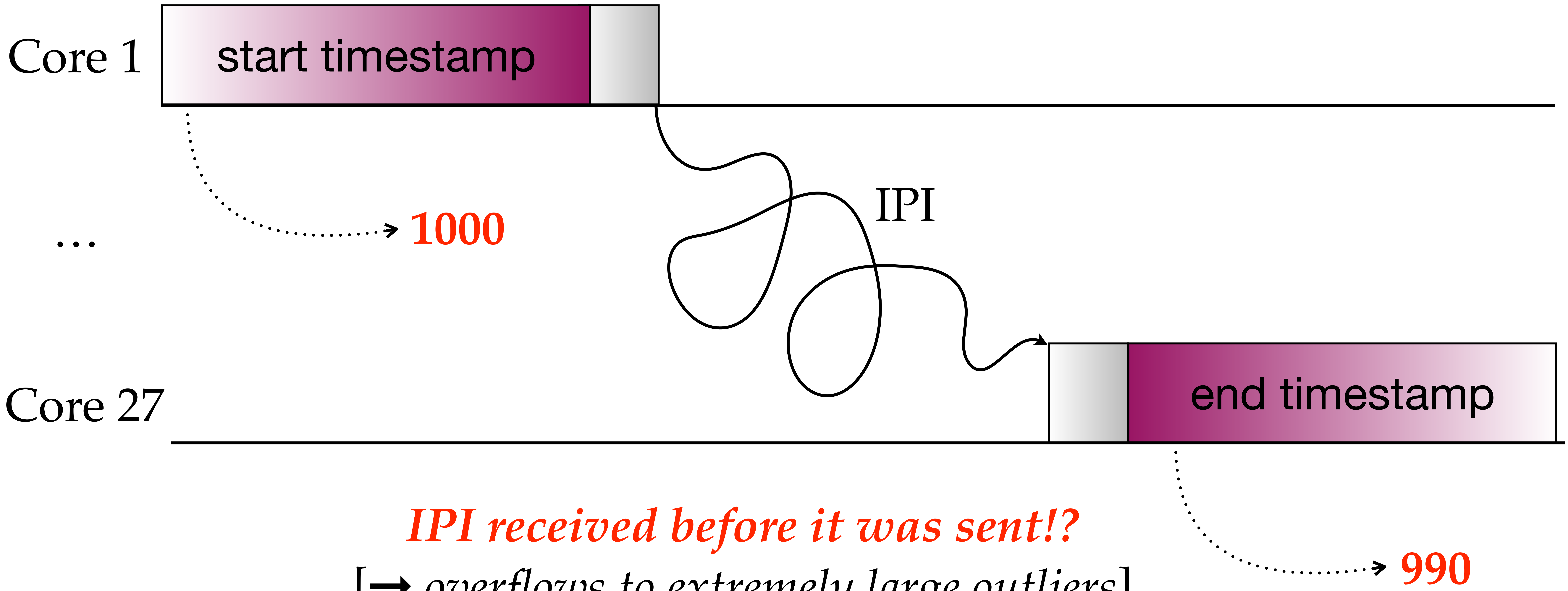
Cycle Counter Skew Compensation

Tracing inter-processor interrupts (IPI):



Cycle Counter Skew Compensation

Tracing inter-processor interrupts (IPI), with **non-aligned** clock sources:

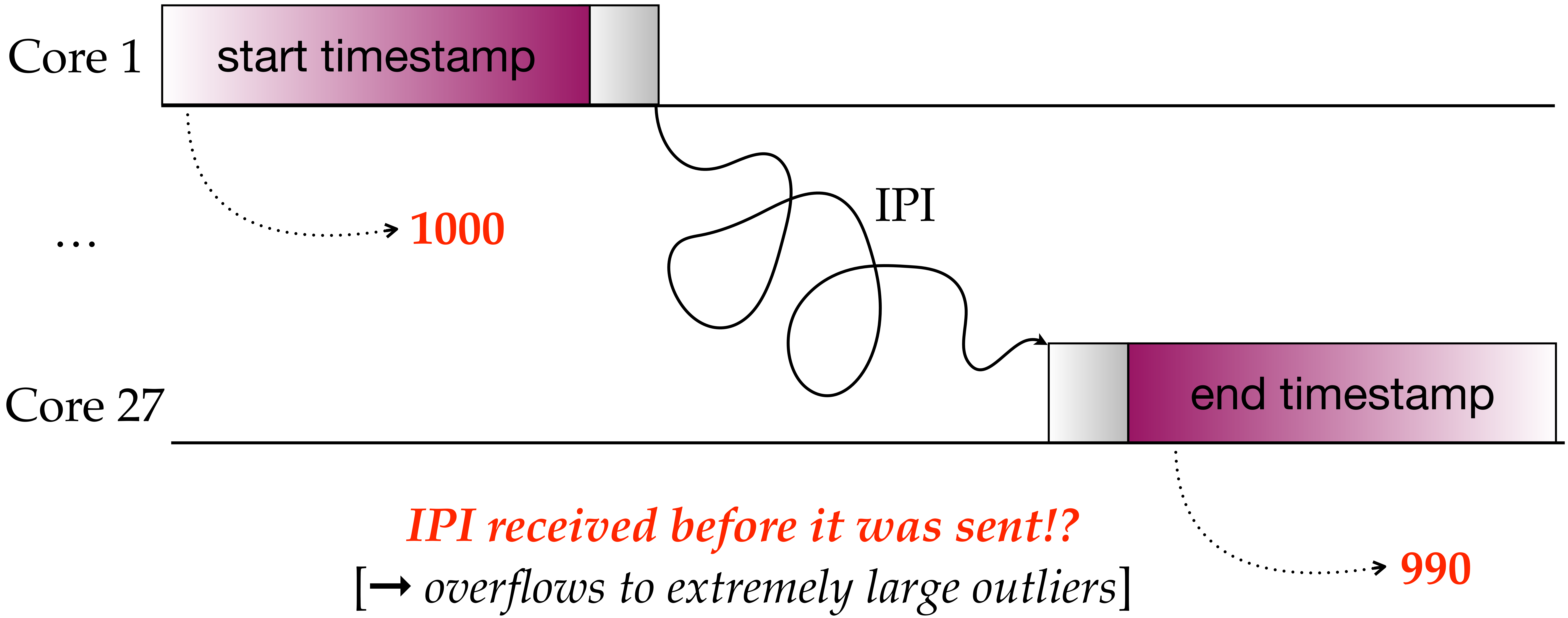


IPI received before it was sent!?

[→ *overflows to extremely large outliers*]

Cycle Counter Skew Compensation

Tracing inter-processor interrupts (IPI), with **non-aligned** clock sources:



IPI received before it was sent!?

[→ *overflows to extremely large outliers*]

In LITMUS^RT, simply run **`ftcat -c`** to measure *and automatically compensate* for unaligned clock sources.

Lightweight Schedule Tracing

task parameters

+

context switches & blocking

+

job releases & deadlines & completions

Built on top of:

feather trace

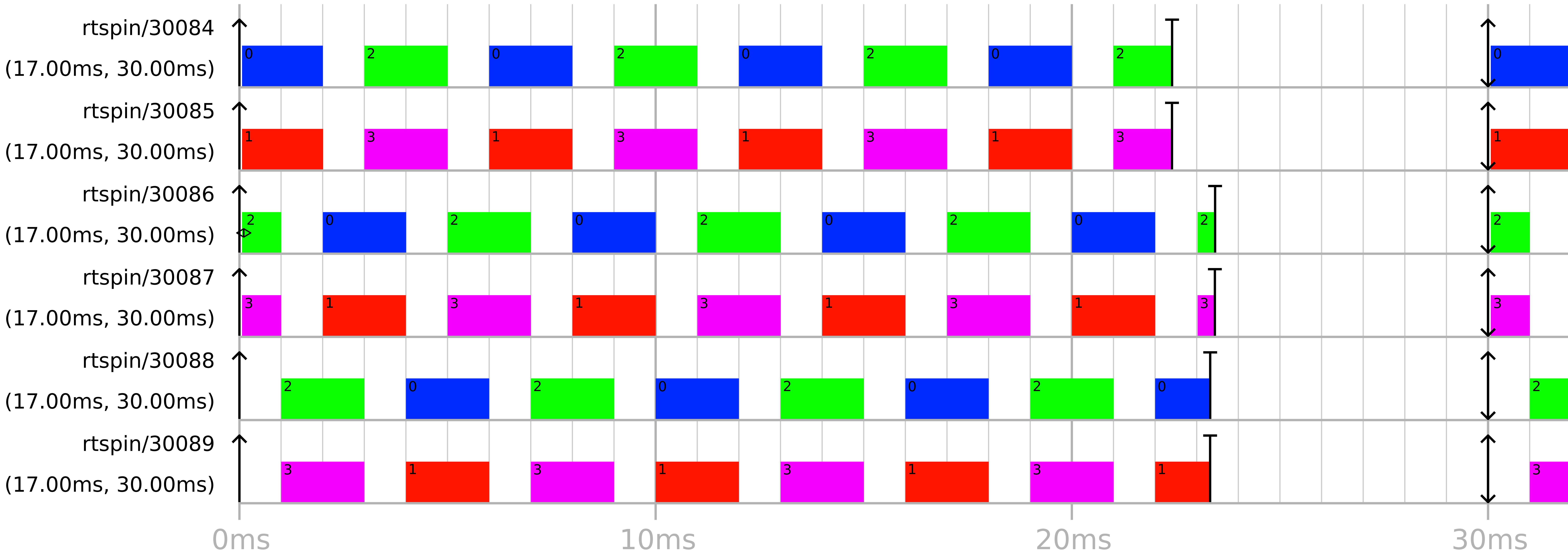
Schedule Visualization: `st-draw`

Ever wondered what a Pfair schedule looks like in reality?

Schedule Visualization: `st-draw`

Ever wondered what a Pfair schedule looks like in reality?

Easy! Just record the schedule with *sched_trace* and run `st-draw`!

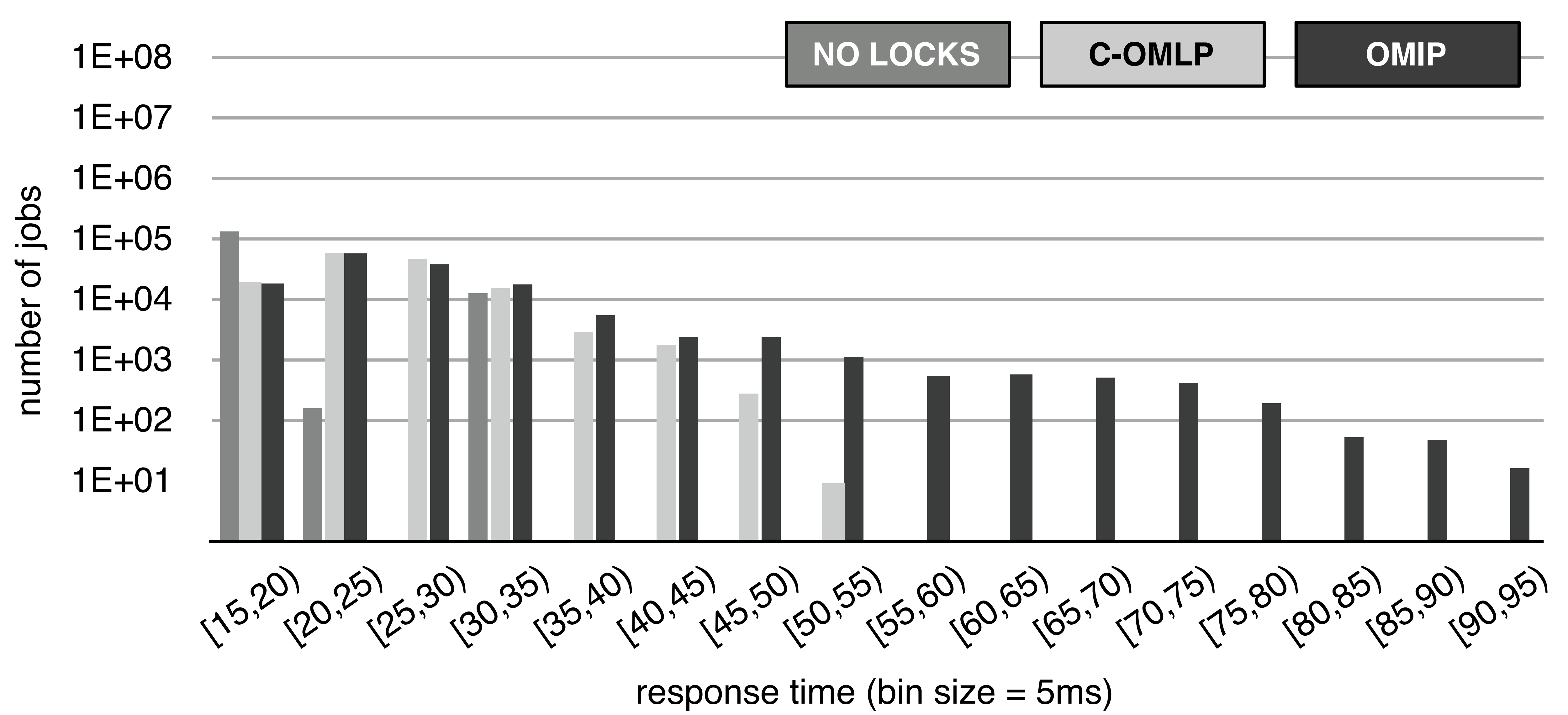


Note: this is ***real*** execution data from a 4-core machine, *not* a simulation! [*Color indicates CPU identity*].

Easy Access to Workload Statistics

"We traced the resulting schedules using LITMUS^{RT}'s ***sched_trace*** *facility and* ***recorded the response times*** *of more than* ***45,000,000*** *individual jobs."*

[—, "A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications", ECRTS'13]



(b) Response times of regular tasks with period $p_i = 100ms$.

Easy Access to Workload Statistics

*"We traced the resulting schedules using LITMUS^{RT} sched_trace facility and **recorded the response times** of more than **45,000,000** individual jobs."*

[—, "A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications", ECRTS'13]

```
(1) st-trace-schedule my-ecrts13-experiments-OMIP
    […run workload…]
```

just a name

```
(2) st-job-stats *my-ecrts13-experiments-OMIP*.bin
```

```
# Task,   Job,     Period,  Response, DL Miss?,   Lateness,  Tardiness, Forced?,       ACET
# task NAME=rtspin PID=29587 COST=1000000 PERIOD=10000000 CPU=0
 29587,      2,   10000000,       1884,        0,   -9998116,          0,        0,       1191
 29587,      3,   10000000,    1019692,        0,   -8980308,          0,        0,    1017922
 29587,      4,   10000000,    1089789,        0,   -8910211,          0,        0,    1030550
 29587,      5,   10000000,    1034513,        0,   -8965487,          0,        0,    1016656
 29587,      6,   10000000,    1032825,        0,   -8967175,          0,        0,    1016096
 29587,      7,   10000000,    1037301,        0,   -8962699,          0,        0,    1016078
 29587,      8,   10000000,    1033699,        0,   -8966301,          0,        0,    1016535
 29587,      9,   10000000,    1037287,        0,   -8962713,          0,        0,    1015794
…
```

Easy Access to Workload Statistics

*"We traced the resulting schedules using LITMUS^RT^ sched_trace facility and **recorded the response times** of more than **45,000,000** individual jobs."*

[—, "A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications", ECRTS'13]

```
(1) st-trace-schedule my-ecrts13-experiments-OMIP
    […run workload…]
```

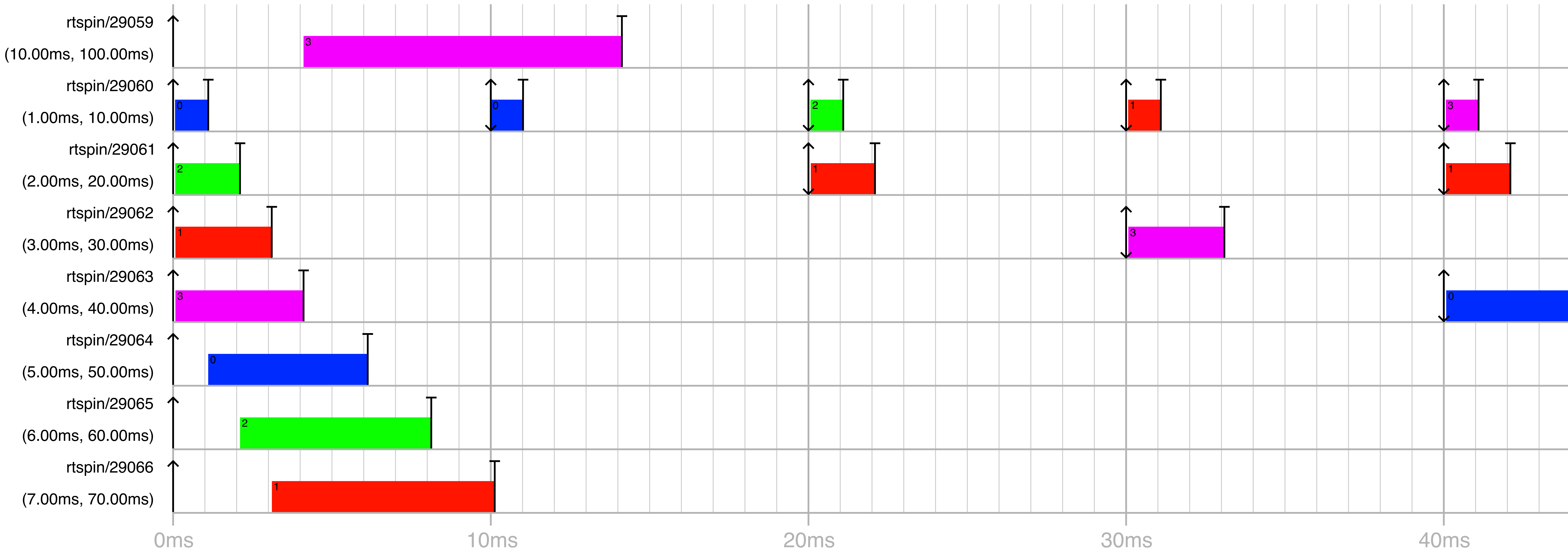
```
(2) st-job-stats *my-ecrts13-experiments-OMIP*.bin
# Task,   Job,     Period,   Response, DL Miss?,   Lateness,  Tardiness, Forced?,       ACET
# task NAME=rtspin PID=29587 COST=1000000 PERIOD=10000000 CPU=0
 29587,      2,   10000000,       1884,         0,   -9998116,          0,        0,       1191
 29587,      3,   10000000,    1019692,         0,   -8980308,          0,        0,    1017922
 29587,      4,   10000000,    1089789,         0,   -8910211,          0,        0,    1030550
 29587,      5,   10000000,    1034513,         0,   -8965487,          0,        0,    1016656
 29587,      6,   10000000,    1032825,         0,   -8967175,          0,        0,    1016096
 29587,      7,   10000000,    1037301,         0,   -8962699,          0,        0,    1016078
 29587,      8,   10000000,    1033699,         0,   -8966301,          0,        0,    1016535
 29587,      9,   10000000,    1037287,         0,   -8962713,          0,        0,    1015794
…
```

How long did each job **use the processor**?

Synchronous Task System Releases

all tasks release their *first job* at a common time "zero."

Synchronous Task System Releases




```
int wait_for_ts_release(void);
```

→ *task sleeps until synchronous release*

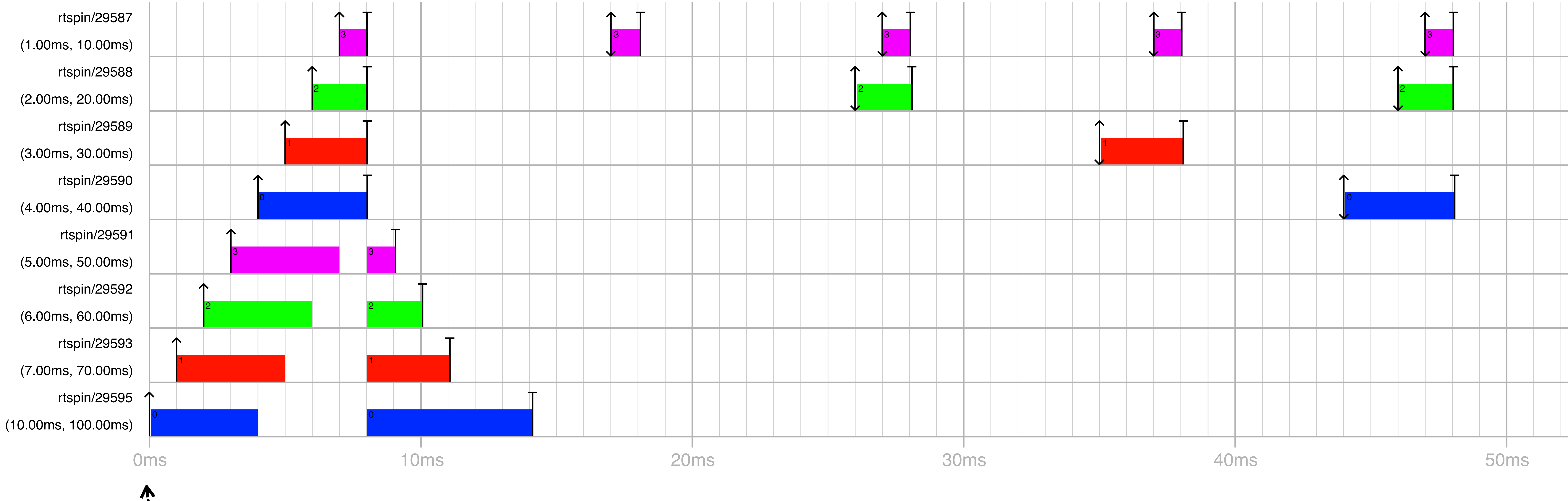
```
int release_ts(lt_t *delay);
```

→ *trigger synchronous release in <delay> nanoseconds*

Asynchronous Releases with Phase/Offset

LITMUS^RT also supports non-zero phase/offset.

➡ release of first job occurs with some *known* offset after task system release.



release of *first job* is staggered w.r.t. time "zero"

➔ *can use schedulability tests for* ***asynchronous periodic*** *tasks*

Easier Starting Point for New Schedulers

simplified scheduler plugin interface

```
struct sched_plugin {
    […]
    schedule_t          schedule;
    finish_switch_t     finish_switch;
    […]
    admit_task_t        admit_task;
    fork_task_t         fork_task;

    task_new_t          task_new;
    task_wake_up_t      task_wake_up;
    task_block_t        task_block;

    task_exit_t         task_exit;
    task_cleanup_t      task_cleanup;
    […]
}
```

simplified interface

+

richer task model

+

plenty of working code to steal from

Many More Features...

Support for true global scheduling

- ➡ supports proper **pull-migrations**
  - ‣ moving tasks among Linux's per-processor runqueues
- ➡ Linux's **SCHED_FIFO** and **SCHED_DEADLINE** global scheduling "emulation" is not 100% correct (**races possible**)

Low-overhead non-preemptive sections

- ➡ Non-preemptive spin locks **without system calls**.

Wait-free preemption state tracking

- ➡ *"Does this remote core need to be sent an IPI?"*
- ➡ Simple API suppresses superfluous IPIs

Debug tracing with `TRACE()`

- ➡ Extensive support for "`printf()` debugging" ➞ *dump from Qemu*

LITMUS^{RT}

Predictable execution platform and research accelerator.

Apply **schedulability** analysis

+

under consideration of **overheads**

+

jump-start your development

Key Concepts

What you need to know to get started

— Part 3 —

Scheduler Plugins

Linux scheduler classes:

pick_next_task()

- SCHED_LITMUS — active plugin
- SCHED_DEADLINE
- SCHED_FIFO/RR
- SCHED_OTHER (CFS)
- SCHED_IDLE

LITMUS^{RT} plugins:

- Linux (dummy)
- PSN-EDF
- GSN-EDF
- C-EDF
- P-FP
- P-RES

SCHED_LITMUS “class” invokes *active plugin*.

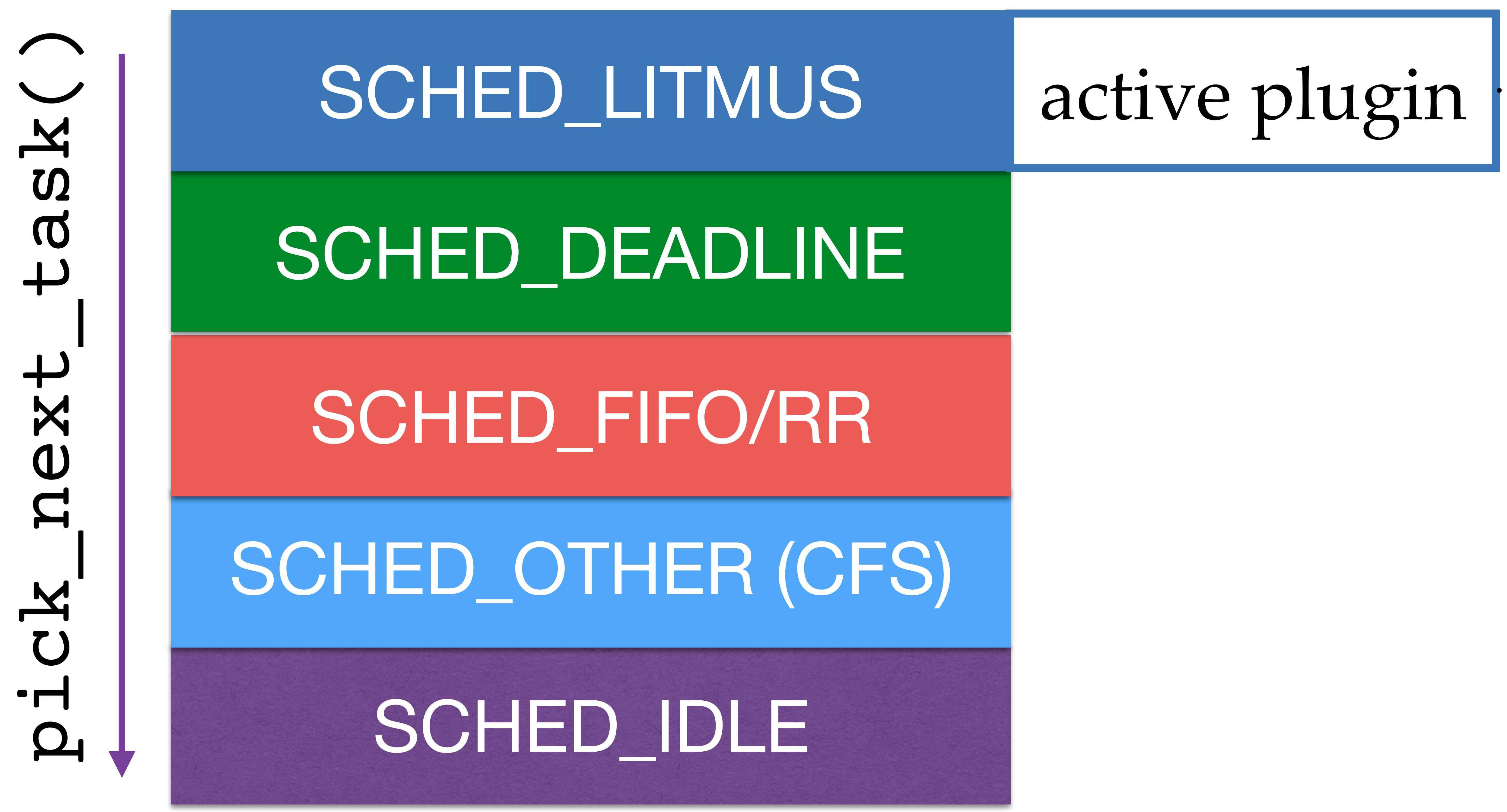
➡ LITMUS^{RT} tasks have highest priority.

➡ SCHED_DEADLINE & SCHED_FIFO/RR:

➔ ***best-effort*** from SCHED_LITMUS point of view

Plugin Switch

Linux scheduler classes:



LITMUS^{RT} plugins:

- Linux (dummy)
- PSN-EDF
- GSN-EDF
- C-EDF
- P-FP
- P-RES

```
$ setsched PSN-EDF
```

Active plugin can be switched at runtime.

➡ But only if no real-time tasks are present.

Classic Process-Based Plugins

SCHED_LITMUS | active plugin

PSN-EDF

T_1 T_2 … T_n

Plugin manages real-time tasks <u>directly</u>

➡ one sporadic task (scheduling theory)
= *exactly* one process in Linux

WCET

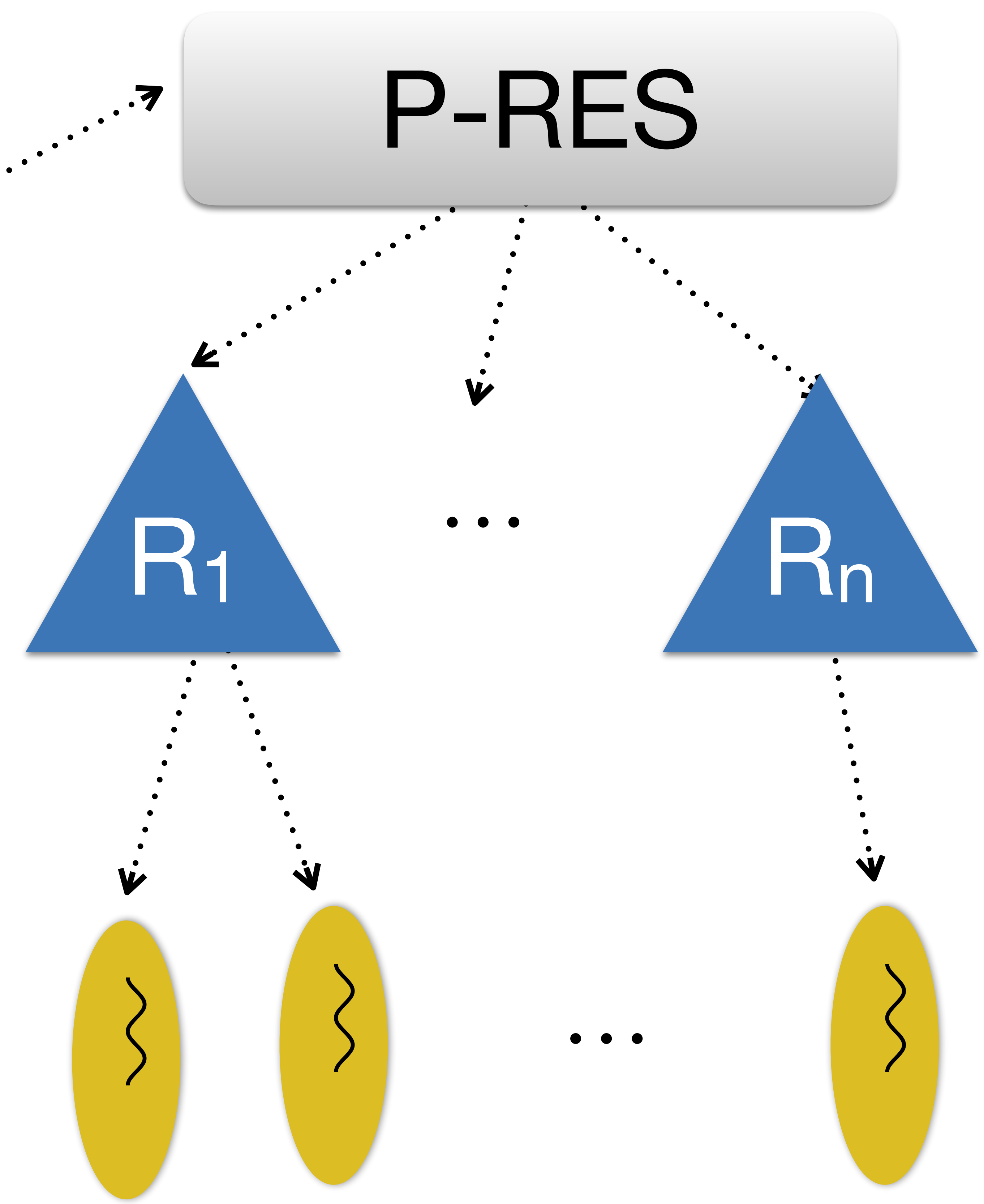
```
$ rt_launch -p 1 10 100 <my-app-binary>
```

core

period

Reservation-Based Scheduling (RBS)

SCHED_LITMUS | active plugin



Plugin schedules <u>reservations</u>
(= process *containers*)

➡ one sporadic task in analysis
= one **reservation** (→ *0..n processes*)

➡ second-level dispatcher in each reservation

arbitrary ID **core** **budget** **period**

```
$ resctl -n 123 -c 1 -b 10 -p 100
$ rt_launch -p 1 -r 123 <my-app-component-1>
$ rt_launch -p 1 -r 123 <my-app-component-2>
[...]
```

RBS: Motivation 1/2

***Temporal* Decomposition: Sequential, Recurrent Tasks**

➡ sequential tasks: basic element of RT scheduling theory

vs

***Logical* Decomposition: Software Modularization**

➡ split complex applications into many *logical* modules or components to manage spheres of responsibility

<u>Example</u>: one real-time "task" may consist of multiple processes

middleware process + redundant, isolated application threads + database process

RBS: Motivation 2/2

Irregular execution patterns

- ➡ e.g., http server triggered by unpredictable incoming requests
- ➡ with RBS, can ***safely encapsulate*** arbitrary activation patterns

"Legacy" and complex software

- ➡ Cannot require existing, complex software (e.g., ROS) to adopt and comply with LITMUS^{RT} API
- ➡ with RBS, can ***transparently encapsulate*** any Linux process
  - *Even kernel threads (e.g., interrupt threads)!*

Worst-case execution time is a fantasy

- ➡ Most practical systems must live with imperfect measurements and cannot always provision on a (measured) worst-case basis
- ➡ with RBS, can ***manage overruns predictably*** and actively ***exploit slack***

Three Main Repositories

Linux kernel patch

➔ `litmus-rt`

+

user-space interface

➔ `liblitmus`

+

tracing infrastructure

➔ `feather-trace-tools`

liblitmus: The User-Space Interface

C API (task model + system calls)

+

user-space tools

➔ `setsched`, `showsched`, `release_ts`, `rt_launch`, `rtspin`

/proc/litmus/* and /dev/litmus/*

/proc/litmus/*

- Used to export information about the plugins and existing real-time tasks.
- Read- and writable files.
- Typically managed by higher-level wrapper scripts.

/dev/litmus/*

- Special device files based on custom character device drivers.
- Primarily, export **trace data** (use only with `ftcat`):
  - `ft_cpu_traceX` — core-local overheads of CPU X
  - `ft_msg_traceX` — IPIs related to CPU X
  - `sched_traceX` — scheduling events on CPU X
- `log` — debug trace (use with regular `cat`)

Control Page: `/dev/litmus/ctrl`

A (private) per-process page mapped by each real-time task

- Shared memory segment between kernel and task.
- Purpose: **low-overhead communication channel**
- interrupt count
- *preemption-disabled* and *preemption-needed* flags
- current deadline, etc.

Second purpose, as of 2016.1

- implements LITMUS^{RT} "system calls" as `ioctl()` operations
- improves portability and reduces maintenance overhead

Transparent use

- `liblitmus` takes care of everything

(Lack of) Processor Affinities

In Linux, each process has a processor affinity mask.

X

Most LITMUS[RT] plugins ignore affinity masks.

➡ In particular, all plugins in the mainline version do so.

- *Global is global; partitioned is partitioned…*

Recent out-of-tree developments

➡ Support for *hierarchical* affinities [ECRTS'16]

Things That Are Not Supported

With limited resources, we cannot possibly support & test all Linux features.

Architectures other than x86 and ARM

➡ *Though not difficult to add support if someone cares…*

Running on top of a hypervisor

➡ It works (→ *hands-on session*), but it's not "officially" supported.

➡ You *can* use LITMUS^{RT} as a real-time hypervisor by encapsulating **kvm** in a reservation.

CPU Hotplug

➡ Not supported by existing plugins.

Processor Frequency Scaling

➡ Plugins "work," but oblivious to speed changes.

Integration with PREEMPT_RT

➡ For historic reasons, the two patches are incompatible

➡ Rebasing on top of PREEMPT_RT has been on the wish list for some time…

LITMUS^{RT}

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

Enable *practical* multiprocessor real-time *systems* research under *realistic conditions*.

Connect theory and practice.

Don't reinvent the wheel.

Use LITMUS^{RT} as a baseline.