RTAS-WiP 2019

A Unified Runtime Framework for Weakly-hard Real-time Systems

Hyunjong Choi, Hyoseung Kim

UNIVERSITY OF CALIFORNIA
UCRIVERSIDE

RTEN
Real-time Embedded and Networked Systems
Laboratory

Weakly-hard real-time systems [‡]

- Improve resource usage efficiency
  - Tolerable to some deadline misses w/o affecting functional correctness

$\boldsymbol{(m, K)}$: at most m jobs can miss their deadlines among any K consecutive jobs

- Various assumptions on handling of deadline-missed jobs

Handling scheme	Prior work
Job abort	Goossens (RTN, 2008), Koren (RTSS, 1995), Ramanathan (1999)
Delayed completion	Hammadeh (ECRTS 2017), Sun (TECS, 2017)
Job pre-skip	Koren (RTSS, 1995), Ramanathan (1999)

< Weakly-hard studies based on job handlings >

➡ No prior work of comparative analysis among various handling schemes

[‡] G. Bernat, A. Burns, and A. Liamosi, “Weakly hard real-time systems,” IEEE transactions on Computers, 2001

Handling of deadline-missed jobs

- Four handling schemes

Job abort

- ✓ Terminate immediately
- ✓ No effect on the next released job
- ✓ Drawback: implementation cost (rollback : system-level vs. **task-level**)

Delayed completion

- ✓ Run until a job completes
- ✓ Can Improve quality of service of a system
- ✓ Drawback: no merits of weakly-hard concept in overloaded situations

Job pre-skip

- ✓ Determine at a job release time
- ✓ **Online** (slack time) and offline (predetermined patterns)
- ✓ Drawback: runtime overhead (slack) and underutilization

Job post-skip

- ✓ Run until a job completes, but discard the next released job
- ✓ Drawback: degradation of quality of service of a system

Runtime framework

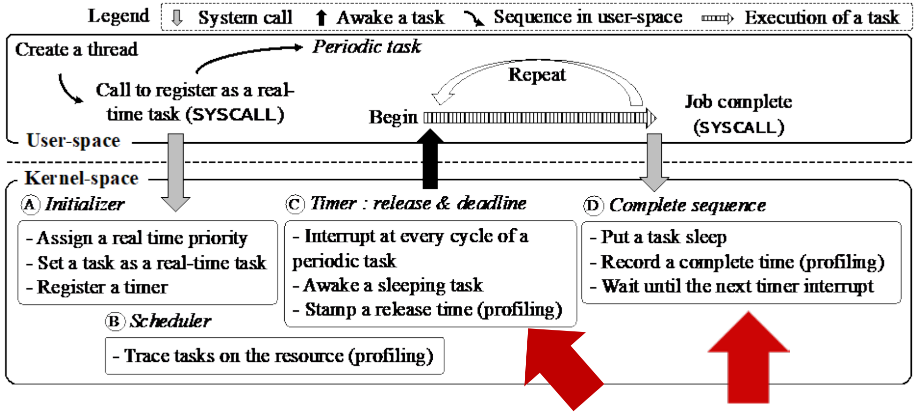
- Job abort
  - *Rollback* mechanism (task-level)

  Step 1. Store a checkpoint

  Step 2. Notify a deadline miss to the user space

  Step 3. Recover from the checkpoint

< Runtime mechanism for periodic task execution >



Legend ⇩ System call ⬆ Awake a task ↘ Sequence in user-space ⇨ Execution of a task

Create a thread

Call to register as a real-time task (SYSCALL)

Periodic task

Repeat

Begin

Job complete (SYSCALL)

User-space

Kernel-space

Ⓐ *Initializer*

- Assign a real time priority
- Set a task as a real-time task
- Register a timer

Ⓑ *Scheduler*

- Trace tasks on the resource (profiling)

Ⓒ *Timer : release & deadline*

- Interrupt at every cycle of a periodic task
- Awake a sleeping task
- Stamp a release time (profiling)

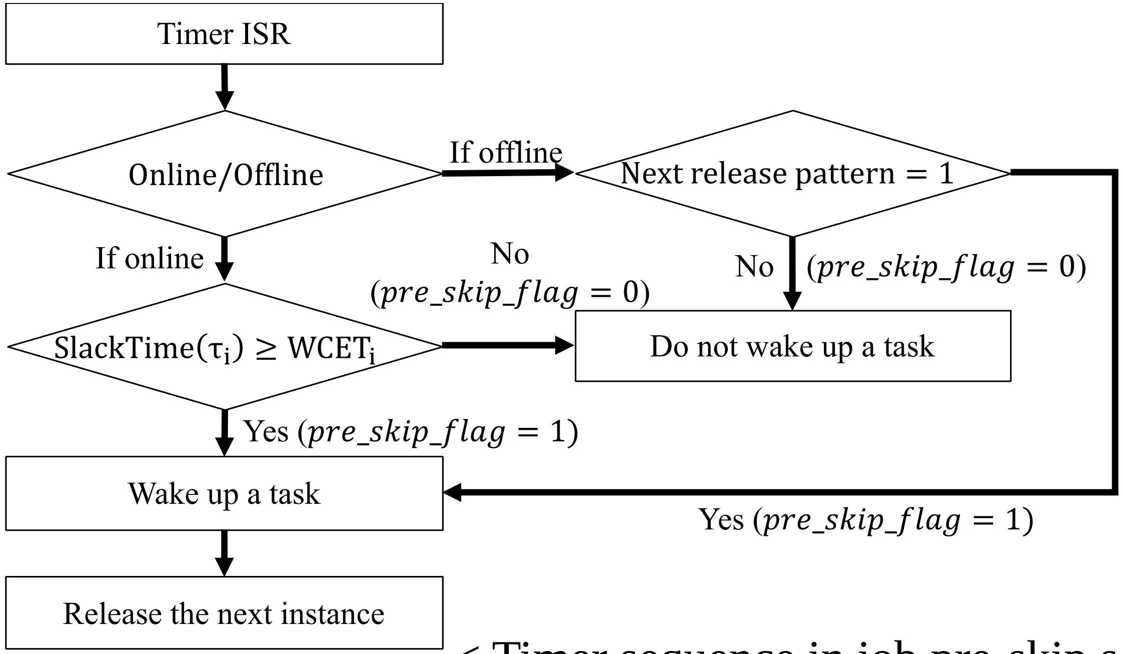
Ⓓ *Complete sequence*

- Put a task sleep
- Record a complete time (profiling)
- Wait until the next timer interrupt

Additional sequences based on handling schemes

- Delayed completion
  - Put in sleep mode when the latest released job is completed

- Job pre-skip
  - Online vs offline



Timer ISR

Online/Offline

If offline

If online

Next release pattern = 1

No (*pre_skip_flag* = 0)

No

(*pre_skip_flag* = 0)

$\mathrm{SlackTime}(\tau_i) \geq \mathrm{WCET}_i$

Do not wake up a task

Yes (*pre_skip_flag* = 1)

Wake up a task

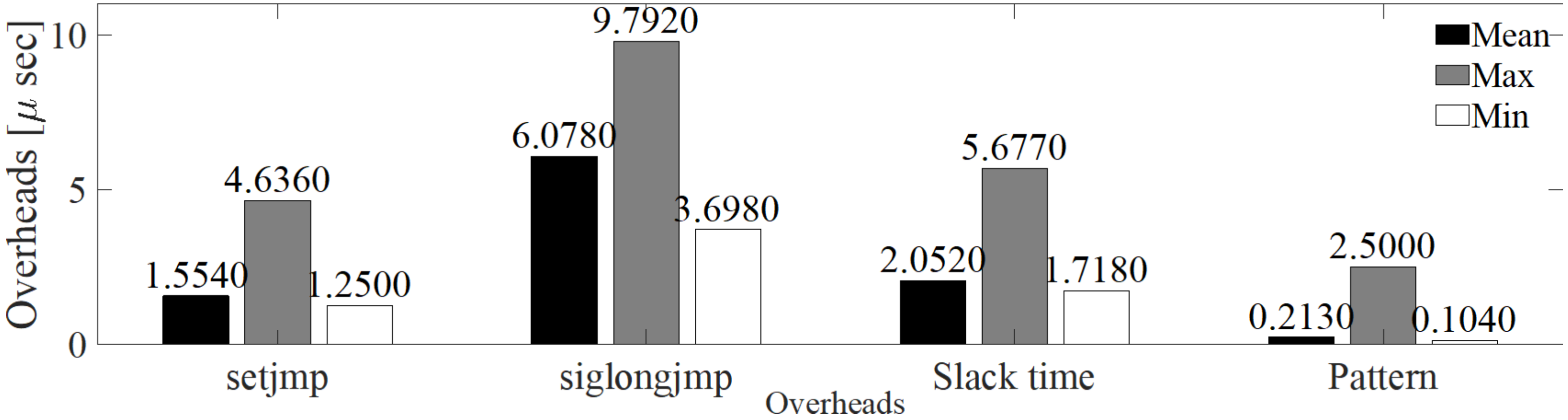
Yes (*pre_skip_flag* = 1)

Release the next instance

< Timer sequence in job pre-skip scheme >

Computational overheads

- Experimental setup
  - Linux kernel running on Raspberry Pi 3 (Quad Cortex A53 @ 1.2GHz)
- Four major sequences that can cause extra runtime overhead
  - `sigsetjmp` (job abort), `siglongjmp` (job abort), slack (job pre-skip), pattern (job pre-skip)



Acceptabily small in μs units, compared to periods of tasks denoted in ms

Conclusion & Future work

- Conclusion
  - Proposed a unified runtime framework for multiple deadline-miss handling schemes in weakly-hard real-time systems
  - Applicable to other OSs using fixed-priority preemptive schedulers
  - Different results (violation of the constraints, utilization) observed depending on the handling scheme for the same taskset

- Future work
  - Will use for the issues that have not studied much in weakly-hard context (e.g., inter-task dependency, shared resources, multicore systems, and contention in cache and main memory)

Thank you

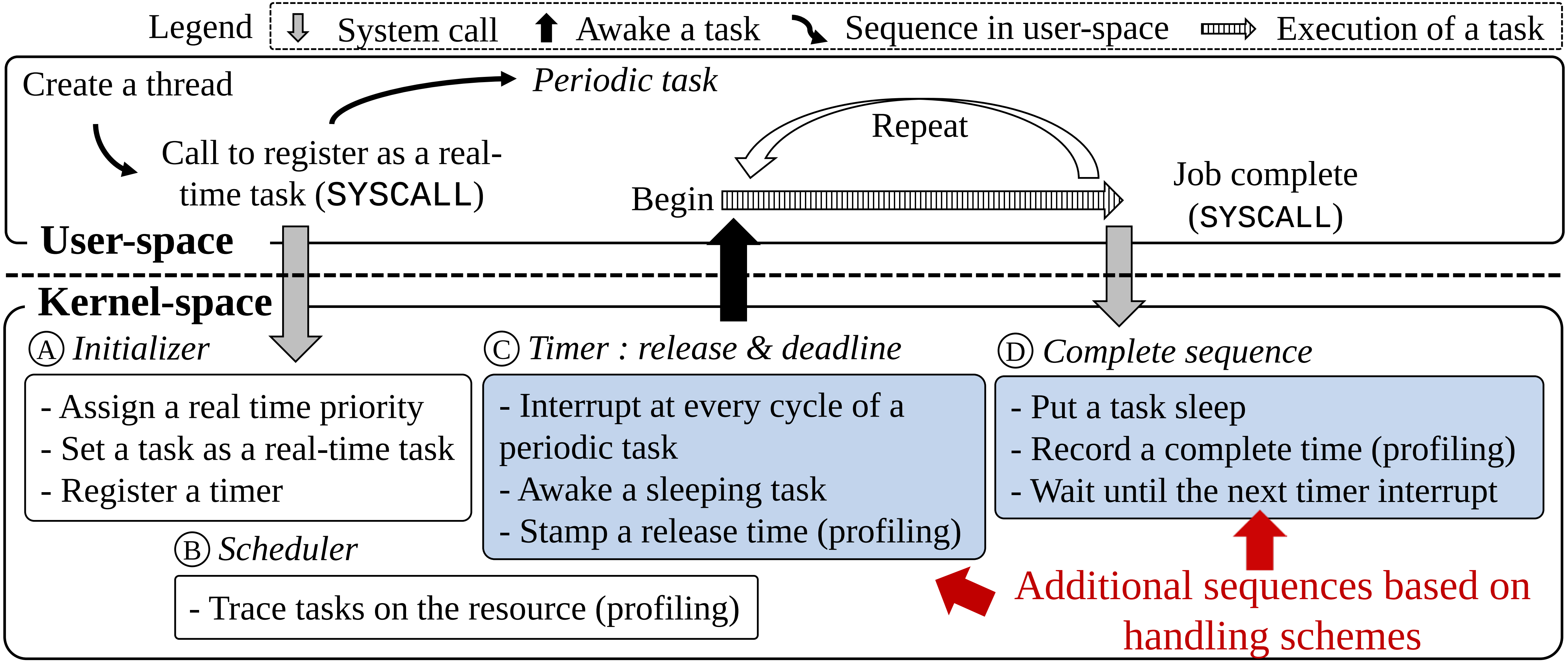
A Unified Runtime Framework for Weakly-hard Real-time Systems

Hyunjong Choi, Hyoseung Kim

Q & A

Runtime mechanism

- A fundamental runtime mechanism for periodic task execution



Legend: ⇩ System call ⬆ Awake a task ↘ Sequence in user-space ⇨ Execution of a task

User-space

Create a thread

Call to register as a real-time task (SYSCALL)

Periodic task

Repeat

Begin

Job complete (SYSCALL)

Kernel-space

Ⓐ *Initializer*

- Assign a real time priority
- Set a task as a real-time task
- Register a timer

Ⓑ *Scheduler*

- Trace tasks on the resource (profiling)

Ⓒ *Timer : release & deadline*

- Interrupt at every cycle of a periodic task
- Awake a sleeping task
- Stamp a release time (profiling)

Ⓓ *Complete sequence*

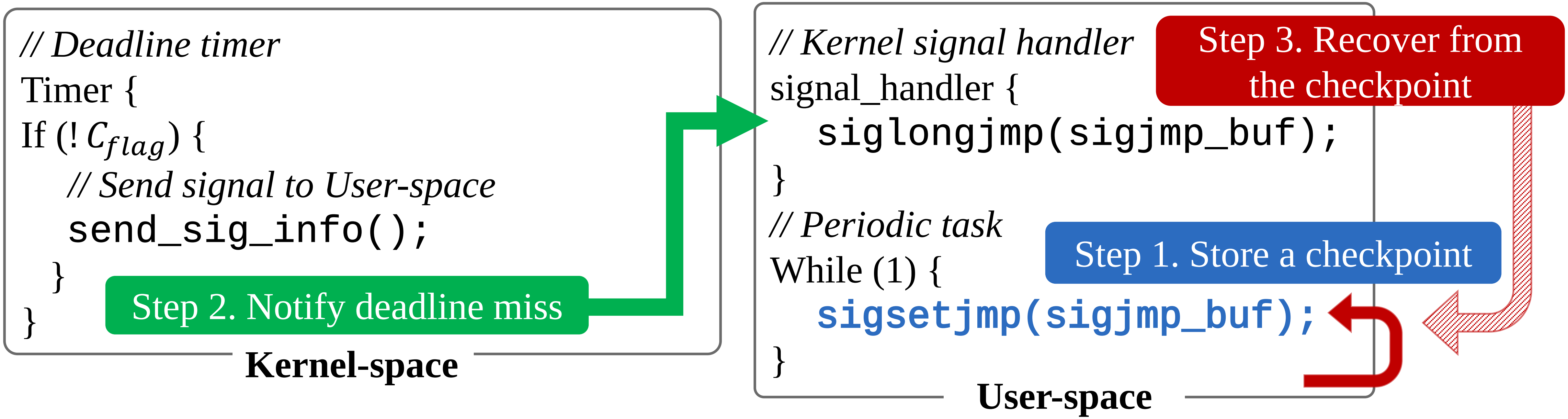
- Put a task sleep
- Record a complete time (profiling)
- Wait until the next timer interrupt

Additional sequences based on handling schemes

Job abort scheme

- Employed *task-level rollback* approach

PC & SP rollback,



```
// Deadline timer
Timer {
If (! C_flag) {
    // Send signal to User-space
    send_sig_info();
  }
}
```

Step 2. Notify deadline miss

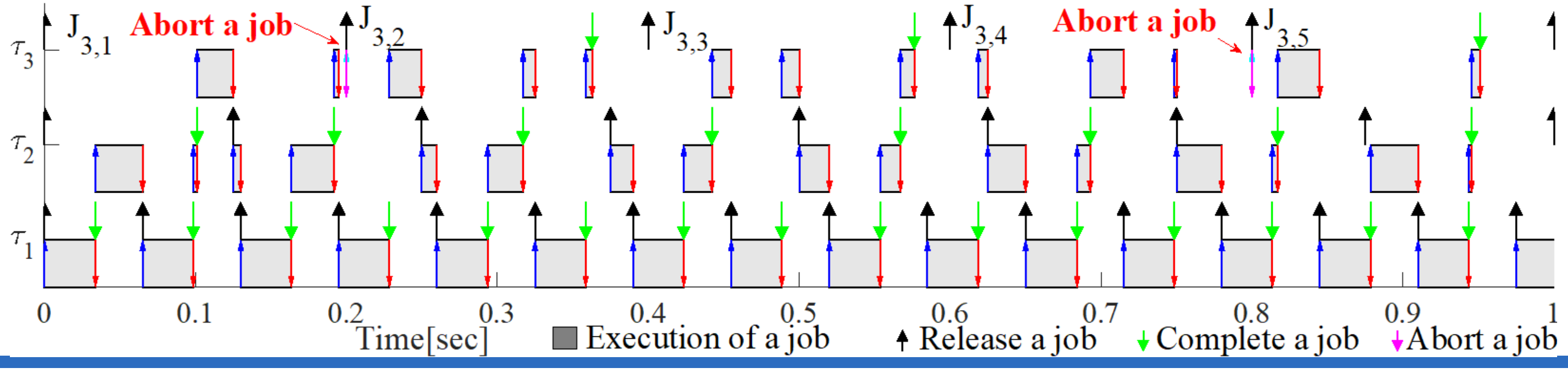
Kernel-space

```
// Kernel signal handler
signal_handler {
   siglongjmp(sigjmp_buf);
}
// Periodic task
While (1) {
   sigsetjmp(sigjmp_buf);
}
```

Step 3. Recover from the checkpoint

Step 1. Store a checkpoint

User-space

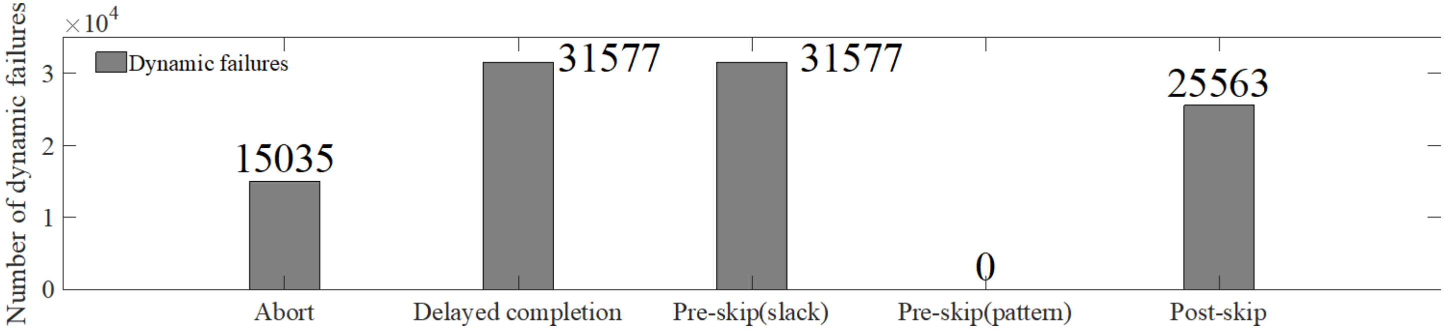


Case study

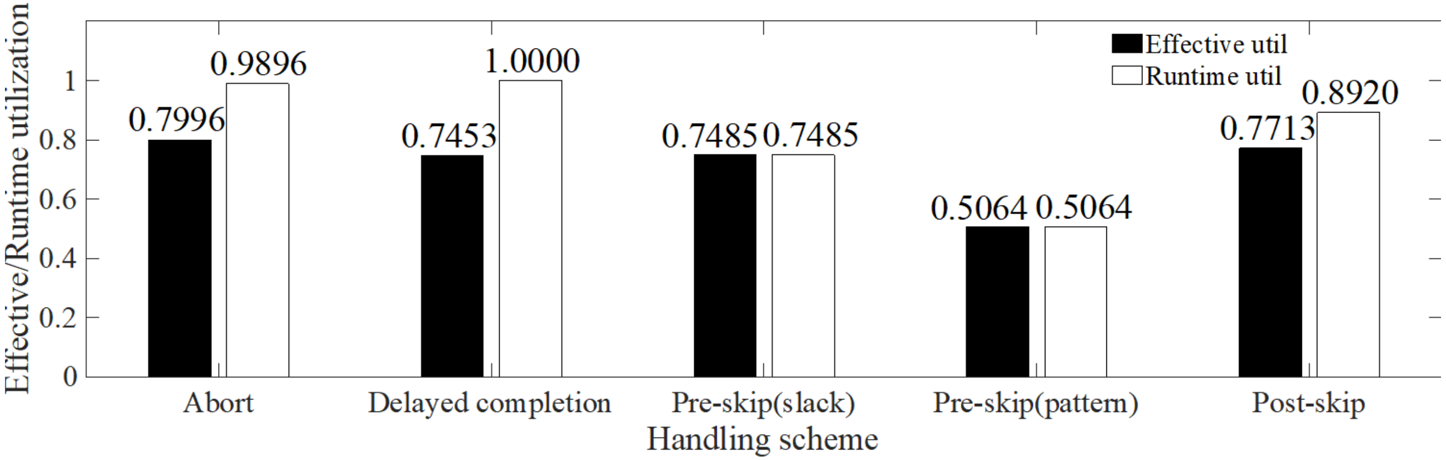
- Select a taskset given in the study[‡]
  - RM-RTO[†] algorithm

Tasks	T [ms]	C [ms]
τ_1	6	1
τ_2	7	4
τ_3	19	5

< Taskset 2 with skip parameter[*] of 2 >



< Dynamic failures[♀] >



[†] RM-RTO stands for Rate Monotonic Red Task Only.

[‡] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *RTSS*, 1995

[*] Tolerance of a task to missing deadlines

[♀] A task experiences more than *m* deadline misses in a window of *K* jobs.