Timing Analysis of Parallel and Accelerated Software with Event-Driven Delay-Induced Tasks

Federico Aromolo

Scuola Superiore Sant’Anna, Pisa, Italy

CAPITAL Workshop 2024: Scalable and Precise Timing Analysis for Multicore Platforms

June 14, 2024 – IRT Saint Exupéry, Toulouse, France

Sant’Anna
School of Advanced Studies – Pisa

TELECOMMUNICATIONS, COMPUTER ENGINEERING, AND PHOTONICS INSTITUTE

Retis
Real-Time Systems Laboratory

Real-Time Systems Laboratory

- The Real-Time Systems Laboratory (RETIS Lab) is part of the TeCIP Institute of Scuola Superiore Sant'Anna – Pisa, Italy
  - Approx. 40 people
- Main topics:
  - Embedded real-time systems
  - Time-critical scheduling algorithms
  - Advanced operating systems
  - Adaptive resource management
  - System-level cyber-security
  - Safe and secure machine learning

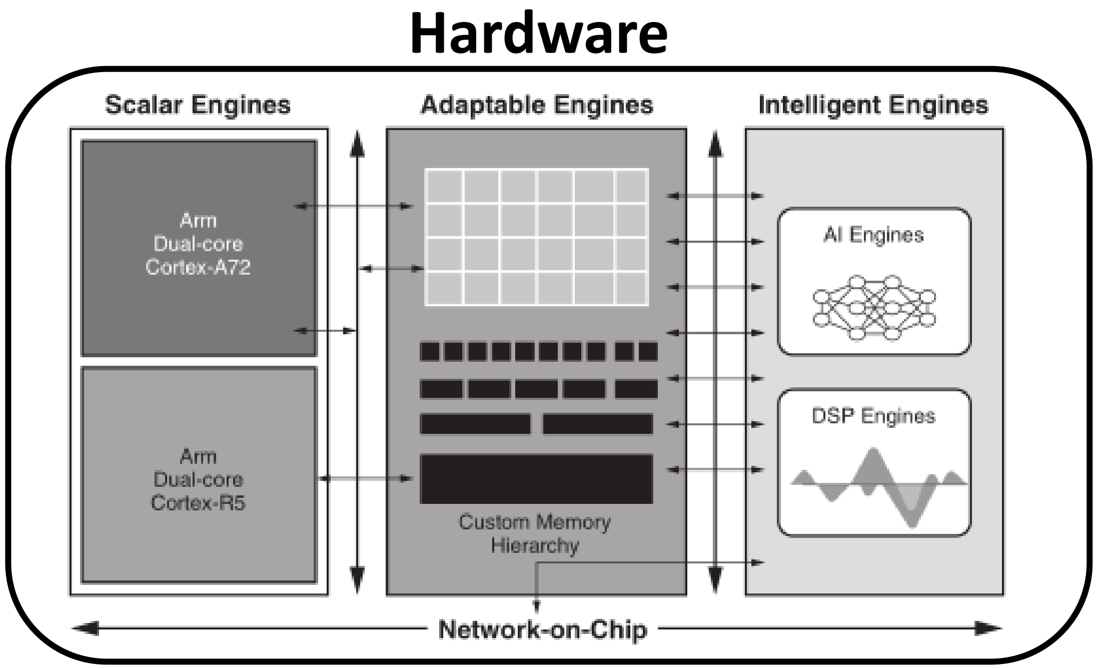
Sant'Anna School of Advanced Studies – Pisa

TELECOMMUNICATIONS, COMPUTER ENGINEERING, AND PHOTONICS INSTITUTE

Retis Real-Time Systems Laboratory

Heterogeneous computing platforms

- **Emerging industry trend in the field of real-time embedded systems:** integrate multiple functionalities onto a single computing platform
- Heterogeneous platforms combine scalar multicores and HW accelerators
  - E.g., FPGAs, GPUs, DSPs, AI engines, …

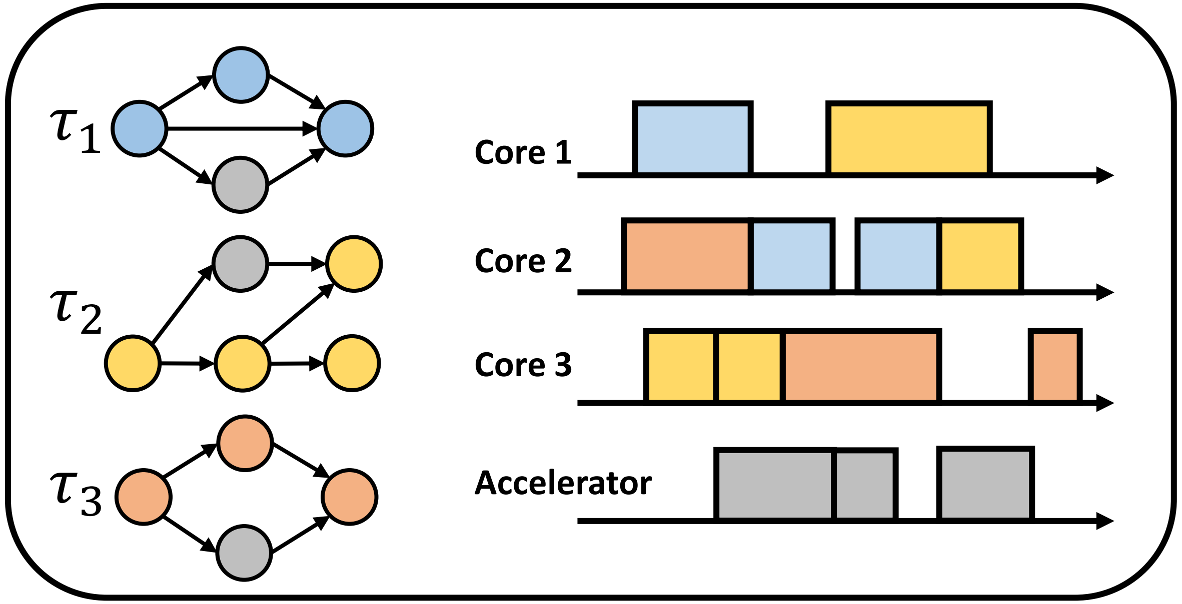


Architecture of the Xilinx Versal ACAP SoC

Heterogeneous computing platforms

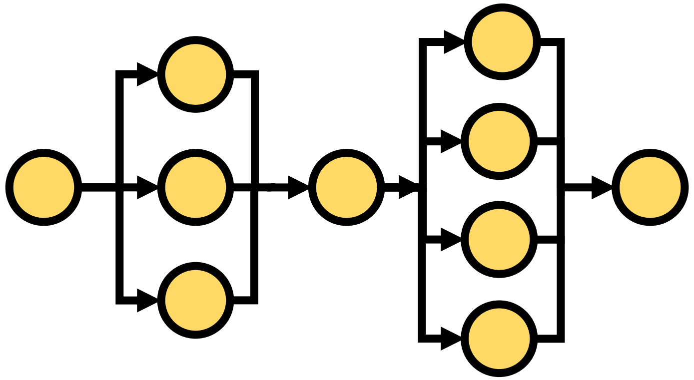
- The typical software workload exploits the available platform capabilities with complex execution patterns:
  - Parallel computation on multiprocessors
  - Hardware acceleration requests
  - Data dependencies and shared resources

Software

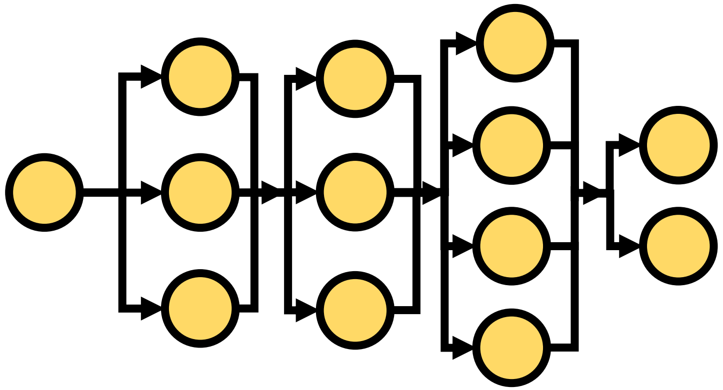


Parallel task models

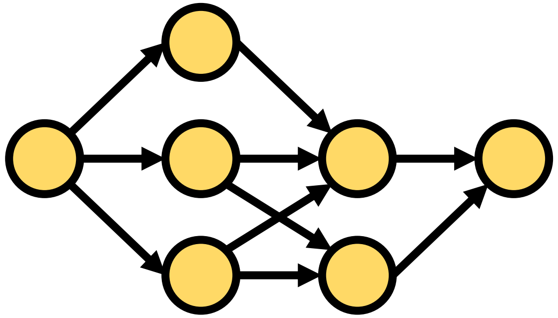
- There are different forms of **sporadic parallel tasks**, representing the internal parallelism of each task in addition to the inter-task parallelism inherent to multitasking
- **Multi-threaded parallel task models:**



Fork-join



Synchronous parallel



Directed Acyclic Graph (DAG)

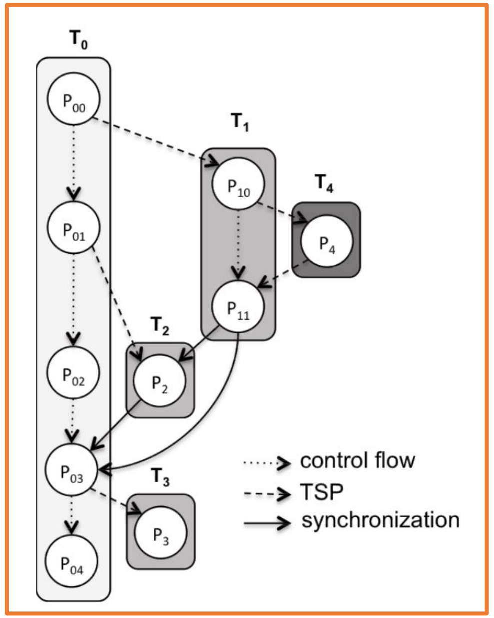
Parallel task models

- An important application of DAG parallel tasks is modeling and analyzing the **structure and scheduling behavior of OpenMP parallel software**

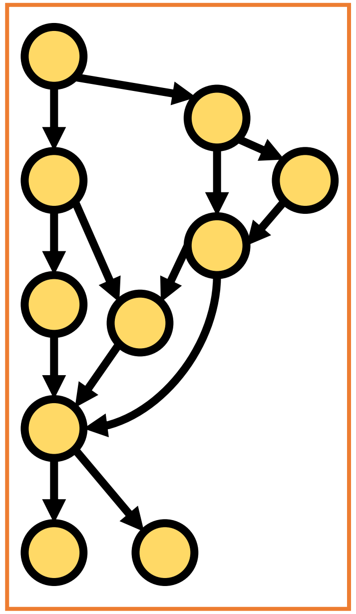
Program code

```
#pragma omp parallel num_threads(10) {
#pragma omp master {
#pragma omp task { // T0
   part00
   #pragma omp task depend(out:x) // T1
                    final(true)
   {
     part10
     #pragma omp task { part4 } // T4
     part11
   }
   part01
   #pragma omp task depend(in:x) // T2
   { part2 }
   part02
   #pragma omp taskwait
   part03
   #pragma omp task { part3 } // T3
   part04
}}}
```

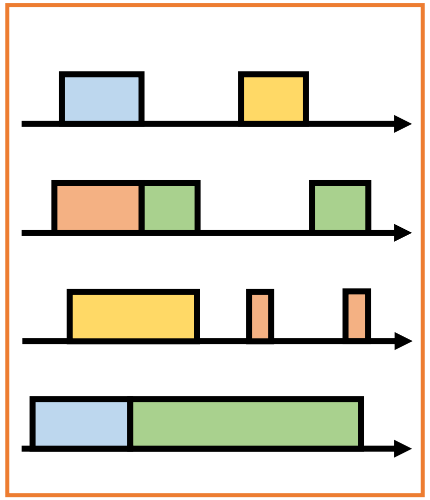
Program structure



DAG model



Timing analysis

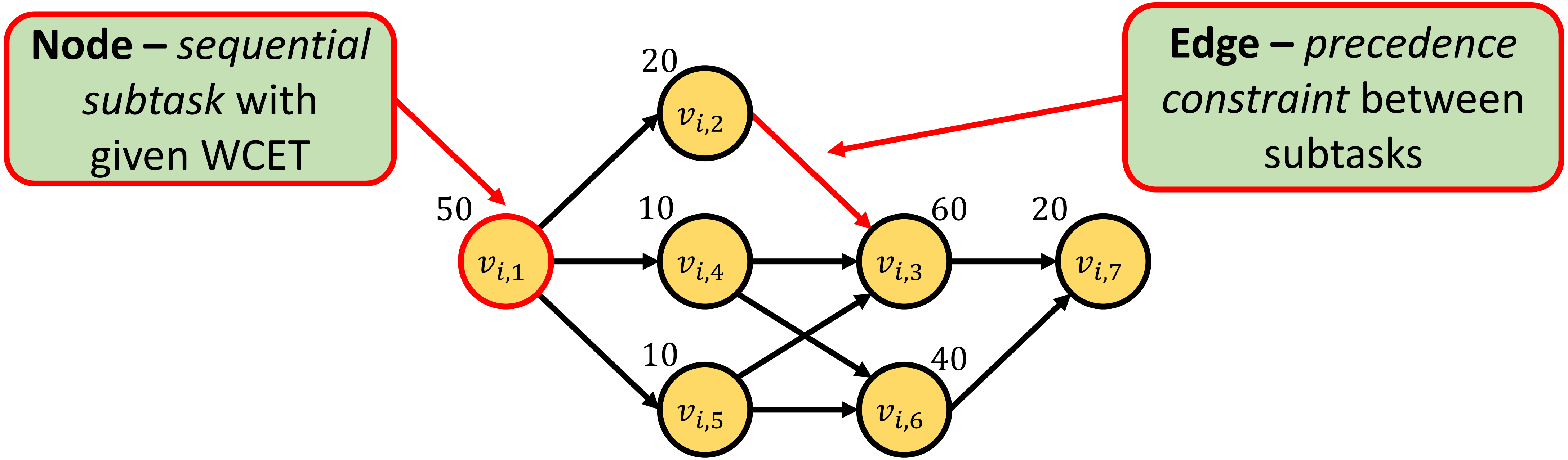


From: Vargas et al. – "OpenMP and Timing Predictability: A Possible Union?" - 2015

Parallel task models

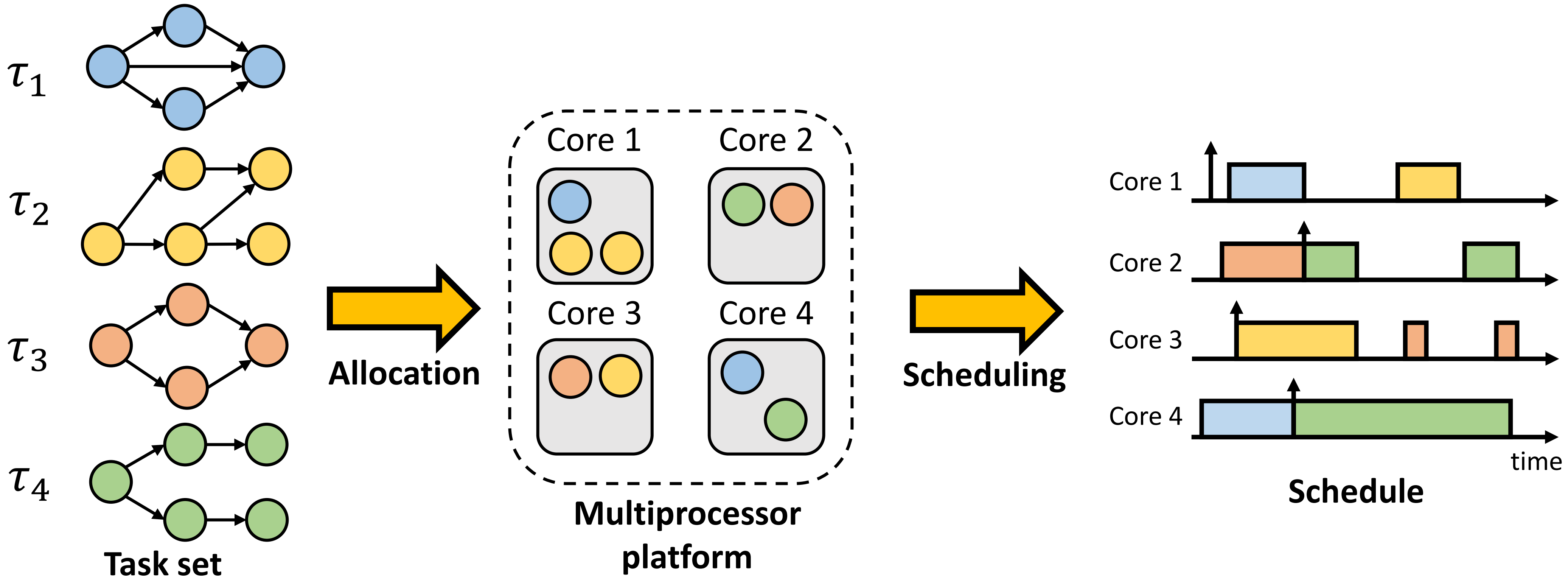
- In the real-time **sporadic parallel DAG model**, each task τ_i:

1. Is released sporadically with minimum period T_i
2. Is subject to a deadline $D_i \leq T_i$
3. Is structured as a directed acyclic graph (DAG)



Scheduling paradigms for parallel tasks

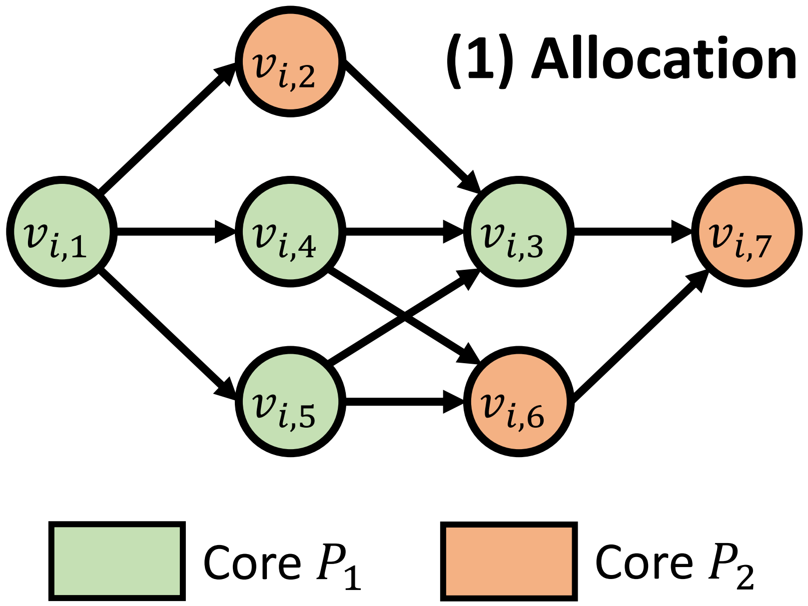
- Different **scheduling paradigms** exist to allocate and schedule subtasks on the cores of a multiprocessor platform

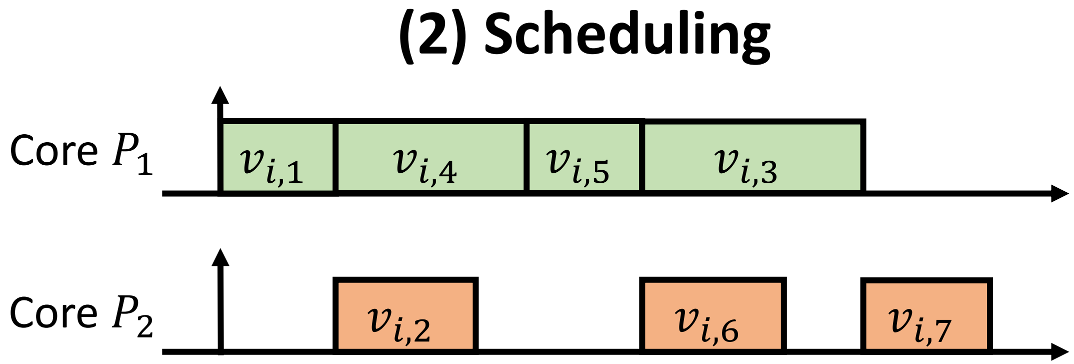


Scheduling paradigms for parallel tasks

- **Partitioned scheduling:**

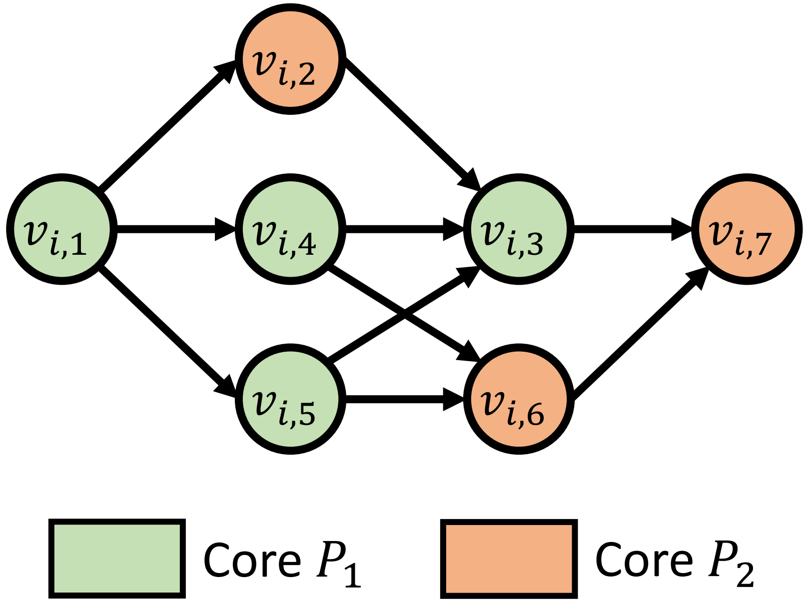
1. **At design time,** each node is statically allocated to a specific processor
2. **At runtime,** nodes are scheduled on the corresponding processor with a uniprocessor policy

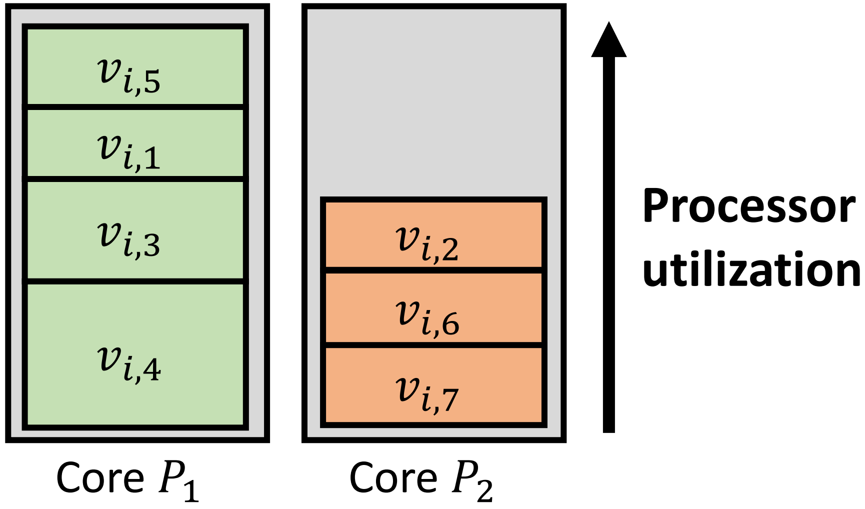




Scheduling paradigms for parallel tasks

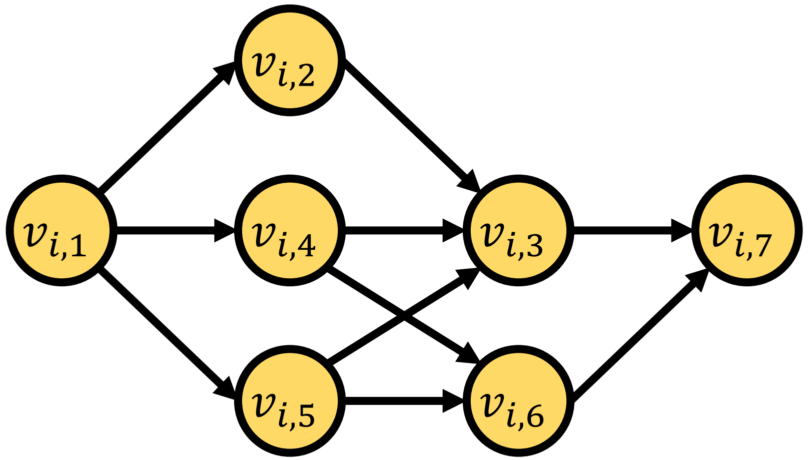
- **Advantage**: uniprocessor scheduling and analysis techniques can be reused
- **Disadvantage**: requires solving a complex allocation problem at design time (typically approached with bin-packing heuristics)

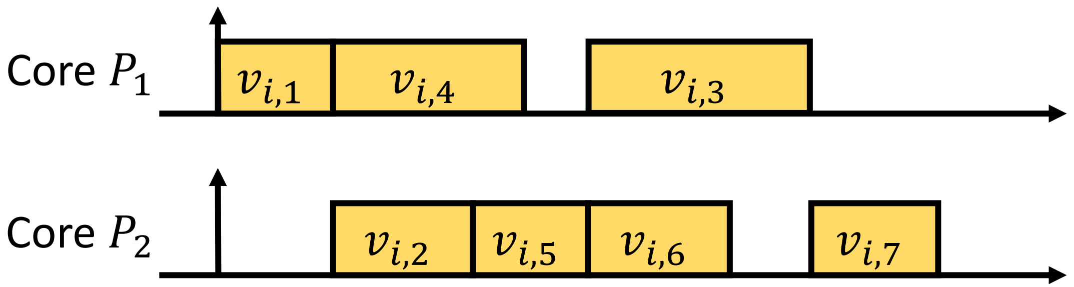




Scheduling paradigms for parallel tasks

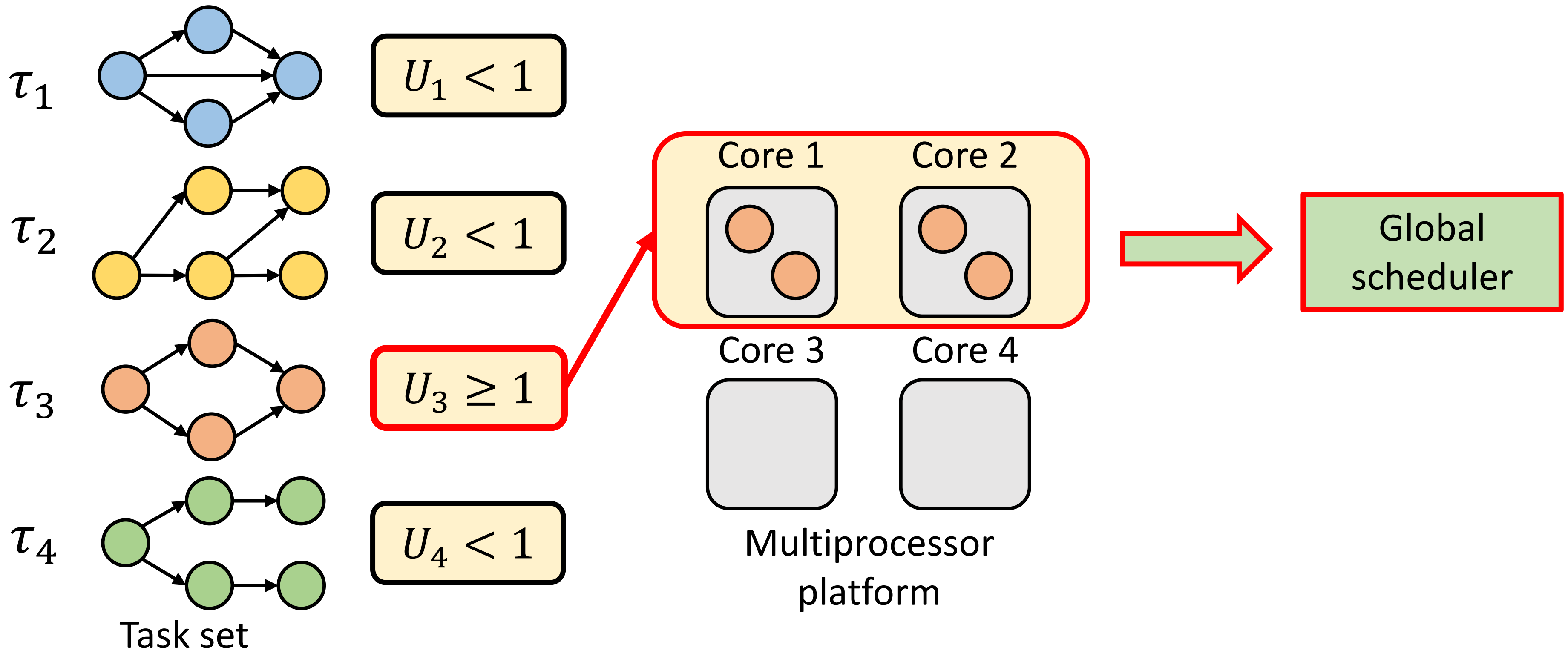
- **Global scheduling:** each subtask can execute on any one of the processors available at a given time, according to their priority level
- **Advantage**: flexible runtime behavior with automatic load balancing
- **Disadvantages**: significant overheads due to migration; complex WCET analysis





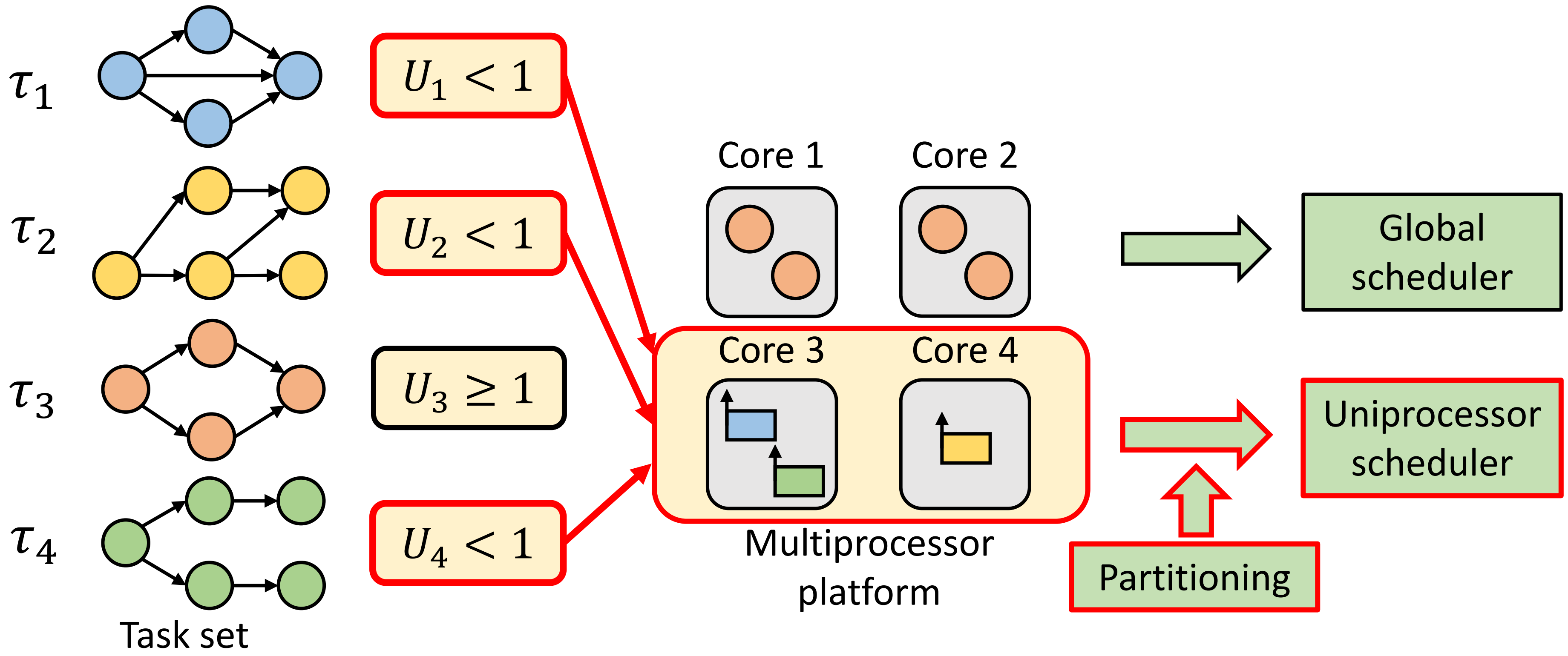
Scheduling paradigms for parallel tasks

- **Federated scheduling**: hybrid approach
  - **1.** Each **heavy task** ($U_i \geq 1$) is assigned a set of dedicated processors, where it is scheduled in isolation by a global scheduler



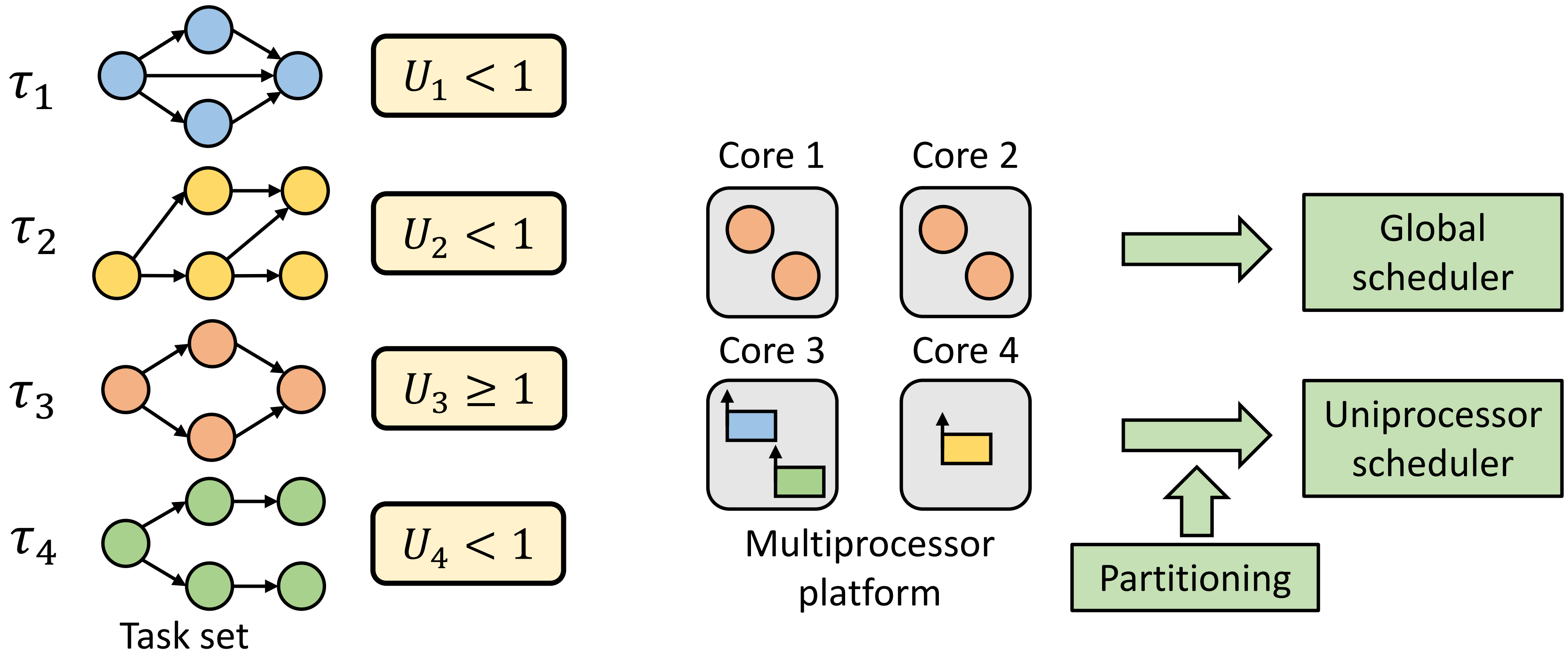
Scheduling paradigms for parallel tasks

- **Federated scheduling**: hybrid approach
  - **2. Light tasks** ($U_i < 1$) are treated as sequential tasks and partitioned on the remaining processors, where they are scheduled with a uniprocessor policy



Scheduling paradigms for parallel tasks

- **Advantage:** simple and efficient analysis
- **Disadvantage:** processors dedicated to a heavy task can be underutilized

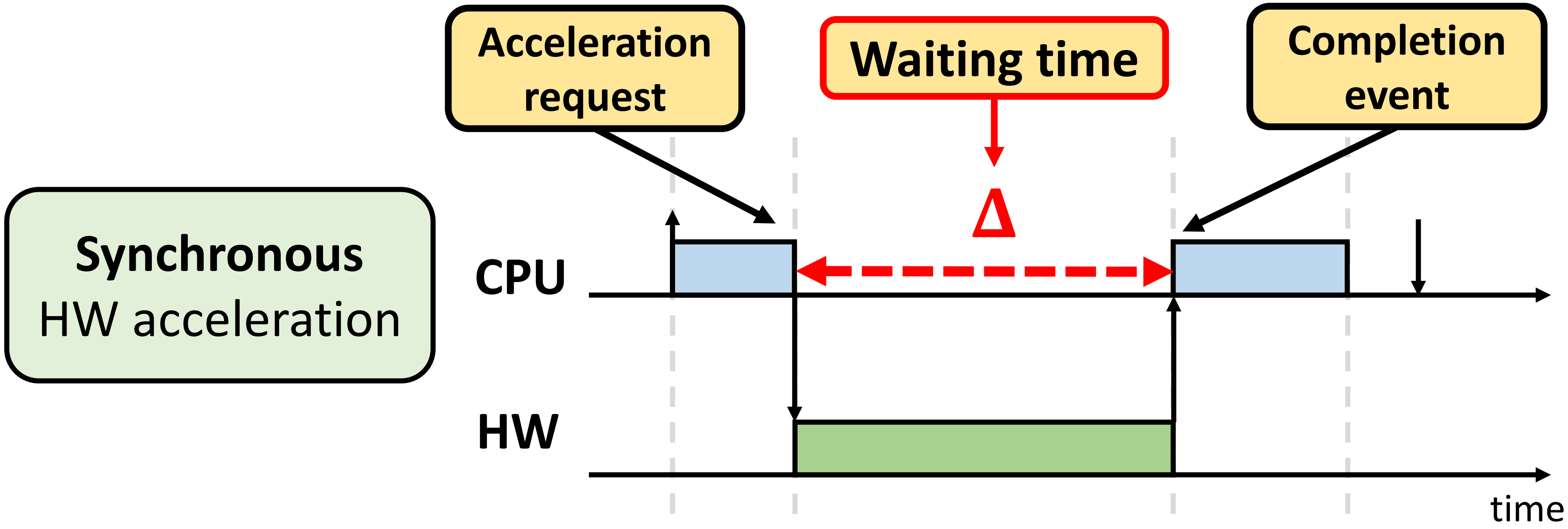


Scheduling paradigms for parallel tasks

- **Partitioned scheduling**:
  - Practical advantages in the implementation
  - Fine-grained control of memory contention and tight blocking bounds in the presence of locking
  - Design-time complexity can be approached with specialized bin packing heuristics
- Empirical evaluations of C=D semi-partitioned EDF scheduling of sequential tasks showed 99%+ schedulable utilization on multiprocessors (Burns et al. 2012, Brandenburg and Gül 2016)
  - C=D semi-partitioned scheduling is a simple and practical approach, as opposed to complex optimal global scheduling algorithms, which incur significant overheads
- However, a **specialized and effective analysis for partitioned parallel tasks is still missing**

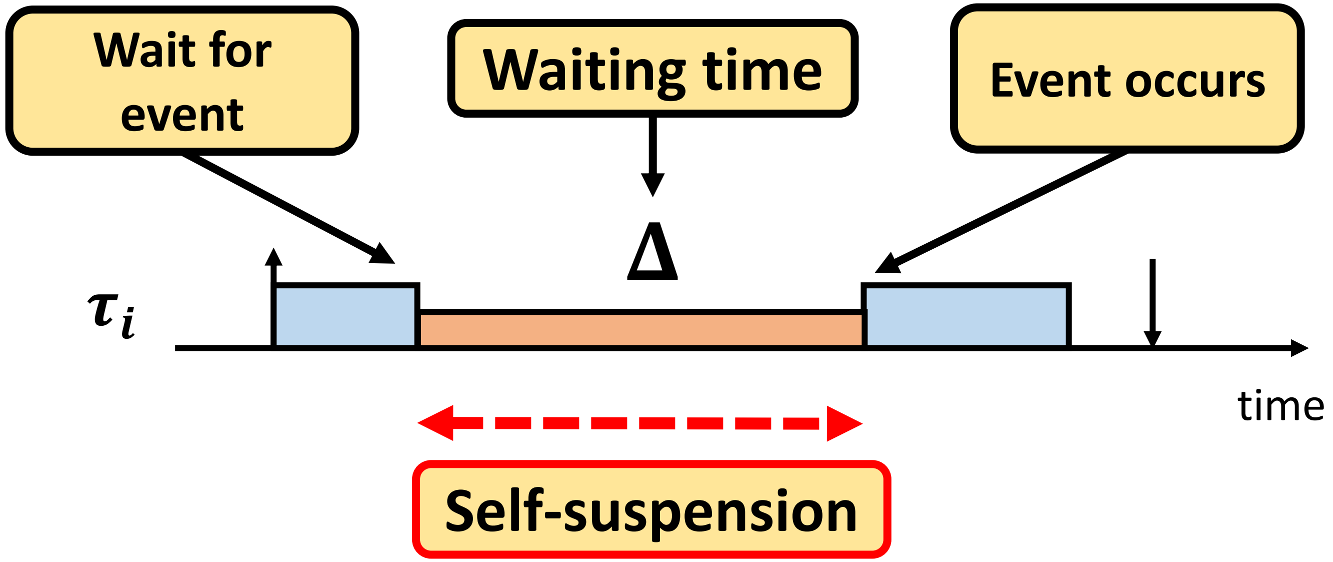
Hardware acceleration

- **Another form of parallelism is due to hardware acceleration**
- **Synchronous hardware acceleration:** when offloading computation to the accelerator, the task must wait for the completion of the acceleration before proceeding



Self-suspending tasks

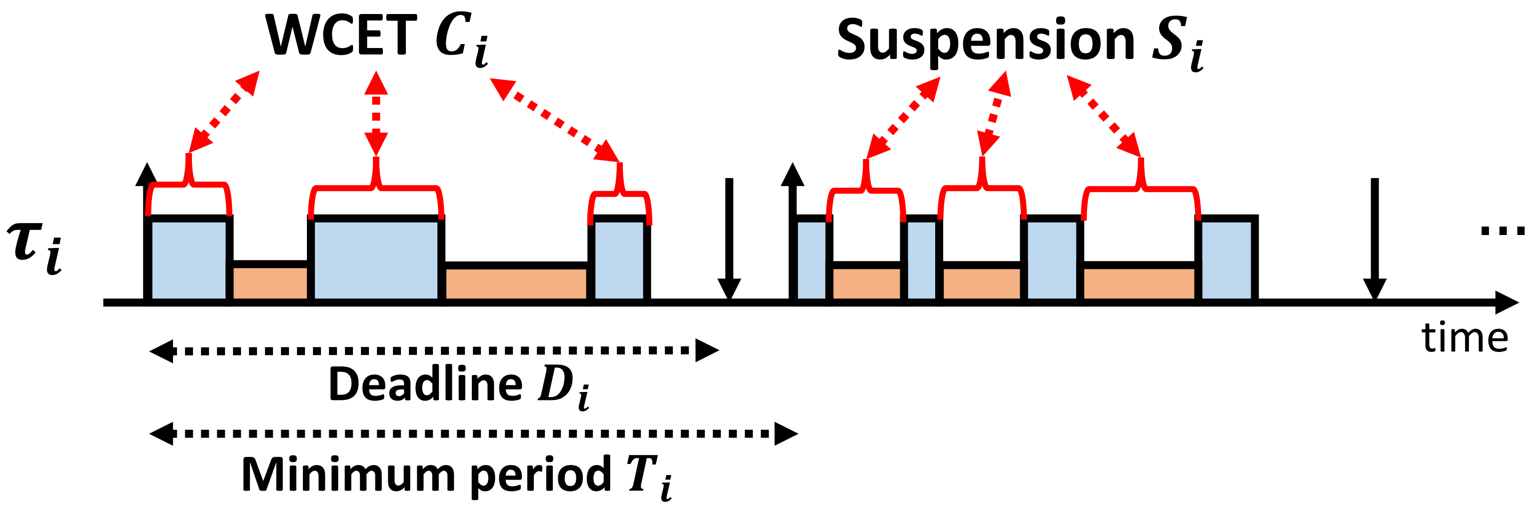
- **Since acceleration delays may be significant,** the typical implementation involves a **self-suspending behavior**
- The **self-suspending task model** was introduced to deal with self-suspending behaviors in the real-time analysis
  - E.g., hardware acceleration, locking protocols, inter-processor synchronization



Self-suspending tasks

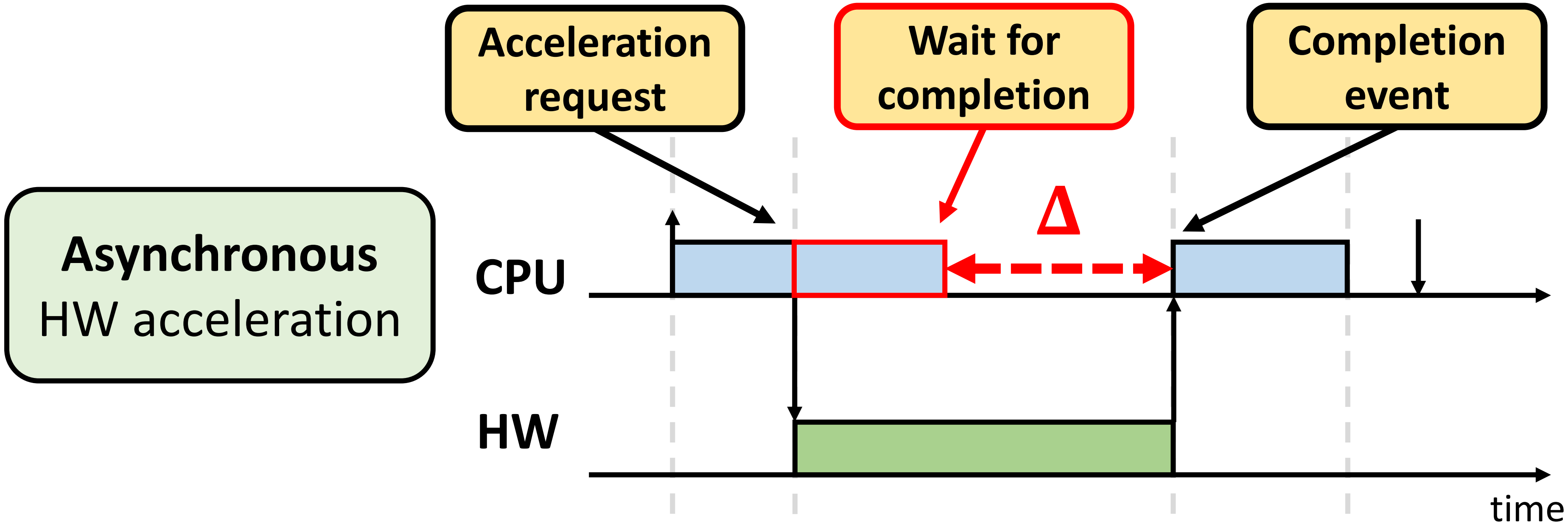
- Under the **dynamic self-suspending task model**, each task τ_i:

1. Is released sporadically with minimum period T_i
2. Is subject to a deadline $D_i \leq T_i$
3. Alternates an arbitrary number of execution and suspension phases up to a cumulative WCET C_i and a cumulative suspension time S_i



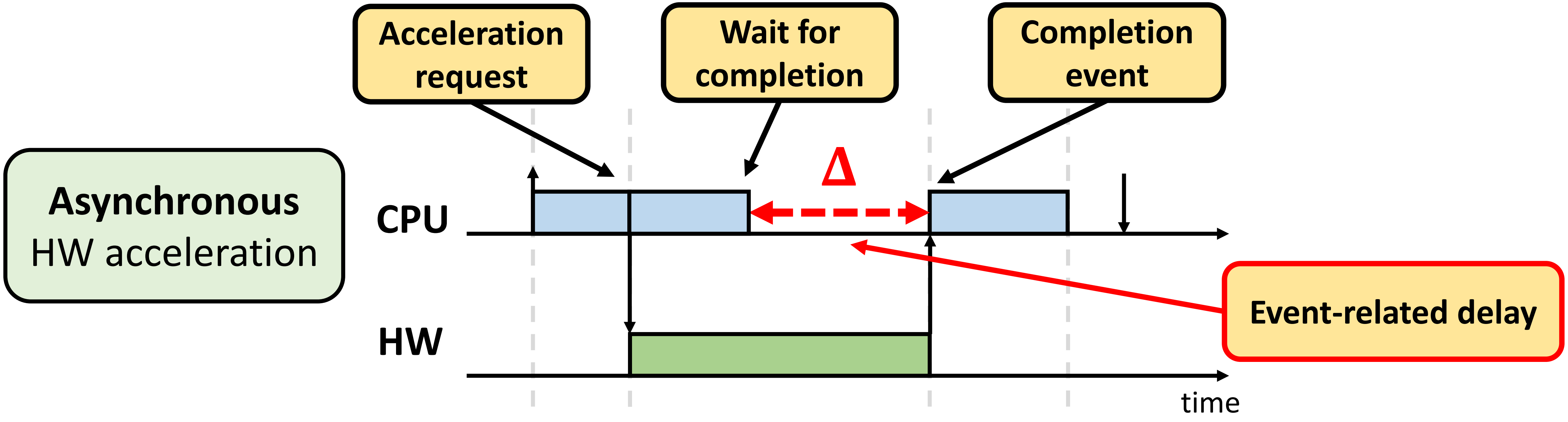
Hardware acceleration

- **Asynchronous hardware acceleration:** after offloading computation to the accelerator, the task can continue executing on the processor before waiting for the completion of the acceleration
- Self-suspending task models **do not explicitly support asynchronous hardware acceleration**



Event-related delays

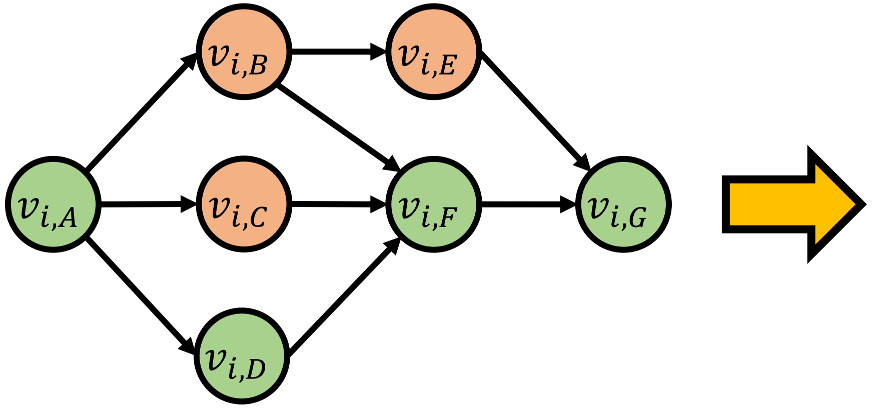
- Asynchronous HW acceleration and partitioned scheduling of parallel tasks share a common scheduling pattern in which the task must wait for an asynchronous event, thus incurring **event-related delays**

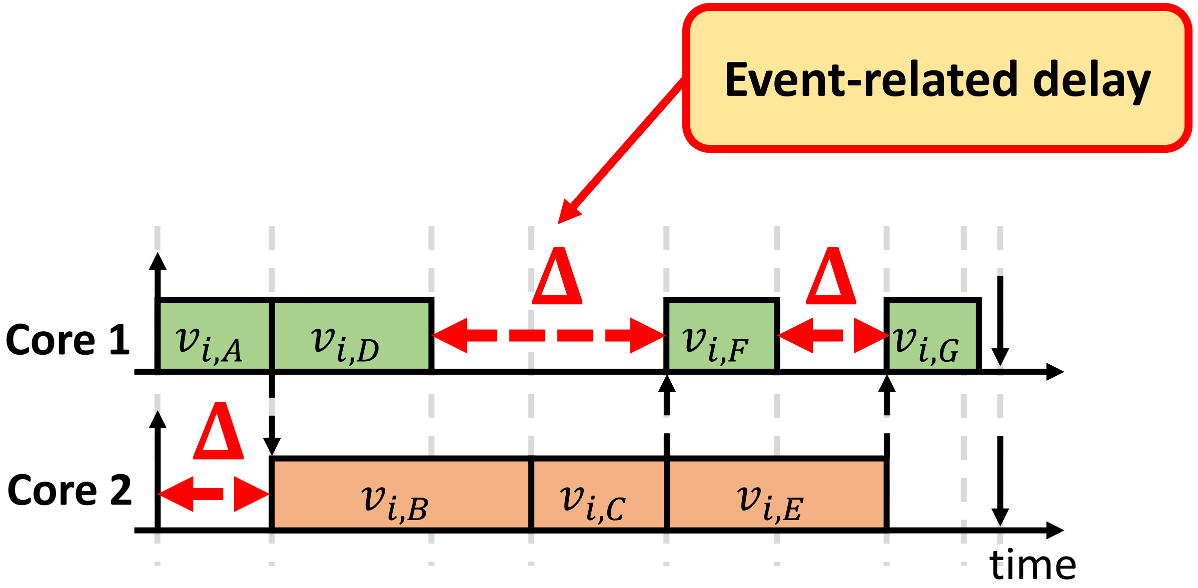


Event-related delays

- Asynchronous HW acceleration and partitioned scheduling of parallel tasks share a common scheduling pattern in which the task must wait for an asynchronous event, thus incurring **event-related delays**

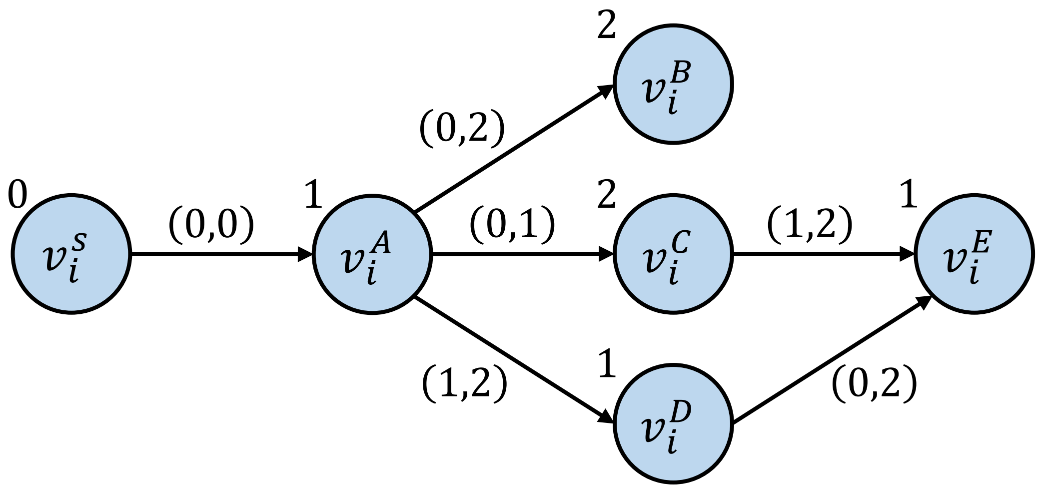
Partitioned
parallel tasks





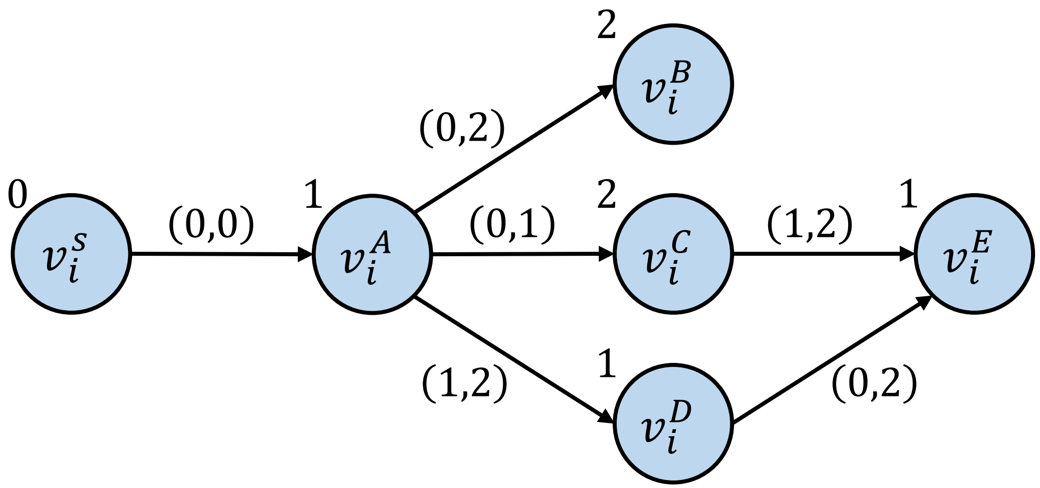
EDD task model

- Existing techniques either deal with event-related delays with considerable analytical pessimism, or can only support specific types of workloads
- **Contribution:** definition of the **event-driven delay-induced (EDD) task model**, which explicitly deals with complex computing workloads incurring event-related delays



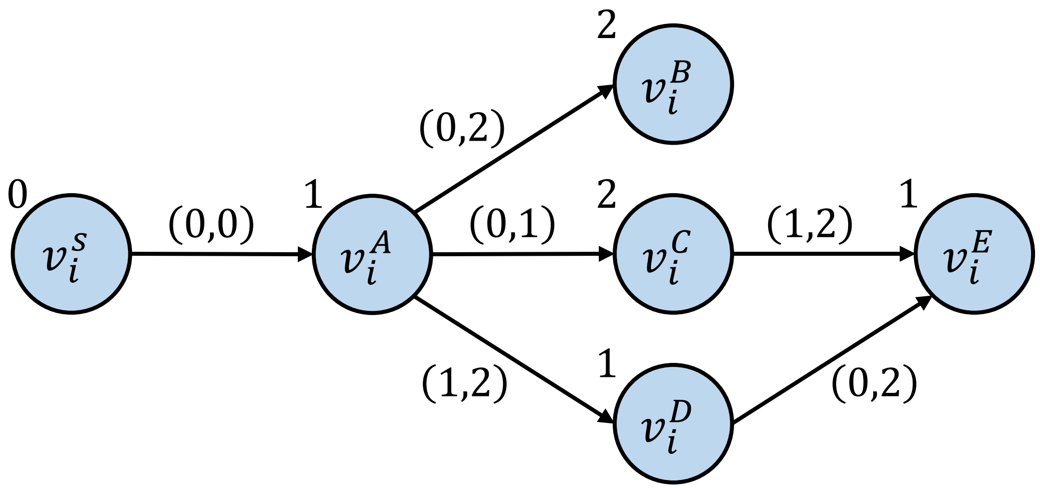
Contributions

- **Event-driven delay-induced (EDD) task model:**
  - Explicitly deals with complex computing workloads that incur event-related delays
- **Analysis techniques:** closed-form and optimization-based
- **Applications:**
  - Modeling of asynchronous hardware acceleration
  - Analysis of partitioned parallel DAG tasks on multicores
  - Generalization of existing task models



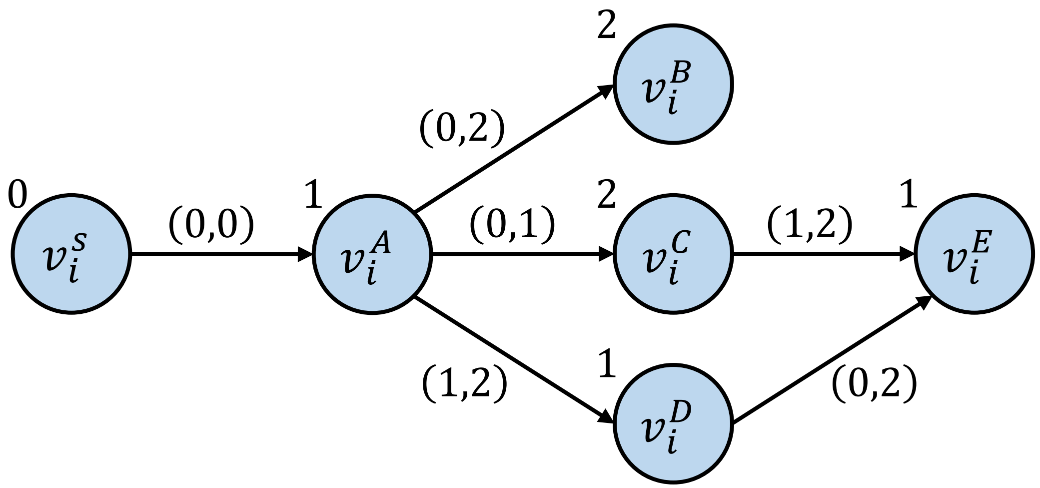
EDD task model

- Preemptive execution on a single processor
- Each EDD task τ_i in a task set τ:
  - is released with a minimum inter-arrival time T_i
  - must complete within a deadline $D_i \leq T_i$
  - is scheduled with fixed priorities π_i



EDD task model

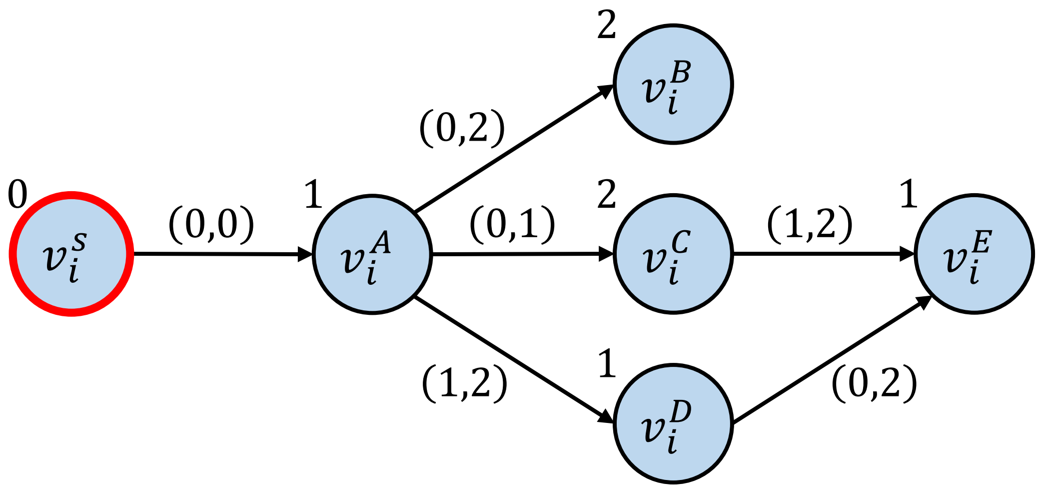
- The workload of a task is described by a **directed acyclic graph** (DAG) G_i



EDD task model

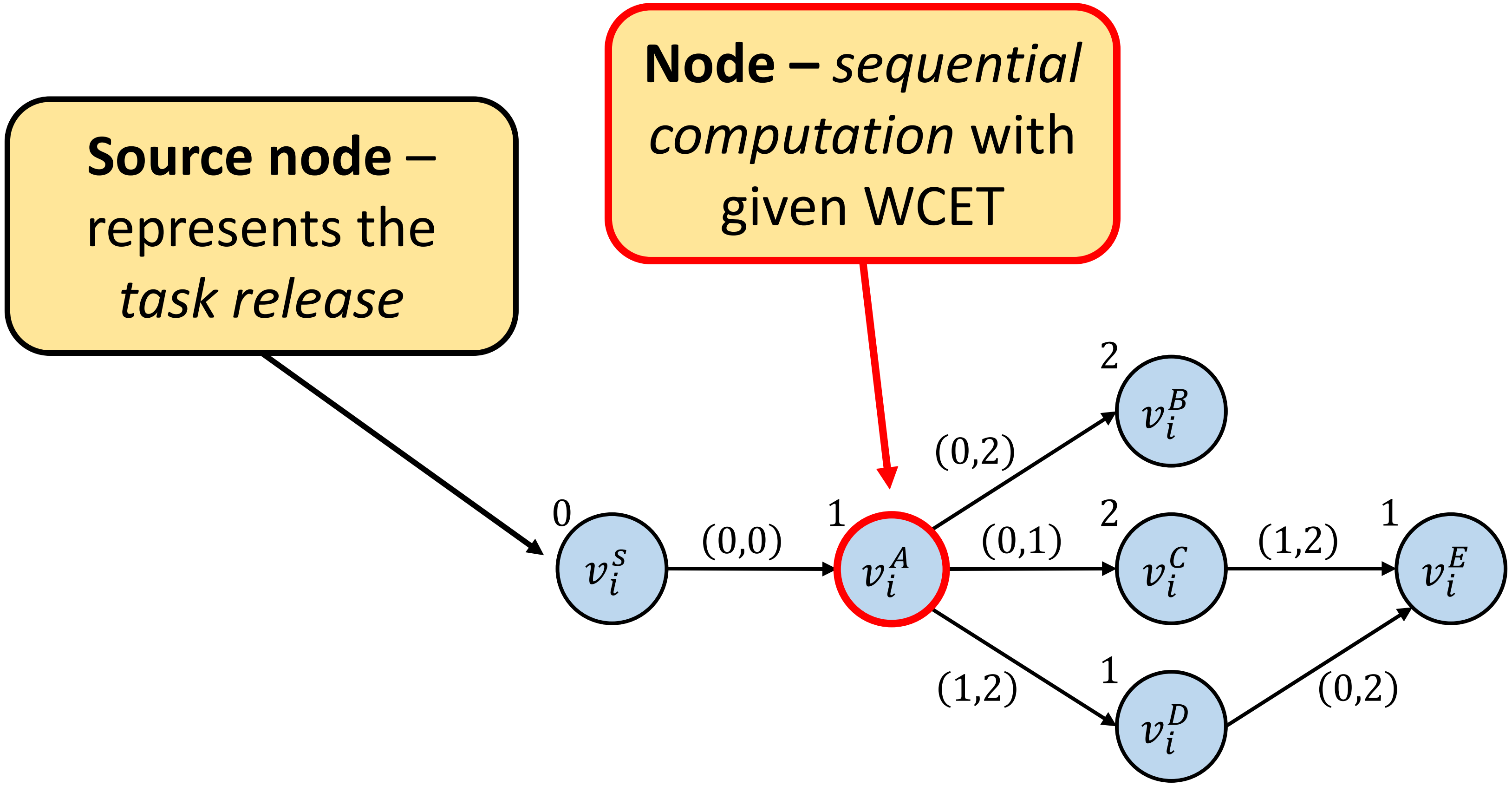
- The workload of a task is described by a **directed acyclic graph** (DAG) G_i

Source node – represents the *task release*



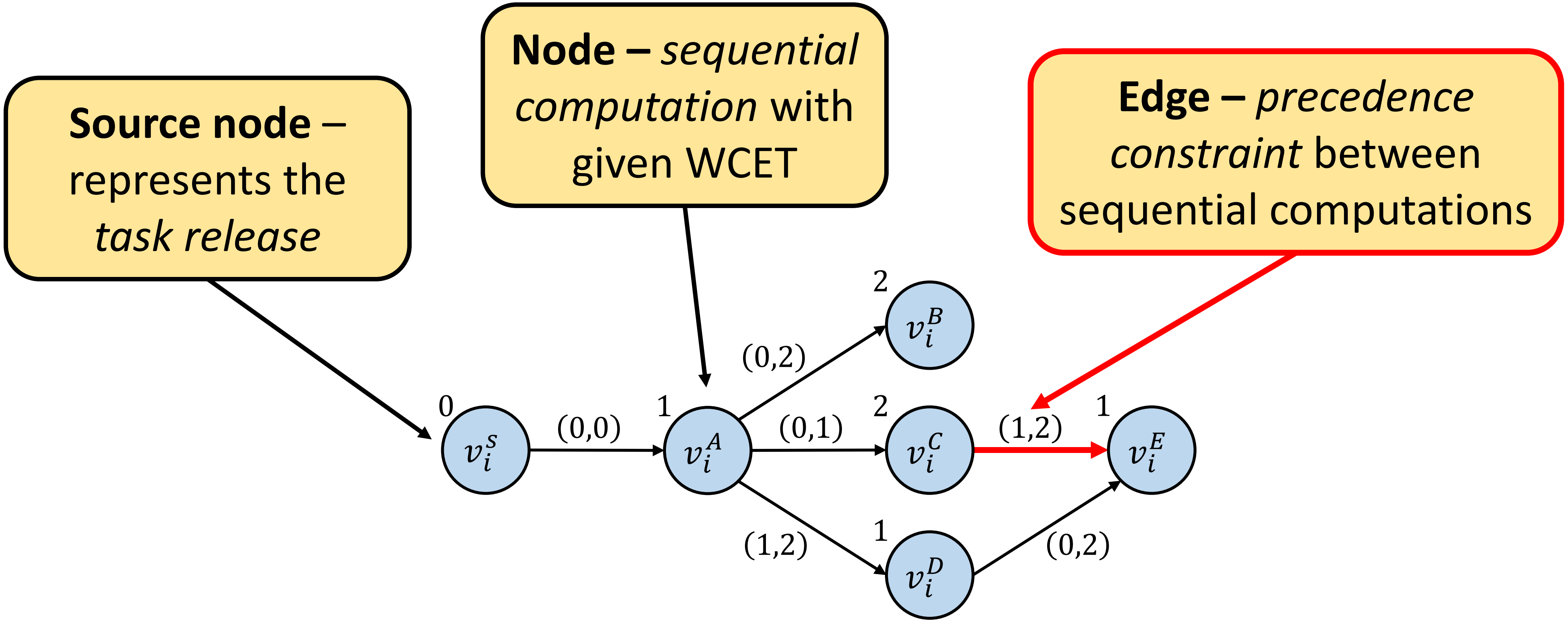
EDD task model

- The workload of a task is described by a **directed acyclic graph** (DAG) G_i



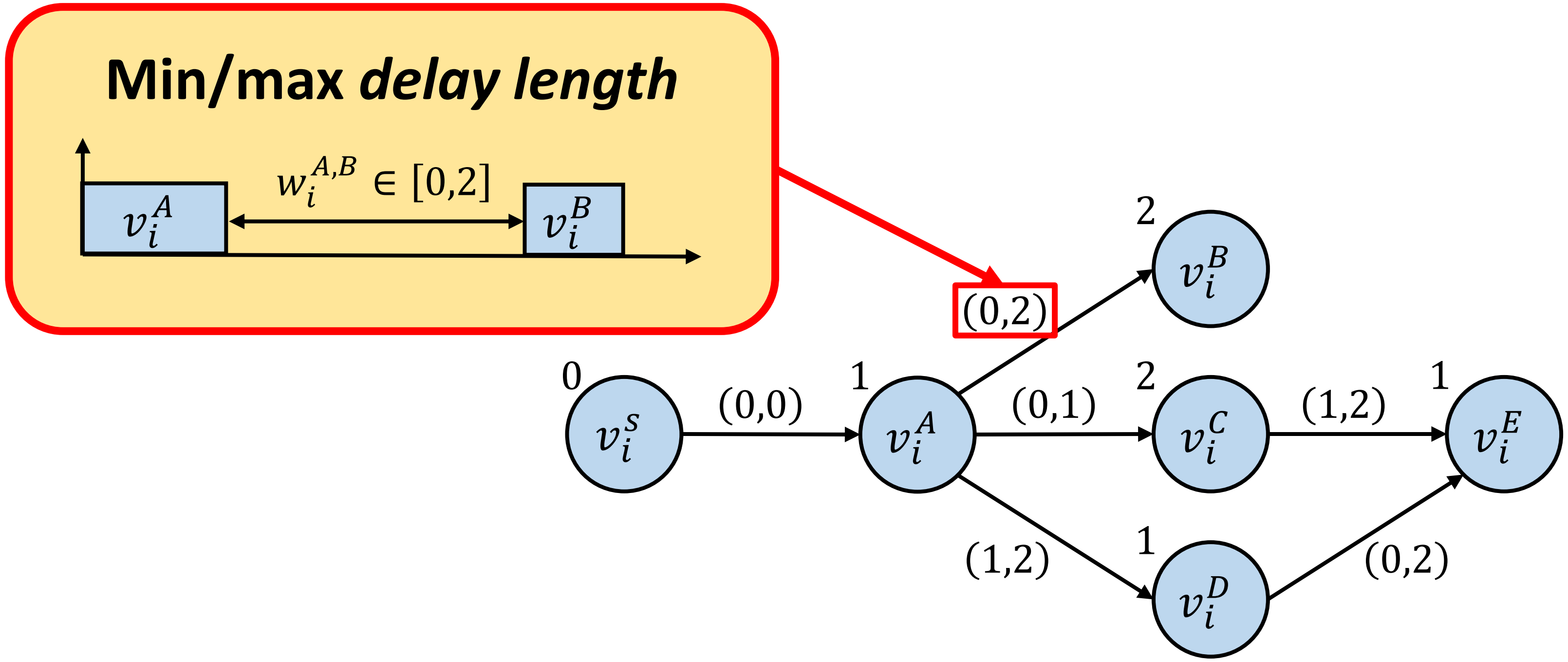
EDD task model

- The workload of a task is described by a **directed acyclic graph** (DAG) G_i



EDD task model

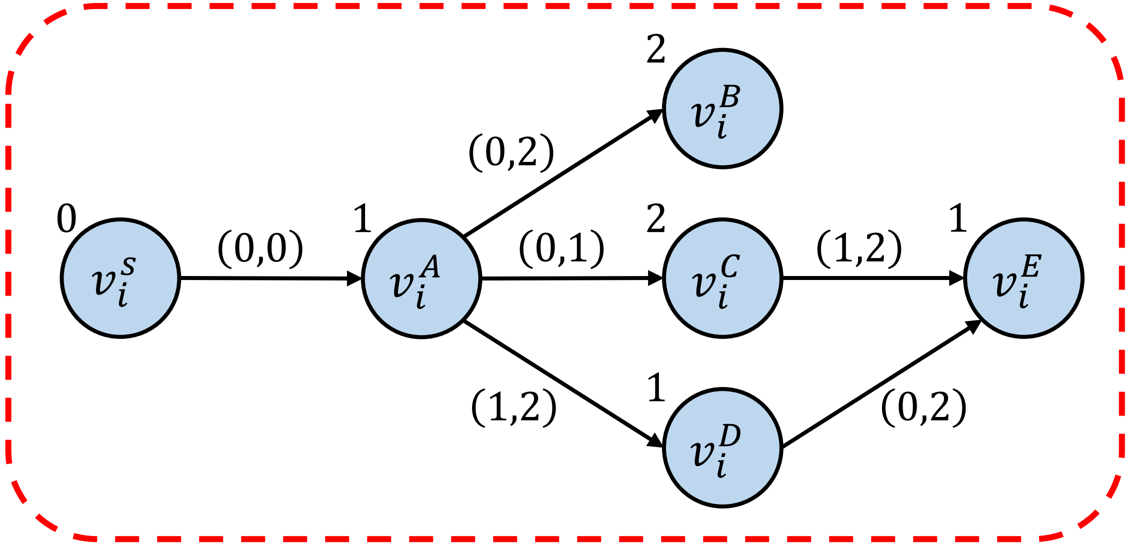
- **Precedence constraint**: satisfied once a variable delay has elapsed after the completion of the predecessor node
  - Models **bounded delays** related to task release or subtask completion events



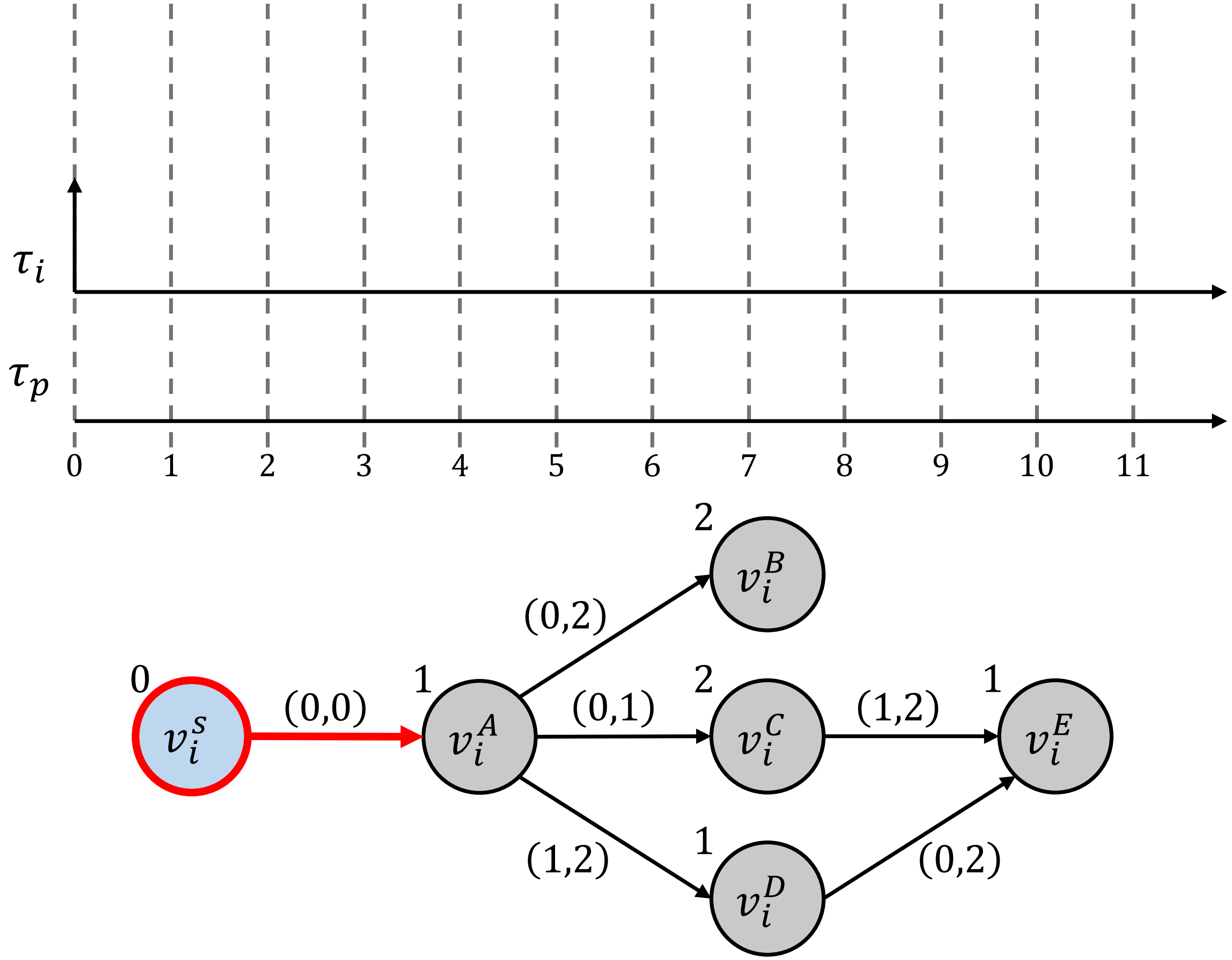
EDD task model

- All subtasks are released simultaneously at task release, but cannot execute until the incoming precedence constraints are satisfied
- If no subtask is ready for execution, the task is suspended

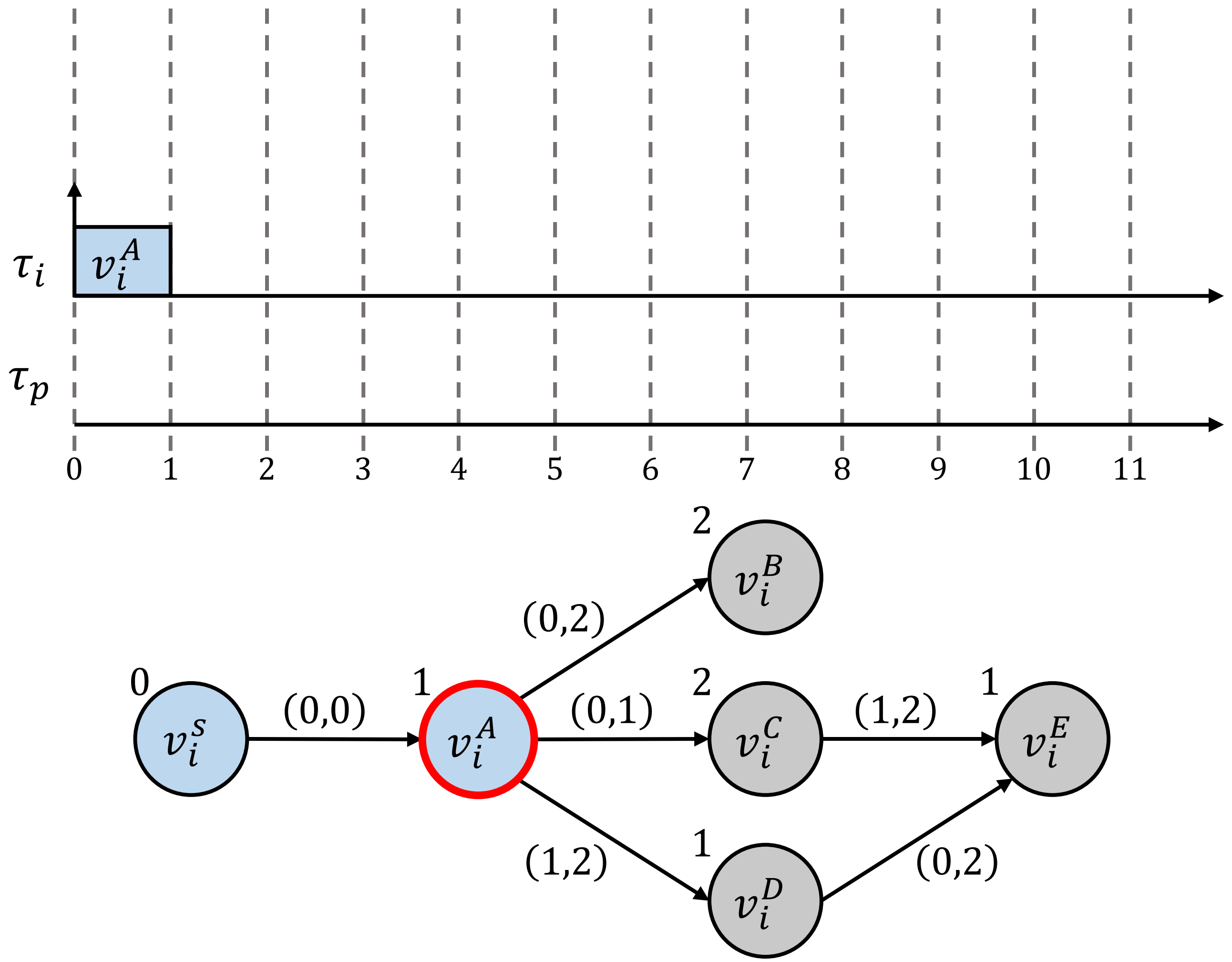
$$\tau_i = (G_i, T_i, D_i, \pi_i)$$



Example schedule

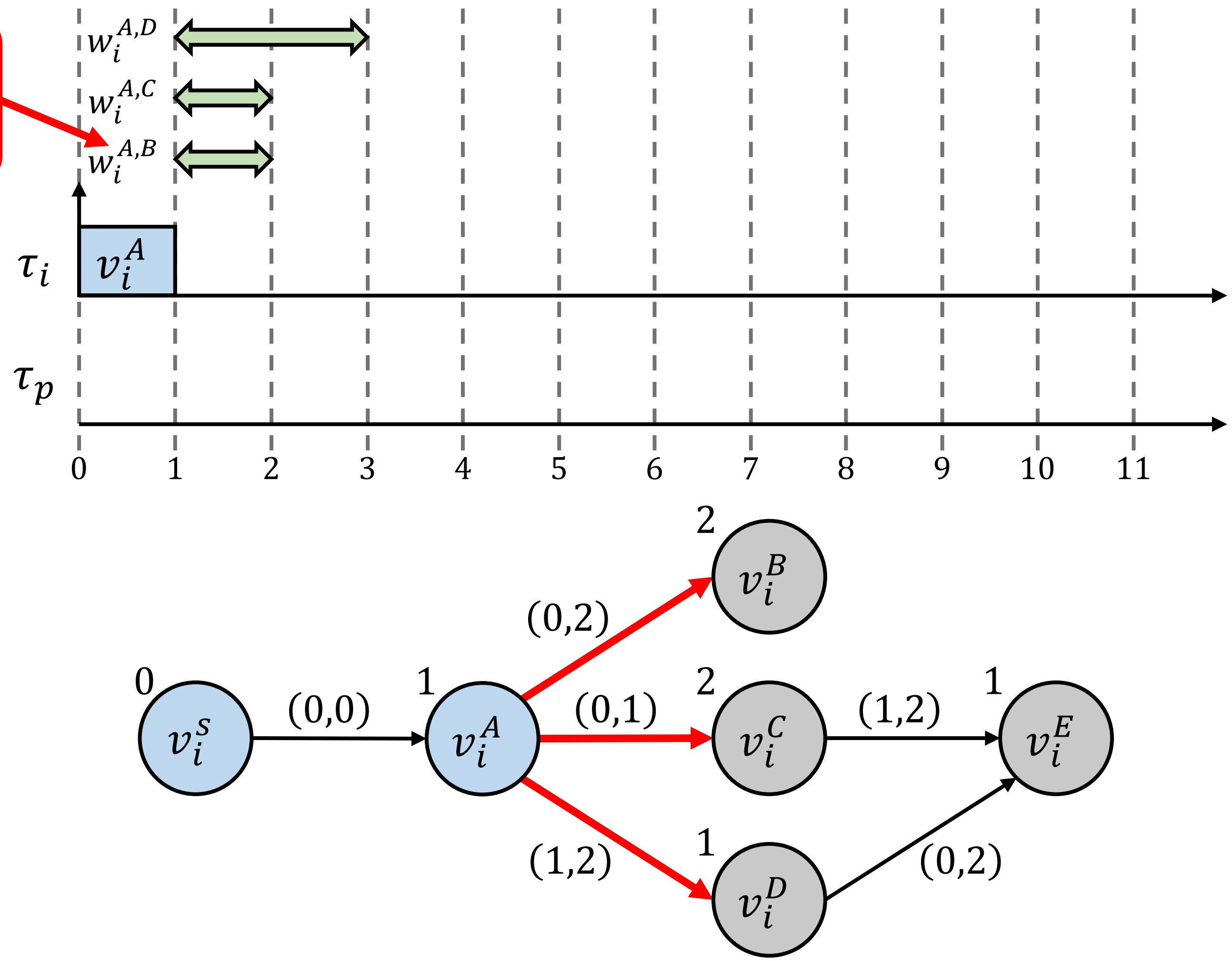


Example schedule



Example schedule

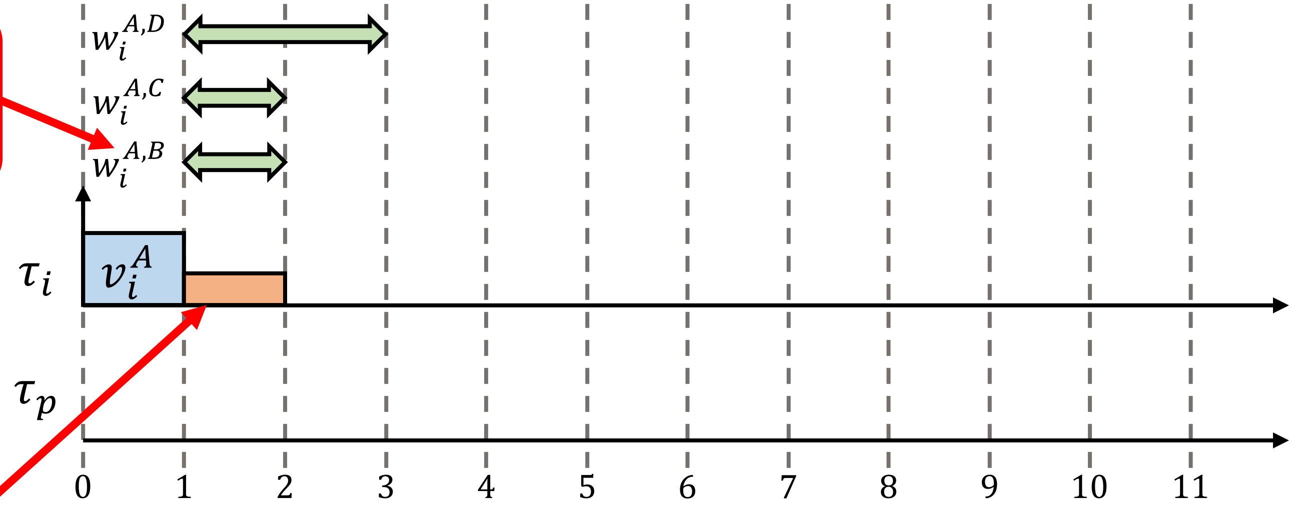
Event-related **delay**
$w_i^{A,B} \in [0,2]$

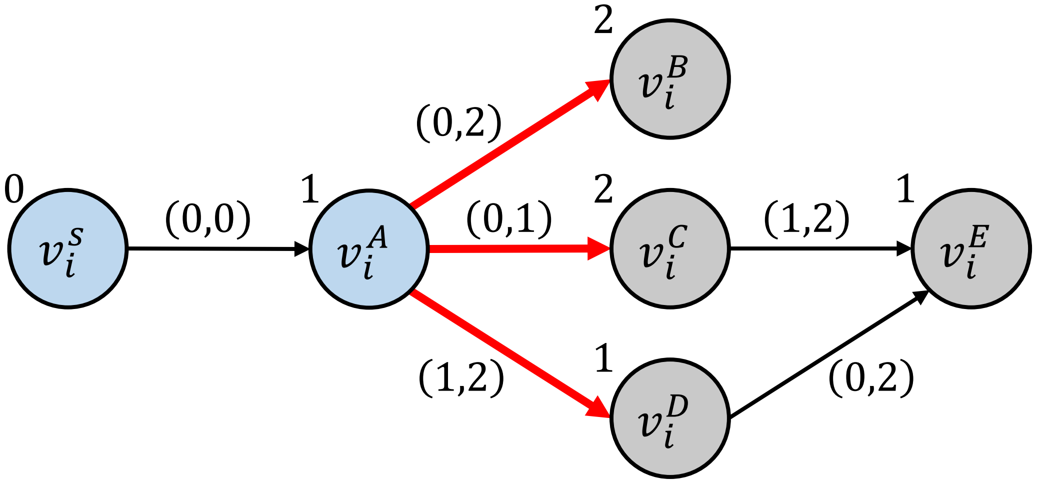


Example schedule

Event-related **delay** $w_i^{A,B} \in [0,2]$

Suspension: no subtask is ready for execution



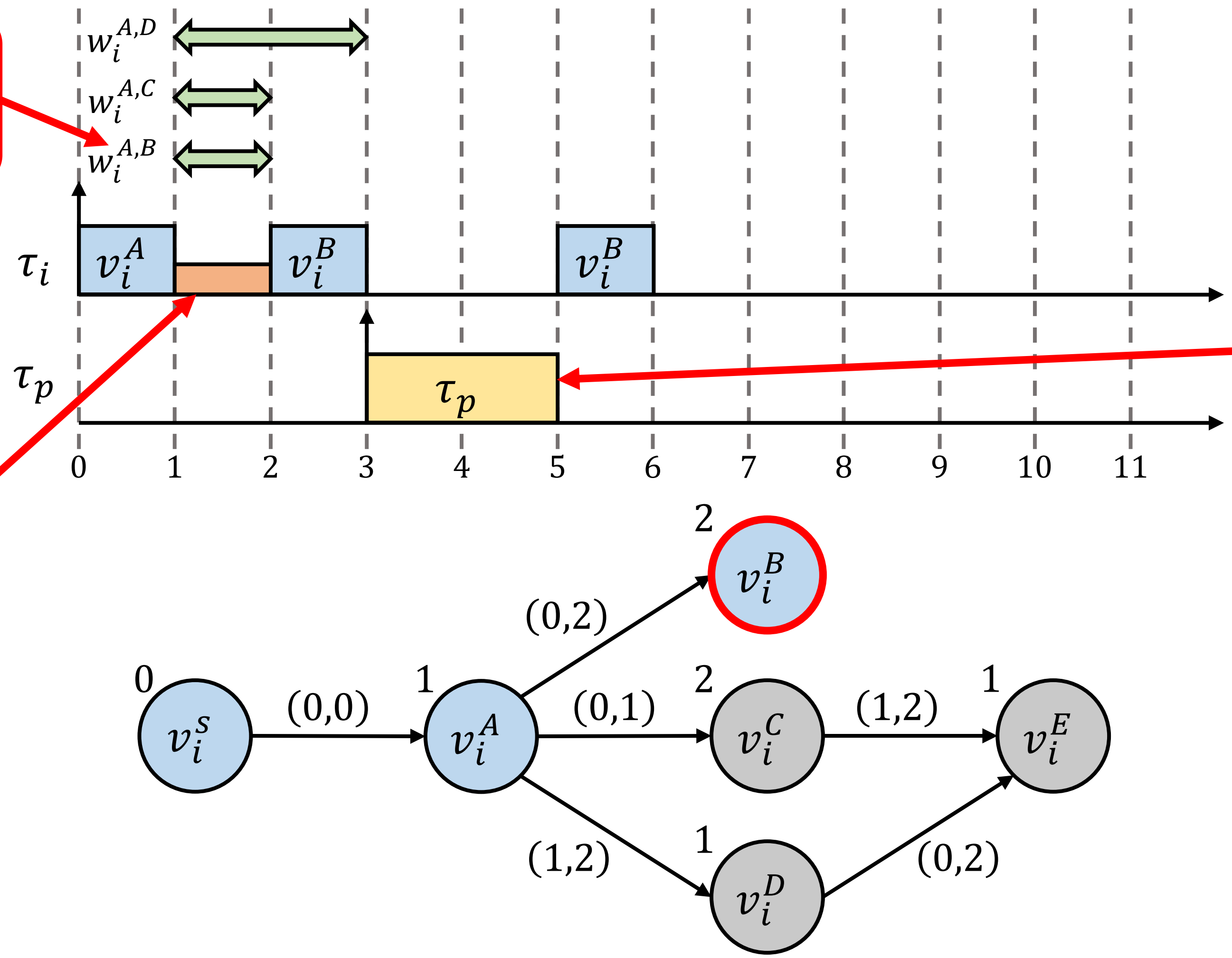


Example schedule

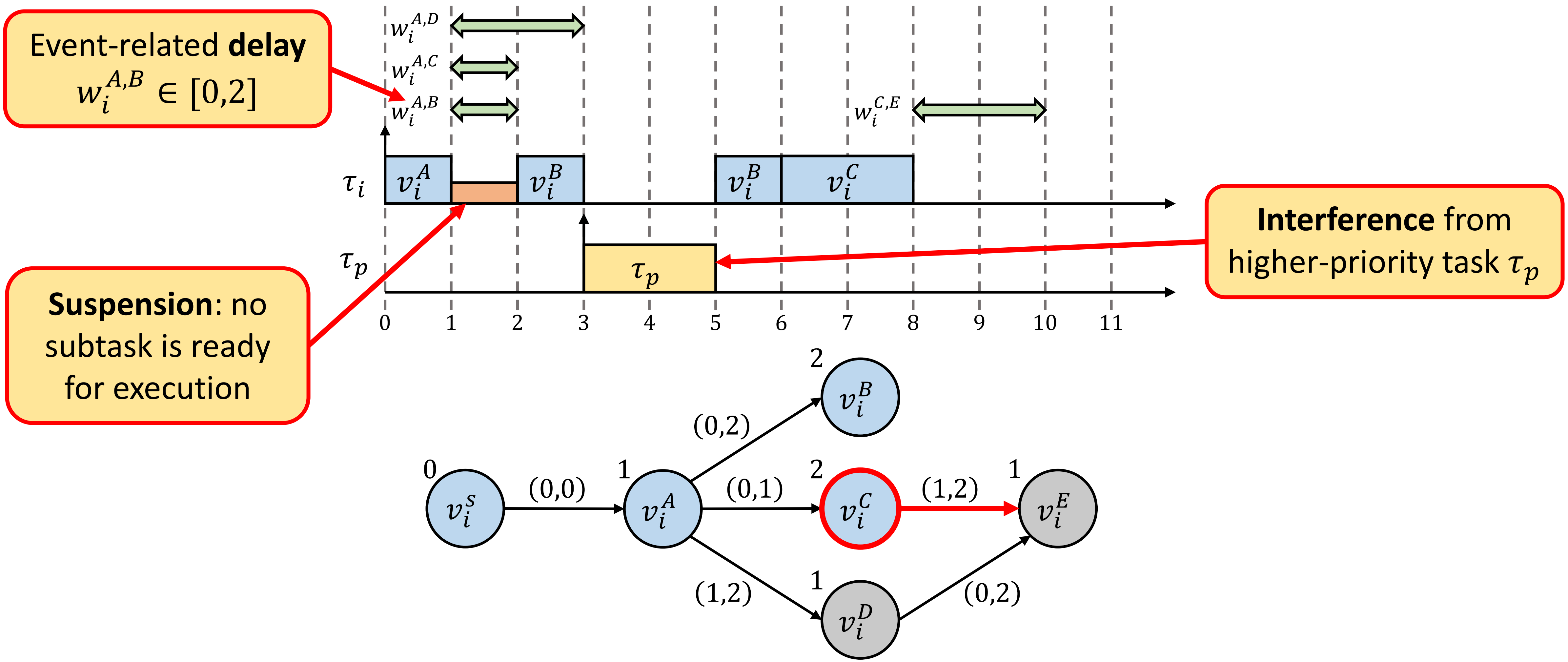
Event-related **delay** $w_i^{A,B} \in [0,2]$

Suspension: no subtask is ready for execution

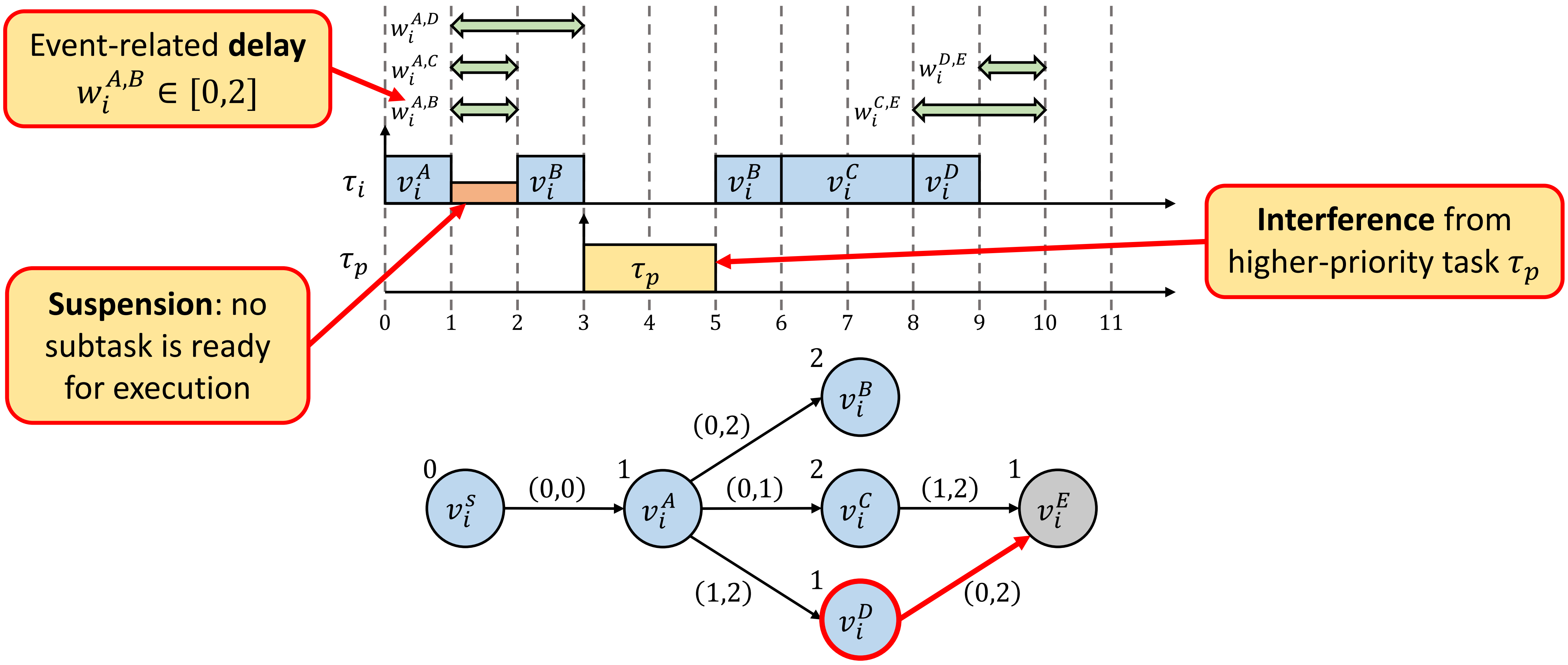
Interference from higher-priority task τ_p



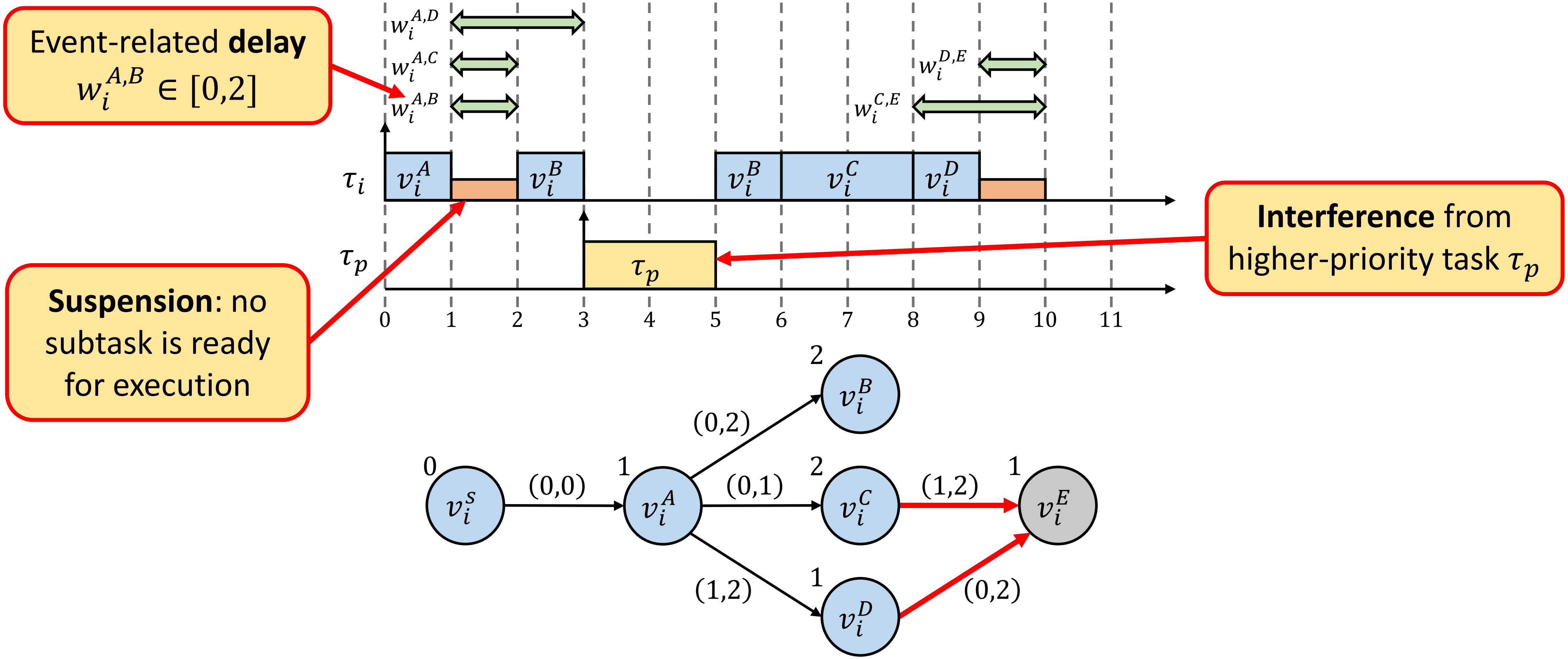
Example schedule



Example schedule



Example schedule



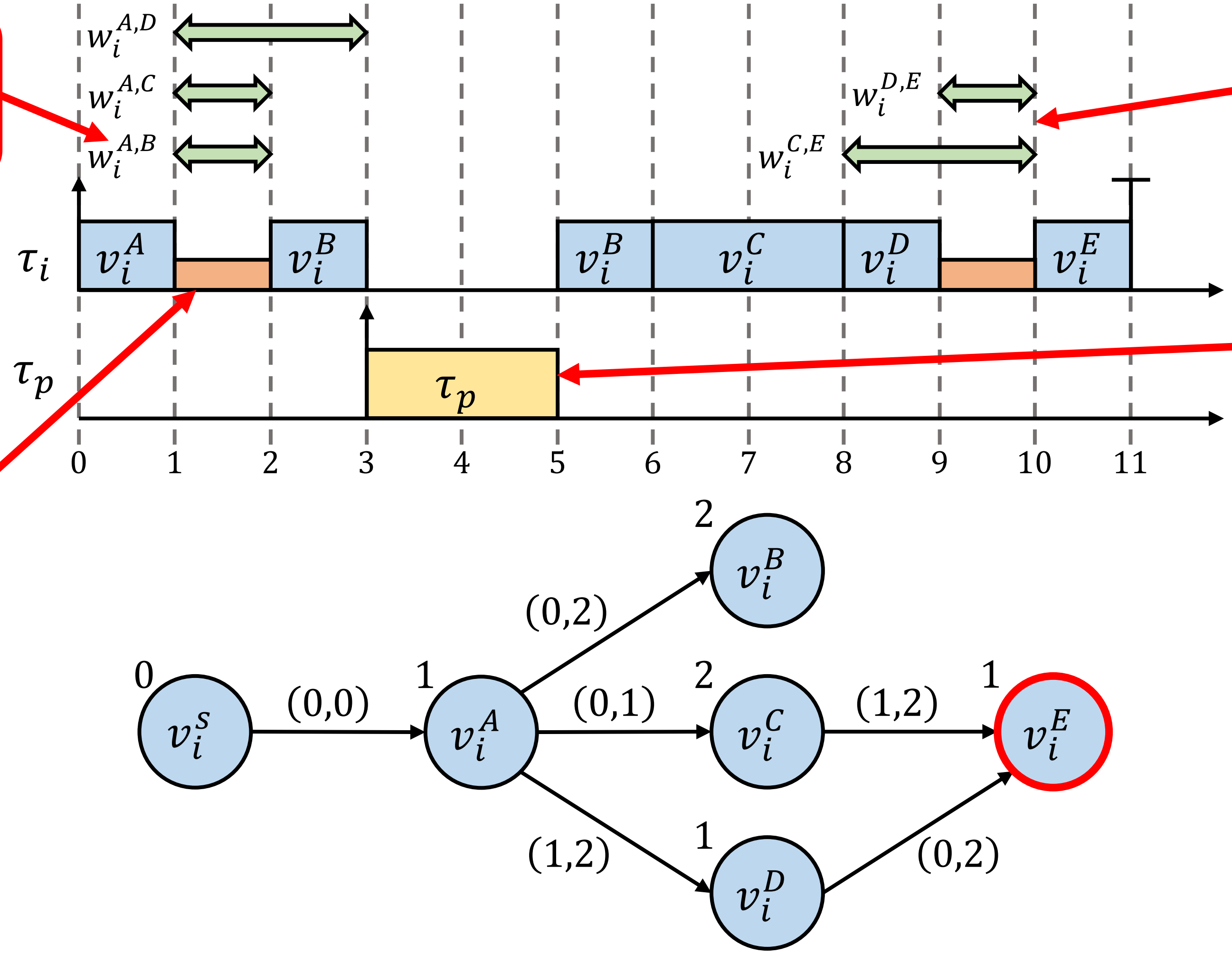
Example schedule

Event-related **delay** $w_i^{A,B} \in [0,2]$

Suspension: no subtask is ready for execution

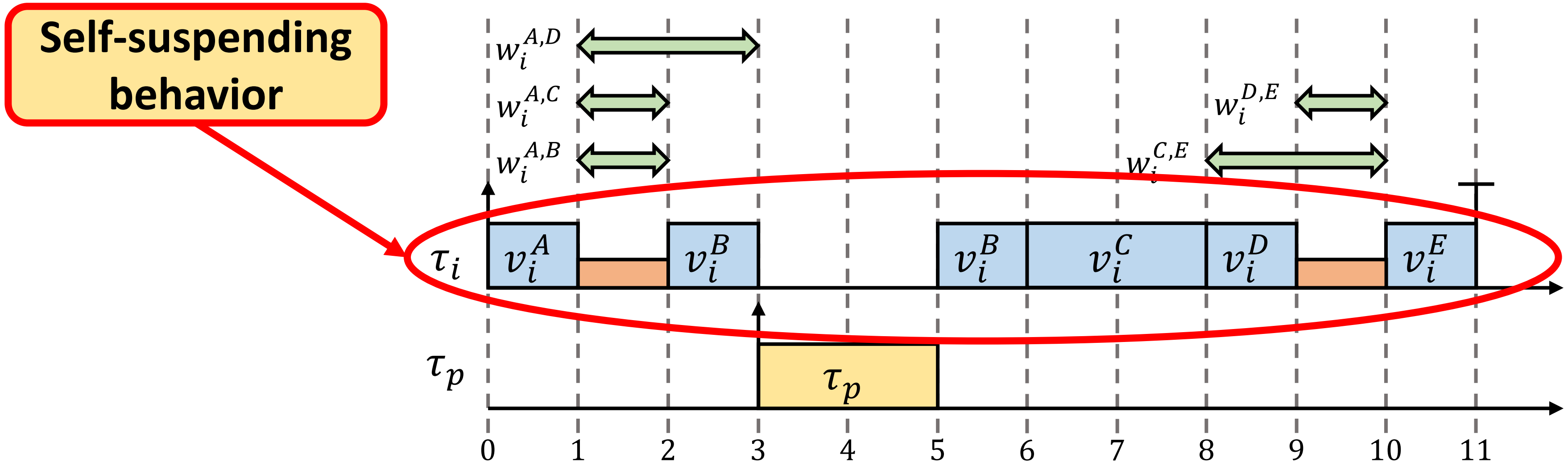
Both **precedence constraints** satisfied

Interference from higher-priority task τ_p



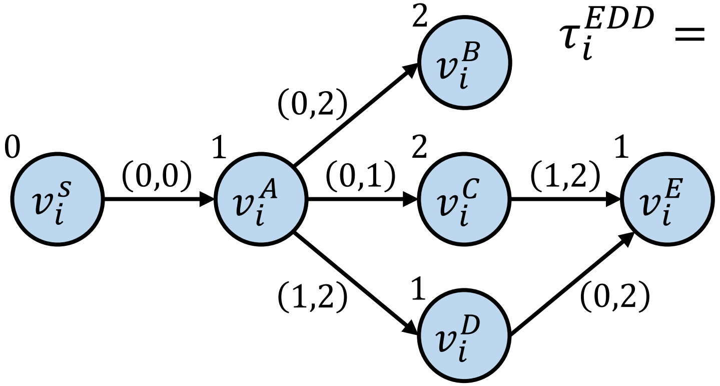
Analysis for EDD tasks

- **Problem**: obtain a response time analysis (RTA) for an EDD task set
  - Determine a worst-case response time (WCRT) upper bound $\bar{R}_i$ for each task
  - Verify if all deadlines are guaranteed: $\bar{R}_i \leq D_i$ for each τ_i
- **Observation**: the scheduling behavior on the processor alternates execution and suspension intervals



Closed-form analysis

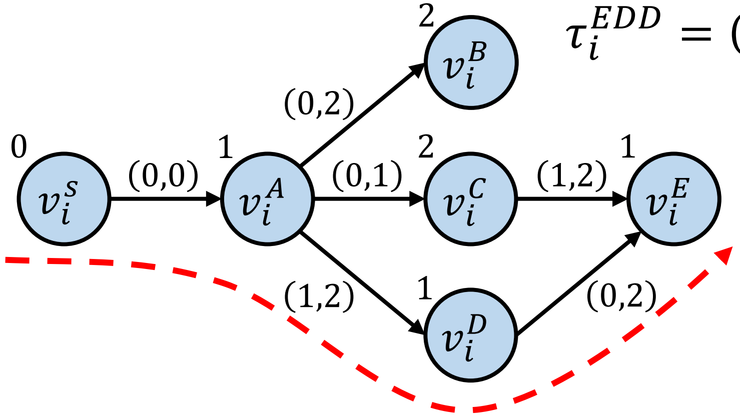
- Dynamic self-suspending (DSS) tasks *alternate execution and suspension phases* up to a cumulative WCET C_i and a cumulative suspension time S_i
- **Theorem**: **the timing behavior of an EDD task can be safely modeled by a DSS task** with
  - WCET equal to the sum of the WCETs of all DAG nodes
  - Maximum suspension time equal to the maximum delay encountered over any path



$\tau_i^{EDD} = (G_i, T_i, D_i, \pi_i)$ **EDD to DSS** $\tau_i^{DSS} = (C_i', S_i', D_i, T_i, \pi_i)$

Closed-form analysis

- Dynamic self-suspending (DSS) tasks *alternate execution and suspension phases* up to a cumulative WCET C_i and a cumulative suspension time S_i
- **Theorem**: **the timing behavior of an EDD task can be safely modeled by a DSS task** with
  - WCET equal to the sum of the WCETs of all DAG nodes
  - Maximum suspension time equal to the maximum delay encountered over any path



$$\tau_i^{EDD} = (G_i, T_i, D_i, \pi_i)$$

EDD to DSS

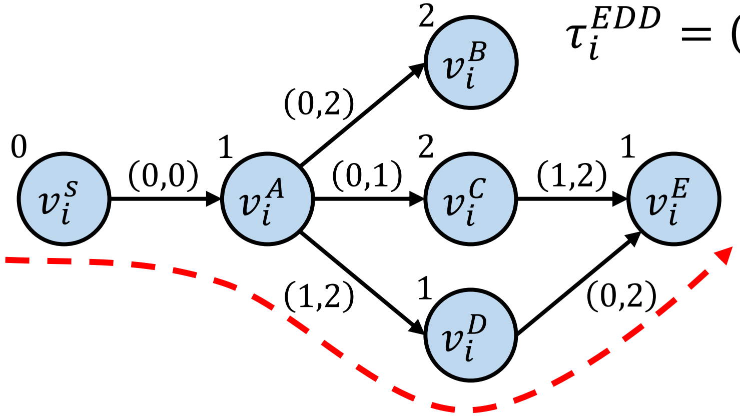
$$\tau_i^{DSS} = (C_i', S_i', D_i, T_i, \pi_i)$$

$$C_i' = \sum_{v_i^a \in V_i} C_i^a = 7$$

$$S_i' = \max_{\lambda \in PATH(G_i)} \left\{ \sum_{e_i^{a,b} \in E(\lambda)} W_i^{a,b} \right\} = 4$$

Closed-form analysis

- Dynamic self-suspending (DSS) tasks *alternate execution and suspension phases* up to a cumulative WCET C_i and a cumulative suspension time S_i
- **Theorem**: **the timing behavior of an EDD task can be safely modeled by a DSS task** with
  - WCET equal to the sum of the WCETs of all DAG nodes
  - Maximum suspension time equal to the maximum delay encountered over any path



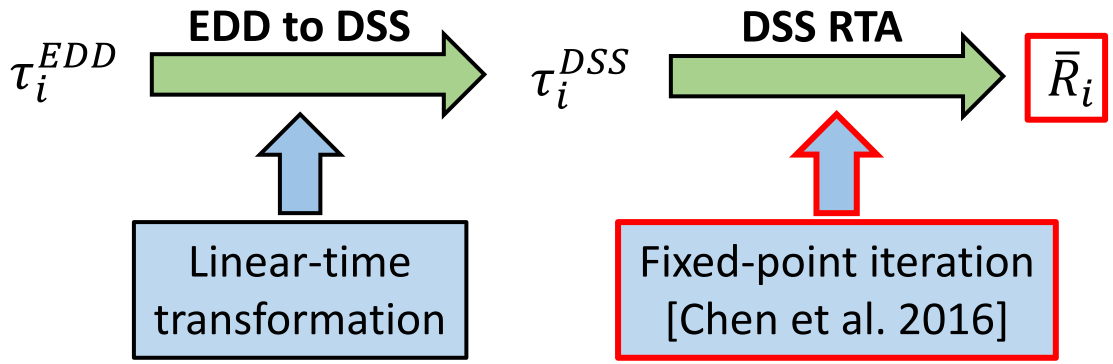
$\tau_i^{EDD} = (G_i, T_i, D_i, \pi_i)$ **EDD to DSS** $\tau_i^{DSS} = (C_i', S_i', D_i, T_i, \pi_i)$

$$C_i' = \sum_{v_i^a \in V_i} C_i^a = 7$$

$$S_i' = \max_{\lambda \in PATH(G_i)} \left\{ \sum_{e_i^{a,b} \in E(\lambda)} W_i^{a,b} \right\} = 4$$

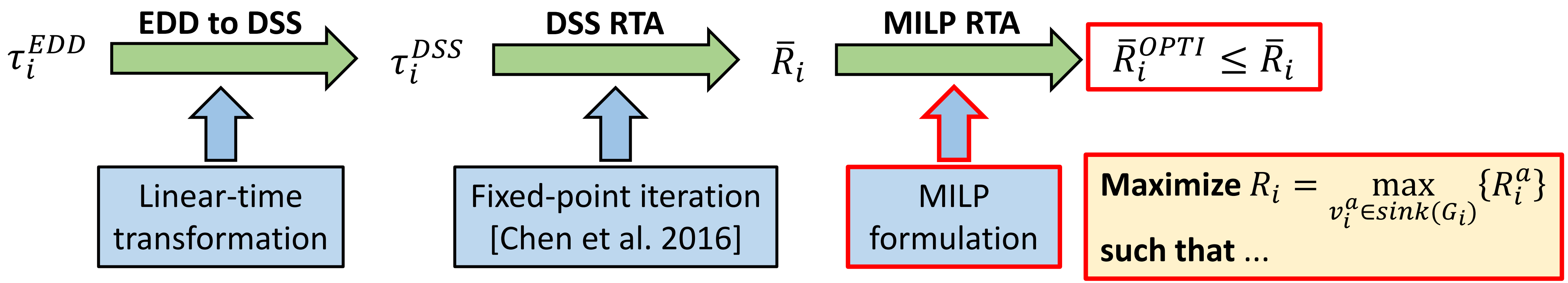
Closed-form analysis

- The resulting DSS tasks can be analyzed by means of a DSS RTA [Chen et al., 2016] to **obtain WCRT upper bounds** $\bar{R}_i$ for each task
- A node-level analysis is also presented to obtain WCRT UBs $\bar{R}_i^a$ for each node
- Pseudo-polynomial time complexity
- **Note:** the transformation is **compatible with both FP and EDF scheduling**



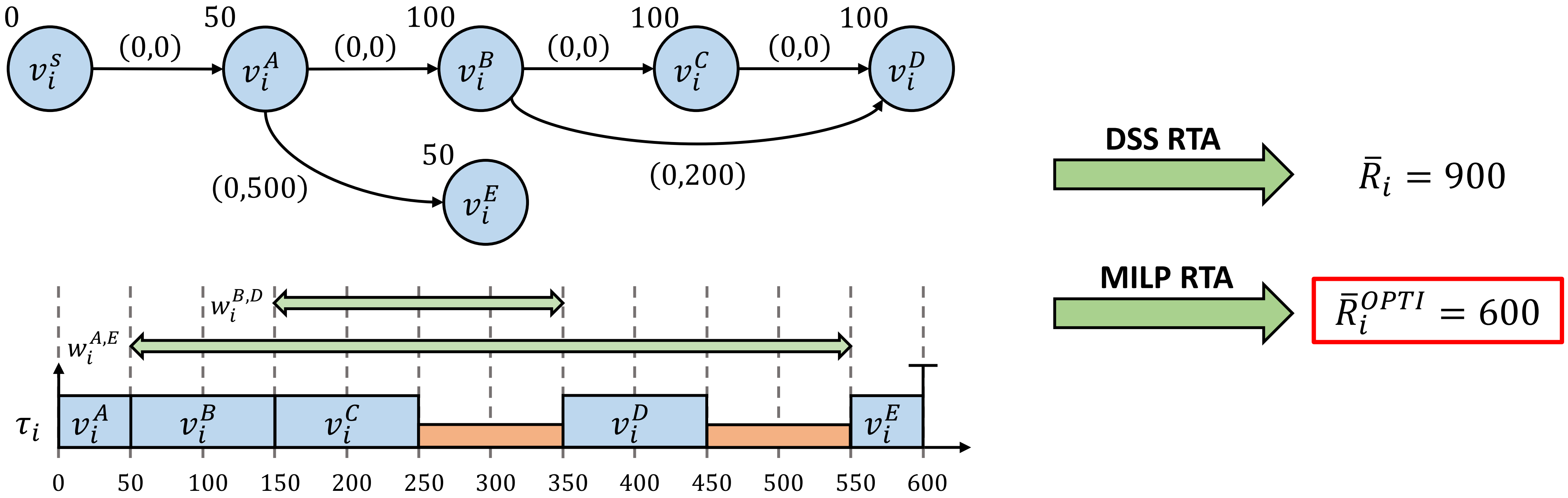
Optimization-based analysis

- A mixed-integer linear programming (MILP) formulation is proposed to **improve upon the WCRT UBs** obtained with the closed-form RTA
  - The MILP models a generic schedule for the task under analysis
  - Objective function: maximize the response time among sink nodes
  - Constraints: impose necessary conditions to exclude impossible schedules



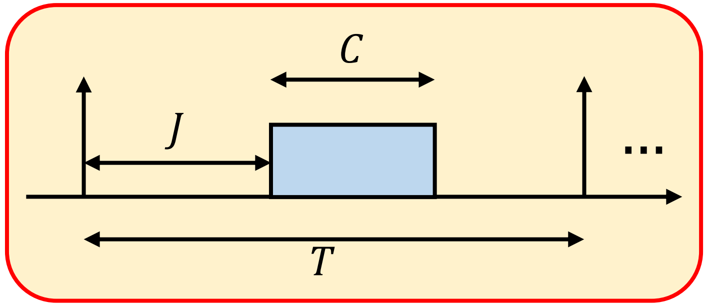
RTA comparison

- **Example:** consider an EDD task τ_i with $T_i = 1000$ executing in isolation
  - The DSS-based RTA gives a WCRT UB of $\bar{R}_i = 900$, since, in this case, $C_i' = 400$ and $S_i' = 500$
  - The MILP-based RTA can more accurately account for the specific DAG topology, giving a WCRT UB of $\bar{R}_i^{OPTI} = 600$
  - In fact, nodes v_i^B, v_i^C and v_i^D can execute even if the event triggering v_i^E has not yet occurred



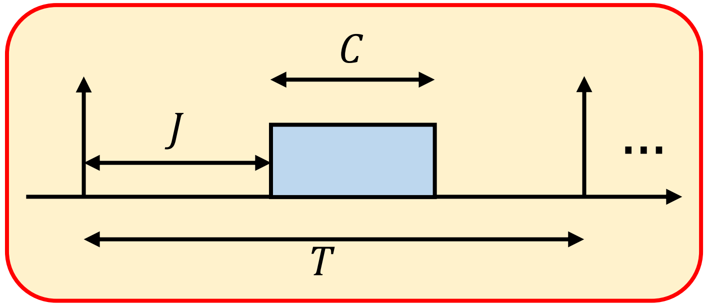
Generalization of other task models

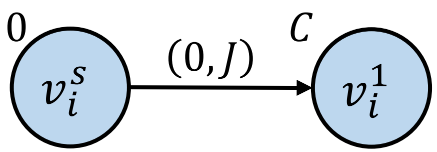
- **Sequential sporadic tasks with release jitter**
  - Sporadic release with jitter J and WCET C



Generalization of other task models

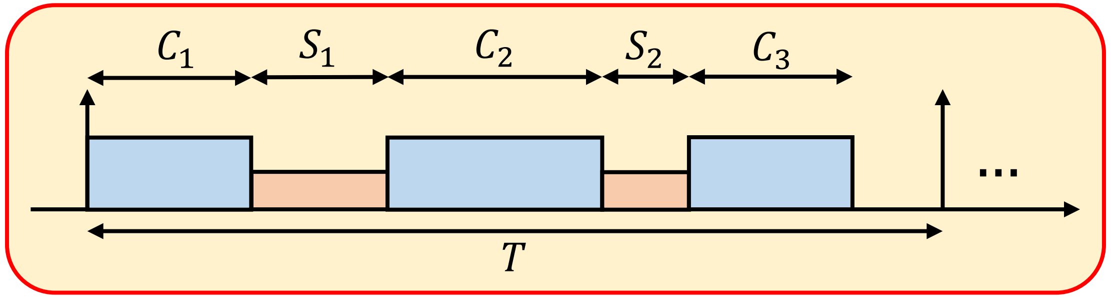
- **Sequential sporadic tasks with release jitter**
  - Sporadic release with jitter J and WCET C
  - Can be represented with a node with WCET C, and an edge with label $(0, J)$ incoming from the source node





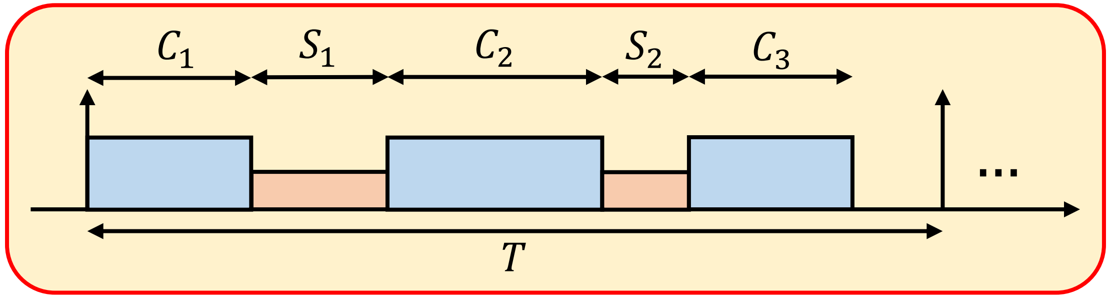
Generalization of other task models

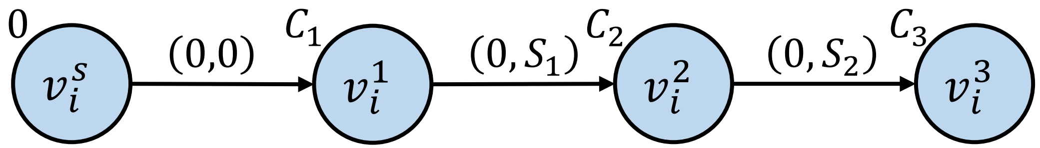
- **Segmented self-suspending tasks**
  - Alternate executions and suspensions with a given pattern: $(C_1, S_1, C_2, S_2, \dots, C_k)$
  - S_j: worst-case suspension time between successive subtasks



Generalization of other task models

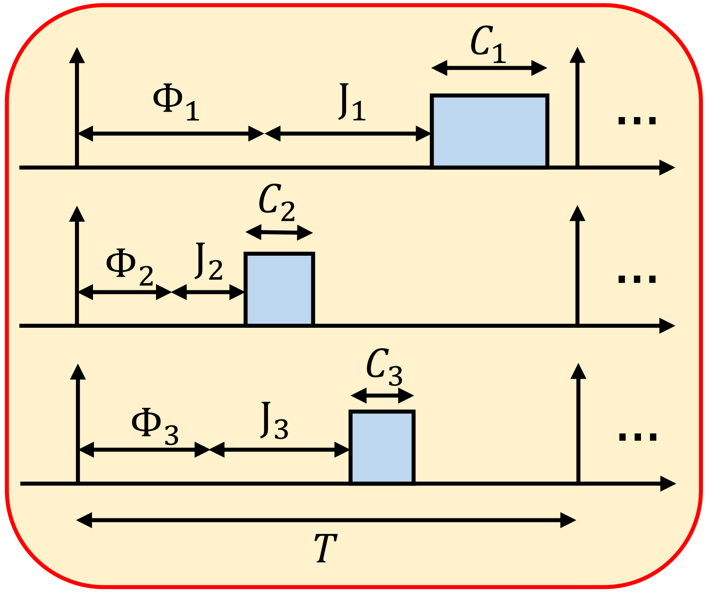
- **Segmented self-suspending tasks**
  - Alternate executions and suspensions with a given pattern: $(C_1, S_1, C_2, S_2, \dots, C_k)$
  - S_j: worst-case suspension time between successive subtasks
  - Can be represented with a linear DAG with $(0, S_j)$ labels on the edges





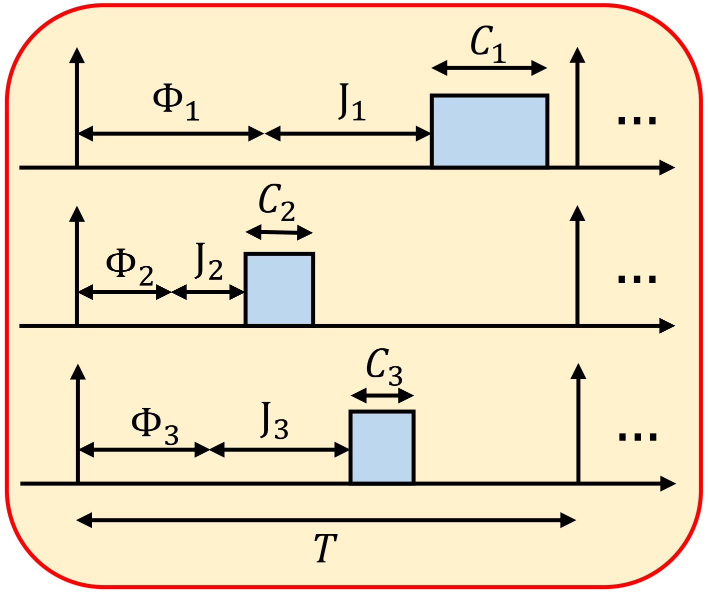
Generalization of other task models

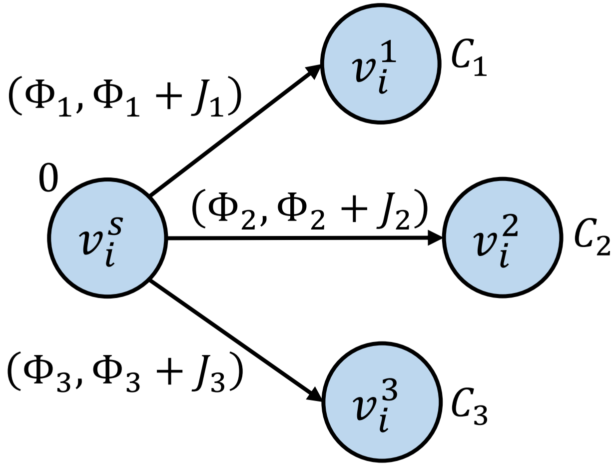
- **Transactional tasks with offsets**
  - Collection of independent subtasks released with fixed offset Φ_j and variable jitter J_j, relative to task release



Generalization of other task models

- **Transactional tasks with offsets**
  - Collection of independent subtasks released with fixed offset Φ_j and variable jitter J_j, relative to task release
  - Can be represented with one node for each subtask in the transaction, each connected to the source node with labels $\left(\Phi_\mathrm{j}, \Phi_\mathrm{j} + J_j\right)$

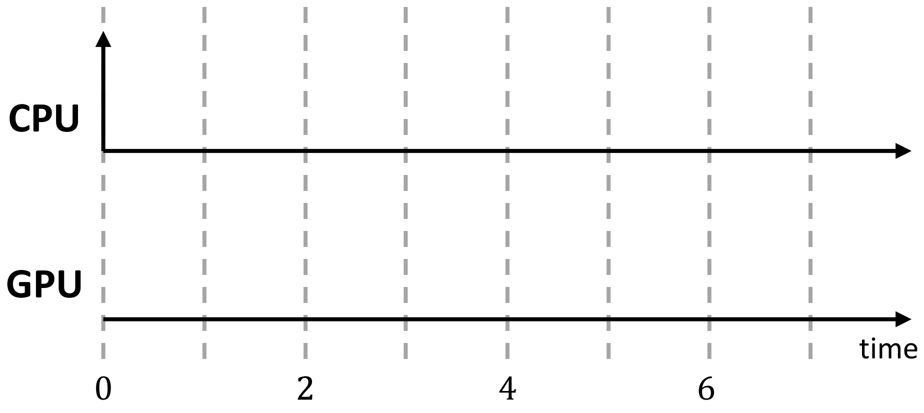




Modeling asynchronous GPU acceleration

- **Example**: asynchronous GPU acceleration with NVIDIA CUDA Runtime API

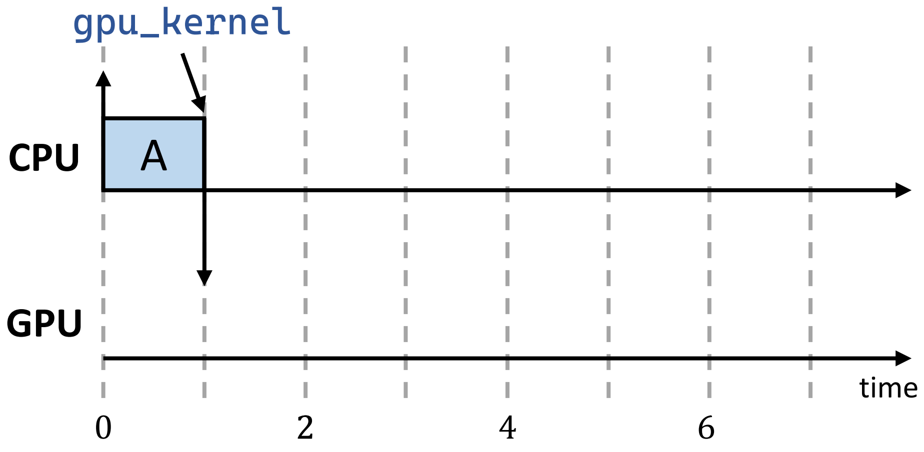
```
TASK(example)
{

}
```



Modeling asynchronous GPU acceleration

- **Example**: asynchronous GPU acceleration with NVIDIA CUDA Runtime API

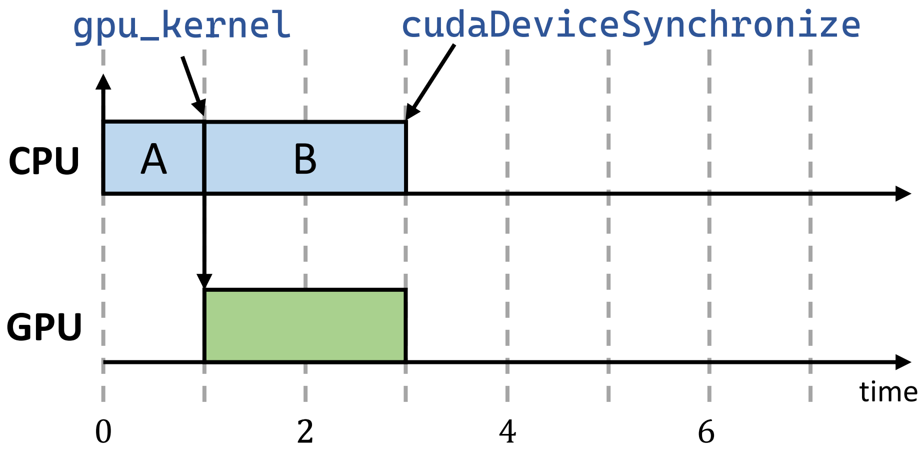
```
TASK(example)
{
    <execute on the CPU (A)>

    // asynchronously launch GPU kernel
    gpu_kernel<<<blocks, threads>>>();

}
```



Modeling asynchronous GPU acceleration

- **Example**: asynchronous GPU acceleration with NVIDIA CUDA Runtime API

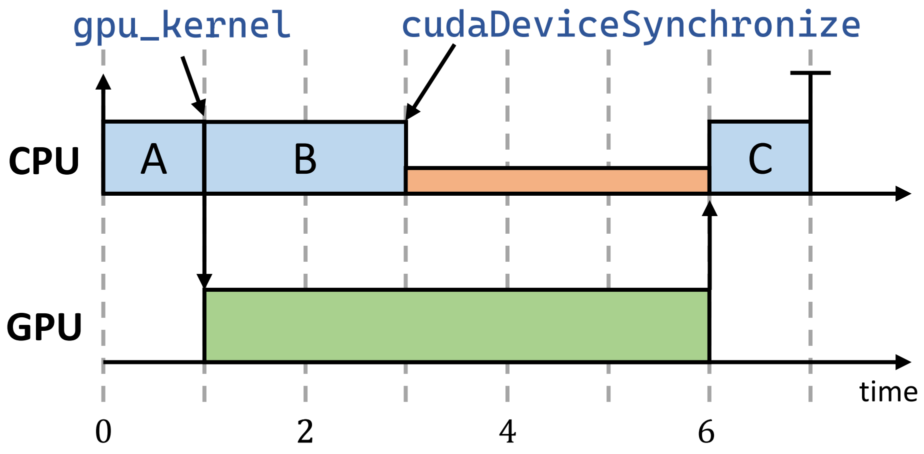
```
TASK(example)
{
    <execute on the CPU (A)>

    // asynchronously launch GPU kernel
    gpu_kernel<<<blocks, threads>>>();

    // execute in parallel on the CPU
    <execute on the CPU (B)>

    // wait for kernel completion
    cudaDeviceSynchronize();


}
```



Modeling asynchronous GPU acceleration

- **Example**: asynchronous GPU acceleration with NVIDIA CUDA Runtime API

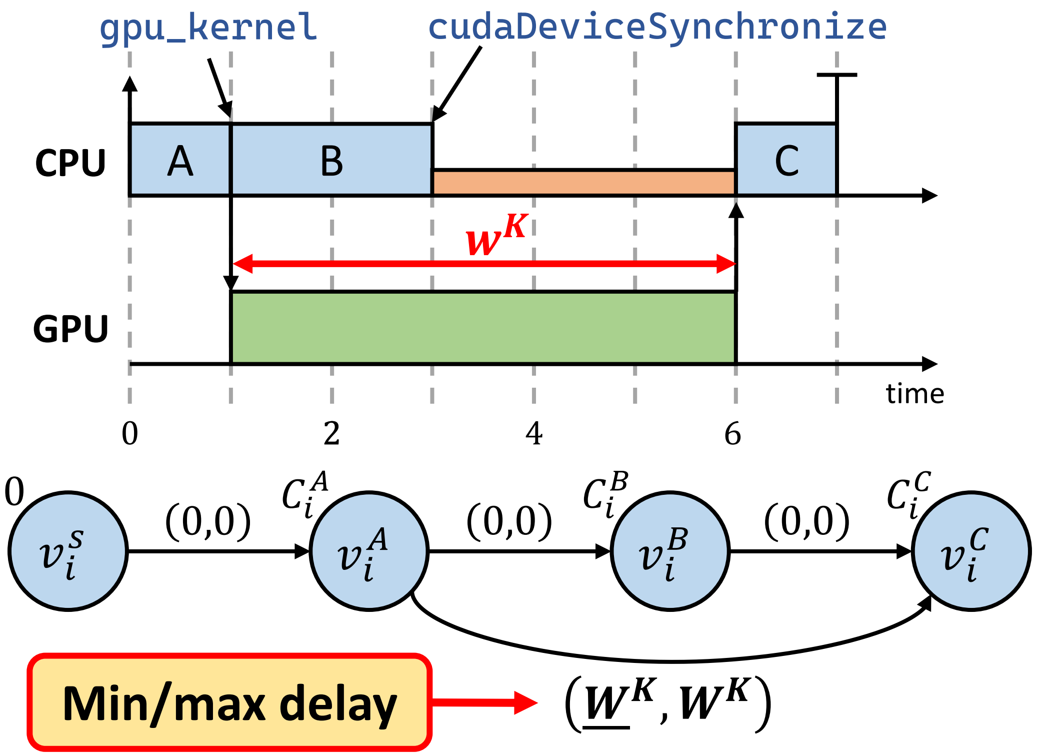
```
TASK(example)
{
    <execute on the CPU (A)>

    // asynchronously launch GPU kernel
    gpu_kernel<<<blocks, threads>>>();

    // execute in parallel on the CPU
    <execute on the CPU (B)>

    // wait for kernel completion
    cudaDeviceSynchronize();

    <execute on the CPU (C)>
}
```



Modeling asynchronous GPU acceleration

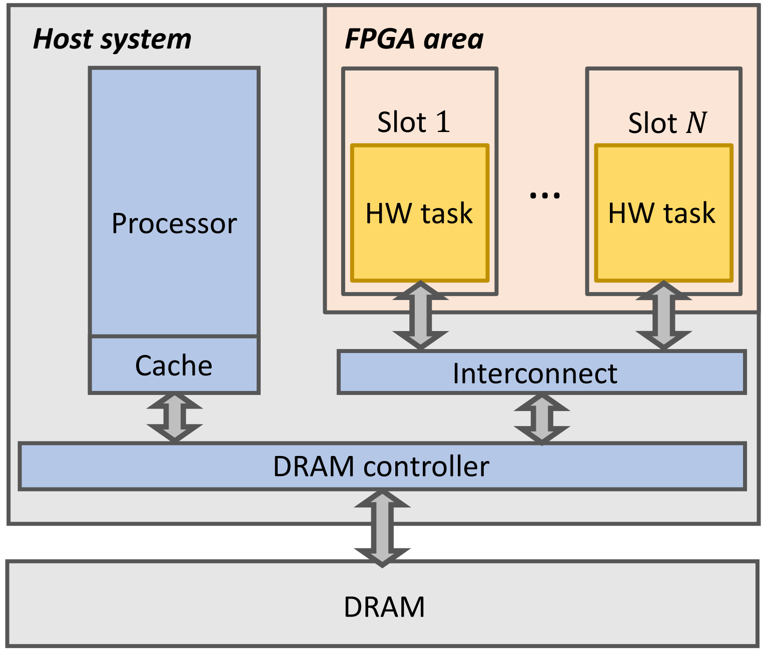
- **Example**: asynchronous GPU acceleration with NVIDIA CUDA Runtime API
- Modeled by an EDD task with nodes representing CPU execution and an edge with delay given by the min/max response time of the GPU kernel

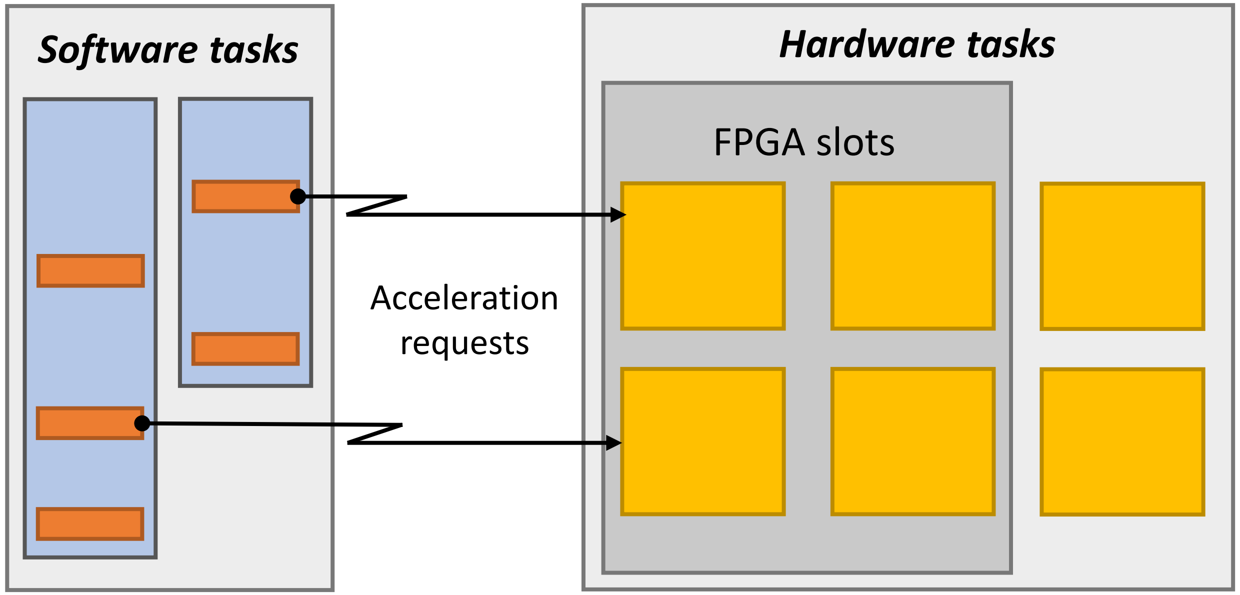
```
TASK(example)
{
    <execute on the CPU (A)>

    // asynchronously launch GPU kernel
    gpu_kernel<<<blocks, threads>>>();

    // execute in parallel on the CPU
    <execute on the CPU (B)>

    // wait for kernel completion
    cudaDeviceSynchronize();

    <execute on the CPU (C)>
}
```



Modeling FPGA hardware acceleration

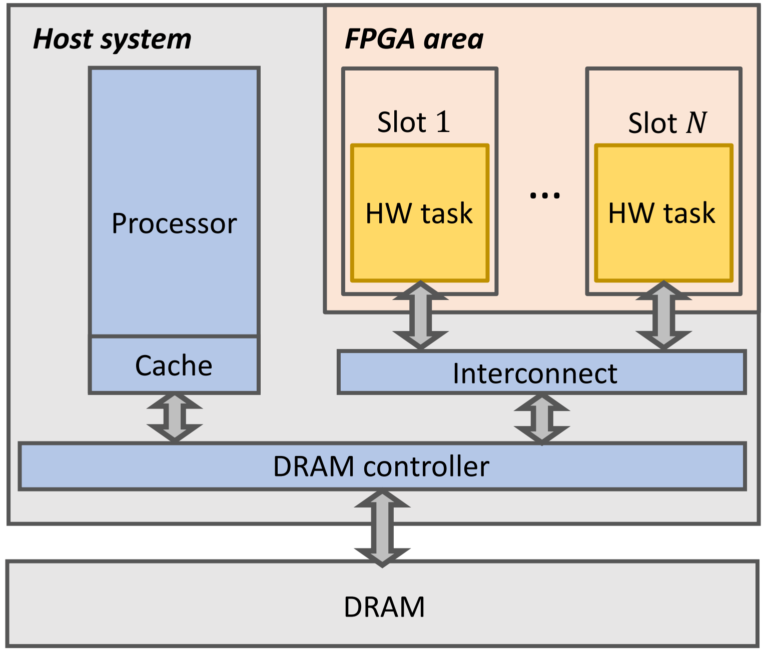
- **Example:** **FRED** is a scheduling framework for time-predictable FPGA hardware acceleration [Biondi et al. 2016]
  - The FPGA area is statically partitioned into slots of fixed size
  - Software tasks can request the execution of FPGA-accelerated functions (hardware tasks)
  - Dynamic partial reconfiguration (DPR) is leveraged to reconfigure the FPGA slots at runtime with different hardware tasks

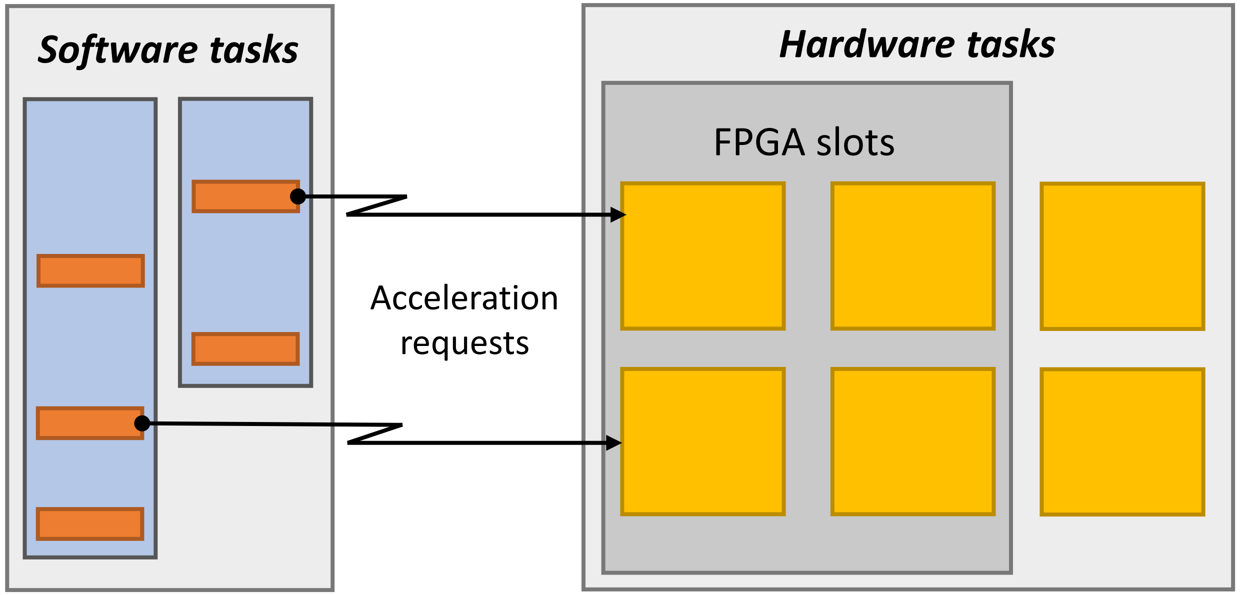




Modeling FPGA hardware acceleration

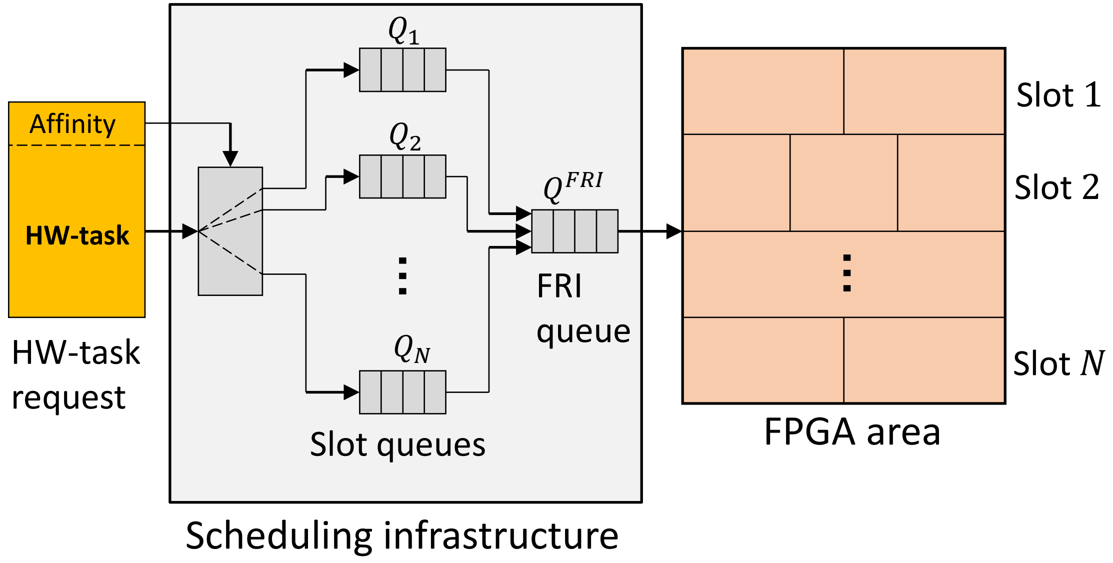
- The FRED framework **enables predictable time multiplexing of FPGA resources** to support sets of hardware tasks with total FPGA area requirements exceeding the physical area

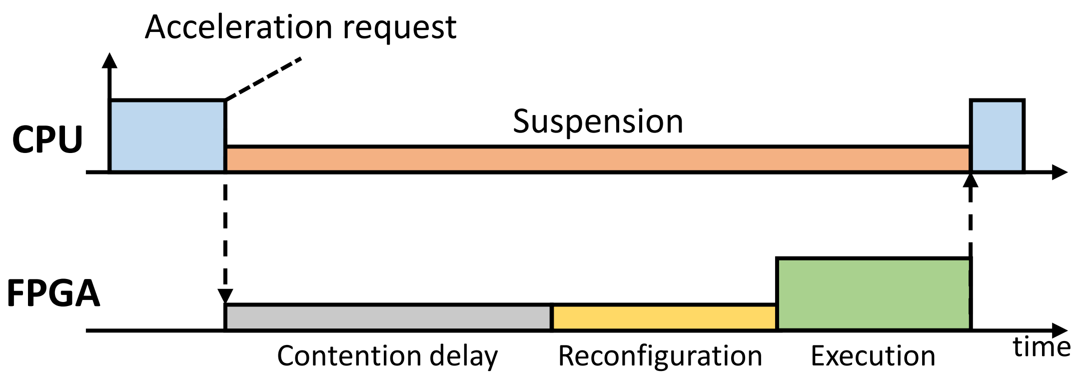




Modeling FPGA hardware acceleration

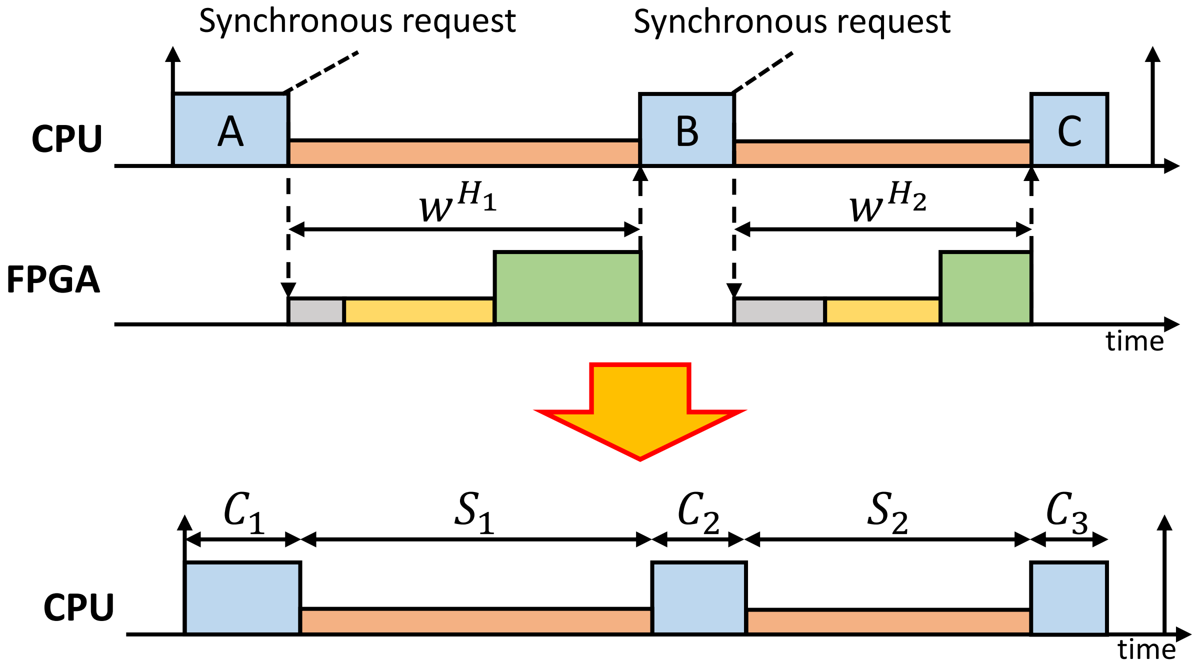
- Differently from GPU-based systems, the acceleration delays are decoupled from the software scheduling behavior, and can be upper bounded using a specialized timing analysis
  - Predictable access to shared resources (FPGA slots and FPGA reconfiguration interface) is guaranteed by a specialized scheduling infrastructure
  - The resulting suspension time is given by the sum of the resource contention delay, the slot reconfiguration time, and the execution time of the hardware task





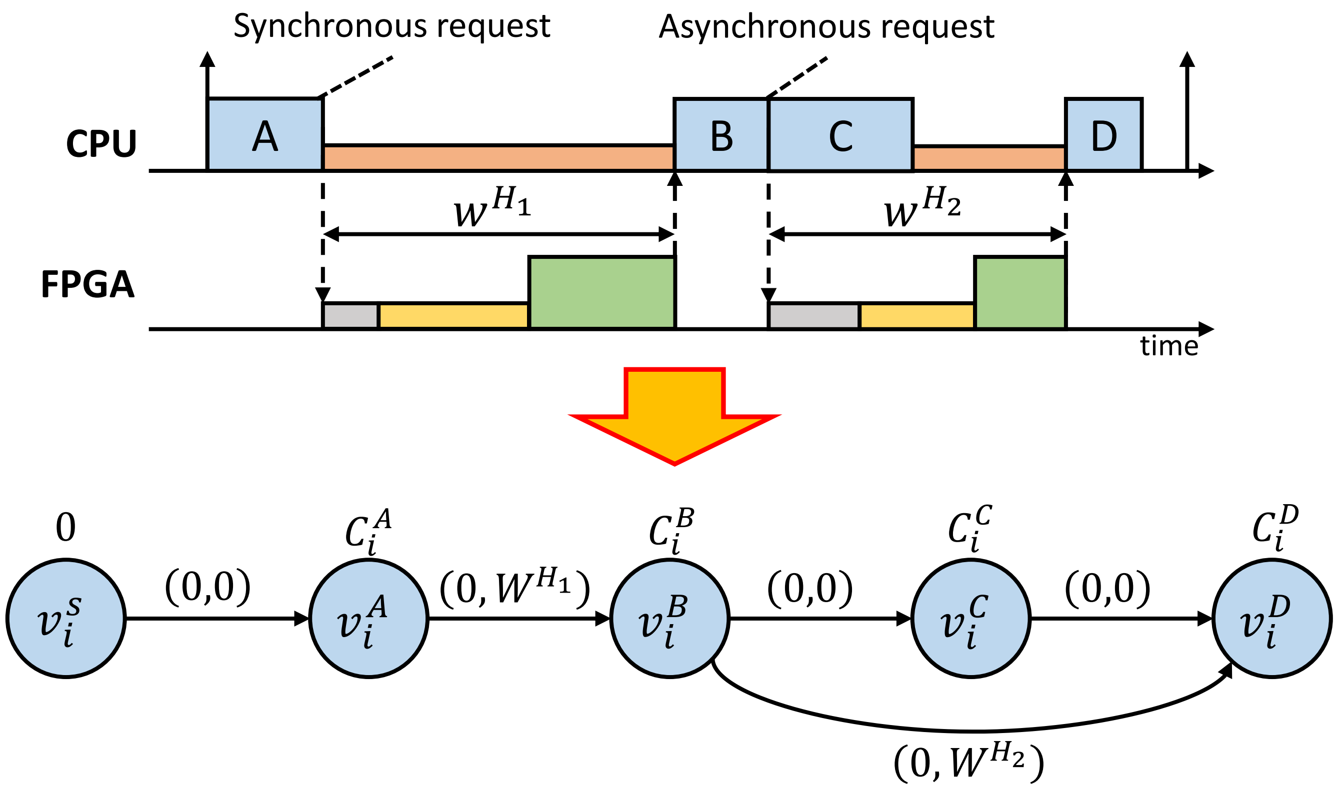
Modeling FPGA hardware acceleration

- In the overall timing analysis, software tasks are treated as segmented self-suspending tasks to account for multiple acceleration requests from each task
- This allows modeling the timing behavior of synchronous HW acceleration



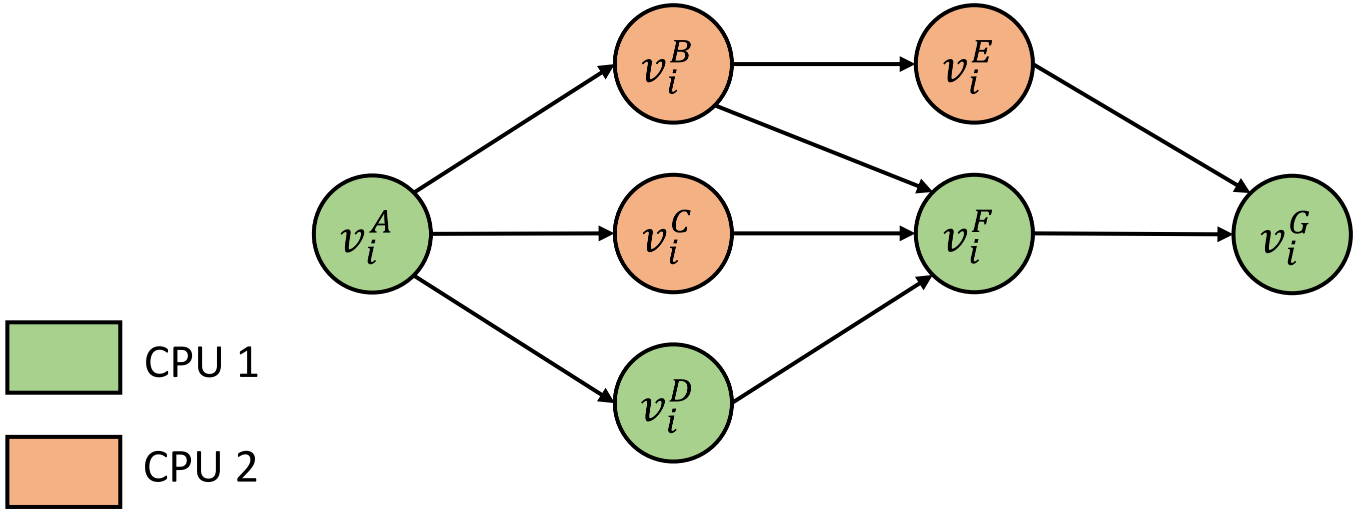
Modeling FPGA hardware acceleration

- The current implementation of the FRED framework is compatible with both synchronous and asynchronous acceleration
- Applying the EDD task model to the FRED timing analysis allows capturing more complex behaviors, including asynchronous acceleration requests



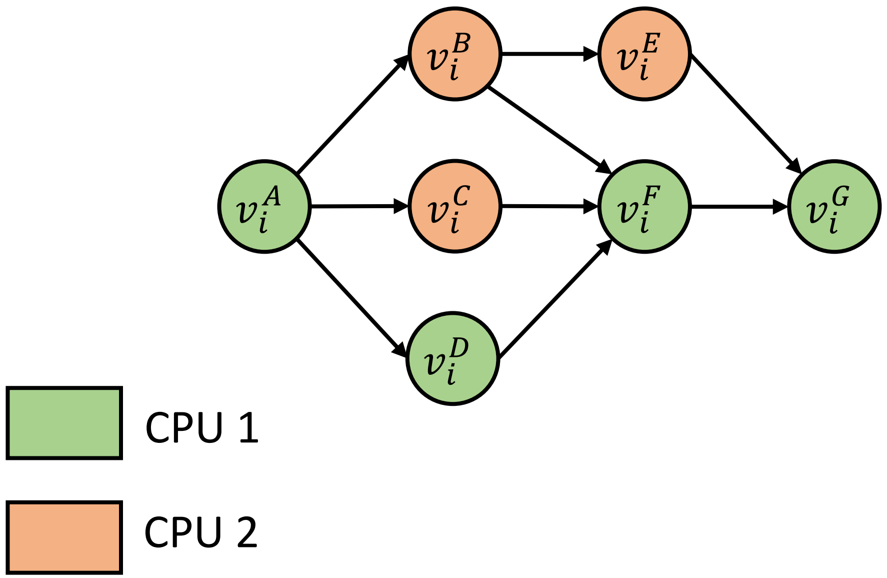
Modeling partitioned parallel DAG tasks

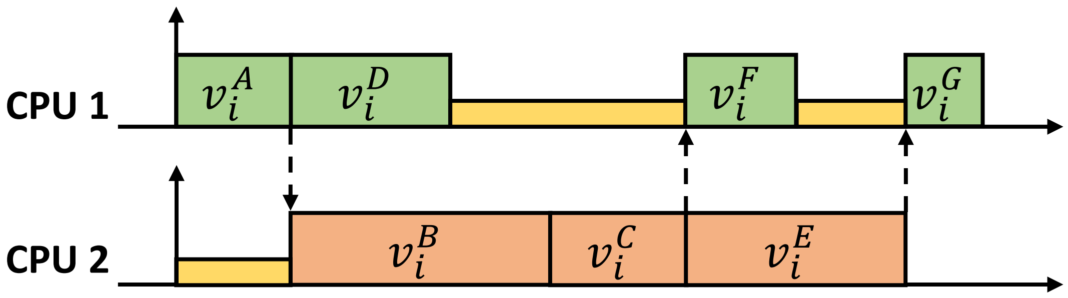
- **Partitioned parallel DAG tasks**:
  - Workload represented by a DAG executing on a multiprocessor system
  - Partitioned scheduling: each node is assigned to a specific processor
    - Nodes are scheduled according to a preemptive, fixed-priority uniprocessor policy



Modeling partitioned parallel DAG tasks

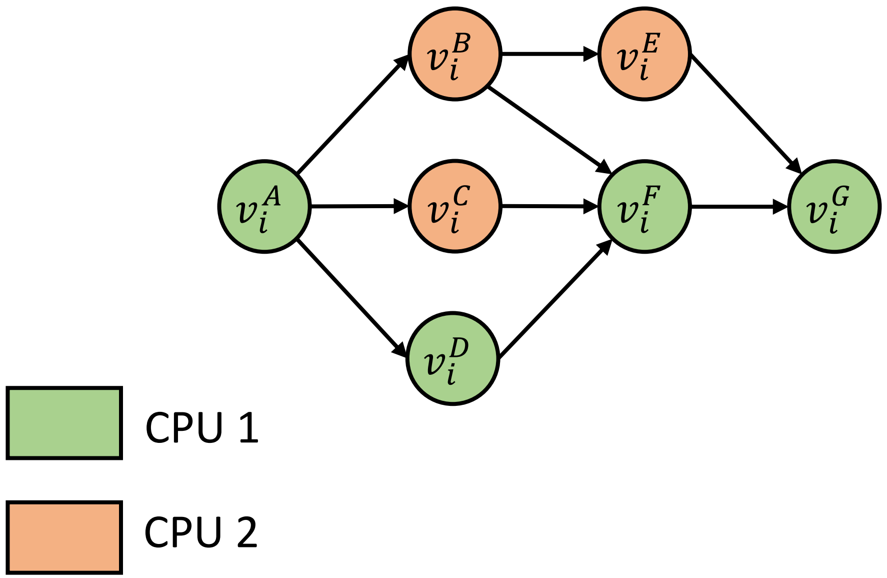
- **Application:** a **partitioned parallel task** can be modeled by a set of EDD tasks (one for each core) for the purpose of real-time analysis





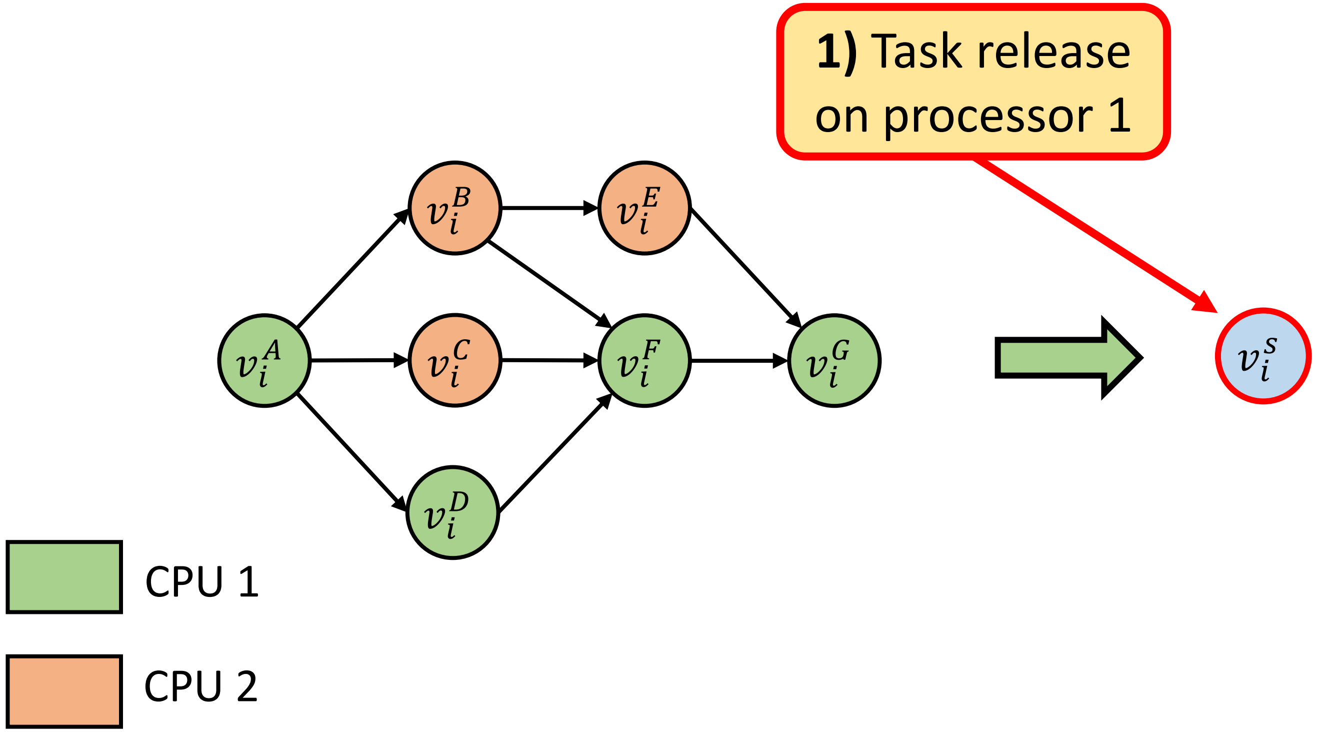
Modeling partitioned parallel DAG tasks

- The scheduling behavior of a parallel task τ^P on a processor P_k can be captured by an EDD task $\mathcal{P}_k(\tau^P)$
- **Projection on processor 1**:



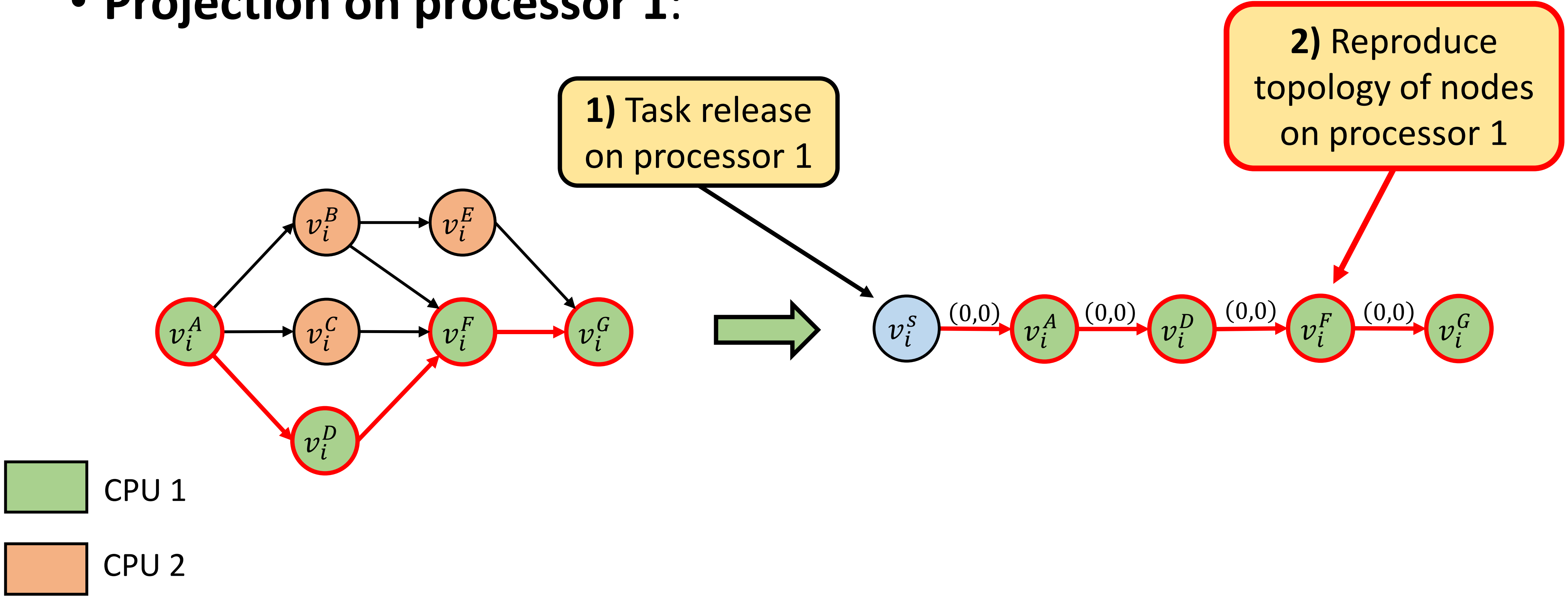
Modeling partitioned parallel DAG tasks

- The scheduling behavior of a parallel task τ^P on a processor P_k can be captured by an EDD task $\mathcal{P}_k(\tau^P)$
- **Projection on processor 1**:



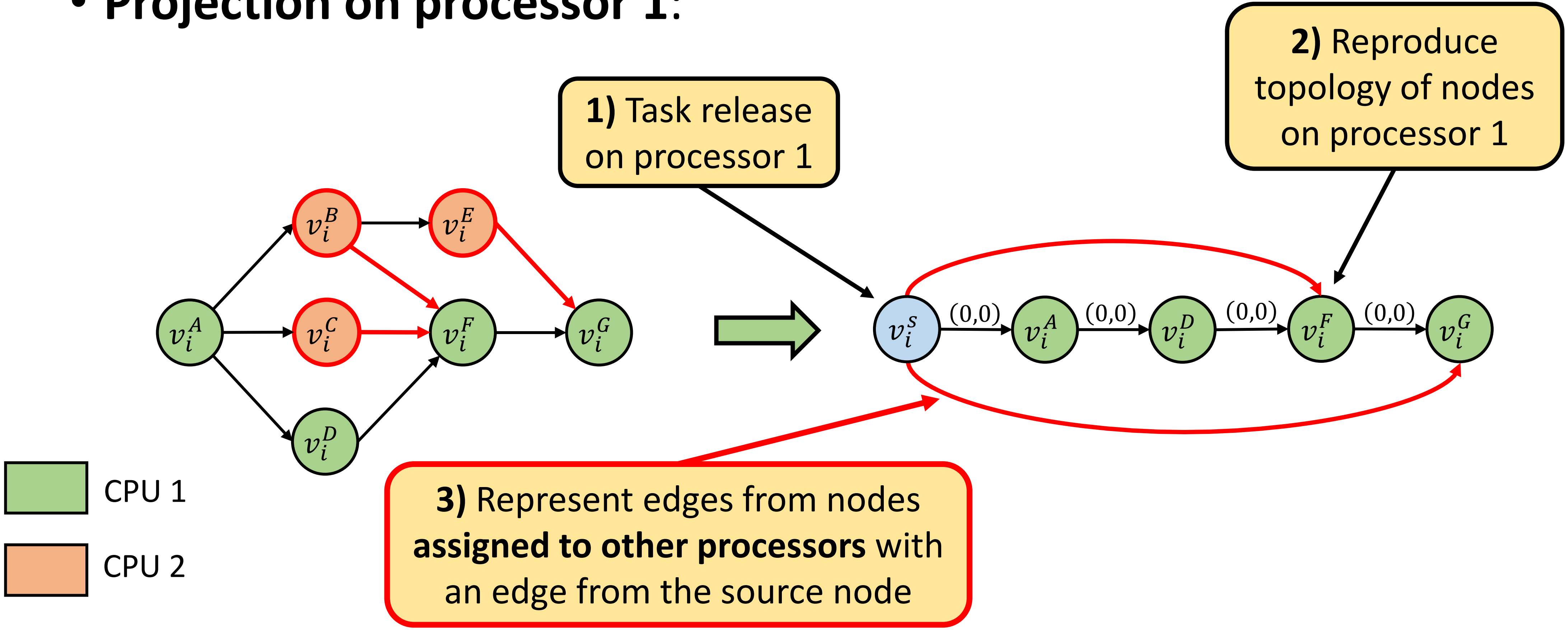
Modeling partitioned parallel DAG tasks

- The scheduling behavior of a parallel task τ^P on a processor P_k can be captured by an EDD task $\mathcal{P}_k(\tau^P)$
- **Projection on processor 1**:



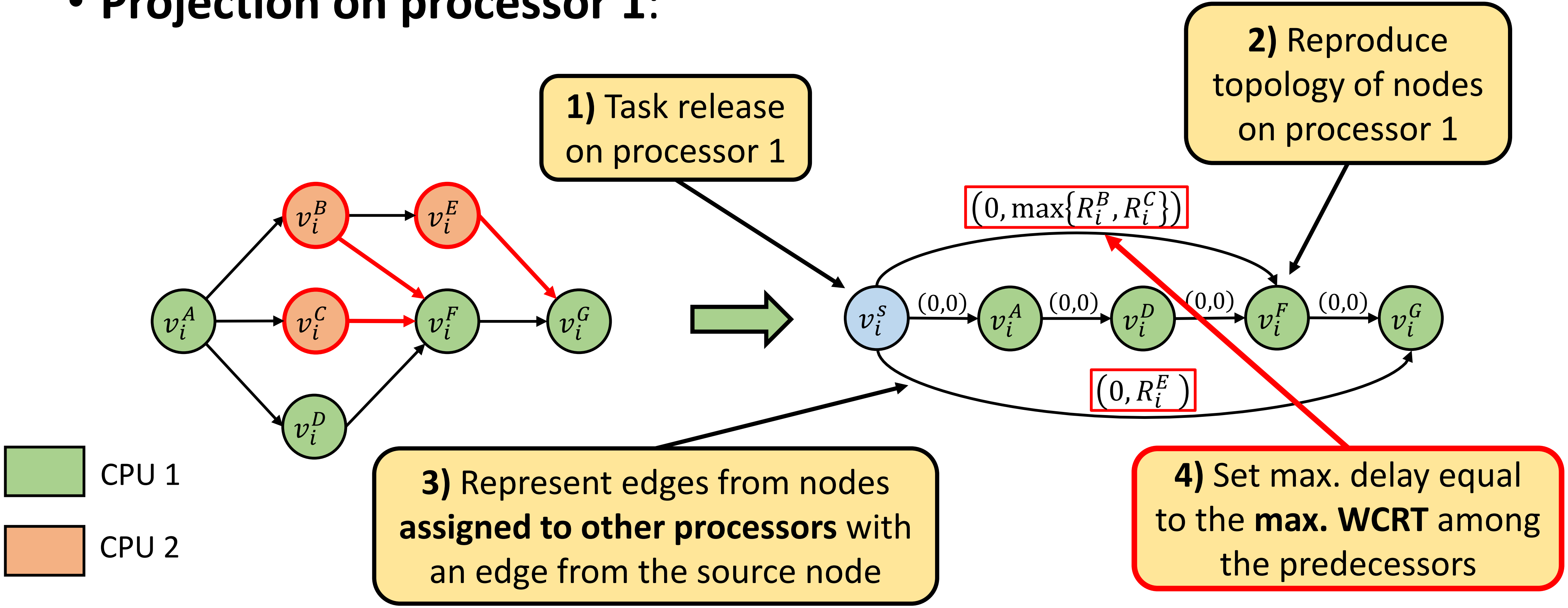
Modeling partitioned parallel DAG tasks

- The scheduling behavior of a parallel task τ^P on a processor P_k can be captured by an EDD task $\mathcal{P}_k(\tau^P)$
- **Projection on processor 1**:



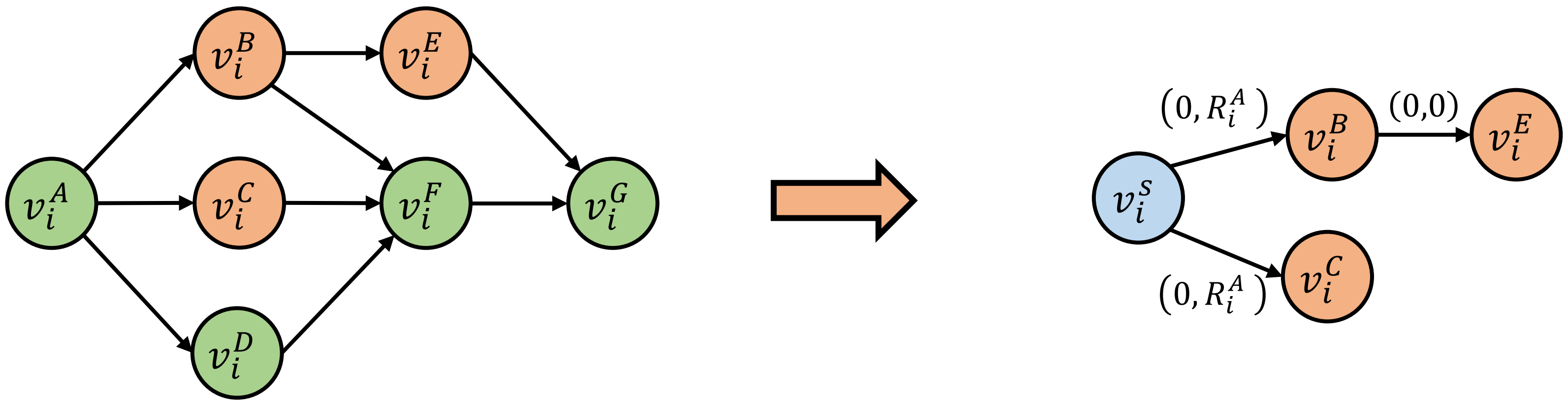
Modeling partitioned parallel DAG tasks

- The scheduling behavior of a parallel task τ^P on a processor P_k can be captured by an EDD task $\mathcal{P}_k(\tau^P)$
- **Projection on processor 1**:



Modeling partitioned parallel DAG tasks

- The scheduling behavior of a parallel task τ^P on a processor P_k can be captured by an EDD task $\mathcal{P}_k(\tau^P)$
- **Projection on processor 2**:

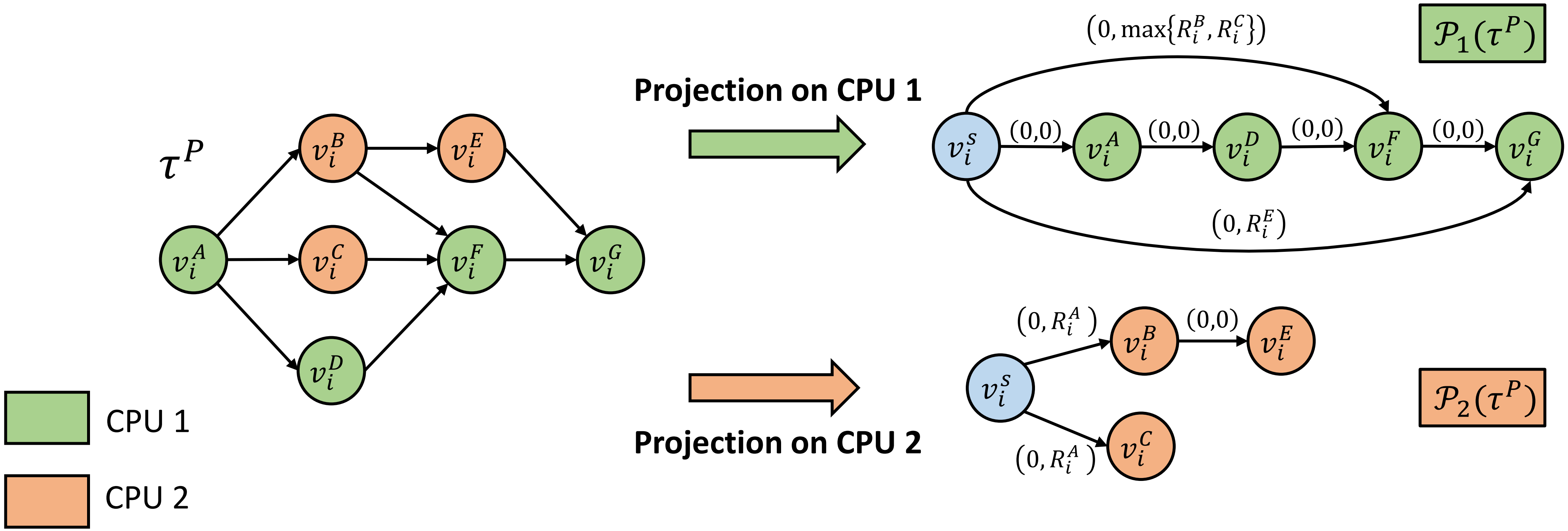


CPU 1

CPU 2

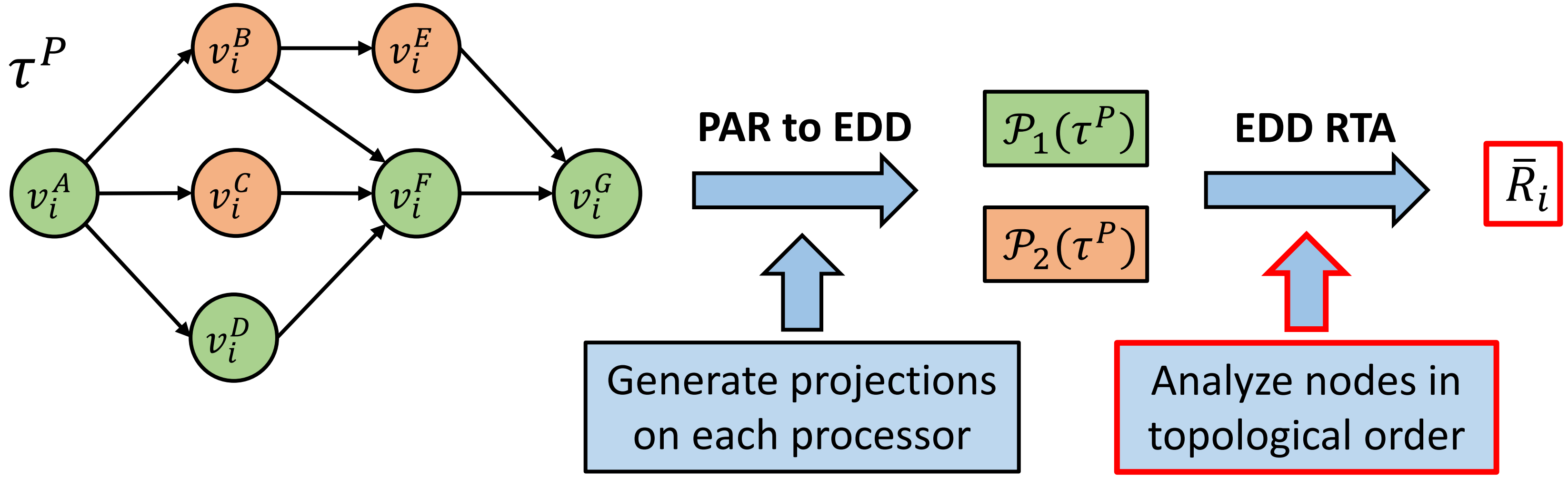
Modeling partitioned parallel DAG tasks

- **Result**: a partitioned parallel task τ^P can be modeled by a set of EDD tasks (one for each processor P_k) for the purpose of real-time analysis
- Note: the WCRTs on the edges introduce circular dependencies



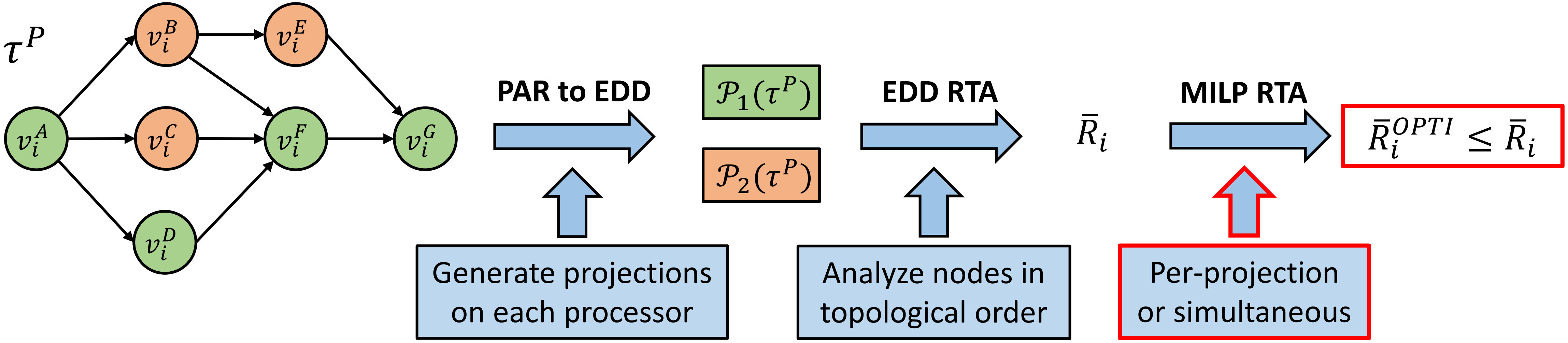
Analysis of partitioned parallel DAG tasks

- **Closed-form**: the EDD projections on each processor are constructed by exploring the DAG of the parallel task in topological order
  - The node-level RTA for EDD tasks is used to obtain a WCRT UB for each node
  - This works around the circular dependencies due to the WCRTs on the edges



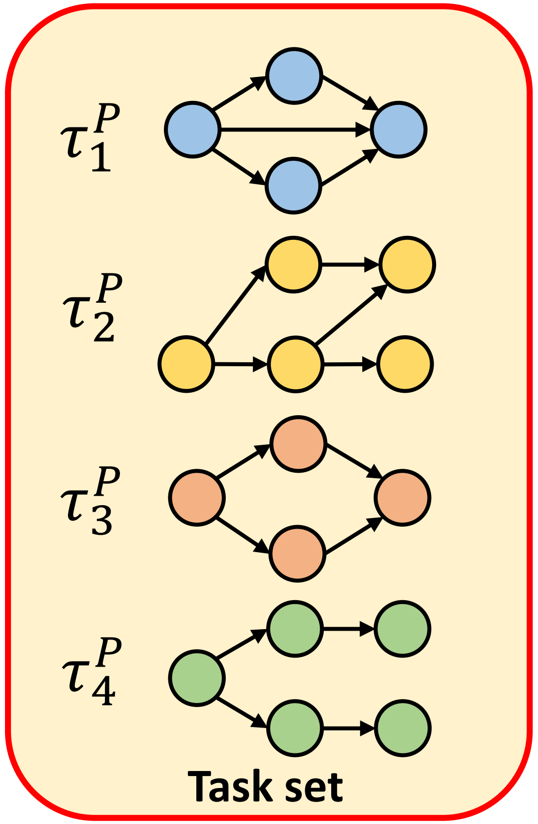
Analysis of partitioned parallel DAG tasks

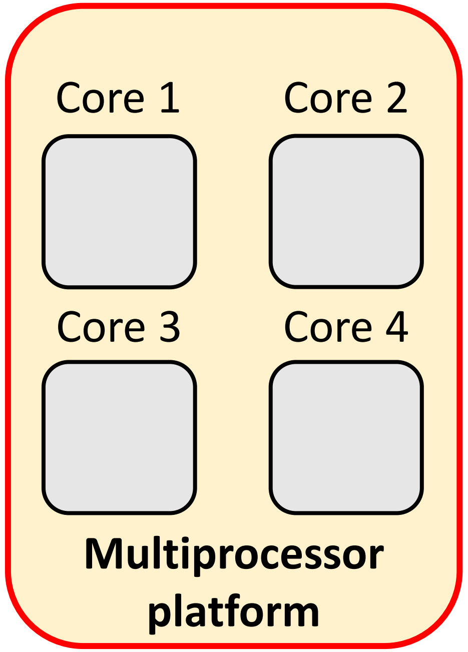
- **Optimization-based**: the proposed EDD MILP analysis can be applied to each projection to **improve upon the obtained WCRT bounds**
- A specialized MILP formulation is also presented to analyze all the projections simultaneously



Experiments on parallel tasks

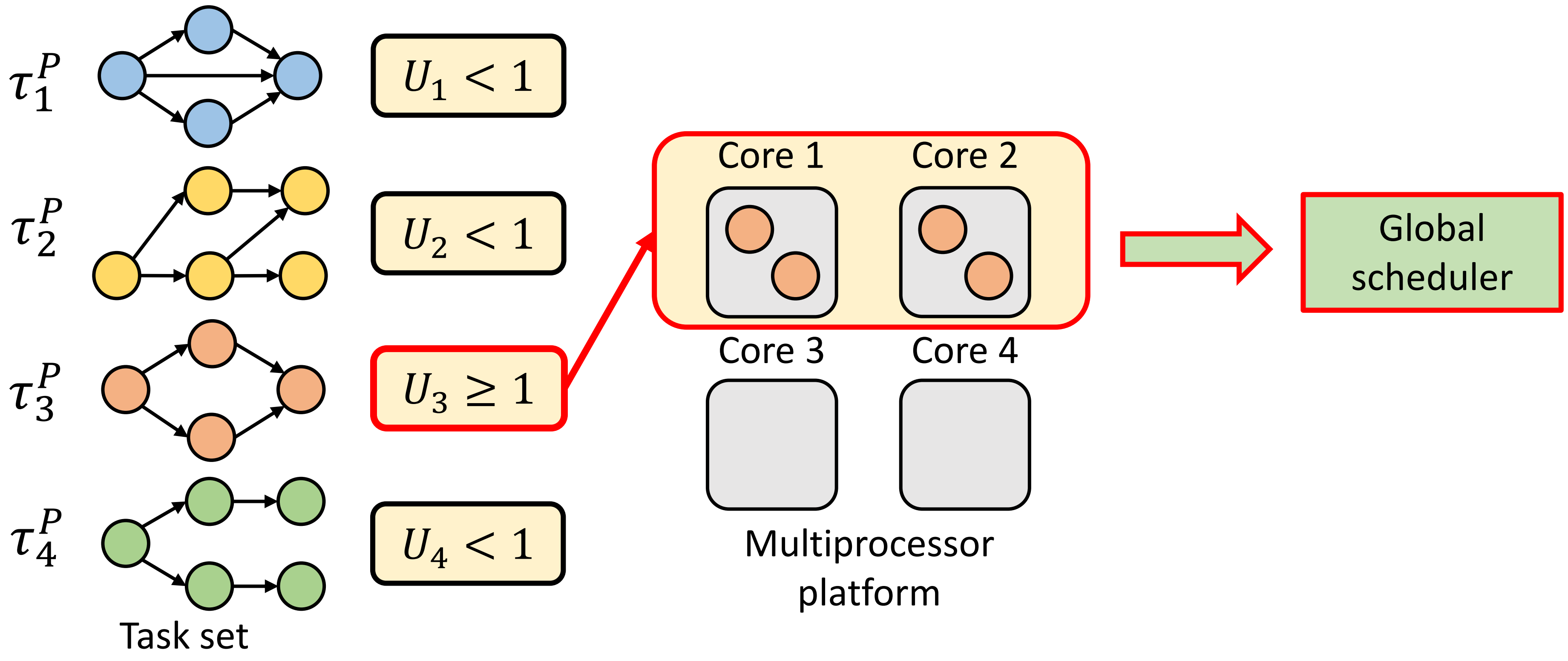
- **Experiments**: comparison of partitioned scheduling (analyzed with EDD tasks) and federated scheduling of parallel tasks on a multiprocessor platform





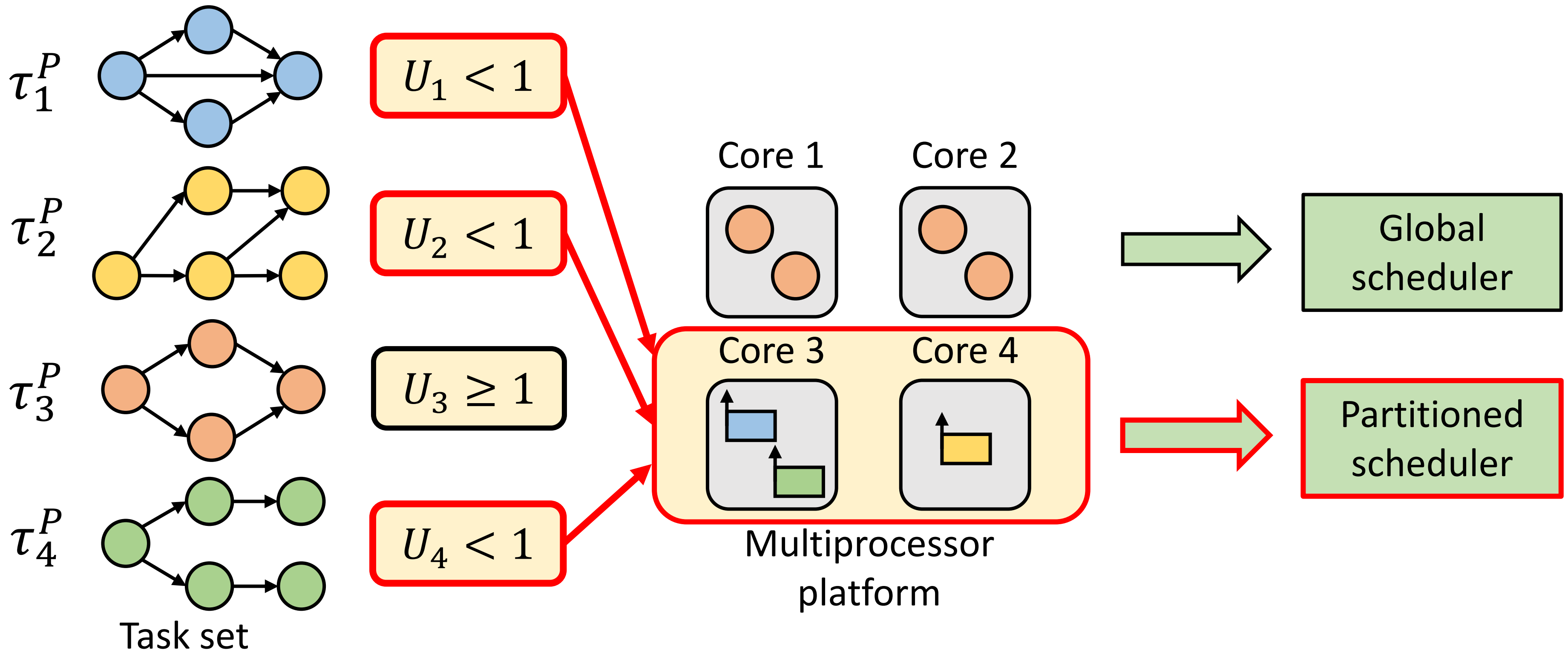
Experiments on parallel tasks

- **Federated scheduling** [Li et al., 2014]:
  - **1.** Each heavy task ($U_i \geq 1$) is assigned a set of dedicated processors, where it is scheduled by a global scheduler



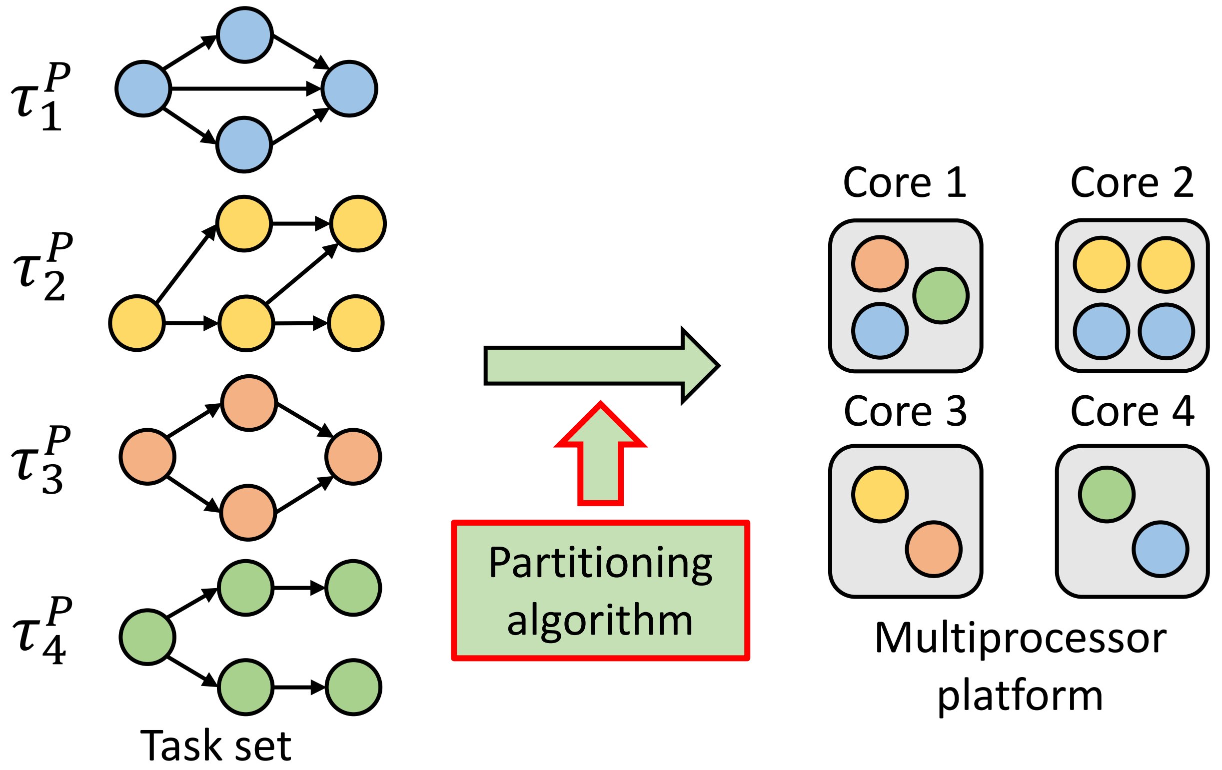
Experiments on parallel tasks

- **Federated scheduling** [Li et al., 2014]:
  - **2.** Light tasks ($U_i < 1$) are treated as sequential tasks and partitioned on the remaining processors, where they are scheduled with a uniprocessor policy



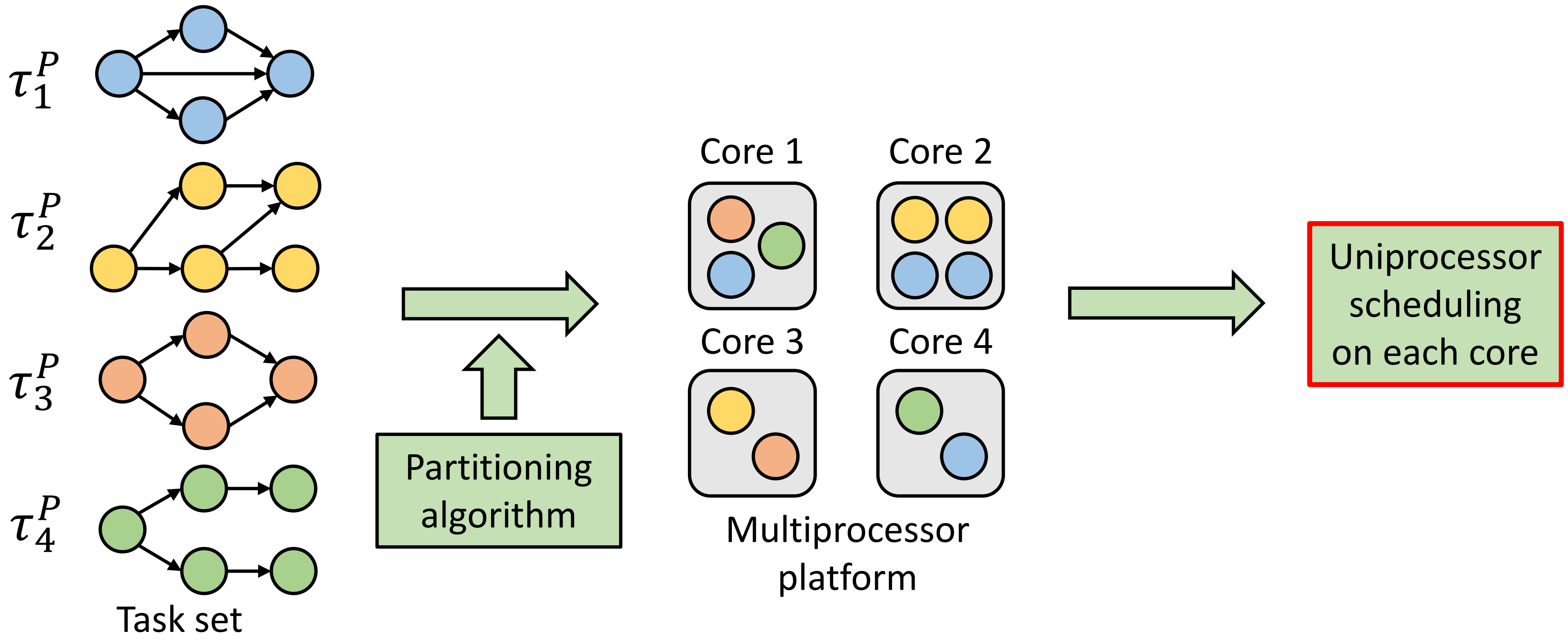
Experiments on parallel tasks

- **Partitioned scheduling**:
  - **1.** Each node is assigned to a specific processor according to a partitioning algorithm



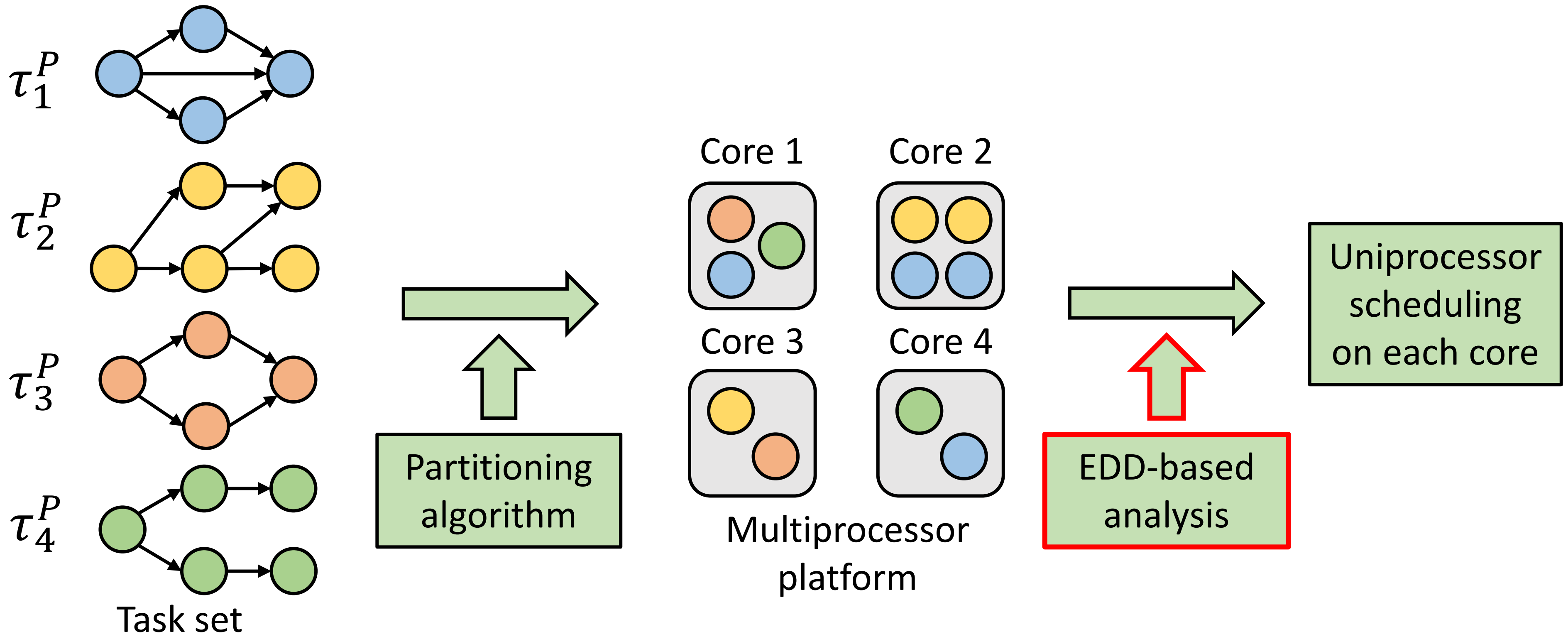
Experiments on parallel tasks

- **Partitioned scheduling**:
  - **2.** Each processor schedules nodes according to a uniprocessor policy



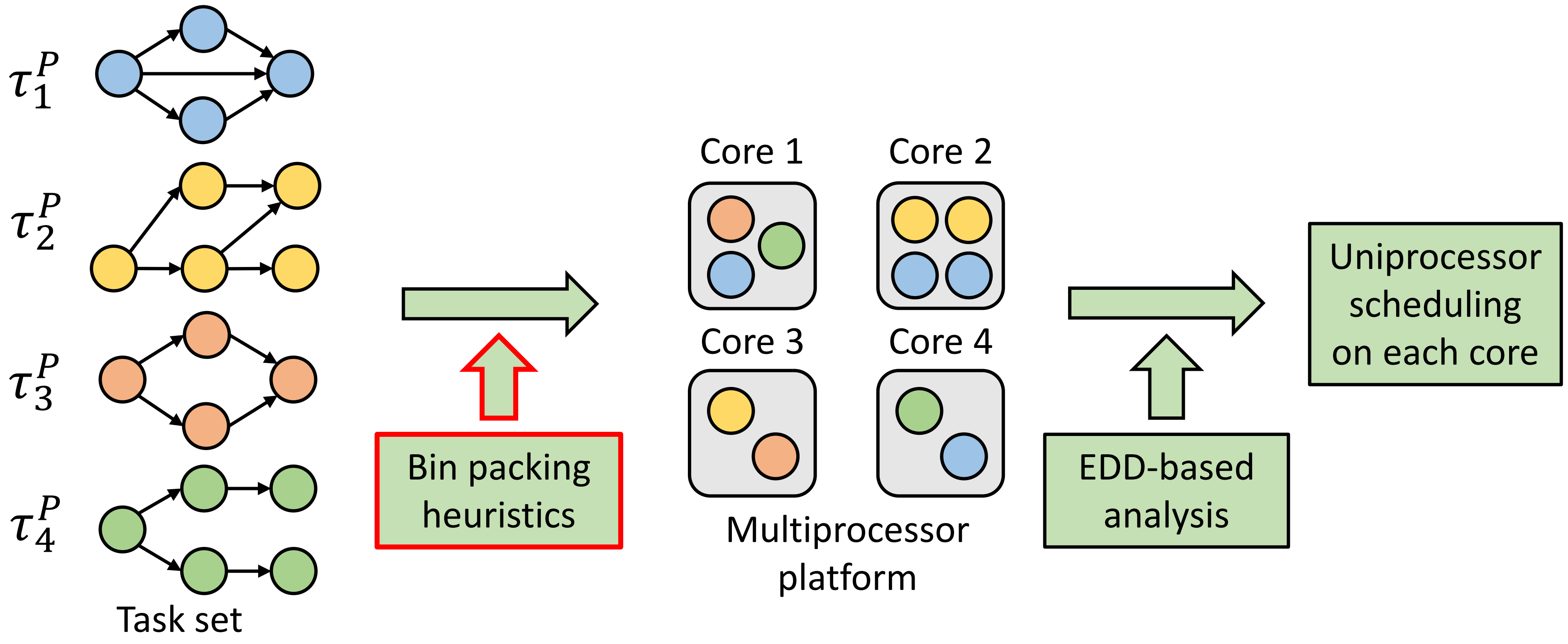
Experiments on parallel tasks

- **Partitioned scheduling**:
  - **3.** Once partitioned, the parallel tasks are analyzed by means of EDD tasks



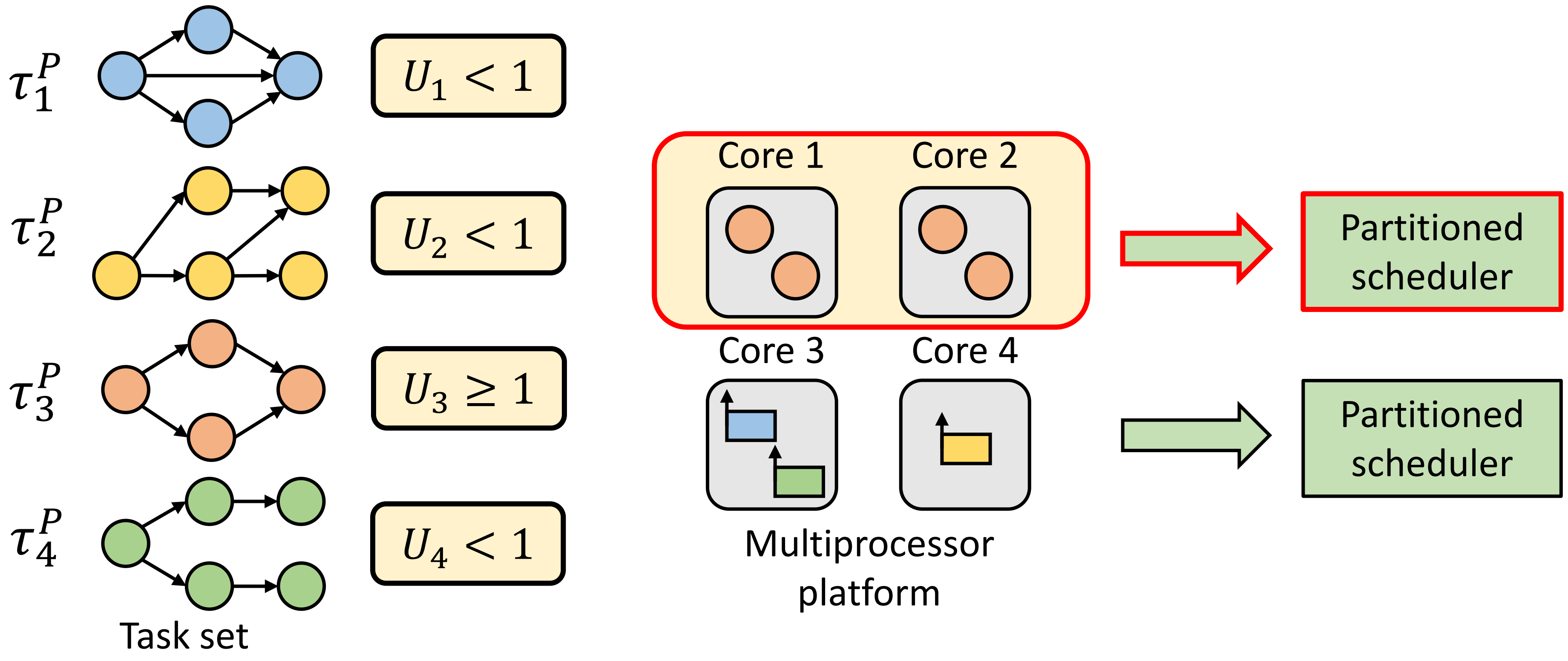
Experiments on parallel tasks

- **Basic partitioning algorithm** considered in the experiments:
  - **WBF**: nodes are sorted by decreasing utilization, and allocated to a processor according to worst-fit, best-fit, or first-fit bin packing heuristics, verifying that processor utilization does not exceed one



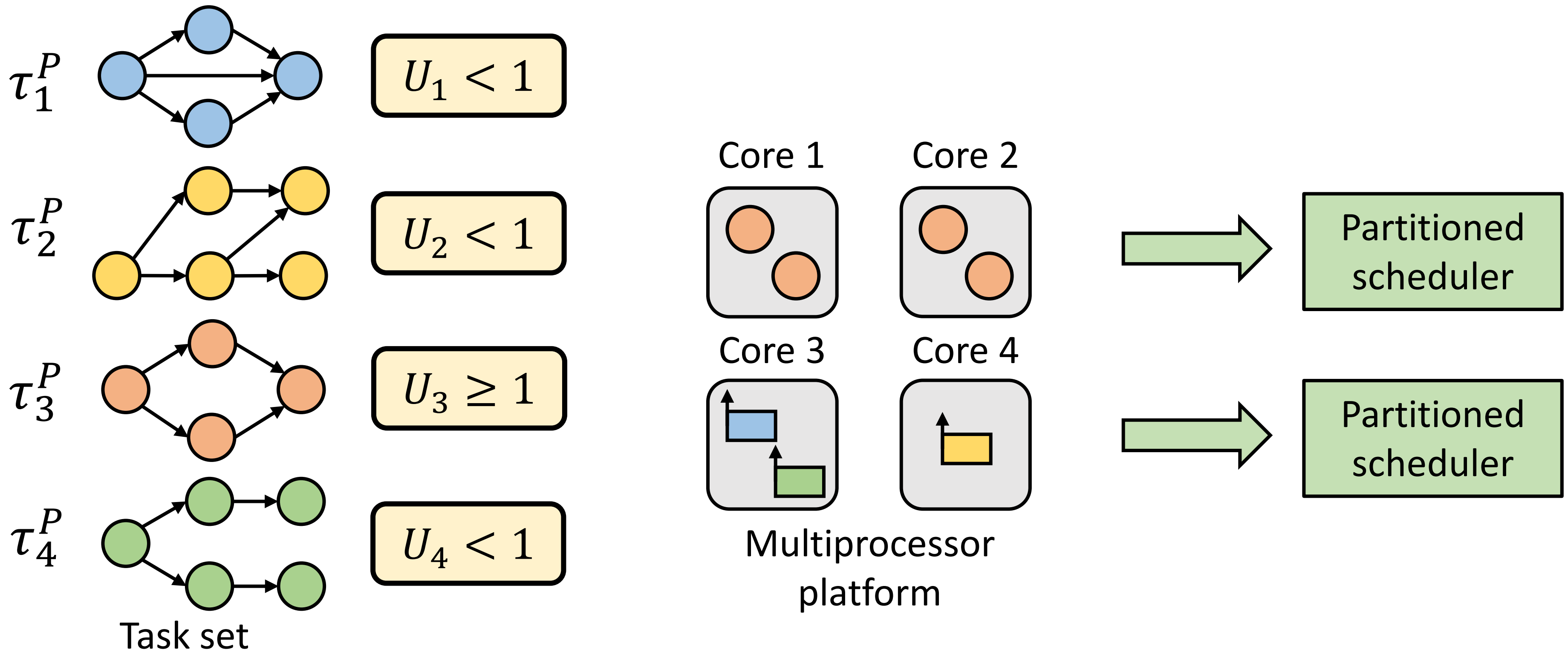
Experiments on parallel tasks

- **Specialized partitioning algorithms** inspired by federated scheduling:
  - **Pseudo-federated** (**P-FED**): like federated scheduling, but heavy tasks are scheduled with partitioned scheduling on the assigned processors



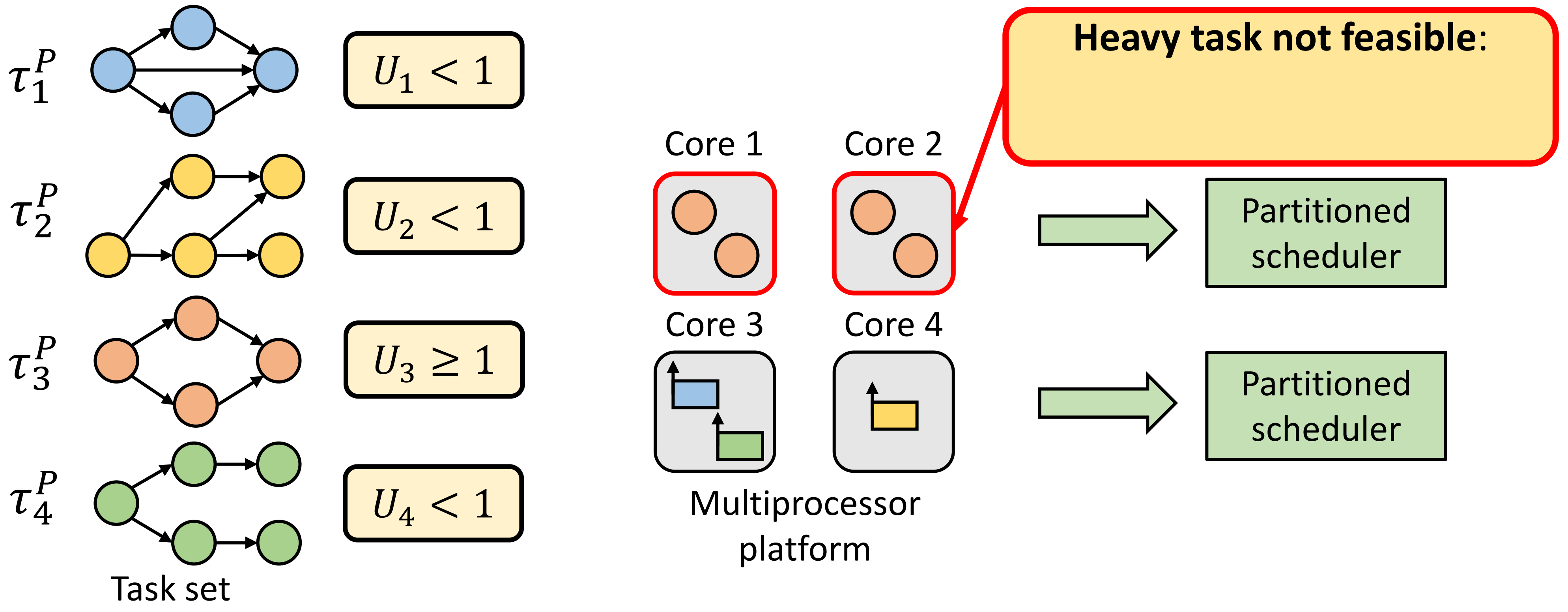
Experiments on parallel tasks

- **Specialized partitioning algorithms** inspired by federated scheduling:
  - **Pseudo-federated++** (**P-FED++**): improves upon P-FED with additional ways to allocate tasks in case a feasible allocation is not found



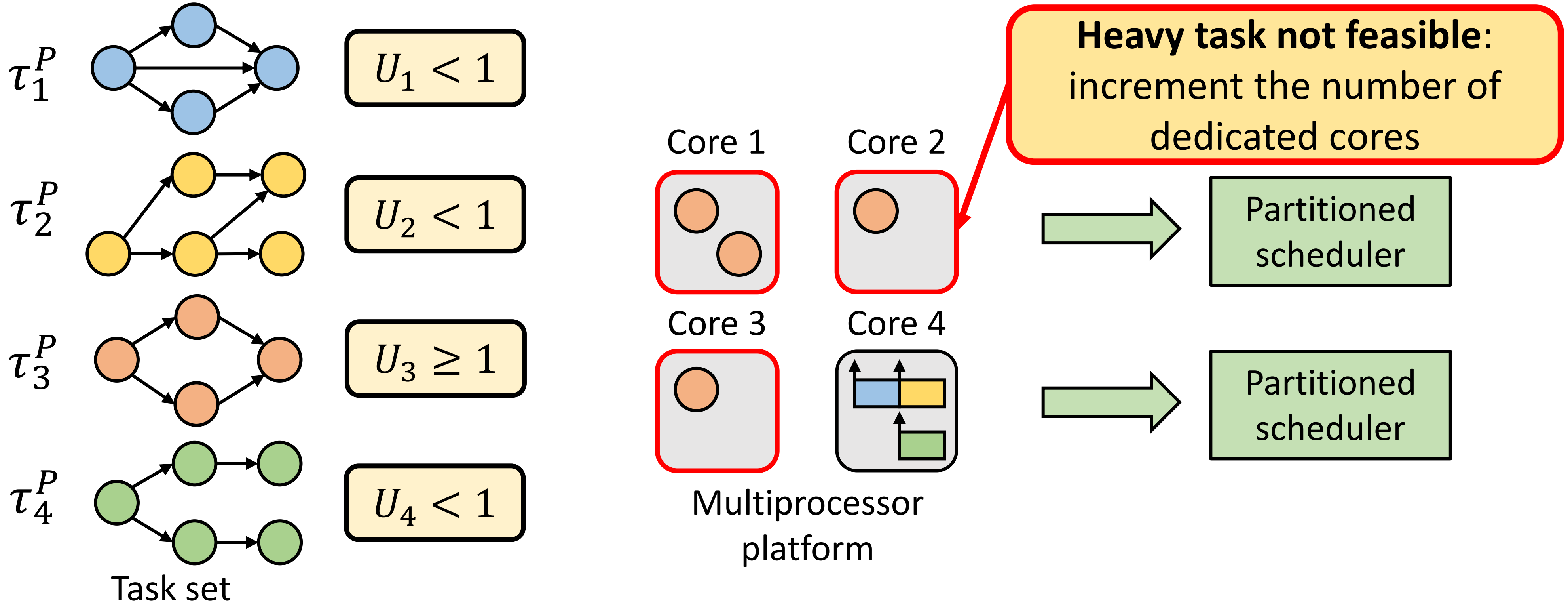
Experiments on parallel tasks

- **Specialized partitioning algorithms** inspired by federated scheduling:
  - **Pseudo-federated++** (**P-FED++**): improves upon P-FED with additional ways to allocate tasks in case a feasible allocation is not found



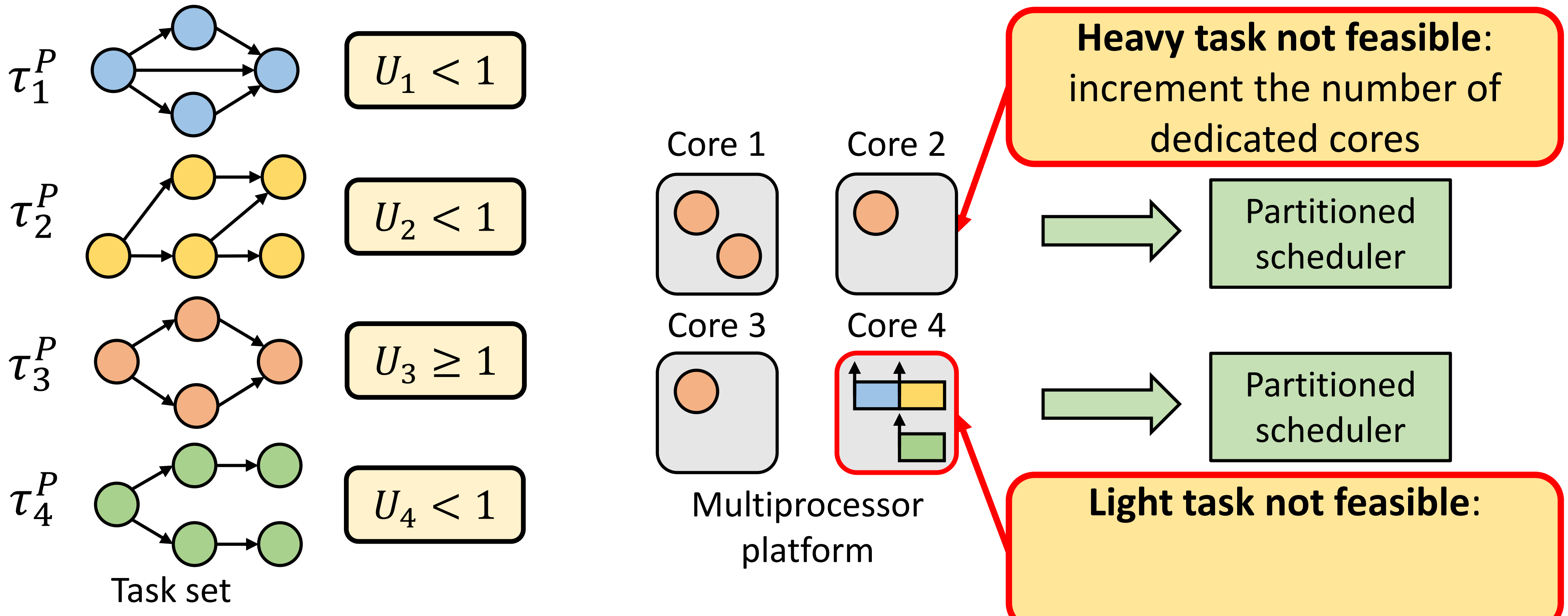
Experiments on parallel tasks

- **Specialized partitioning algorithms** inspired by federated scheduling:
  - **Pseudo-federated++** (**P-FED++**): improves upon P-FED with additional ways to allocate tasks in case a feasible allocation is not found



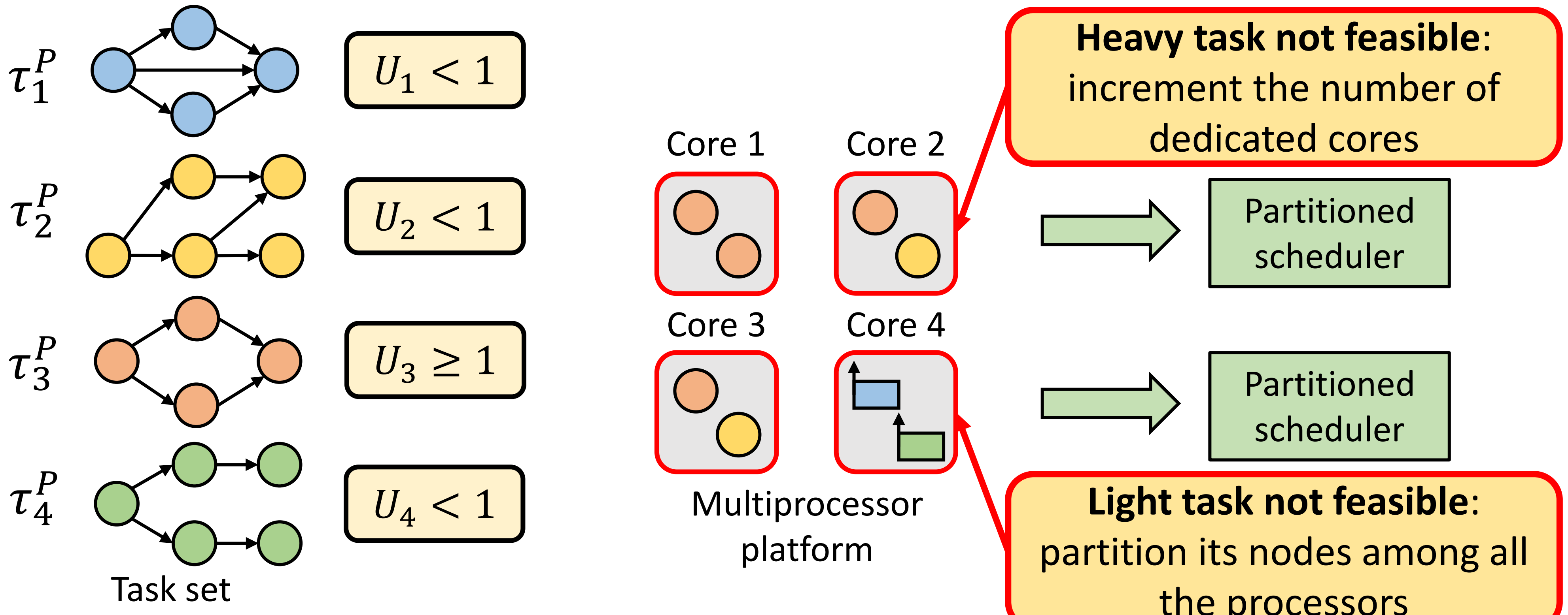
Experiments on parallel tasks

- **Specialized partitioning algorithms** inspired by federated scheduling:
  - **Pseudo-federated++** (**P-FED++**): improves upon P-FED with additional ways to allocate tasks in case a feasible allocation is not found



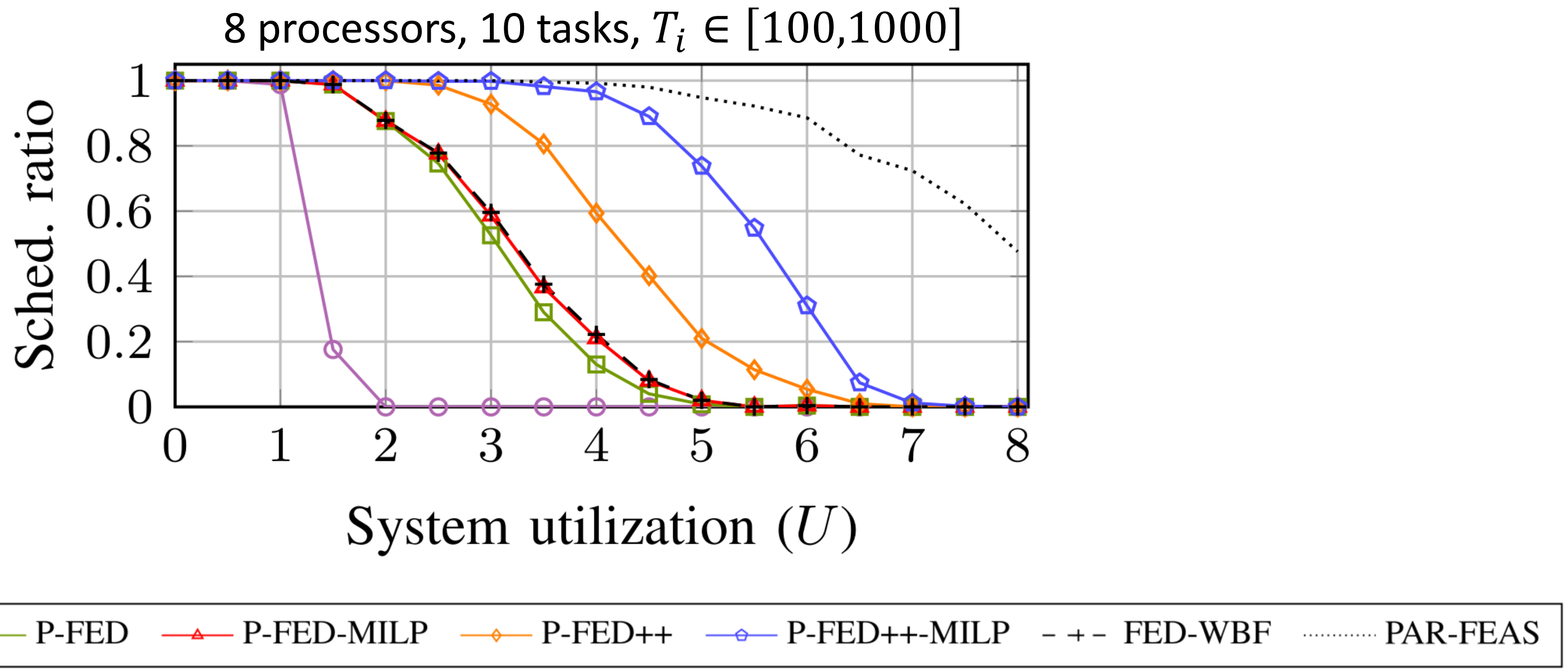
Experiments on parallel tasks

- **Specialized partitioning algorithms** inspired by federated scheduling:
  - **Pseudo-federated++** (**P-FED++**): improves upon P-FED with additional ways to allocate tasks in case a feasible allocation is not found



Experiments on parallel tasks

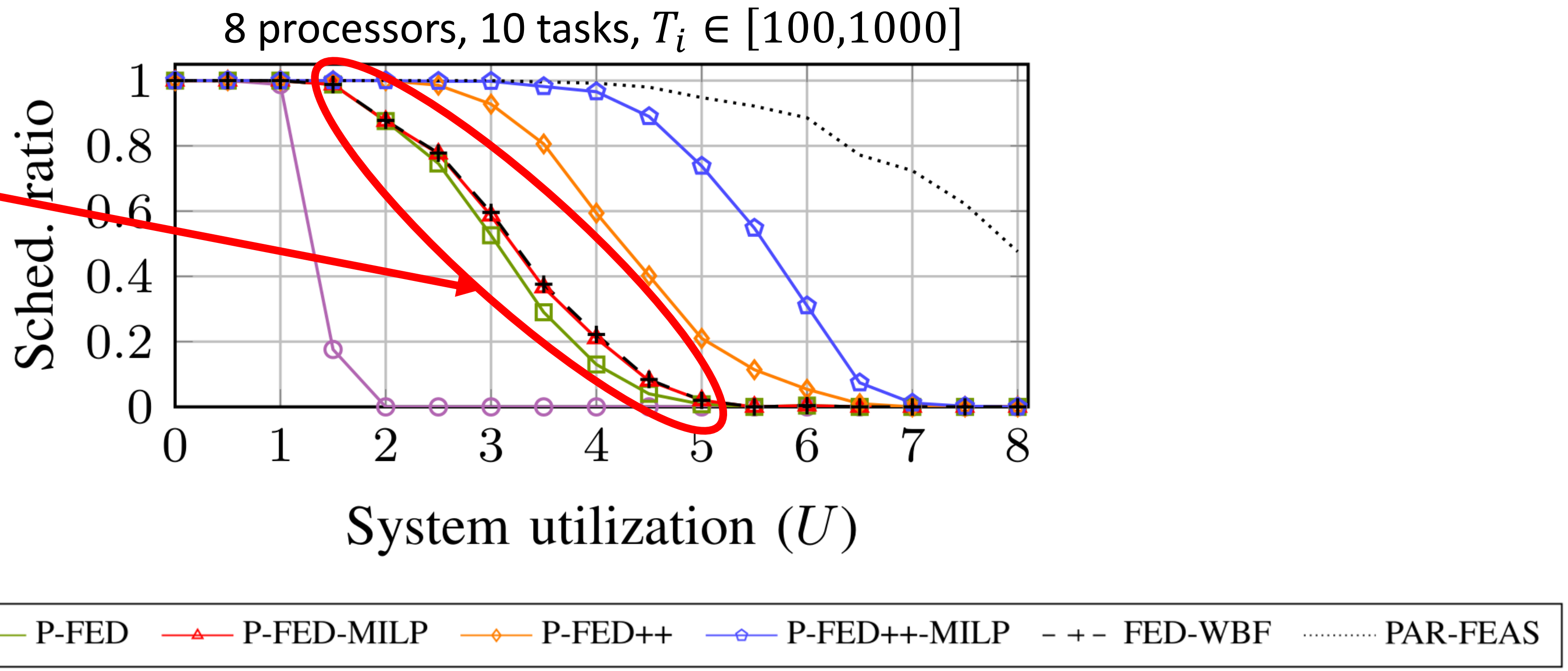
- Comparison of federated scheduling (FED-WBF) and partitioned scheduling
  - Schedulability ratio over randomly generated DAG tasks (Melani et al., 2015)
  - PAR-FEAS: schedulability limit



Experiments on parallel tasks

- Comparison of federated scheduling (FED-WBF) and partitioned scheduling
  - Schedulability ratio over randomly generated DAG tasks (Melani et al., 2015)
  - PAR-FEAS: schedulability limit

Similar performance for **federated** and **pseudo-federated**

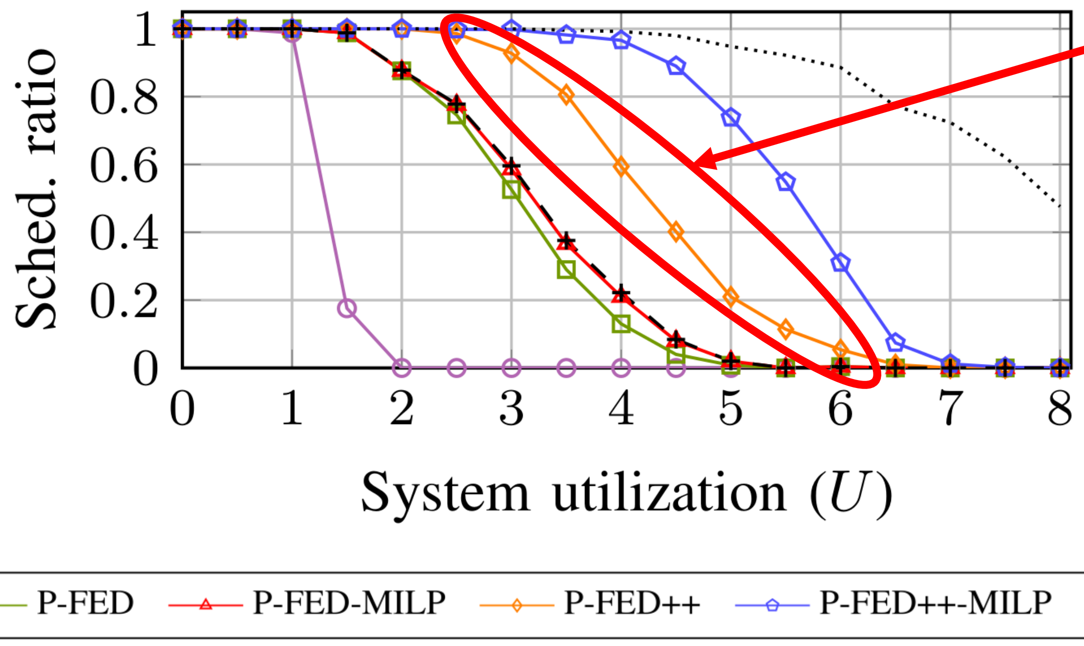


Experiments on parallel tasks

- Comparison of federated scheduling (FED-WBF) and partitioned scheduling
  - Schedulability ratio over randomly generated DAG tasks (Melani et al., 2015)
  - PAR-FEAS: schedulability limit

Similar performance for **federated** and **pseudo-federated**

8 processors, 10 tasks, $T_i \in [100{,}1000]$



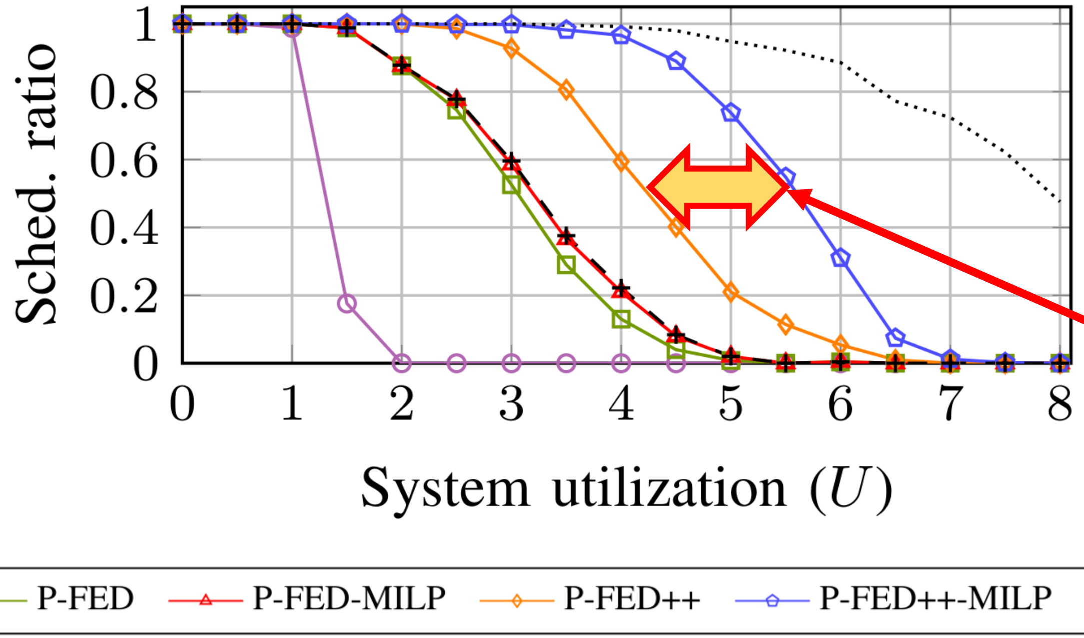
P-FED++ significantly improves upon EDD-WBF and federated scheduling

Experiments on parallel tasks

- Comparison of federated scheduling (FED-WBF) and partitioned scheduling
  - Schedulability ratio over randomly generated DAG tasks (Melani et al., 2015)
  - PAR-FEAS: schedulability limit

Similar performance for **federated** and **pseudo-federated**

8 processors, 10 tasks, $T_i \in [100{,}1000]$

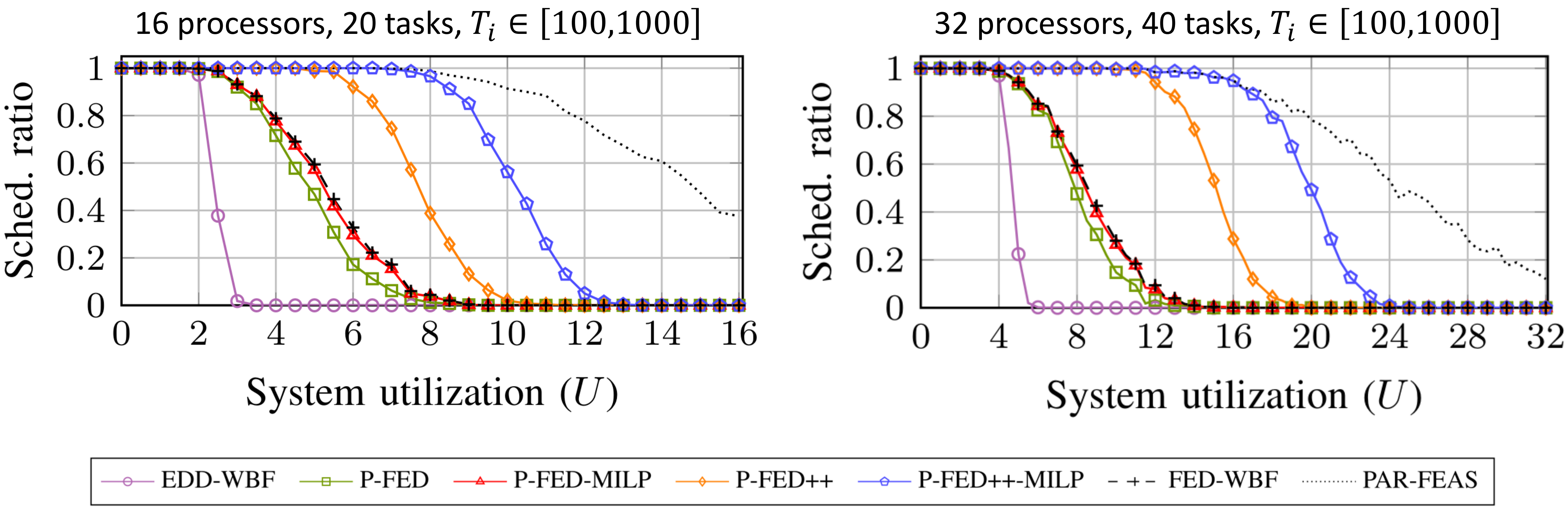


P-FED++ significantly improves upon EDD-WBF and federated scheduling

The **MILP approach** shows significant improvement over the closed-form analysis

Experiments on parallel tasks

- Similar results are observed for other system configurations, with even larger performance gaps



Conclusions

- The **EDD task model** was proposed to explicitly deal with complex computing workloads that incur event-related delays
- Applications include:
  - Analysis of asynchronous HW acceleration
  - Analysis of partitioned parallel tasks on multicores
  - Generalization of other task models
- Two response time analysis techniques were proposed
  - The optimization approach was shown to generally improve upon the closed-form approach, especially in the experiments on parallel tasks
- Partitioned scheduling of parallel tasks analyzed by means of EDD tasks was shown to significantly outperform federated scheduling, without the need for global scheduling

Future work

- Evaluate the applicability of the EDD model to other kinds of workloads
  - Inference of Deep Neural Networks (DNN) on GPU- and FPGA-based heterogeneous platforms
  - Multiprocessor version of FRED with support for asynchronous HW acceleration
- Devise a suitable MILP analysis for EDF scheduling of EDD tasks
- Explore additional partitioning approaches for parallel tasks
- Investigate possible applications of semi-partitioning of nodes on multiprocessors
- Investigate the analysis of locking protocols in parallel task models

Publication

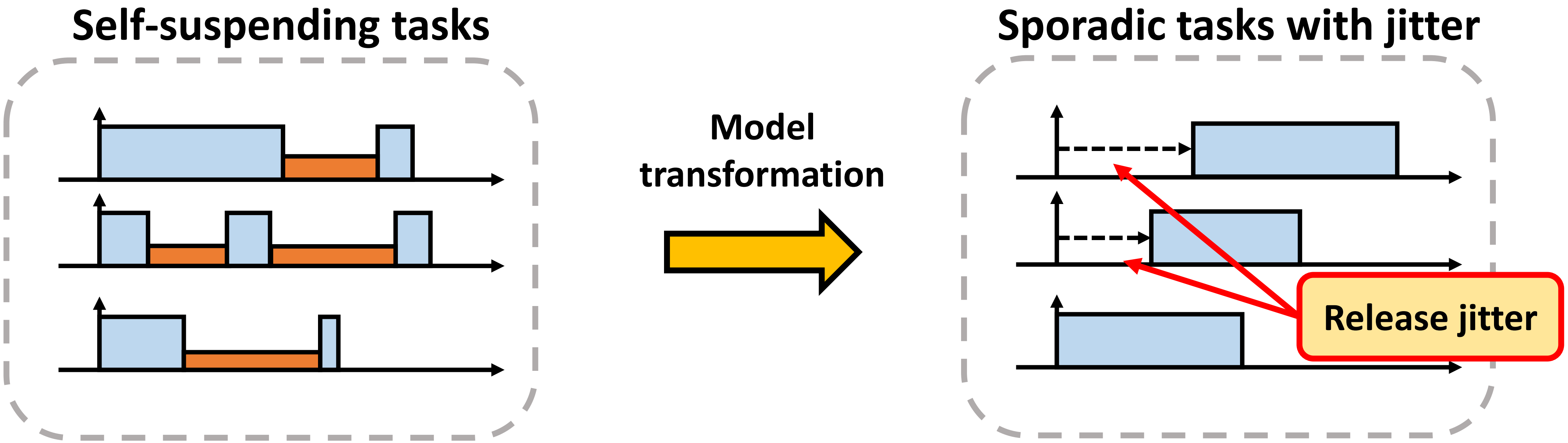
- **Publication** describing the EDD modeling and analysis framework:
  - F. Aromolo, A. Biondi, G. Nelissen, and G. Buttazzo, "**Event-Driven Delay-Induced Tasks: Model, Analysis, and Applications**," In Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021)

Event-Driven Delay-Induced Tasks: Model, Analysis, and Applications

Federico Aromolo*, Alessandro Biondi*, Geoffrey Nelissen†, and Giorgio Buttazzo*
*Scuola Superiore Sant'Anna, Pisa, Italy
†Eindhoven University of Technology, Eindhoven, The Netherlands

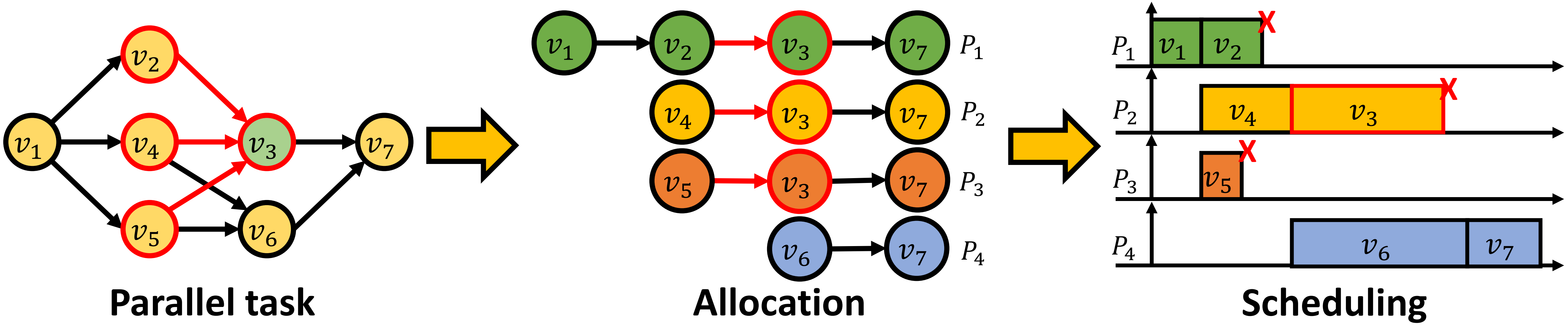
Related publications

- **Publication** proposing a response-time analysis for dynamic self-suspending tasks under EDF based on a transformation to sporadic tasks with jitter, applicable to the analysis of EDD tasks under EDF:
  - F. Aromolo, A. Biondi, and G. Nelissen, "**Response-Time Analysis for Self-Suspending Tasks Under EDF Scheduling**," in Proceedings of the 34th Euromicro Conference on Real-Time Systems (ECRTS 2022)



Related publications

- **Publication** proposing the **Replication-Based Scheduling paradigm for parallel tasks** as a specialized alternative to partitioned, global, and federated scheduling
  - F. Aromolo, G. Nelissen, and A. Biondi, "**Replication-Based Scheduling of Parallel Real-Time Tasks**," in Proceedings of the 35th Euromicro Conference on Real-Time Systems (ECRTS 2023)



References

- F. Aromolo, A. Biondi, G. Nelissen, and G. Buttazzo, "Event-Driven Delay-Induced Tasks: Model, Analysis, and Applications," In Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2021), May 18-21, 2021.
- A. Burns, R. Davis, P. Wang, and F. Zhang, "Partitioned EDF scheduling for multiprocessors using a C=D scheme," in Proceedings of the 18th International Conference on Real-Time and Network Systems (RTNS 2010), 2010, pp. 169–178.
- B. B. Brandenburg and M. Gül, "Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations," in Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016). IEEE, 2016, pp. 99–110.
- A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable FPGAs," in Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS 2016). IEEE, 2016, pp. 1–12.
- J.-J. Chen, G. Nelissen, and W.-H. Huang, "A unifying response time analysis framework for dynamic self-suspending tasks," in Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS 2016). IEEE, 2016, pp. 61–71.
- A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional DAG tasks in multiprocessor systems," in Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS 2015). IEEE, 2015, pp. 211–221.
- J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS 2014). IEEE, 2014, pp. 85–96.

Timing Analysis of Parallel and Accelerated Software with Event-Driven Delay-Induced Tasks

Federico Aromolo

Scuola Superiore Sant’Anna, Pisa, Italy

CAPITAL Workshop 2024: Scalable and Precise Timing Analysis for Multicore Platforms

June 14, 2024 – IRT Saint Exupéry, Toulouse, France