# Blinder: Partition-Oblivious Hierarchical Scheduling

**Man-Ki Yoon**, Mengqi Liu, Hao Chen, Jung-Eun Kim, Zhong Shao

Yale University

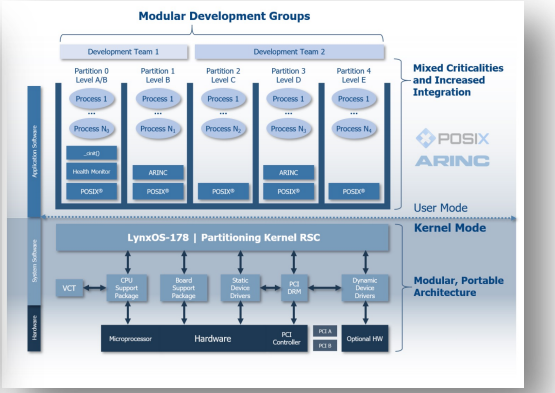
Yale

# Temporal Isolation

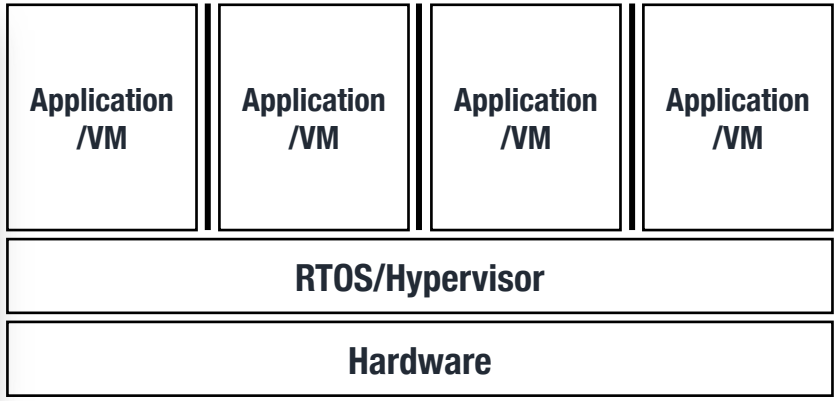
- Key requirement for safety-critical systems
  - Especially when integrating applications with different levels of criticality (e.g., DO-178B, ASIL)
- Time partitioning
  - Isolates potential misuse of CPU resource
  - Enables modular reasoning about temporal behavior

Source: Blackberry QNX

Source: Wind River

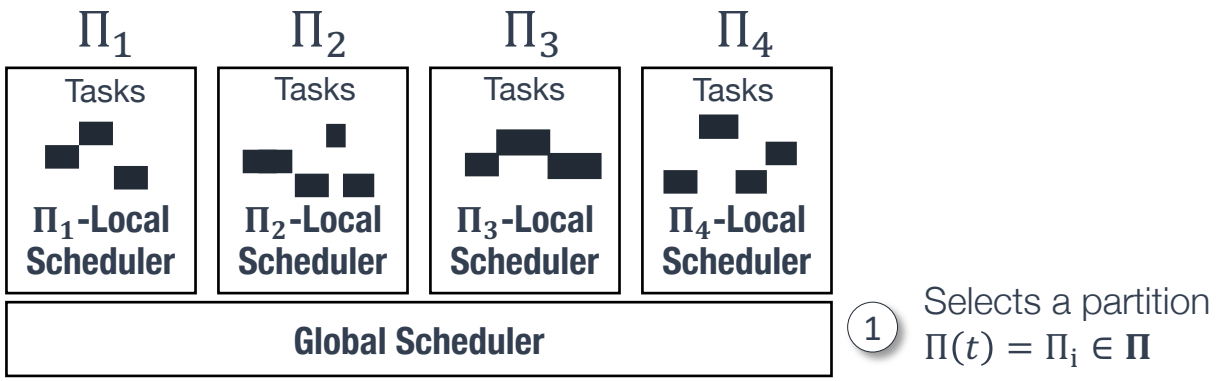



Source: Lynx



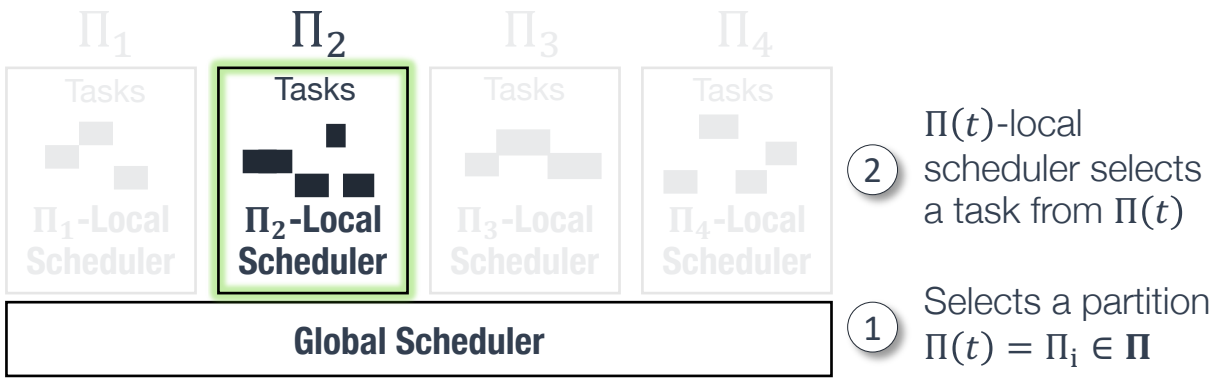
# Real-Time Hierarchical Scheduling

- Two-level scheduling: global and local
  - Provides each application (=partition) with the illusion of exclusive CPU resource



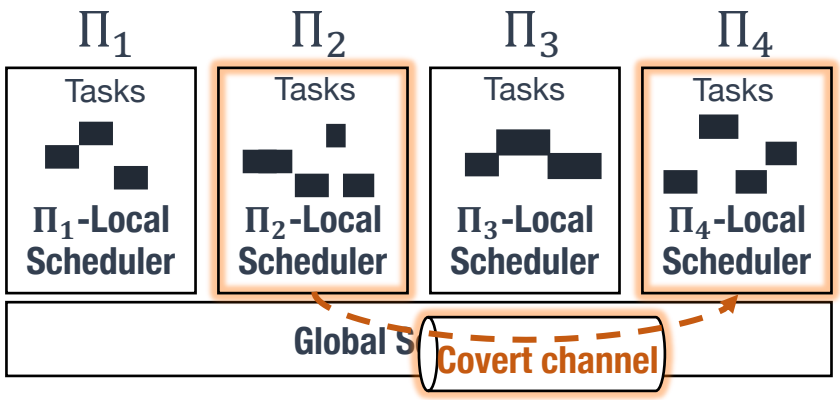
# Real-Time Hierarchical Scheduling

- Two-level scheduling: global and local
  - Provides each application (=partition) with the illusion of exclusive CPU resource



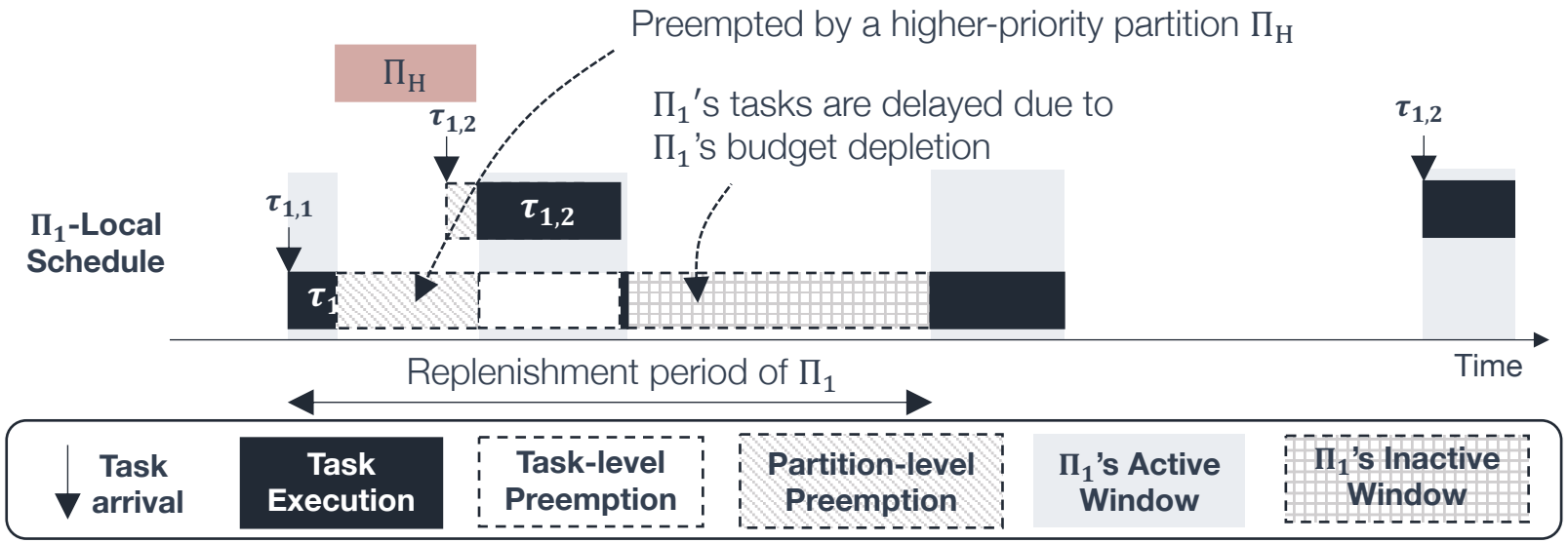
# What This Paper is About

- Temporal isolation from a security perspective
  - Algorithmic covert timing channel through hierarchical scheduling
  - Scheduling approach to making partitions oblivious to other's varying behavior

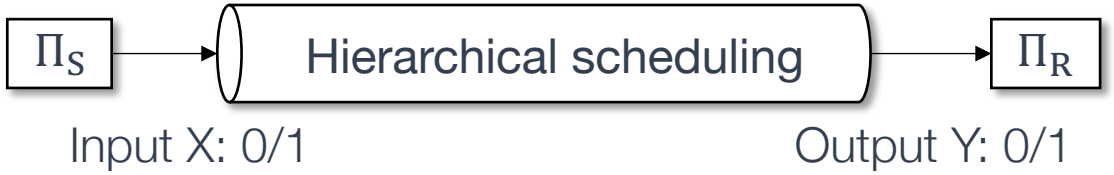


# Preliminary: Non-static Time Partitioning

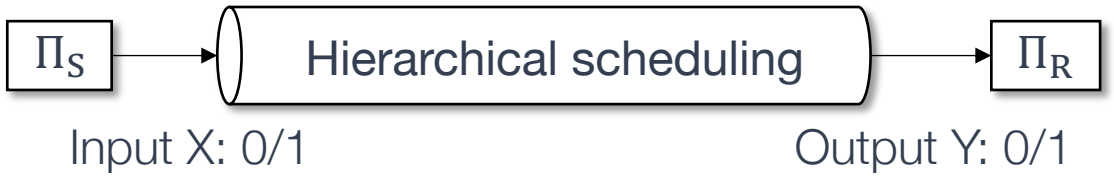
- Priority-based partition scheduling
  - Real-time server algorithms (e.g., sporadic server, deferrable server)
- Budget and replenishment period
- Better CPU utilization and responsiveness than static partitioning



# Algorithmic Covert Timing Channel in Hierarchical Scheduling

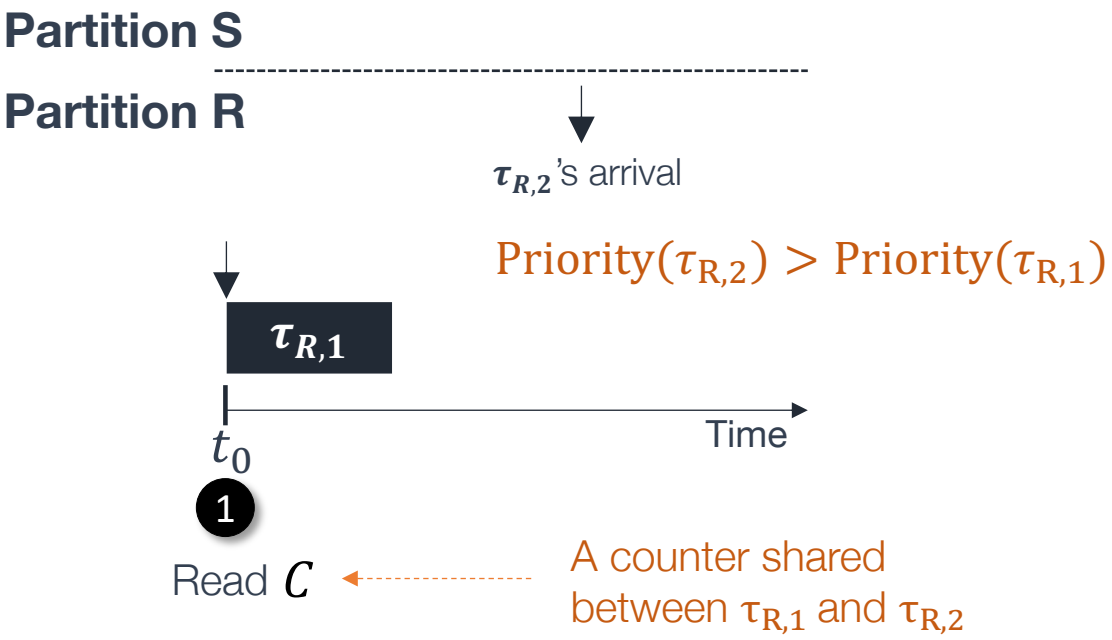


# Algorithmic Covert Timing Channel in Hierarchical Scheduling




**(a) $\Pi_S$ sends bit 0**

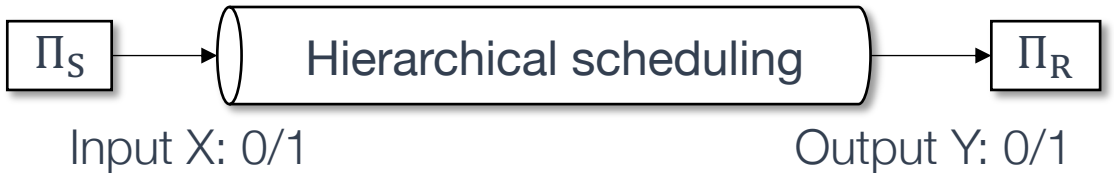
$\text{Priority}(\Pi_S) > \text{Priority}(\Pi_R)$



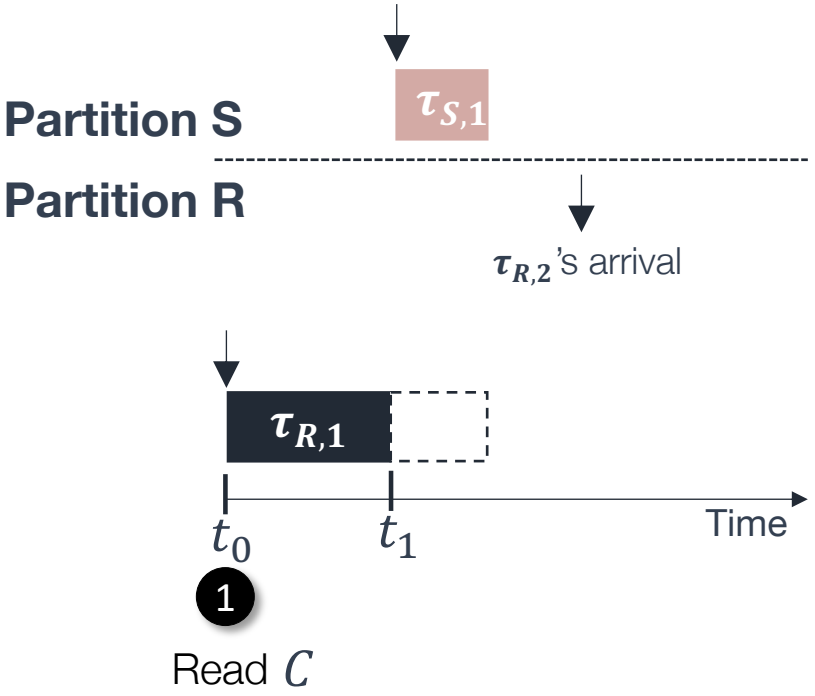

```
int receiver1_job(void) {
  prev_c = shared_c;
  n_loops = 6000000;

  for (i=0; i<n_loops; i++)
    asm("nop");



}
```

1

# Algorithmic Covert Timing Channel in Hierarchical Scheduling




**(a) $\Pi_S$ sends bit 0**

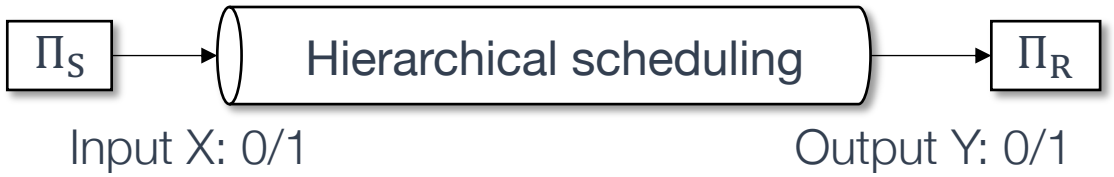



```
void sender_job(int bit) {
  if (bit==1)
    n_loops = 10000000;
  else
    n_loops =  2000000;

  for (i=0; i<n_loops; i++)
    asm("nop");
}
```

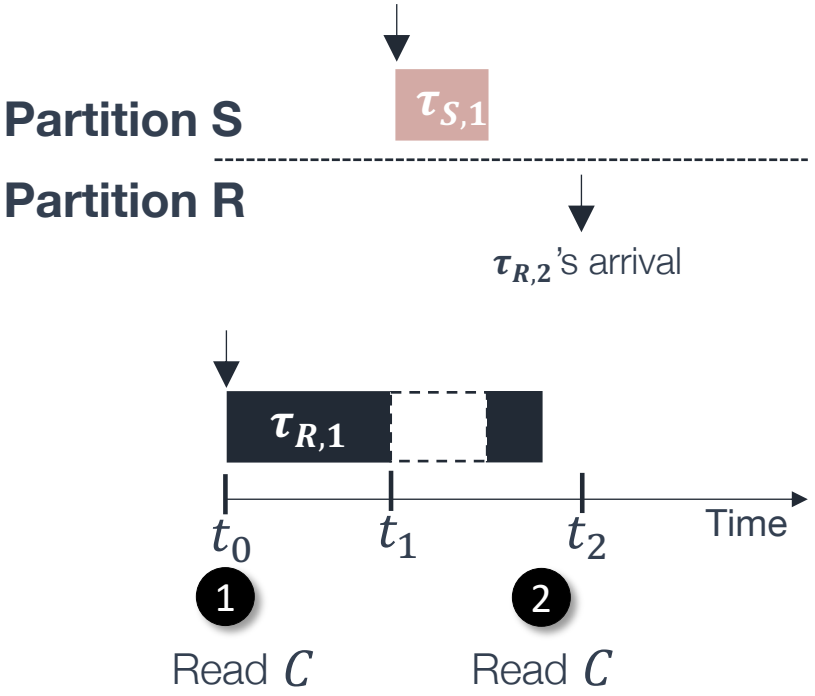
```
int receiver1_job(void) {
  prev_c = shared_c;
  n_loops = 6000000;

  for (i=0; i<n_loops; i++)
    asm("nop");


}
```

1

# Algorithmic Covert Timing Channel in Hierarchical Scheduling



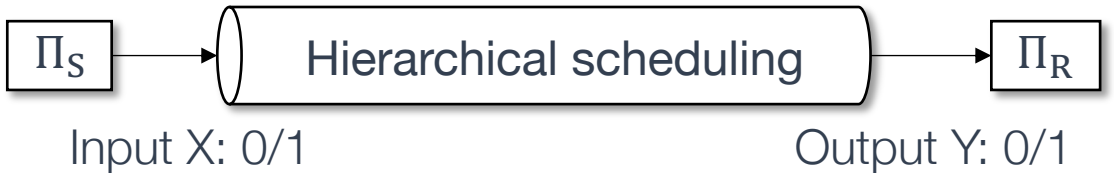

**(a) $\Pi_S$ sends bit 0**



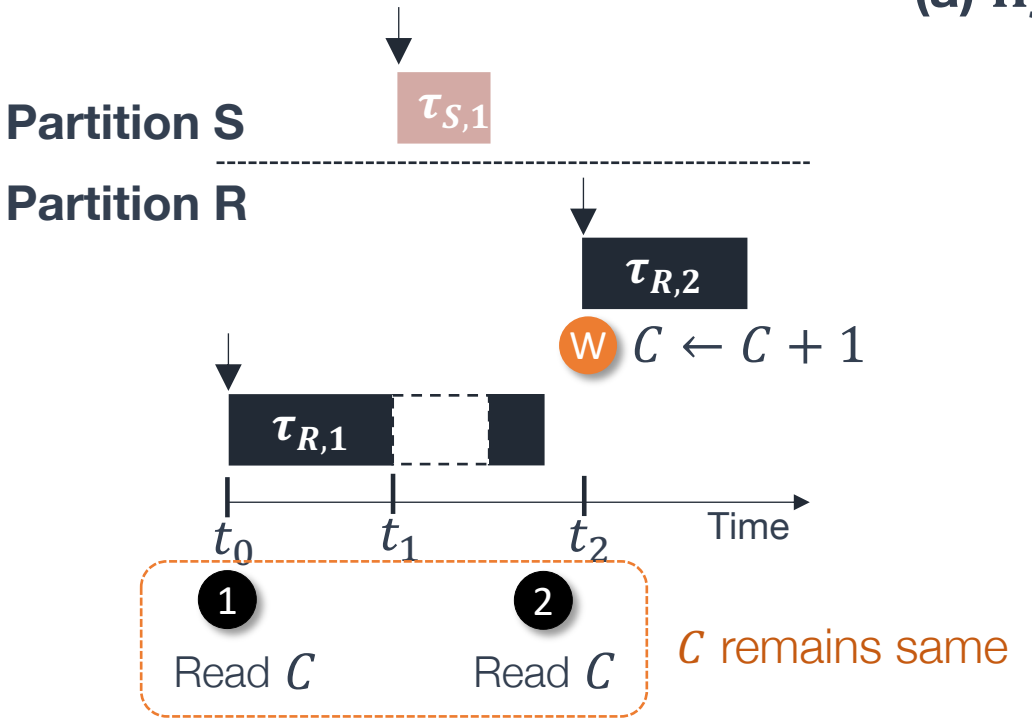

```
void sender_job(int bit) {
  if (bit==1)
    n_loops = 10000000;
  else
    n_loops =  2000000;

  for (i=0; i<n_loops; i++)
    asm("nop");
}
```

```
int receiver1_job(void) {
  prev_c = shared_c;          1
  n_loops = 6000000;

  for (i=0; i<n_loops; i++)
    asm("nop");

  curr_c = shared_c;          2
  return curr_c - prev_c;
}
```

# Algorithmic Covert Timing Channel in Hierarchical Scheduling




**(a) $\Pi_S$ sends bit 0**




```
void sender_job(int bit) {
  if (bit==1)
    n_loops = 10000000;
  else
    n_loops =  2000000;

  for (i=0; i<n_loops; i++)
    asm("nop");
}
```

```
int receiver1_job(void) {
  prev_c = shared_c;
  n_loops = 6000000;

  for (i=0; i<n_loops; i++)
    asm("nop");

  curr_c = shared_c;
  return curr_c - prev_c;
}
```

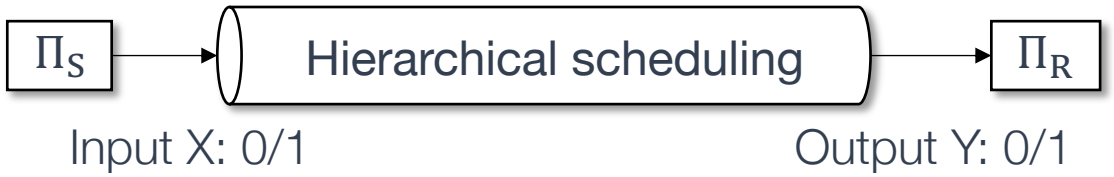
1

2

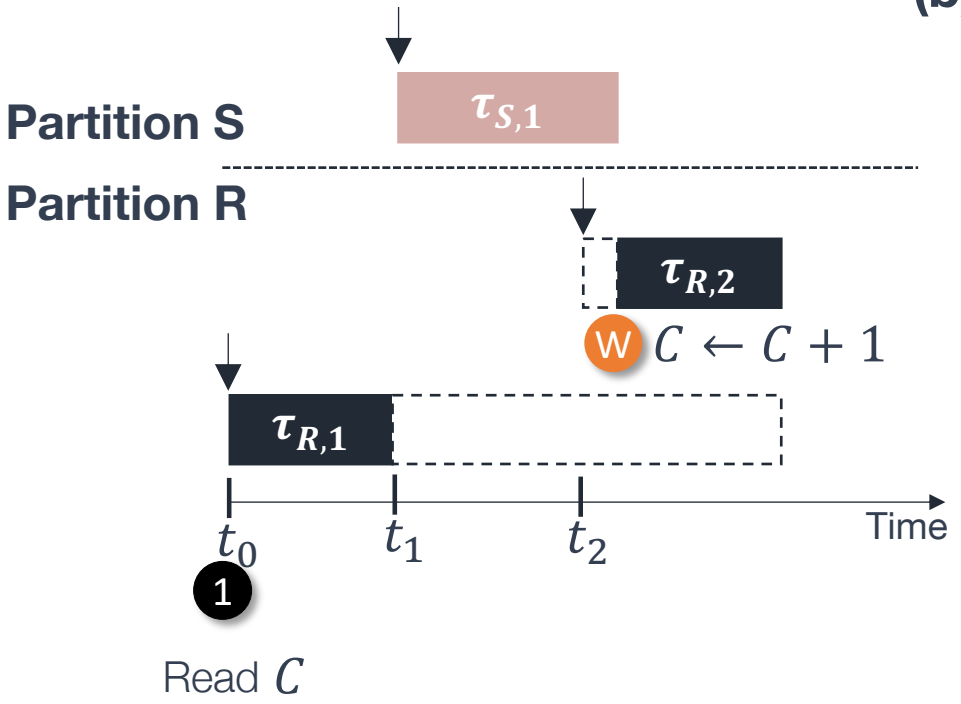
```
void receiver2_job(void) {
    shared_c++;
}
```

W

# Algorithmic Covert Timing Channel in Hierarchical Scheduling




**(b) $\Pi_S$ sends bit 1**



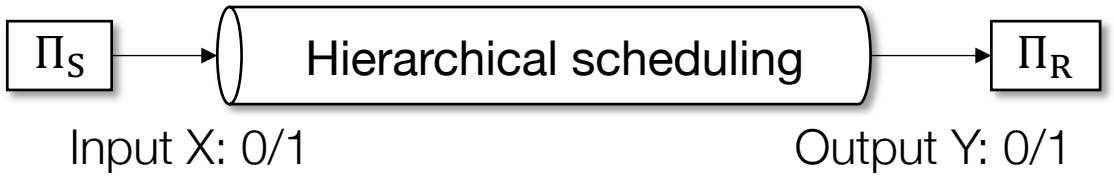

```
void sender_job(int bit) {
  if (bit==1)
    n_loops = 10000000;
  else
    n_loops =  2000000;

  for (i=0; i<n_loops; i++)
    asm("nop");
}
```

```
int receiver1_job(void) {
  prev_c = shared_c;
  n_loops = 6000000;

  for (i=0; i<n_loops; i++)
    asm("nop");



}


void receiver2_job(void) {
    shared_c++;
}
```

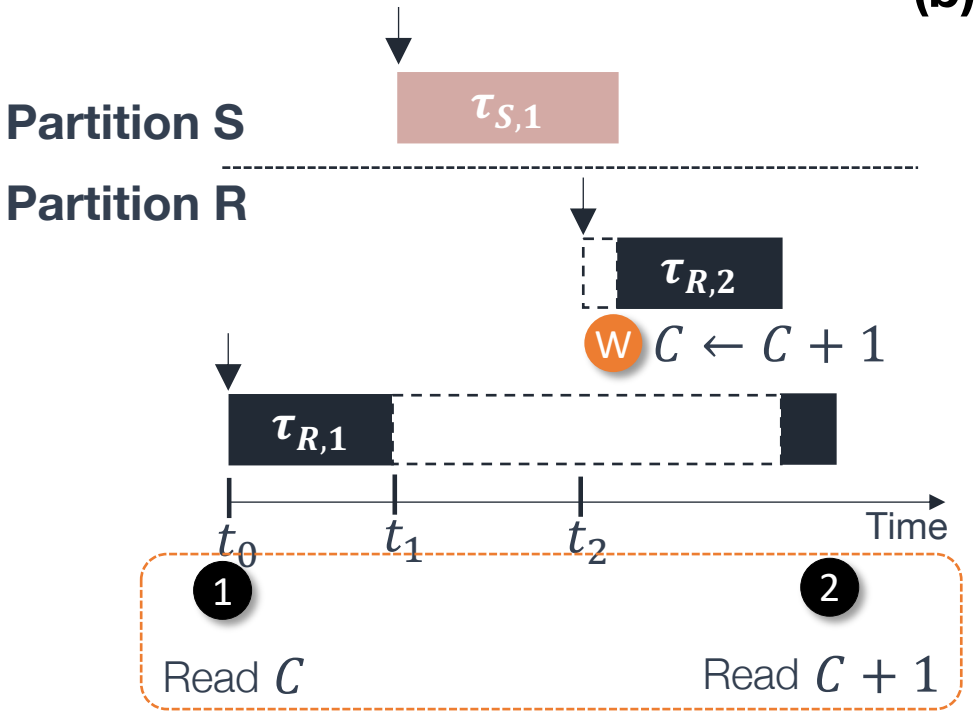
1

W

# Algorithmic Covert Timing Channel in Hierarchical Scheduling




**(b) $\Pi_S$ sends bit 1**



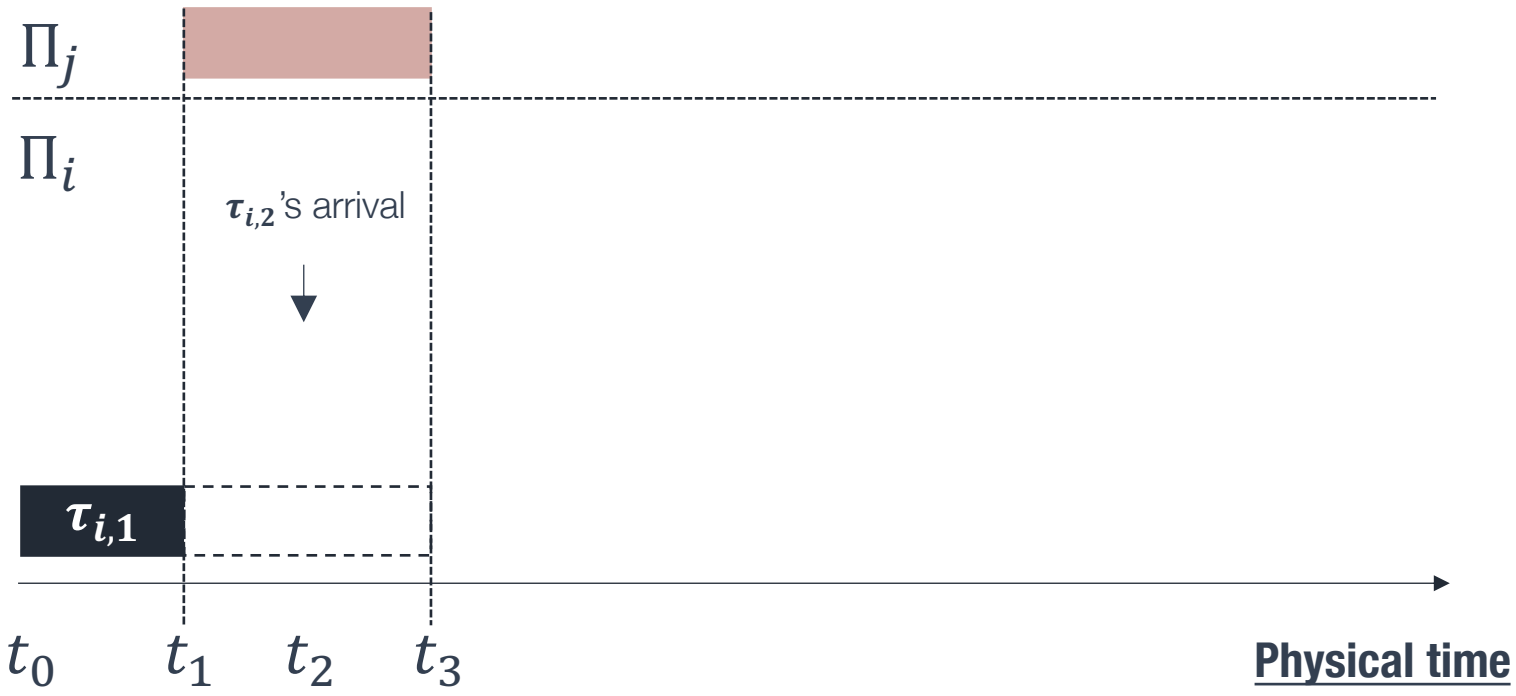

$C$ has incremented by 1

```
void sender_job(int bit) {
  if (bit==1)
    n_loops = 10000000;
  else
    n_loops =  2000000;

  for (i=0; i<n_loops; i++)
    asm("nop");
}
```

```
int receiver1_job(void) {
  prev_c = shared_c;          1
  n_loops = 6000000;

  for (i=0; i<n_loops; i++)
    asm("nop");

  curr_c = shared_c;          2
  return curr_c - prev_c;
}


void receiver2_job(void) {
    shared_c++;               W
}
```

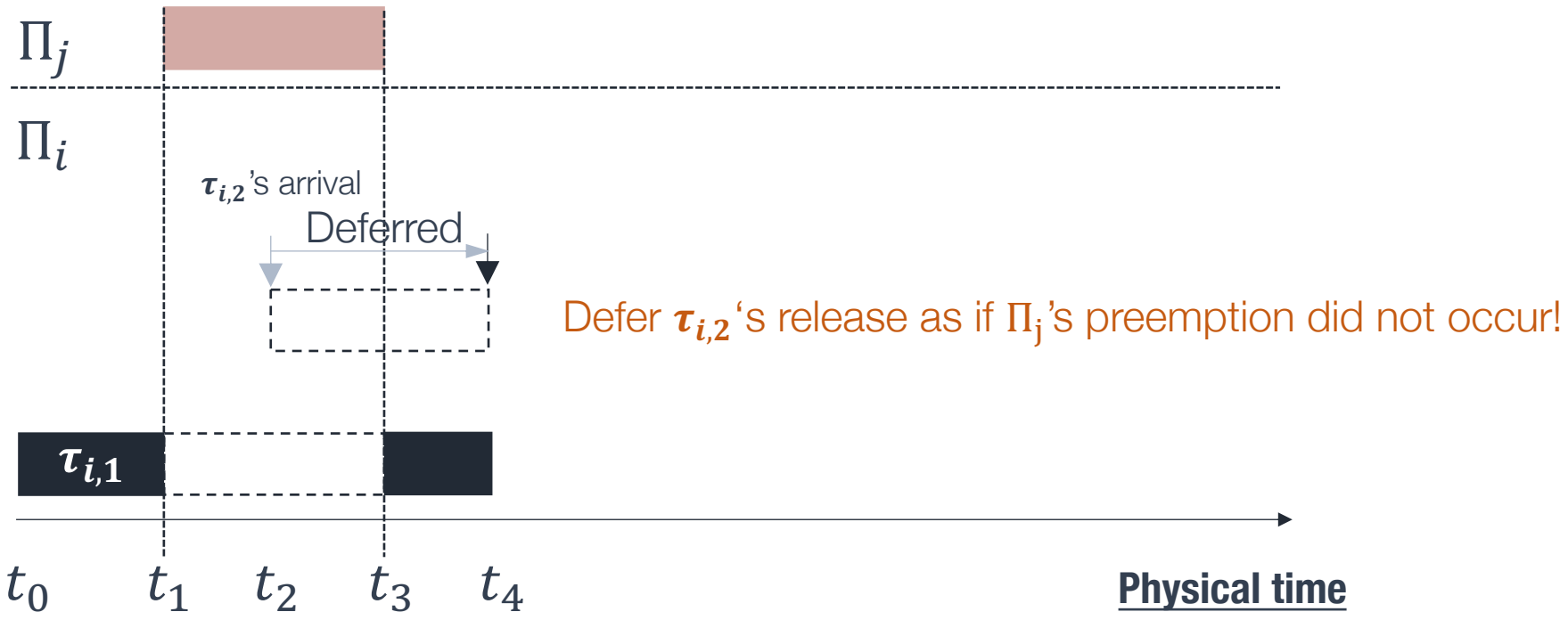
# Blinder: Partition-Oblivious Hierarchical Scheduling

- Idea: making partition-local schedule deterministic no matter how other partitions behave



# Blinder: Partition-Oblivious Hierarchical Scheduling

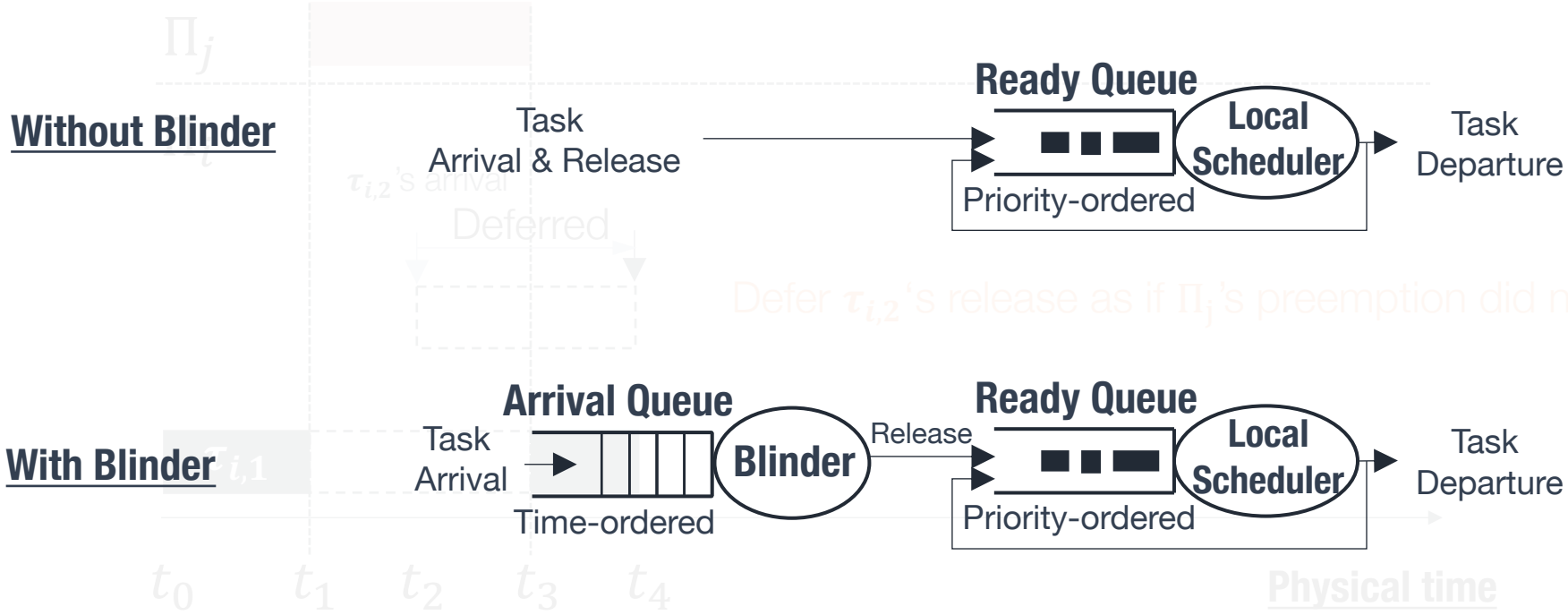
- Idea: making partition-local schedule deterministic no matter how other partitions behave




Defer $\tau_{i,2}$'s release as if $\Pi_j$'s preemption did not occur!

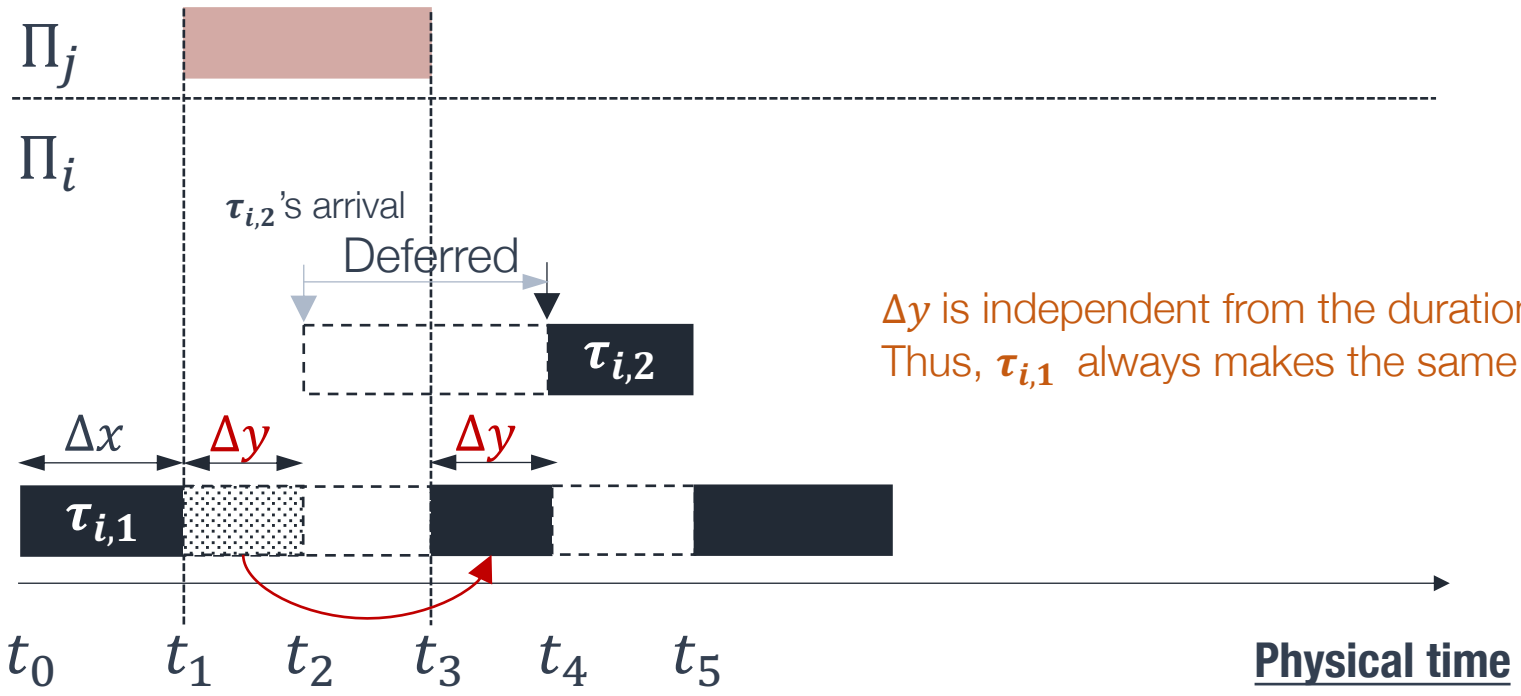
# Blinder: Partition-Oblivious Hierarchical Scheduling

- Idea: making partition-local schedule deterministic no matter how other partitions behave

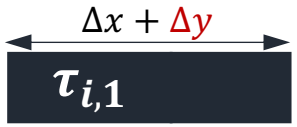


# Blinder: Partition-Oblivious Hierarchical Scheduling

- Idea: making partition-local schedule deterministic no matter how other partitions behave



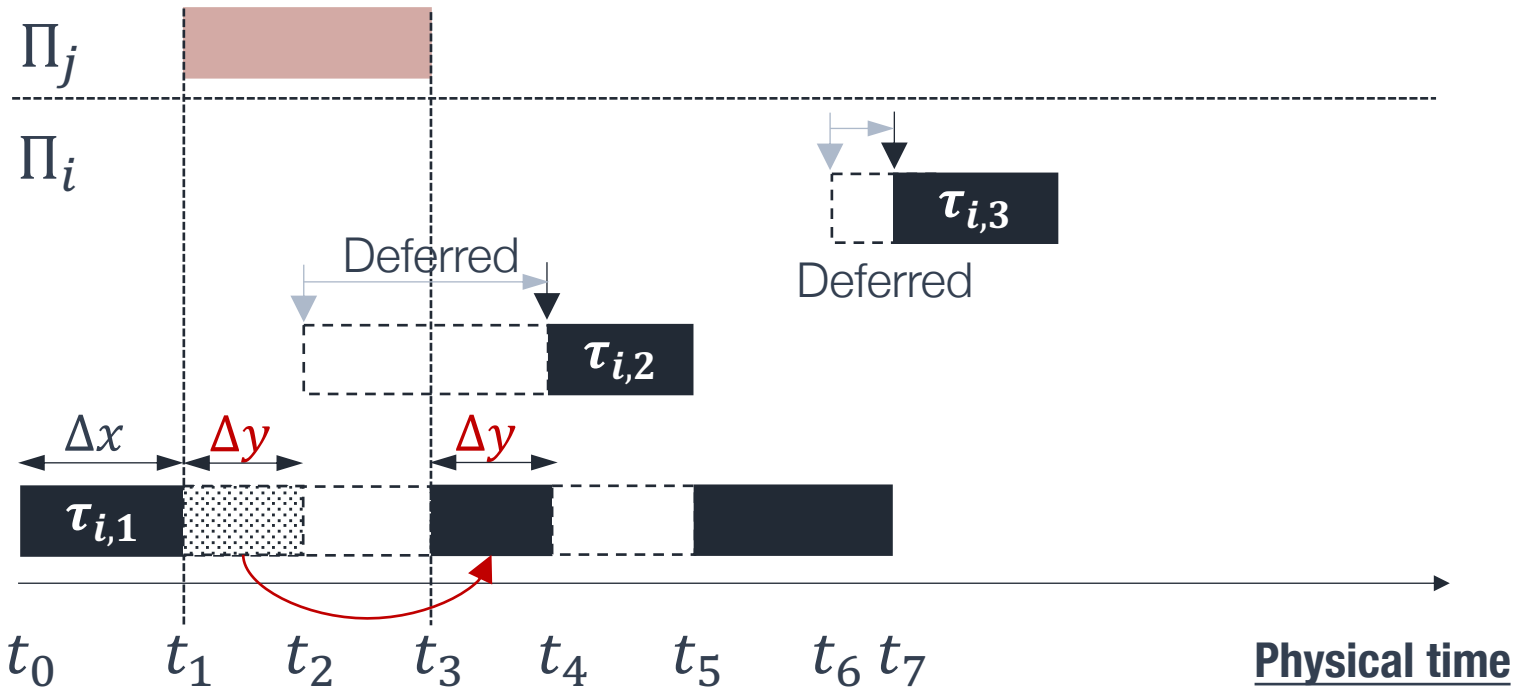

$\Delta y$ is independent from the duration of .
Thus, $\tau_{i,1}$ always makes the same amount of progress.




$\Delta x + \Delta y$ = the amount of time that $\tau_{i,1}$ would have executed for until $\tau_{i,2}$ arrives if the partition-level preemption by $\Pi_j$ did not happen.

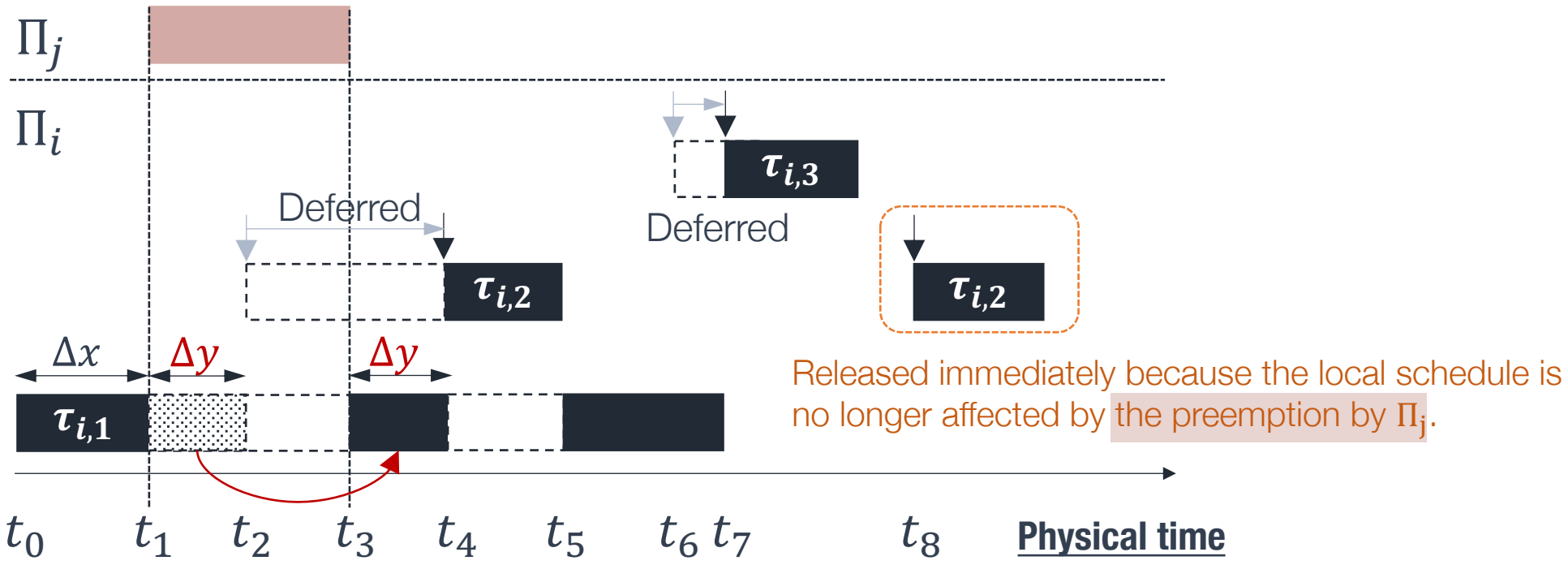
# Blinder: Partition-Oblivious Hierarchical Scheduling

- Idea: making partition-local schedule deterministic no matter how other partitions behave



# Blinder: Partition-Oblivious Hierarchical Scheduling

- Idea: making partition-local schedule deterministic no matter how other partitions behave

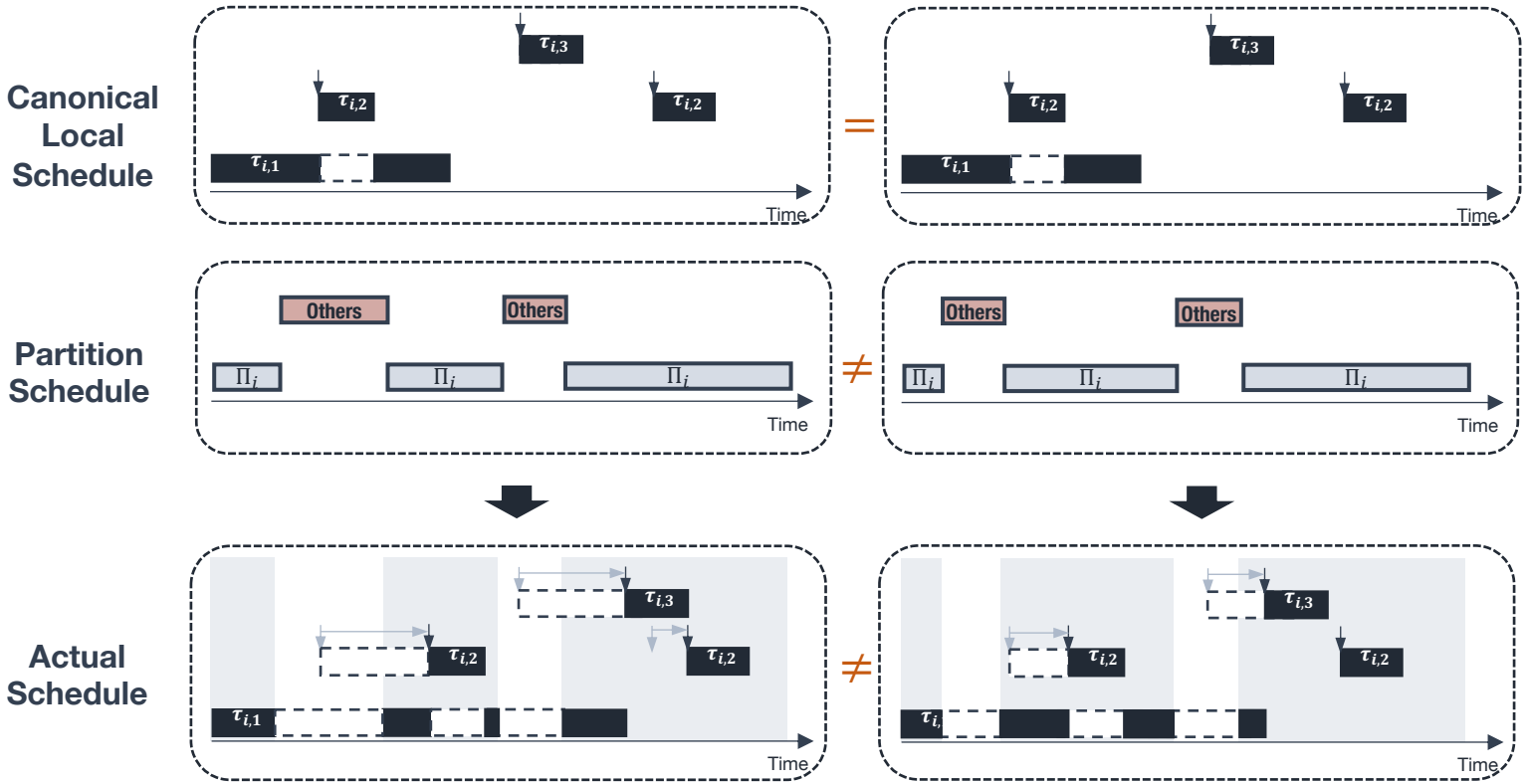


# Blinder: Partition-Oblivious Hierarchical Scheduling

- Idea: making partition-local schedule deterministic no matter how other partitions behave

Guarantees that partition-local schedule always follows the partition's **canonical local schedule**

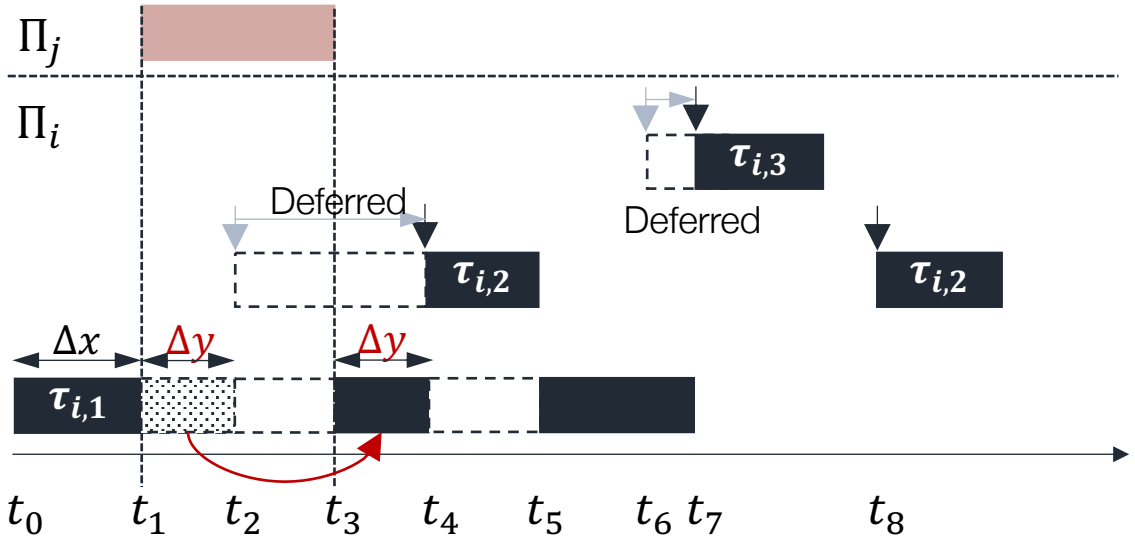
The local schedule when no other partitions run




Canonical schedule being stretched out by higher-priority partitions

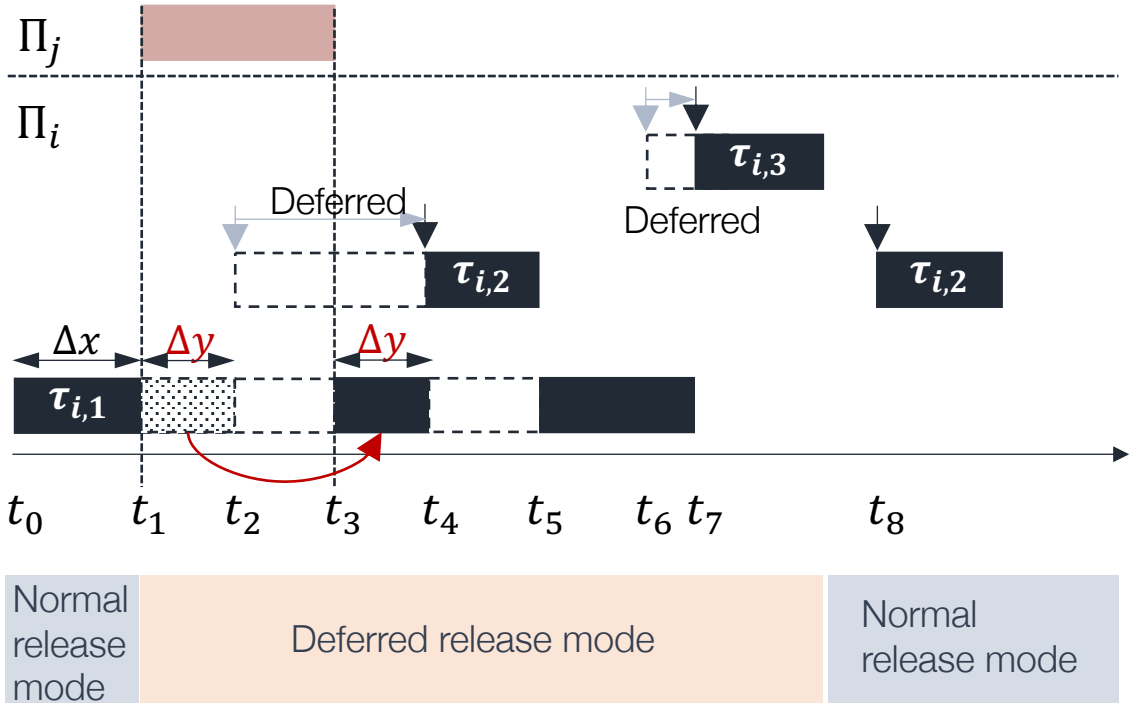
# Lag-based Task Release

- Canonical local schedule cannot be statically constructed
  - Tasks may arrive at arbitrary times and have variable execution times
  - Also, these affect partition budget depletion and replenishment
- Hence, Blinder constructs canonical local schedule on the fly



# Lag-based Task Release

- Canonical local schedule cannot be statically constructed
  - Tasks may arrive at arbitrary times and have variable execution times
  - Also, these affect partition budget depletion and replenishment
- Hence, Blinder constructs canonical local schedule on the fly




## Task release mode

Normal mode

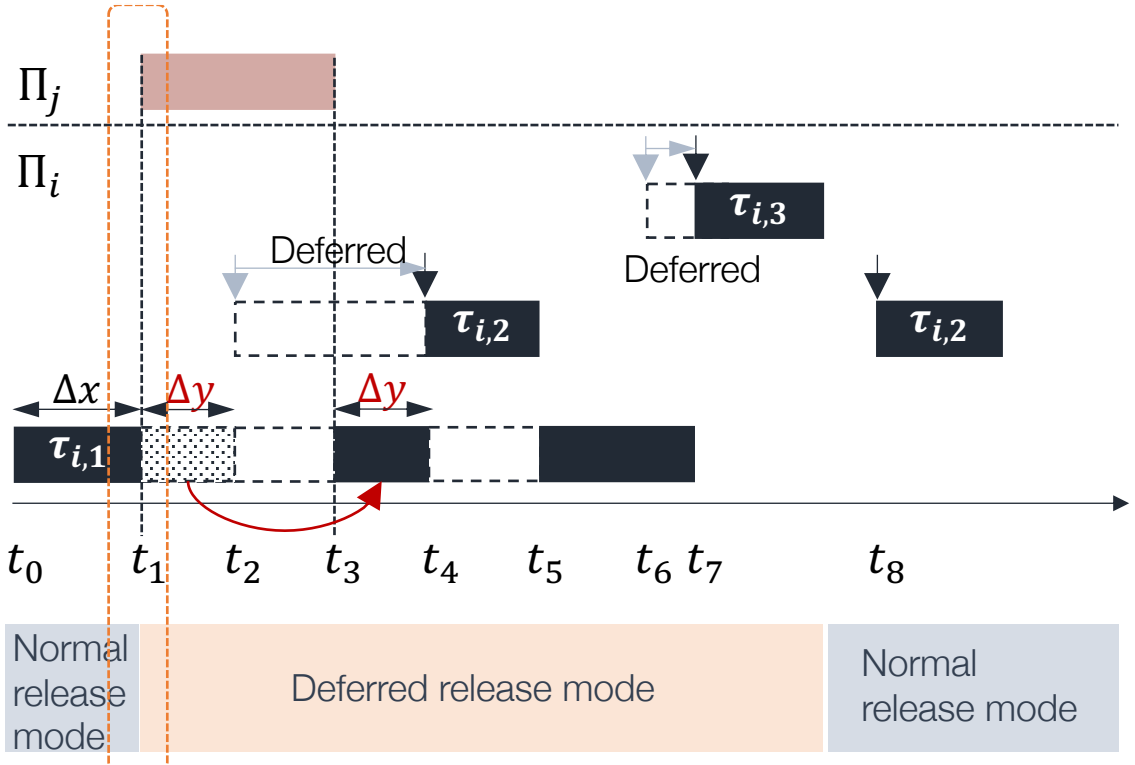

Deferred mode

# Lag-based Task Release

- Canonical local schedule cannot be statically constructed
  - Tasks may arrive at arbitrary times and have variable execution times
  - Also, these affect partition budget depletion and replenishment
- Hence, Blinder constructs canonical local schedule on the fly

**Normal mode → Deferred mode**

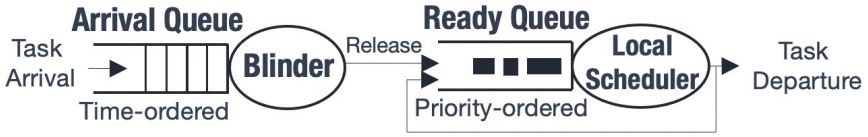
When the local schedule starts deviating from the canonical one.




## Task release mode

### Normal mode


### Deferred mode



# Lag-based Task Release

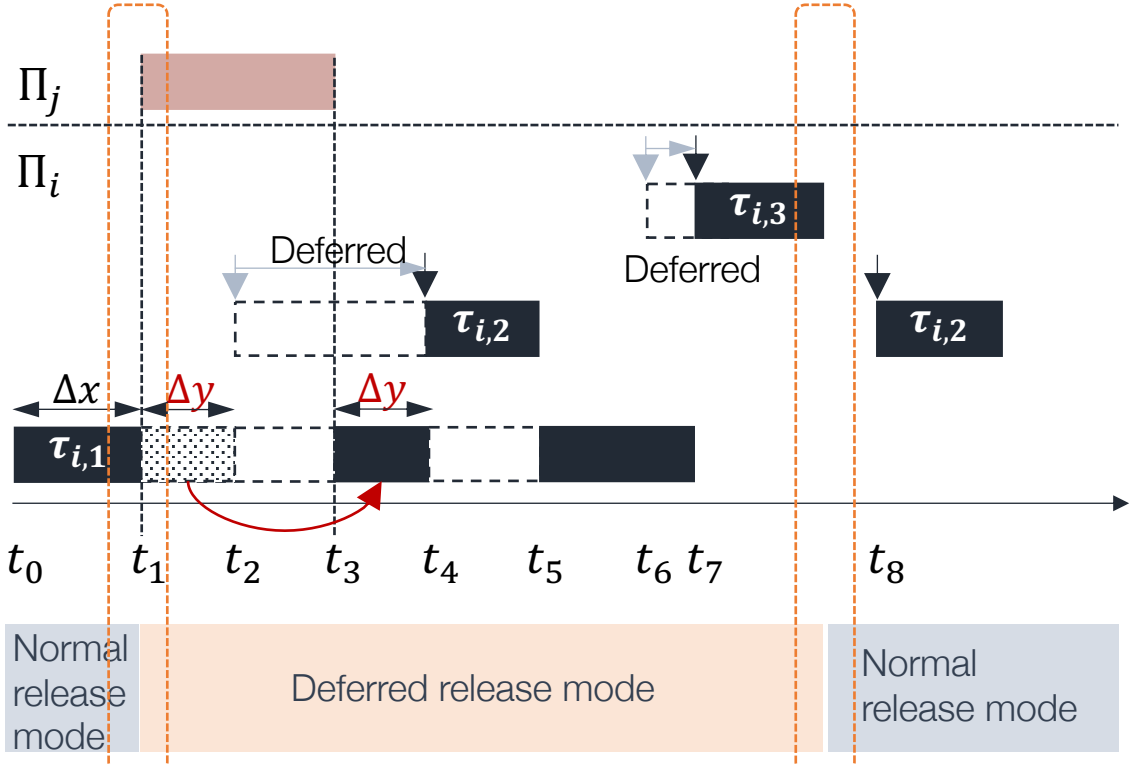
- Canonical local schedule cannot be statically constructed
  - Tasks may arrive at arbitrary times and have variable execution times
  - Also, these affect partition budget depletion and replenishment
  - Hence, Blinder constructs the canonical local schedule on the fly

**Normal mode → Deferred mode**

When the local schedule starts deviating from the canonical one.

**Deferred mode → Normal mode**

When the local schedule is synchronized with the canonical one.



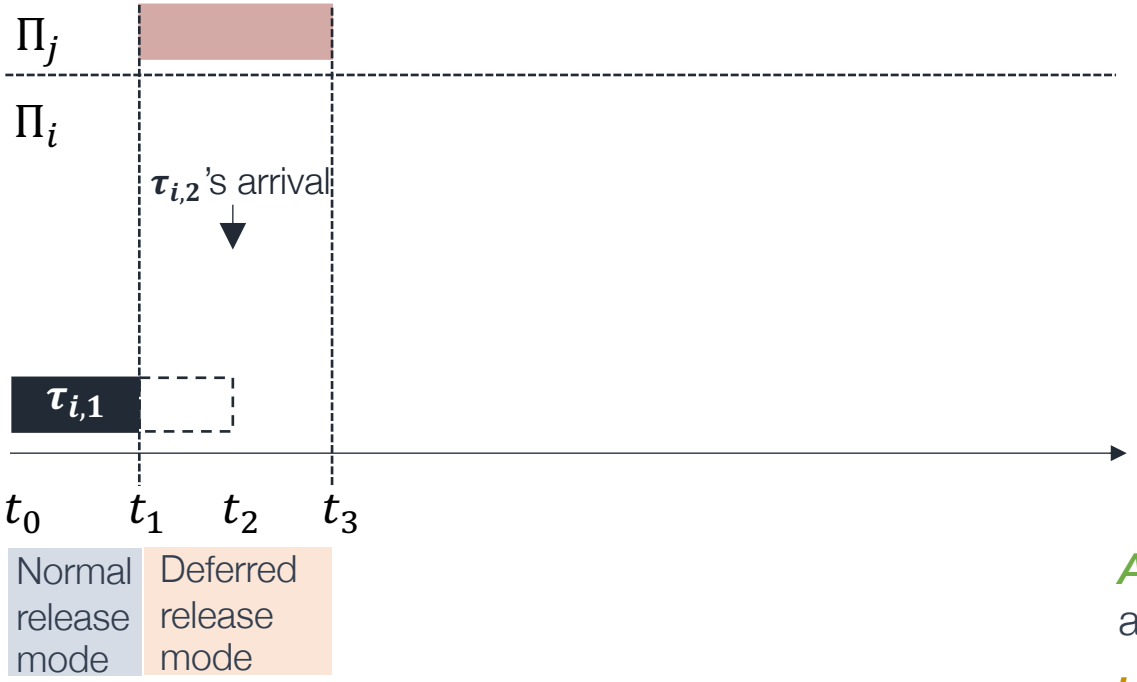

## Task release mode

### Normal mode


### Deferred mode

# Lag-based Task Release

- When to release task?
  - When 'lag' becomes 0





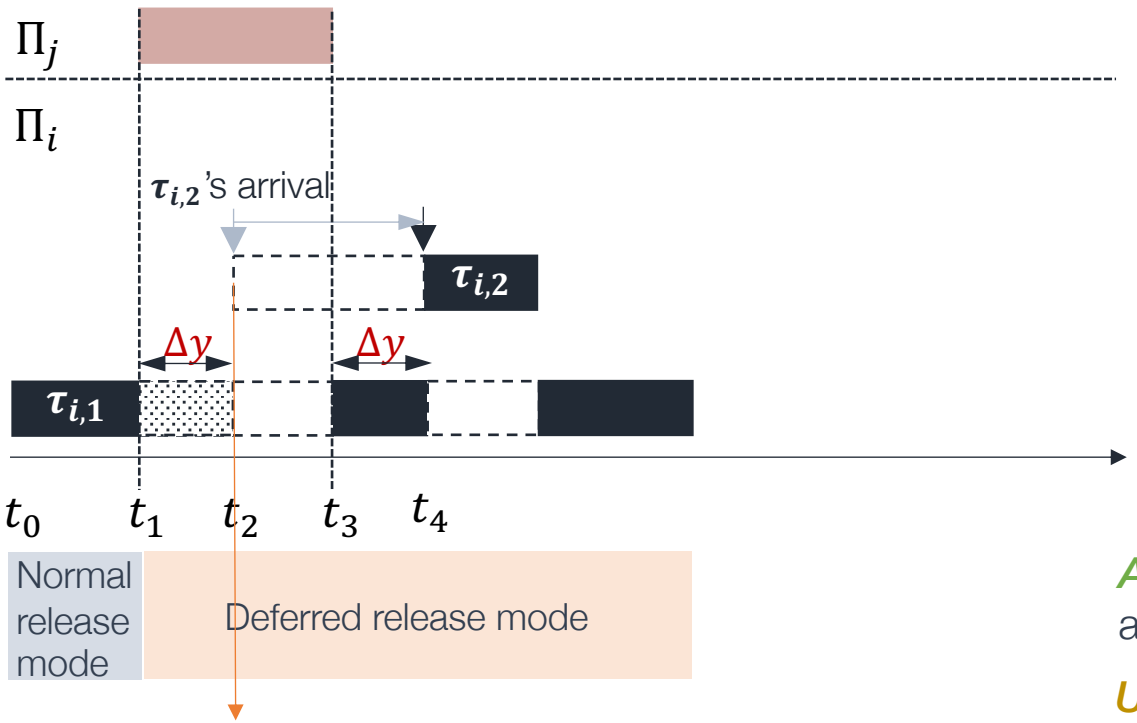
***Available time***: the maximum amount of time that would have been available since entering into the deferred release mode

***Used time***: the amount of time that has actually been used since entering into the deferred release mode

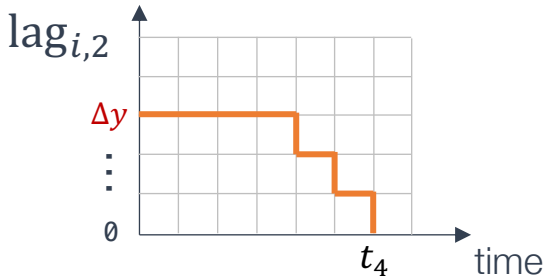
***Lag***: available time – used time (i.e., the maximum amount of time by which the current local schedule lags behind the canonical schedule)

# Lag-based Task Release

- When to release task?
  - When 'lag' becomes 0




available time = $\Delta y$
used time = 0
lag = $\Delta y$




Lag is reduced as long as partition runs.

Hence, the release of $\tau_{i,2}$ is deferred until $\Pi_i$ will have executed for $\Delta y$.

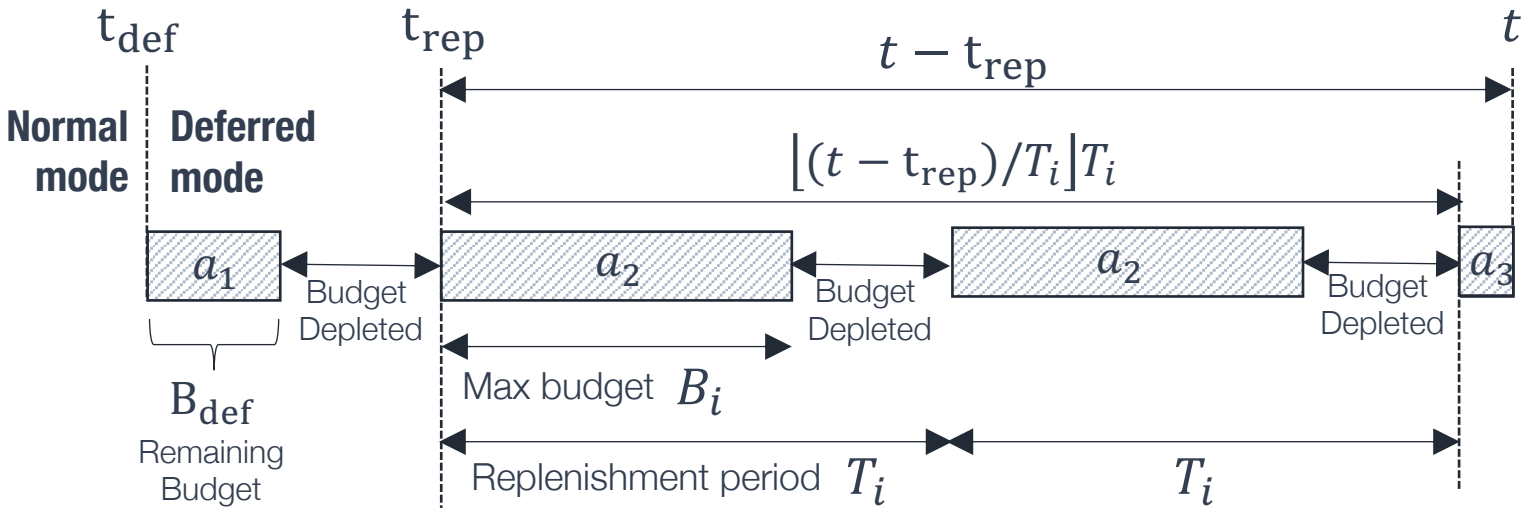
***Available time***: the maximum amount of time that would have been available since entering into the deferred release mode

***Used time***: the amount of time that has actually been used since entering into the deferred release mode

***Lag***: available time – used time (i.e., the maximum amount of time by which the current local schedule lags behind the canonical schedule)

# Available Time Under Budget Constraint

- Available time can be limited by budget constraint
  - Assuming all budgets are used up as soon as they become available




**Remaining budget until the next replenishment or t**

$$a_1 = \min\left[B_{def}, (t - t_{def}), (t_{rep} - t_{def})\right]$$

**Full-budget replenishments**

$$a_2 = \left\lfloor \frac{t - t_{rep}}{T_i} \right\rfloor B_i$$

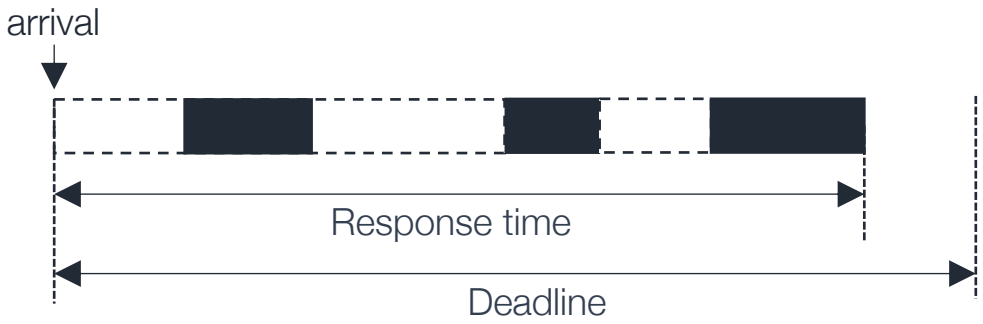
**Last replenishment**

$$a_3 = \min\left[B_i, (t - t_{rep}) - \left\lfloor \frac{t - t_{rep}}{T_i} \right\rfloor T_i\right]$$

$$\boxed{available_i(t) = a_1 + a_2 + a_3}$$

# Real-time Schedulability

- Will real-time tasks meet their deadlines even with Blinder?



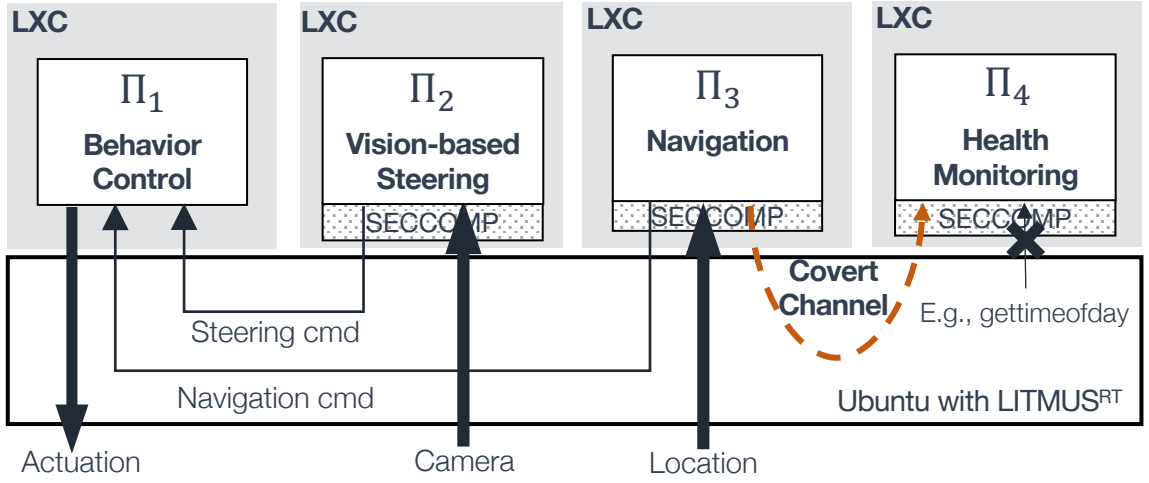

Schedulable iff worst-case response time <= deadline

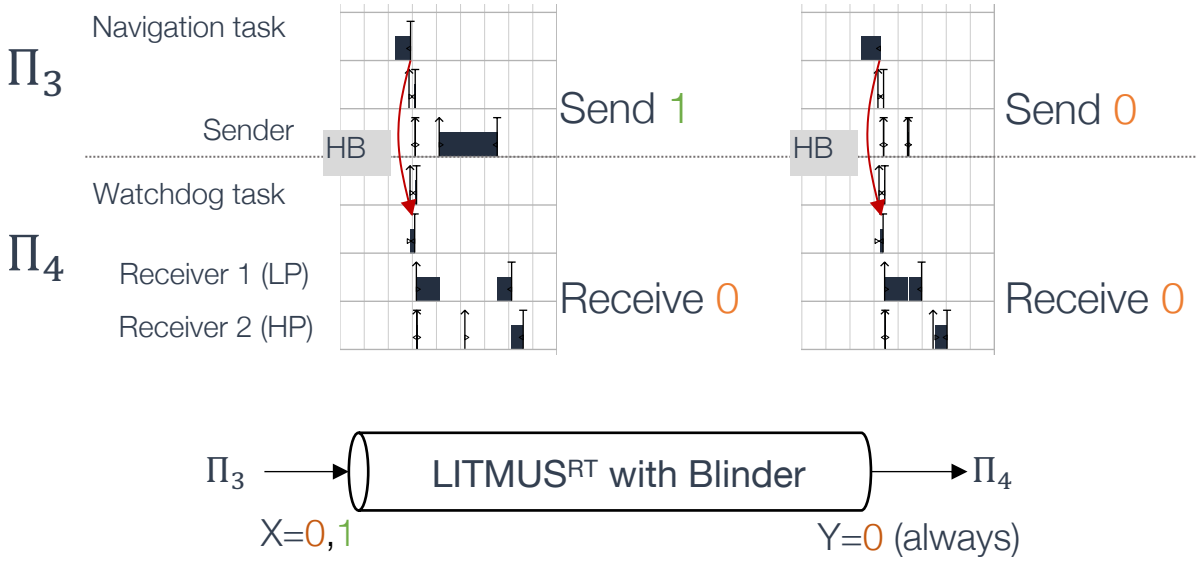
- If partition is schedulable*, Blinder does not increase the worst-case response time of tasks
  - That is, tasks were schedulable without Blinder → they are still schedulable with Blinder
  - See the proof for Theorem 2 in the paper

* Partition $\Pi_i$ is said to be schedulable if it is guaranteed to execute for $B_i$ over every replenishment period $T_i$.

# Implementation and Use Case

- 1/10th-scale self-driving car
  - OS: LITMUS$^{RT}$
    - Ubuntu 16.04 with kernel version 4.9.30
    - Global: Sporadic-polling server, Partition-local: fixed-priority preemptive scheduling
  - Platform: Intel NUC mini PC
    - Intel Core i5-7260U processor (@ 2.20 GHz) + 8 GB main memory



# Evaluation – Impact on Responsiveness

- Synthetic partition set

High priority → Low priority

(Replenishment period, Budget)

(Min inter-arrival time, Exec time)

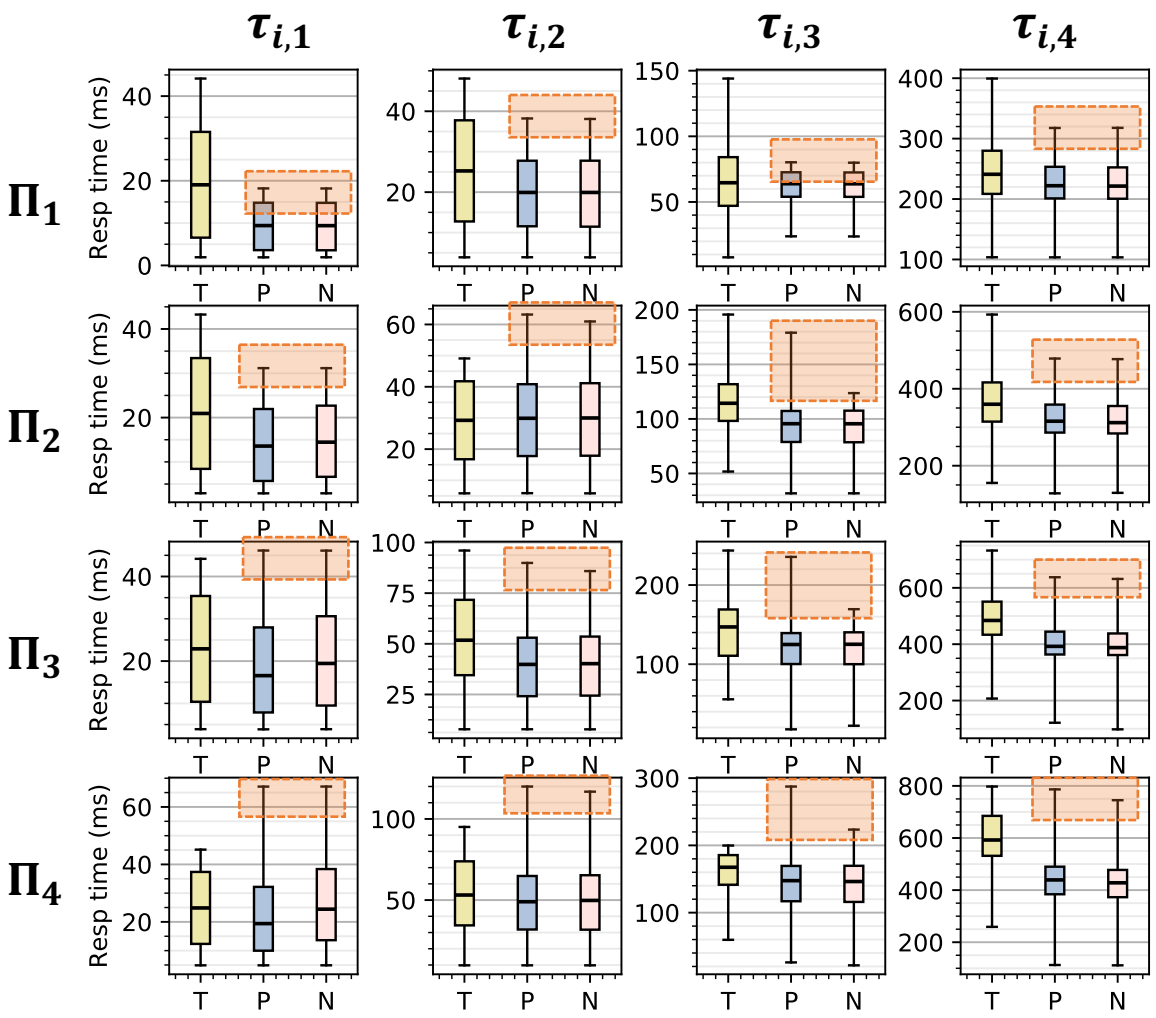
| | $\tau_{i,1}$ | $\tau_{i,2}$ | $\tau_{i,3}$ | $\tau_{i,4}$ |
|---|---|---|---|---|
| $\Pi_1$ $(20, 4\alpha)$ | $(40, 2\beta)$ | $(80, 4\beta)$ | $(160, 8\beta)$ | $(320, 16\beta)$ |
| $\Pi_2$ $(30, 6\alpha)$ | $(60, 3\beta)$ | $(120, 6\beta)$ | $(240, 12\beta)$ | $(480, 24\beta)$ |
| $\Pi_3$ $(40, 8\alpha)$ | $(80, 4\beta)$ | $(160, 8\beta)$ | $(320, 16\beta)$ | $(640, 32\beta)$ |
| $\Pi_4$ $(50, 10\alpha)$ | $(100, 5\beta)$ | $(200, 10\beta)$ | $(400, 20\beta)$ | $(800, 40\beta)$ |

High priority ↓ Low priority

System load=80% when $\alpha = \beta = 1$

- Various rate groups
- Task inter-arrival times are allowed to vary by 20%
- Compare
  - **T**: Static partitioning (TDMA),
  - **P**: Non-static partitioning without Blinder
  - **N**: Non-static partitioning with Blinder
- Collected task response times for 10 hours/config

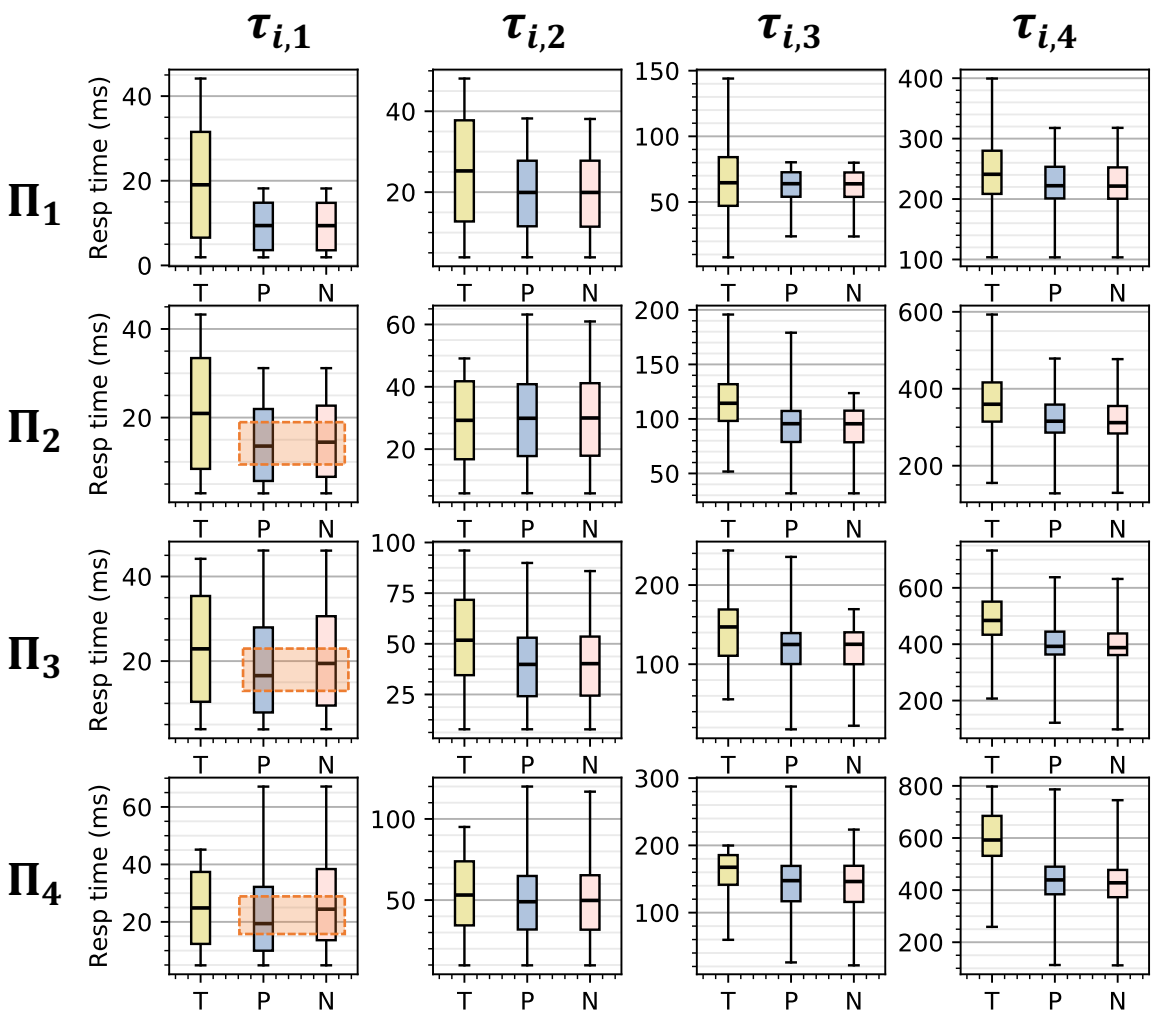
# Evaluation – Impact on Responsiveness




**T**: static, **P**: non-static w/o Blinder, **N**: non-static w/ Blinder

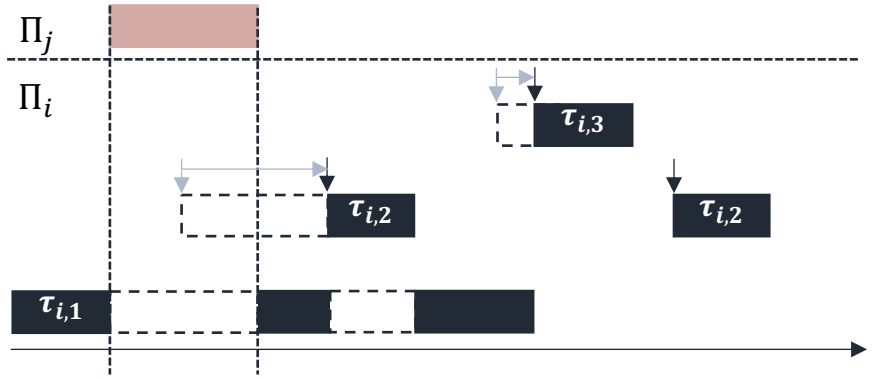
- Blinder does not increase the WCRTs

# Evaluation – Impact on Responsiveness




**T**: static, **P**: non-static w/o Blinder, **N**: non-static w/ Blinder

- Blinder does not increase the WCRTs
- But, Blinder may increase the average-case response times
  - In particular of high-priority tasks
  - Due to the lag-based deferred release




- Impact on low-priority tasks is small
  - Delayed by higher-priority ones anyway

# Summary

- Blinder closes algorithmic covert timing channel over hierarchical scheduling
  - By making partition-local schedules deterministic (i.e., lag-based task release)

- Blinder achieves
  - Flexible CPU resource usage (that non-static partitioning achieves)
  - Partition obliviousness (that static partitioning achieves)

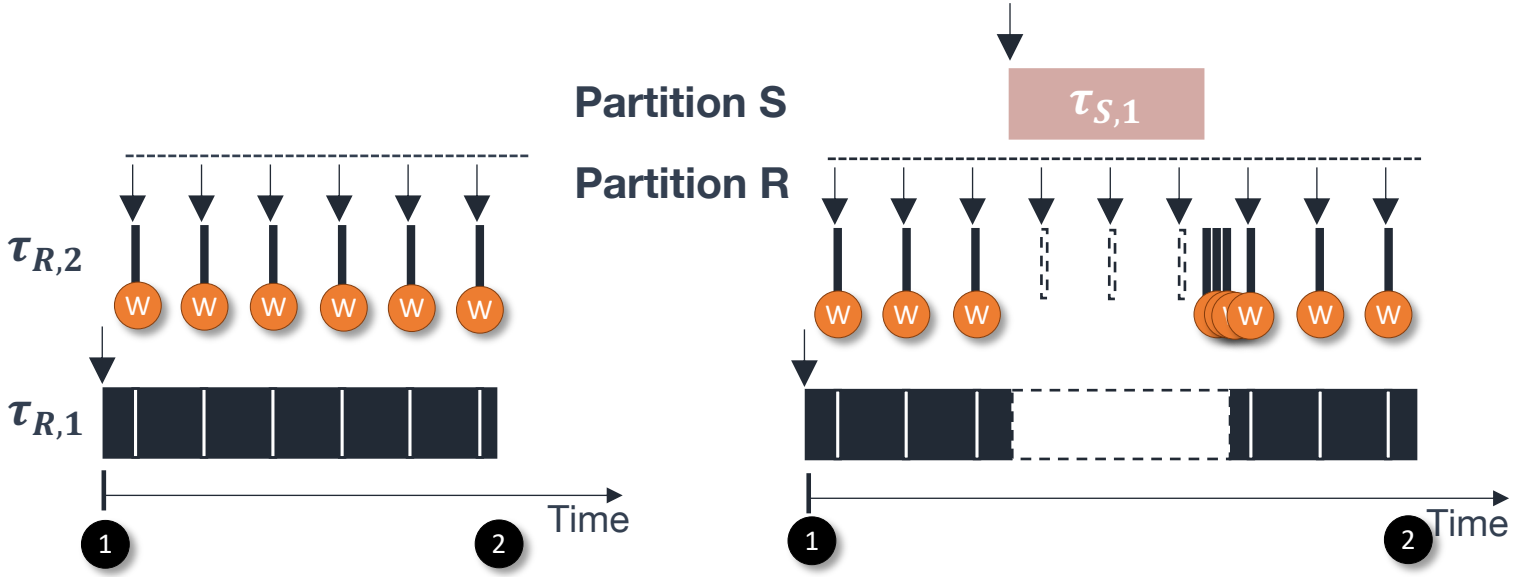
- Blinder is backward-compatible and minimally-intrusive
  - No modification is required to global and local scheduling policies
  - Advantageous to existing safety-critical systems that require high re-certification costs

✉ man-ki.yoon@yale.edu

# Appendix

# Algorithmic Covert Timing Channel in Hierarchical Scheduling

- Multi-bit channel
  - $\tau_{R,2}$ acts as a regular tick counter
  - Message = length of preemption

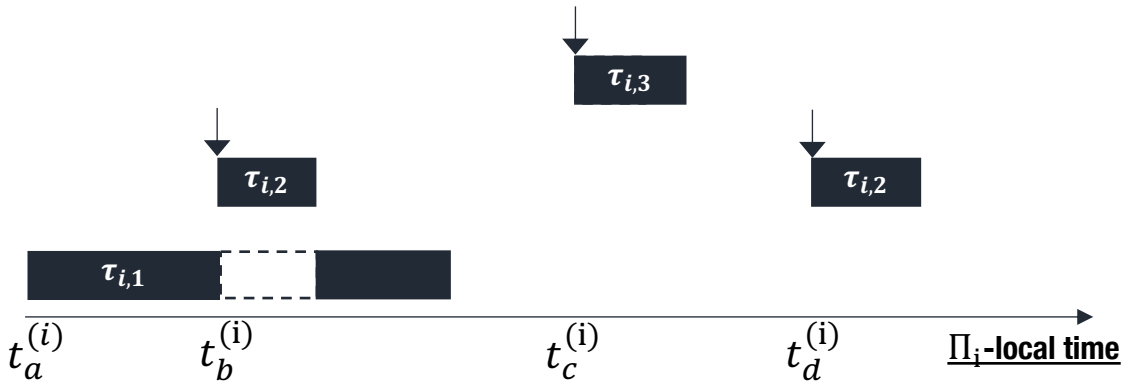



**(a)** $\Delta\text{counter} = 6$ **(b)** $\Delta\text{counter} = 9$

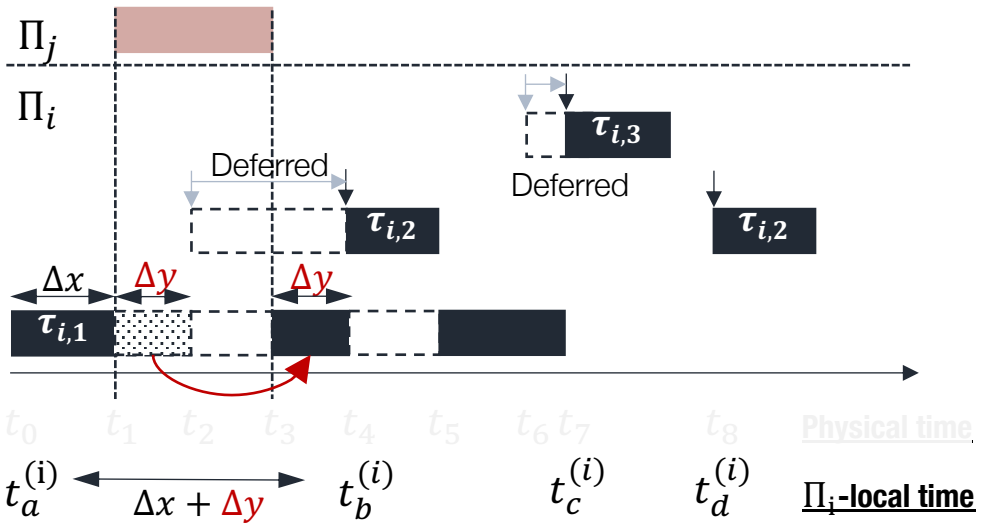
# Blinder: Partition-Oblivious Hierarchical Scheduling

- Idea: making partition-local schedule deterministic no matter how other partitions behave

Guarantees that tasks are released at **deterministic partition-local times**



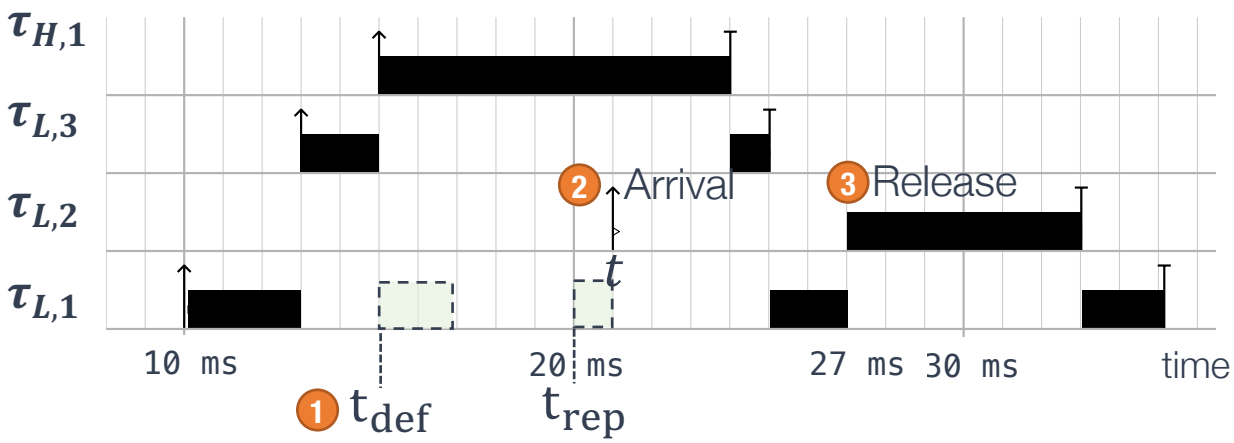

**Canonical schedule**

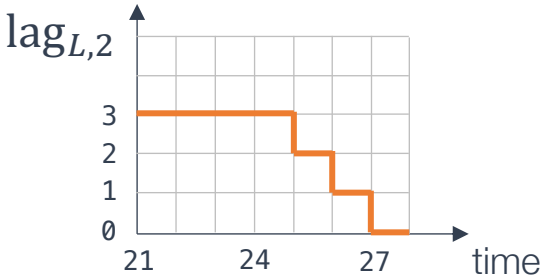



**Actual schedule**

# Example Schedule Trace (1)

- From LITMUS$^{RT}$
- $T_L = 10\ ms, B_L = 7\ ms$



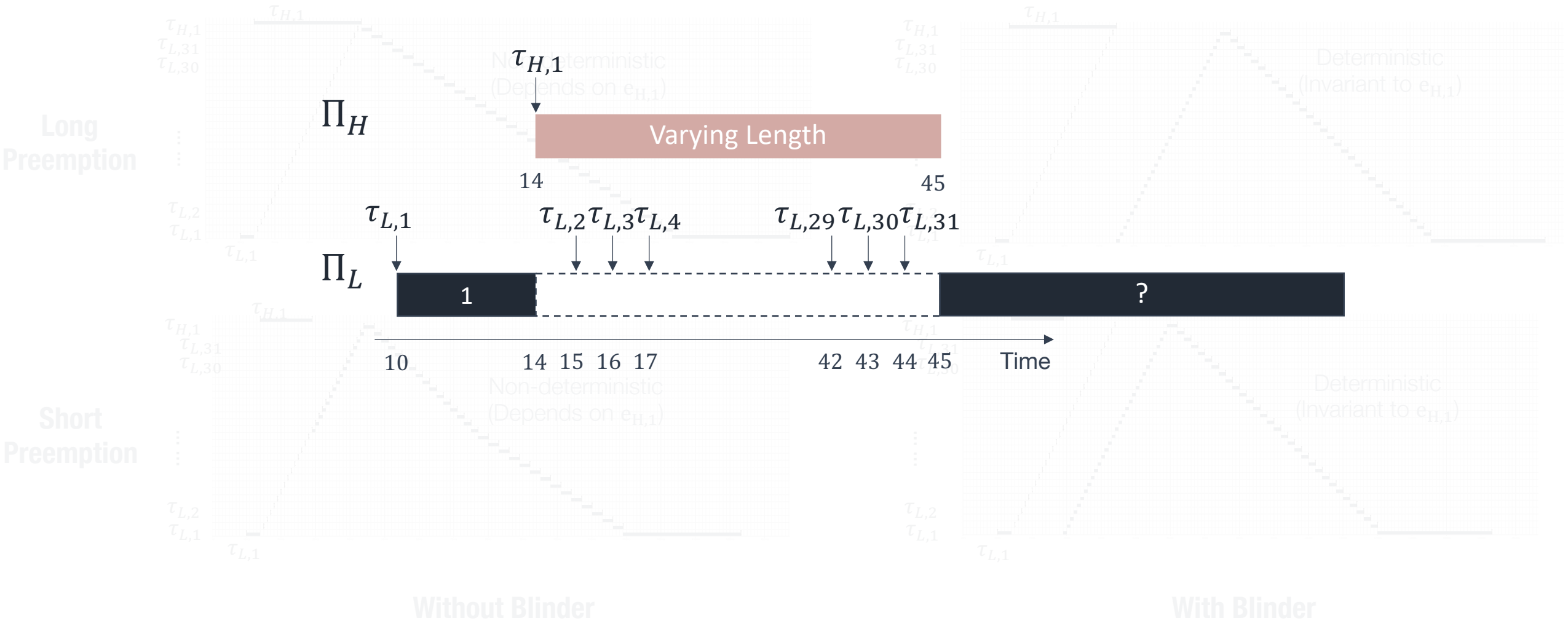




1. Normal mode → Deferred mode
   $B_{def} = 7 - 5 = 2\ ms$

2. $available_L(t = 21) = 2 + 1\ ms$, $used_L = 0\ ms$
   → $lag_{L,2} = 3\ ms$

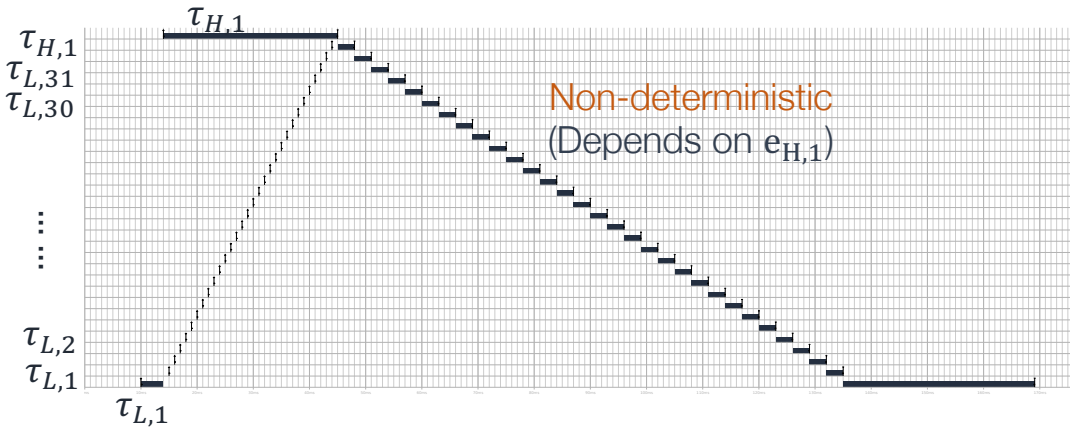
3. $lag_{L,2}$ becomes 0 → $\tau_{L,2}$ is released

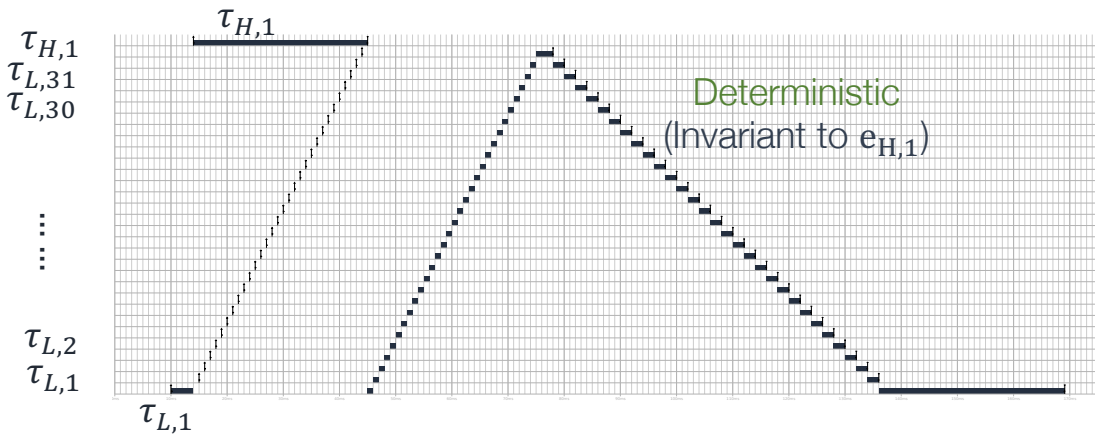
# Example Schedule Trace (2)



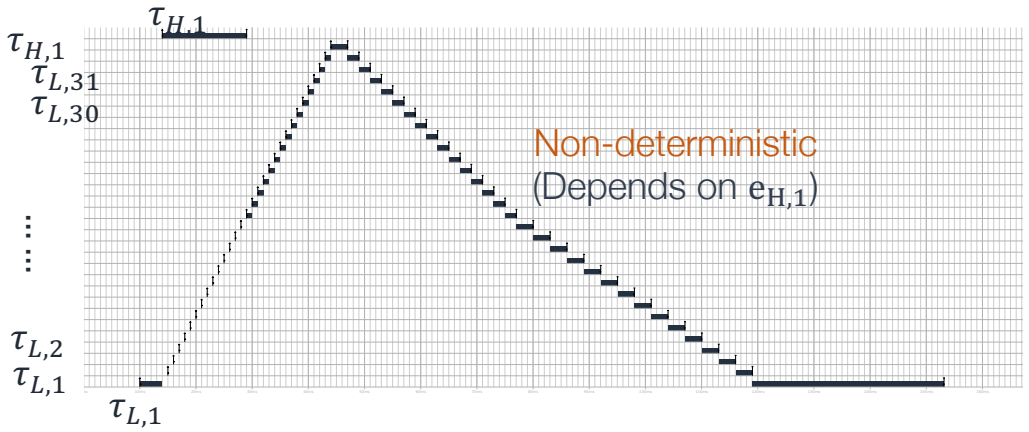
# Example Schedule Trace (2)

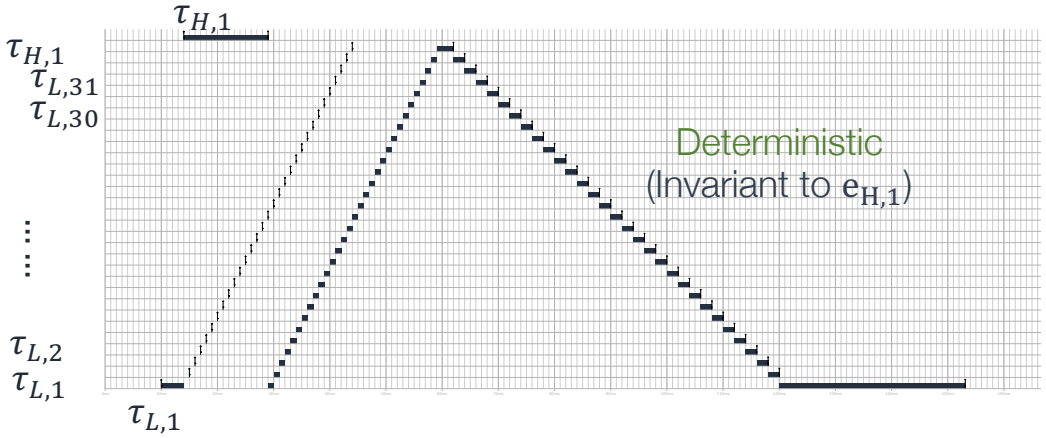
**Long Preemption**



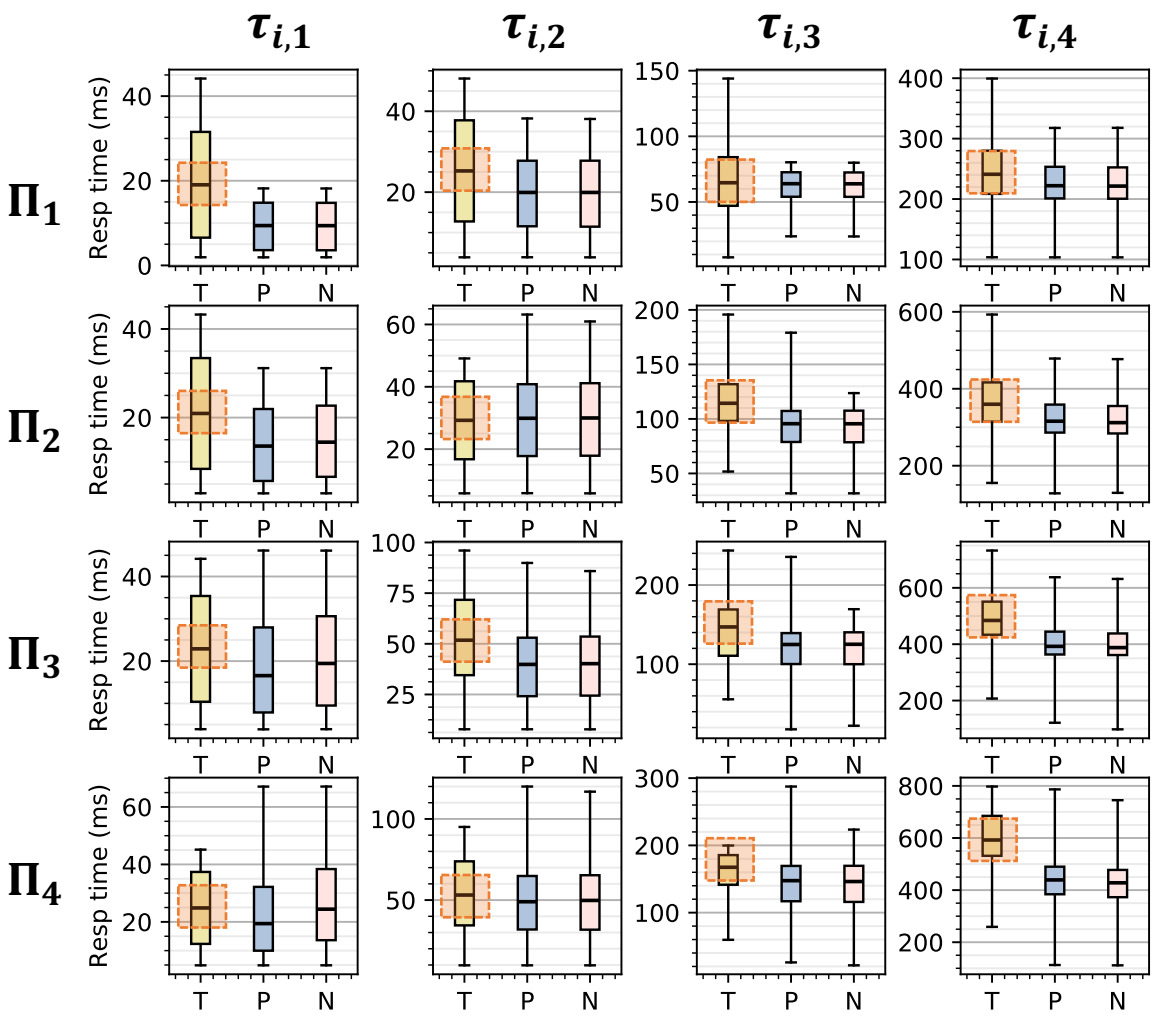

**Short Preemption**




**Without Blinder** **With Blinder**

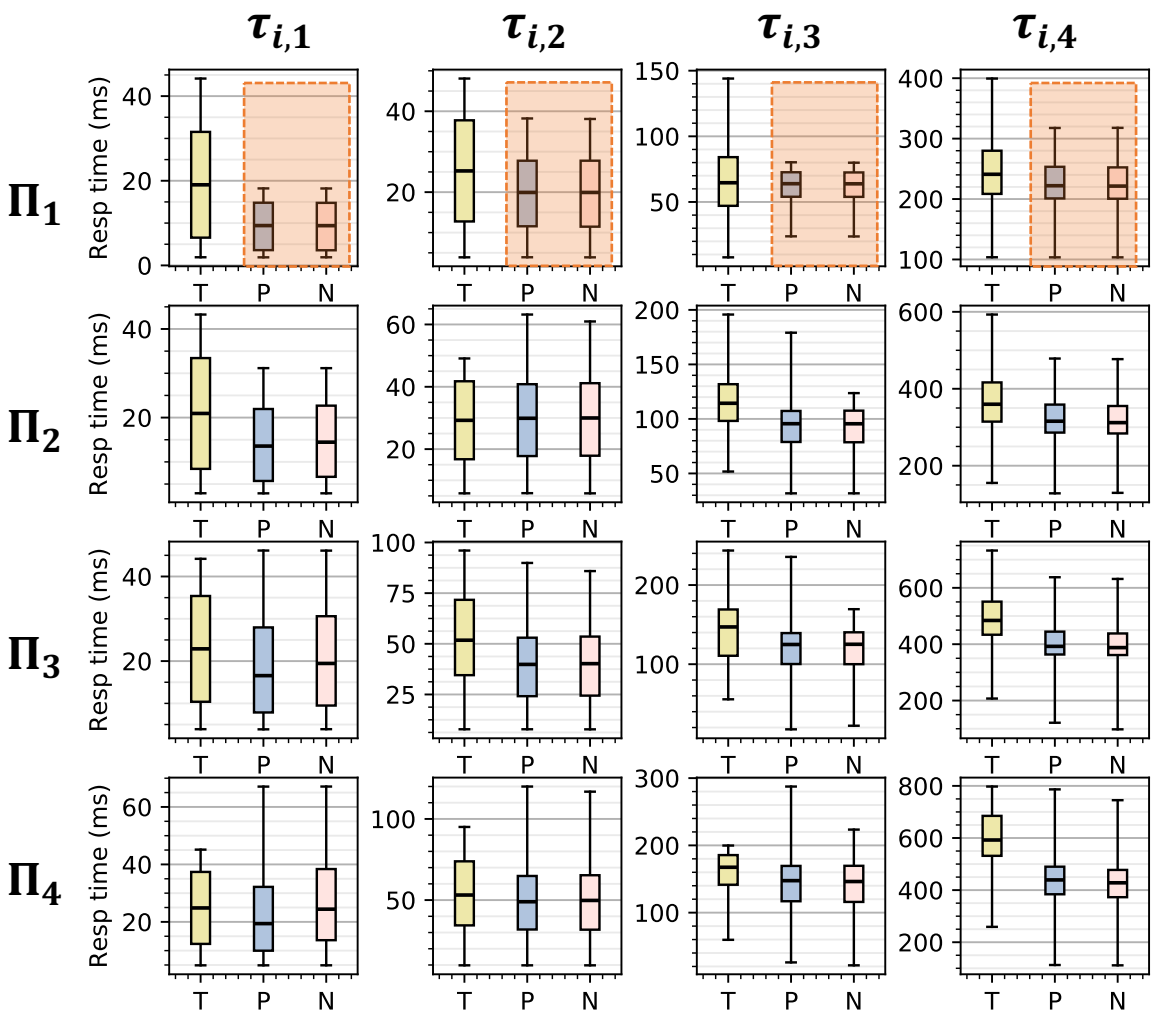
# Evaluation – Impact on Responsiveness




**T**: static, **P**: non-static w/o Blinder, **N**: non-static w/ Blinder

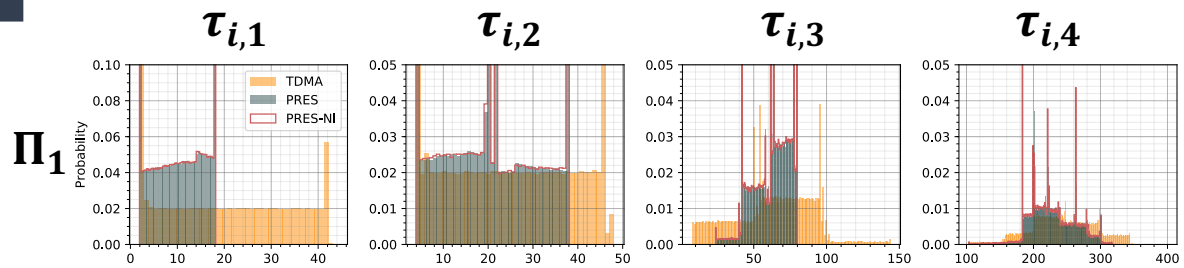
- TDMA (non-static partitioning) degrades the average responsiveness

# Evaluation – Impact on Responsiveness

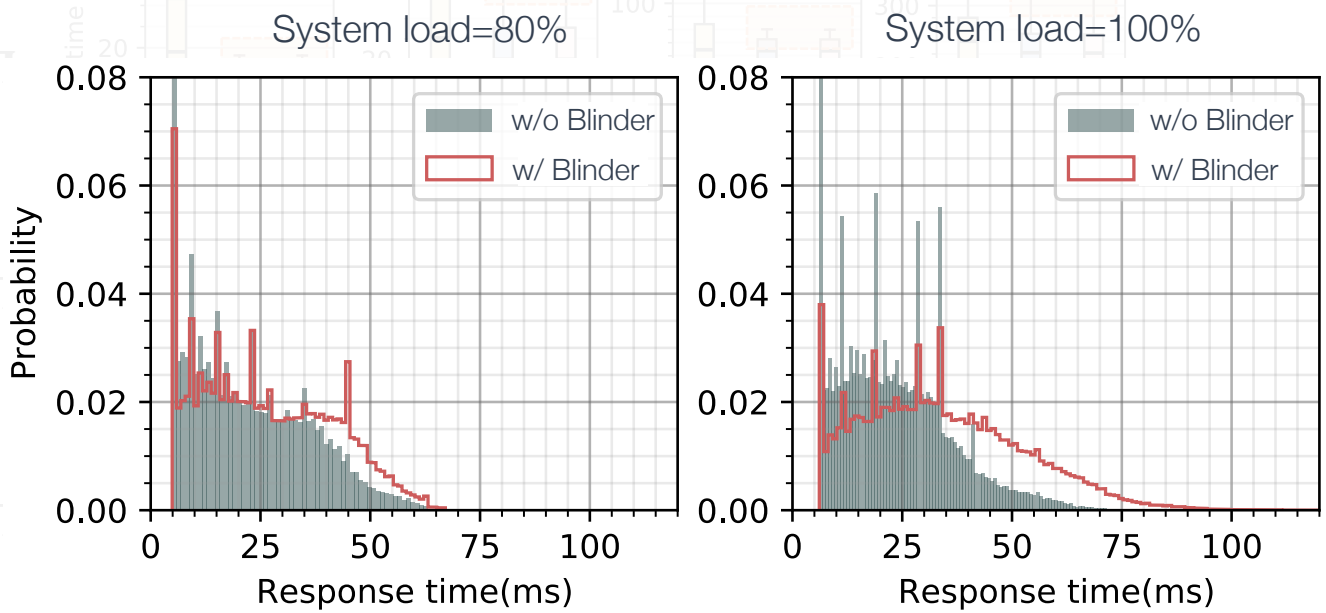



**T**: static, **P**: non-static w/o Blinder, **N**: non-static w/ Blinder

- The highest-priority partition ($\Pi_1$) is not affected by Blinder
  - Because it never enters the deferred release mode



# Evaluation – Impact on Responsiveness




(a) $\Pi_4$ is schedulable

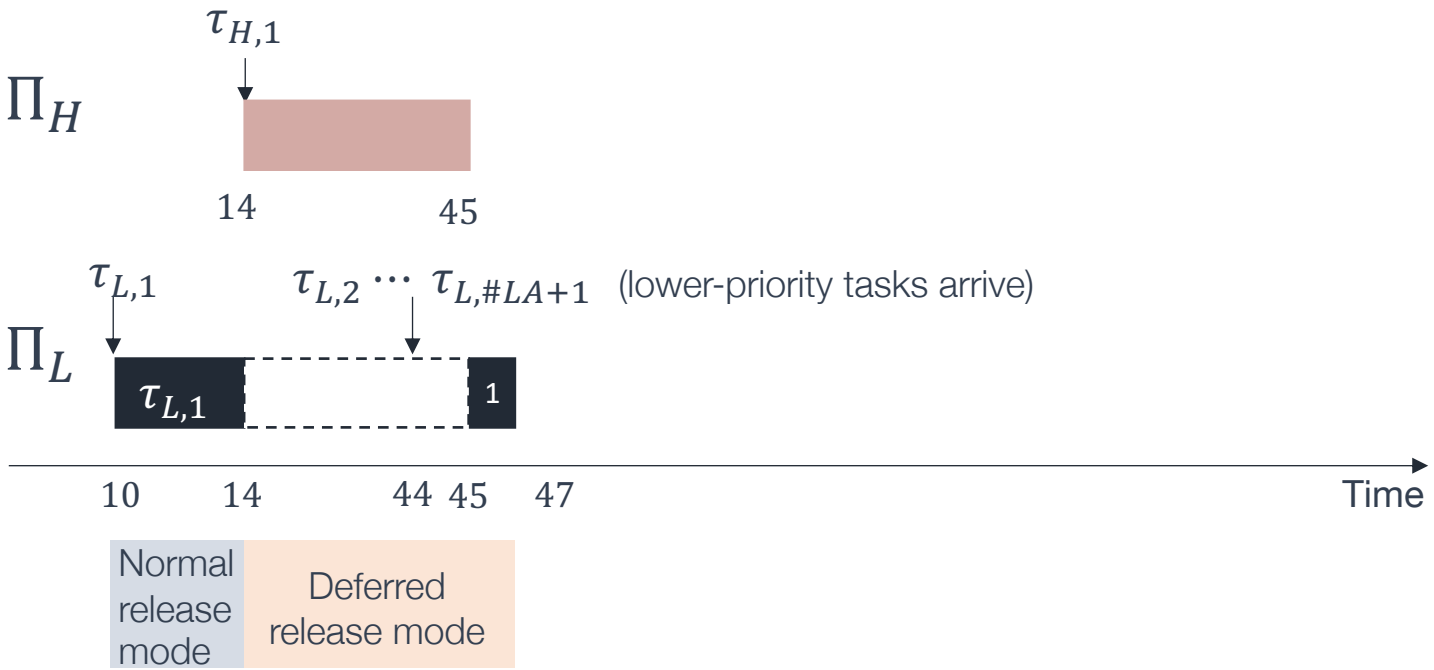
(b) $\Pi_4$ is <u>not</u> schedulable

(=$\Pi_4$ cannot guarantee $B_4$ over every $T_4$)

**Probability distribution of $\tau_{4,1}$'s response times**

- When partition is not schedulable, tasks may experience longer WCRT
  - System is ill-configured

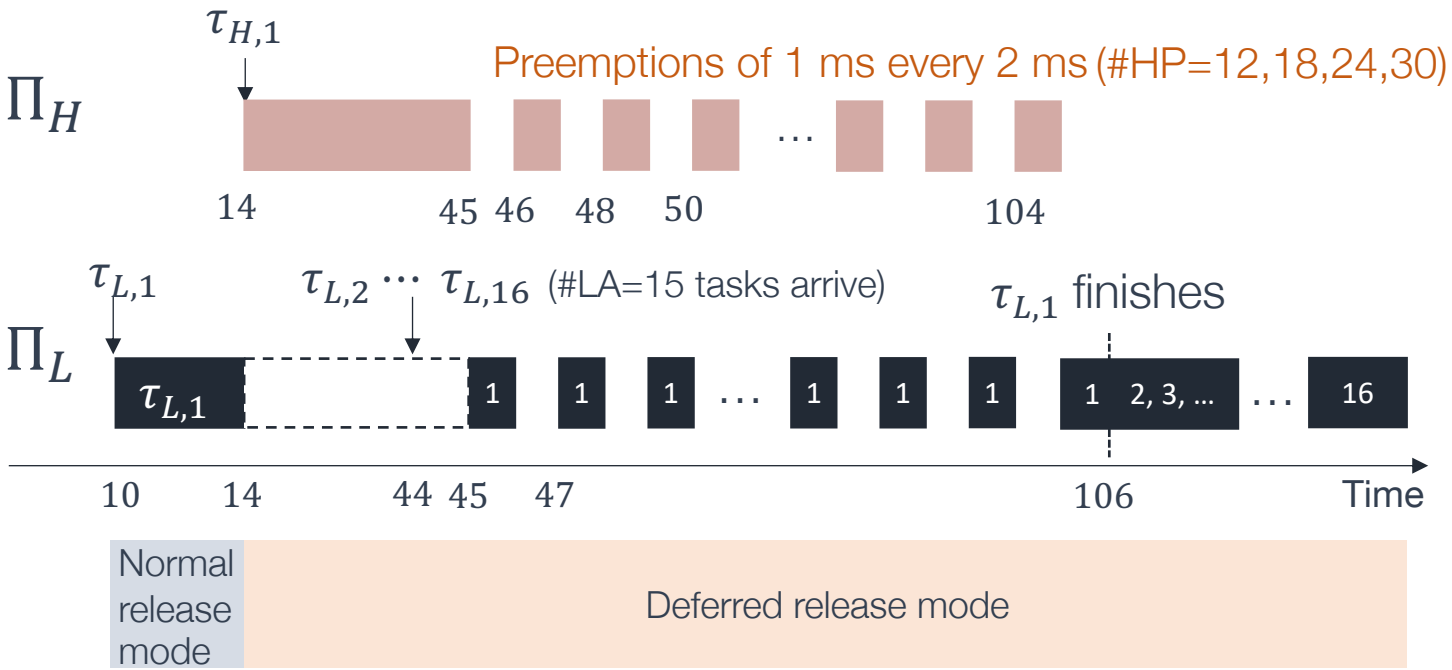
# Evaluation – Scheduling Overhead




Impacts of

1) Number of partition-level preemptions
2) Size of arrival queue

# Evaluation – Scheduling Overhead

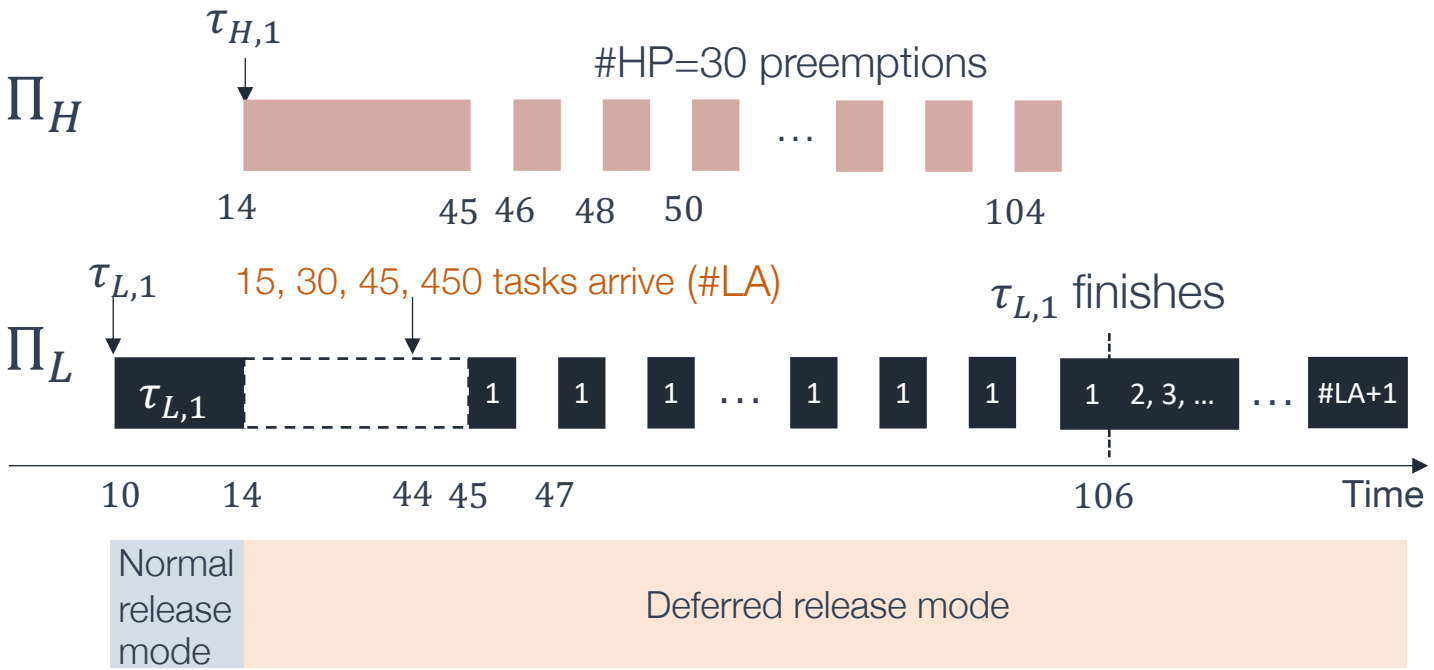



- Number of partition-level preemptions
  - More preemptions = more frequent timer updates

| #HP | 12 | 18 | 24 | 30 | 30 | | |
|---|---|---|---|---|---|---|---|
| #LA | 15 | | | | 30 | 45 | 450 |
| w/o Blinder | 96.15 (0.04) | 96.17 (0.06) | 96.19 (0.07) | 96.22 (0.09) | 96.22 (0.08) | 96.23 (0.08) | 96.39 (0.08) |
| w/ Blinder | 96.17 (0.04) | 96.19 (0.05) | 96.22 (0.06) | 96.24 (0.08) | 96.25 (0.08) | 96.26 (0.07) | 96.73 (0.07) |

Average response time of $\tau_{L,1}$ (in ms) and stdev

# Evaluation – Scheduling Overhead




- Size of arrival queue
  - More tasks in arrival queue = more frequent lag update

| #HP | 12 | 18 | 24 | 30 | 30 | | |
|---|---|---|---|---|---|---|---|
| #LA | 15 | | | | 30 | 45 | 450 |
| w/o Blinder | 96.15 (0.04) | 96.17 (0.06) | 96.19 (0.07) | 96.22 (0.09) | 96.22 (0.08) | 96.23 (0.08) | 96.39 (0.08) |
| w/ Blinder | 96.17 (0.04) | 96.19 (0.05) | 96.22 (0.06) | 96.24 (0.08) | 96.25 (0.08) | 96.26 (0.07) | 96.73 (0.07) |

Average response time of $\tau_{L,1}$ (in ms) and stdev