DAG Scheduling and Analysis on Multiprocessor Systems: Exploitation of Parallelism and Dependency

Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, Wanli Chang

University of York

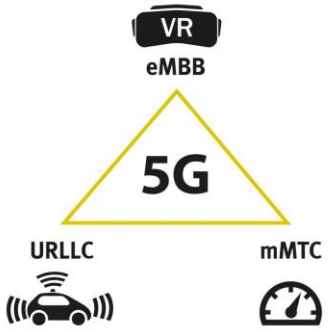
Outline

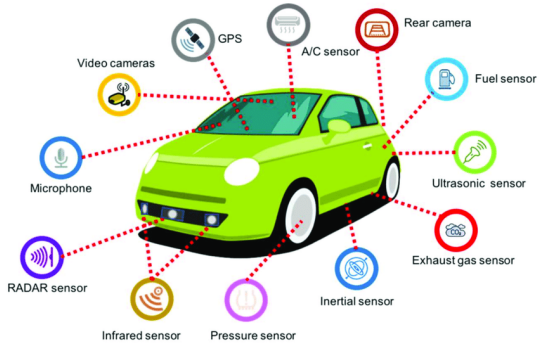
- Background
- Definition of a DAG task
- State-of-the-art
- Concurrent provider and consumer (CPC) model
- A rule-based DAG schedule
- Response time analysis (brief)
- Experimental results
- Conclusion and Future Work

Background

With ever more complex functionalities in emerging real-time applications, directed acyclic graphs (DAGs) are used to model functional dependencies.

Notable examples include 5G base stations under ultra reliable low-latency communication and automotive systems, etc. In these applications, system activities are effectively modelled as a single recurrent DAG task.





In this work, we study a single periodic non-preemptive (NP) DAG running on a homogeneous multiprocessor platform.

We propose novel methods on modelling, scheduling and analysing a DAG task, to achieve shorter makespan and tighter analytical bounds.

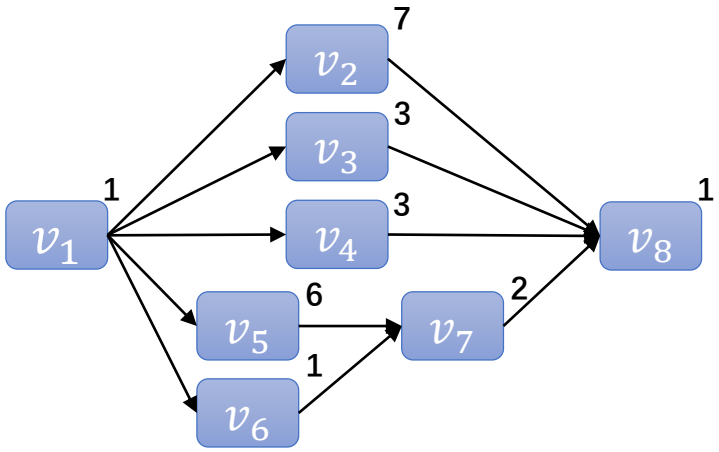
Definition of a DAG task

A DAG task τ_x is defined by a graph $\mathcal{G} = \{V_x, E_x\}$.

V_x denotes nodes in the graph. A node $v_{x,j} \in V_x$ is a computation unit that must be executed sequentially. Each node $v_{x,j}$ has a worst-case execution time (WCET) of $C_{x,j}$.

E_x is the set of directed edges that connect any two nodes. An edge connecting two nodes $(v_{x,j}, v_{x,k})$ defines the execution dependency.

Example:



This DAG has 8 Nodes: $V = \{v_1, \ldots, v_8\}$.

The number at top right of each node gives its WCET.

For any two nodes connected by an edge, e.g., (v_5, v_7), the later node can start only if the former node has finished.

*For simplicity, the subscript of the DAG task (i.e., x for τ_x) is omitted in the context with only one DAG task.

Definition of a DAG task

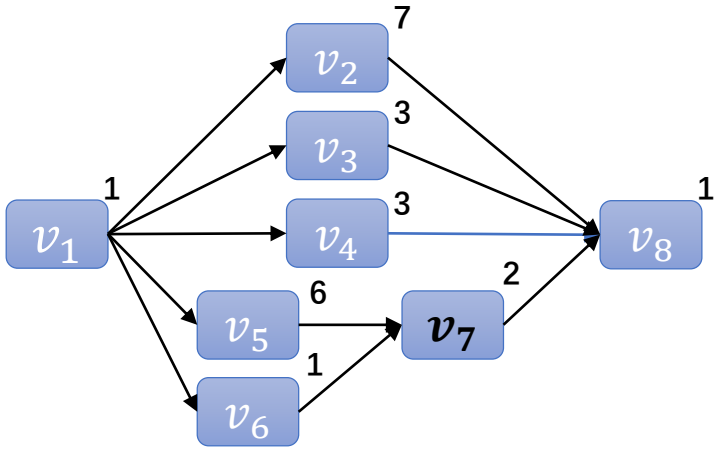
For a given node v_j:

- $pre(v_j) = \{v_k | (v_k, v_j) \in E\}$: the set of *predecessor* nodes that are connected to v_j.
- $suc(v_j) = \{v_k | (v_j, v_k) \in E\}$: the set of *successor* nodes that are connected from v_j.

- $anc(v_j)$: the set of *ancestor* nodes that are (transitively) connected to v_j.
- $des(v_j)$: the set of *descendant* nodes that are (transitively) connected from v_j.

- $\mathcal{C}(v_j) = \{v_k | v_k \notin anc(v_j) \cup des(v_j)\}$: the set of nodes that execute *concurrently* with v_j.

Example:



For v_7 in this example:

$pre(v_7) = \{v_5, v_6\}$, $suc(v_7) = \{v_8\}$,

$anc(v_7) = \{v_1, v_5, v_6\}$, $des(v_7) = \{v_8\}$,

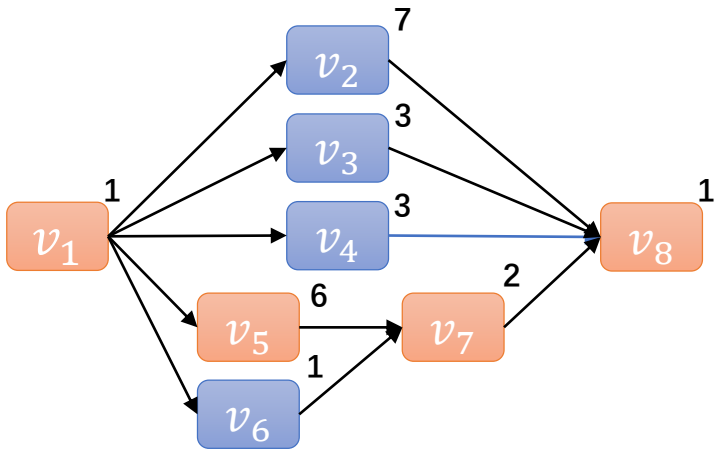
$\mathcal{C}(v_7) = \{v_2, v_3, v_4\}$.

Definition of a DAG task

In addition, a DAG has the following fundamental features:

- A path λ_a is a node sequence in the DAG.
  - $len(\lambda_a)$ gives the length of λ_a, i.e., the sum of WCETs of all nodes in a path.
- λ^* is the longest path of the DAG, termed as the *critical path*.
- Nodes in λ^* is termed as *critical* nodes. Other nodes are termed as *non-critical* nodes $V^{\neg}$.
- $L = len(\lambda^*)$: length of a DAG. $W = \sum_{v_j \in V} C_j$: workload of a DAG.

Example:



Critical path: $\lambda^* = \{v_1, v_5, v_7, v_8\}$.

Non-critical nodes: $V^{\neg} = \{v_2, v_3, v_4, v_6\}$.

Length and workload of the DAG: $L = 10$, $W = 24$.

State-of-the-art

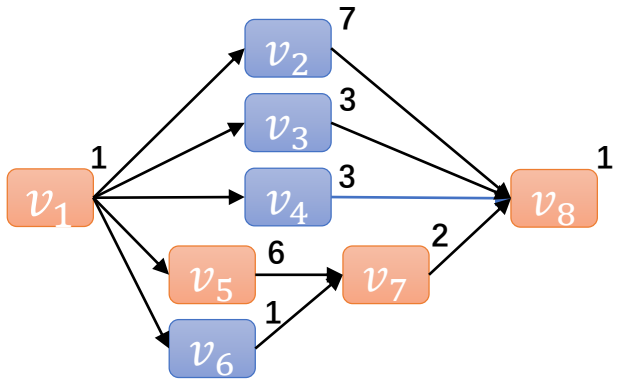
Majority of existing work schedules DAG tasks by a *work-conserving* scheduler, which never idles a core if there exists pending workload.

For any *work-conserving* schedule, the worst-case finish of a DAG is effectively bounded by the worst-case finish of its critical path, known as the classic approach.

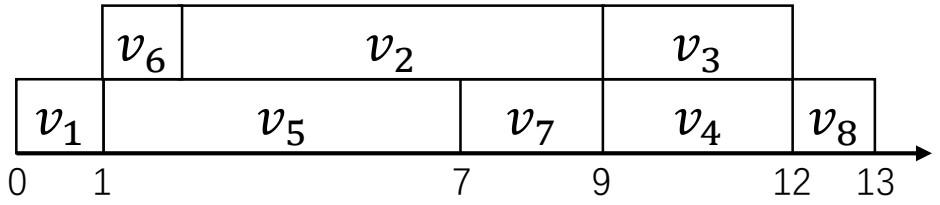
$$R = L + \left\lceil \frac{1}{m}(W - L) \right\rceil$$

However, this analysis can be pessimistic if the execution order of nodes is known a prior.

Example:



With $L = 10$ and $W = 24$, $R = 10 + \left\lceil \frac{1}{2}(24 - 10) \right\rceil = 17$.



With a schedule that guarantees the above execution order, a worst-case makespan of 13 can be achieved, instead of 17.

State-of-the-art

For a DAG with an explicit order known a prior, a tighter bound can be obtained.

In this case, a node v_j can only be delayed by its concurrent nodes $\mathcal{C}(v_j)$ that are scheduled before v_j, denoted as $\mathcal{I}(v_j)$.

$$R = \max_{v_j \in V} f(v_j)$$

$$f(v_j) = \underbrace{\max_{v_k \in pre(v_j)} f(v_k) + C_j}_{①} + \underbrace{\left\lceil \frac{1}{m} \times \sum_{v_k \in \mathcal{I}(v_j)} C_k \right\rceil}_{②}$$

① v_j can becomes ready only if all nodes in $pre(v_j)$ have finished.

② After ready, v_j is delayed by concurrent nodes that are scheduled before v_j, i.e., $\mathcal{I}(v_j)$.

State-of-the-art

However, this analysis still has certain degree of pessimism, by assuming each node in $\mathcal{I}(v_j)$ can cause a delay to v_j, which is not always true due to potential parallel execution.

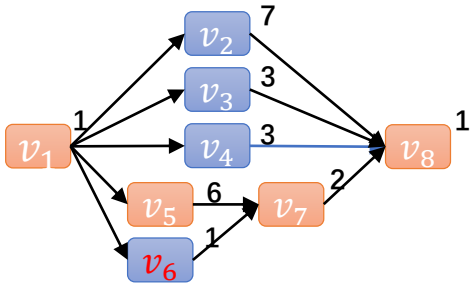
$$R = \max_{v_j \in V} f(v_j)$$

$$f(v_j) = \underbrace{\max_{v_k \in pre(v_j)} f(v_k)}_{①} + C_j + \underbrace{\left\lceil \frac{1}{m} \times \sum_{v_k \in \mathcal{I}(v_j)} C_k \right\rceil}_{②}$$

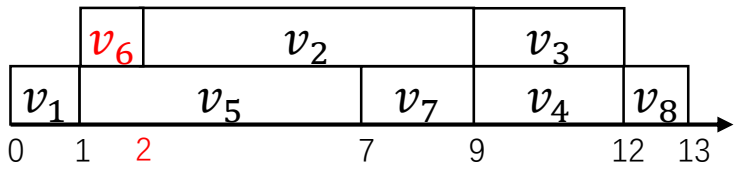
① v_j can becomes *ready* only if all nodes in $pre(v_j)$ have finished.

② After ready, v_j is delayed by concurrent nodes that are scheduled before v_j, i.e., $\mathcal{I}(v_j)$.

Example:



$$v_1 \rightarrow v_5 \rightarrow v_7 \rightarrow v_6 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_8$$



$$f(v_6) = f(v_1) + C_6 + \left\lceil \frac{1}{m}(C_5 + C_7) \right\rceil$$
$$= 1 + 1 + \left\lceil \frac{1}{2}(6 + 2) \right\rceil$$
$$= 6$$

State-of-the-art

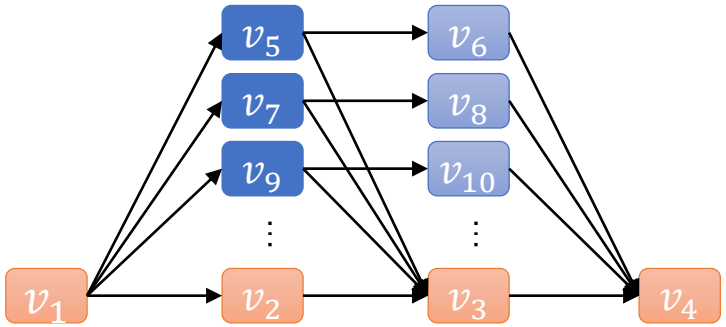
As for scheduling, the state-of-the-art method schedules nodes by guaranteeing:

1. Critical path first.
2. Early predecessor paths of the critical path first.

However, for the predecessor paths with the same earliness, they are ordered by the length of their longest complete paths.

This cannot maximize parallelism and can lead to a prolonged finish time.

Example:



With this method, nodes are scheduled by the order:

1. v_1, v_2
2. v_5, v_7, v_9 ... ← nodes with an early interference.
3. v_3
4. v_6, v_8, v_{10} ... ← nodes with a late interference.
5. v_4

For $\{v_5, v_7, v_9\}$ or $\{v_6, v_8, v_{10}\}$, the order of their lengths can be exact opposite to their complete paths. This leads to a prolonged finish.

Concurrent provider and consumer model

Based on the above, minimizing the delay on the critical path effectively reduces the makespan.

- This implies the critical path first execution.

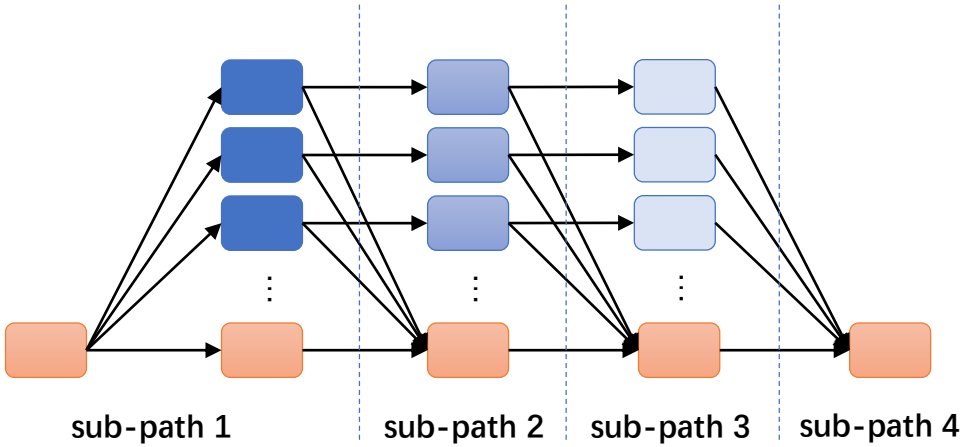
However, this can be difficult to achieve because:

1. The critical path can be delayed multiple times by different non-critical nodes.
2. The execution order of these nodes has a direct impact on the delay.

To fully exploit node dependency and parallel, a concurrent provider and consumer (CPC) model is proposed.

This model divides the critical path to a set of consecutive sub-paths. For each sub-path, CPC identifies non-critical nodes that can:

1. Execute concurrently with the sub-path.
2. Impose a delay to the start of the next sub-path.



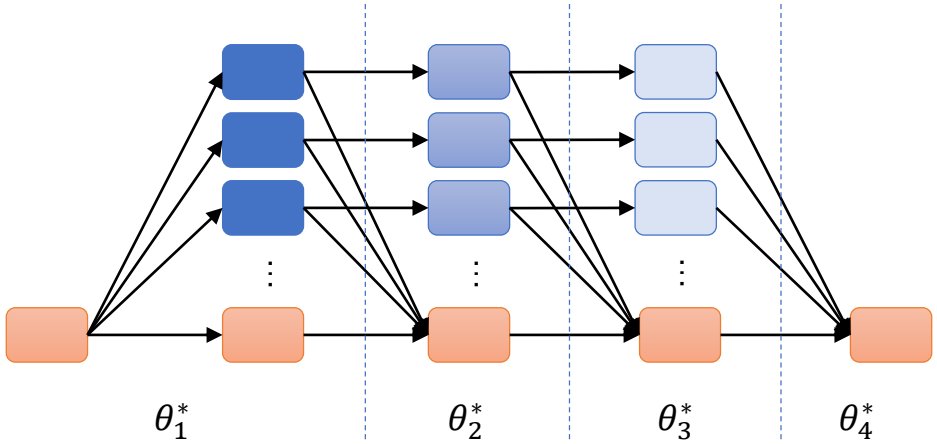
Concurrent provider and consumer model

The intuition is, with each critical sub-path executed first on one core, it provides a *conceptual "capacity"* for the concurrent non-critical nodes to run on the rest costs in parallel.

This capacity can be well-utilized to execute non-critical nodes, and will not cause any delay the critical path.

For this reason, the critical path is termed as the *capacity providers* Θ^* and the non-critical nodes are consumers Θ.

Each provider $\theta_i^* \in \Theta^*$ offers certain capacity (the length of the provider) for concurrent *consumers* to execute in parallel.



Concurrent provider and consumer model

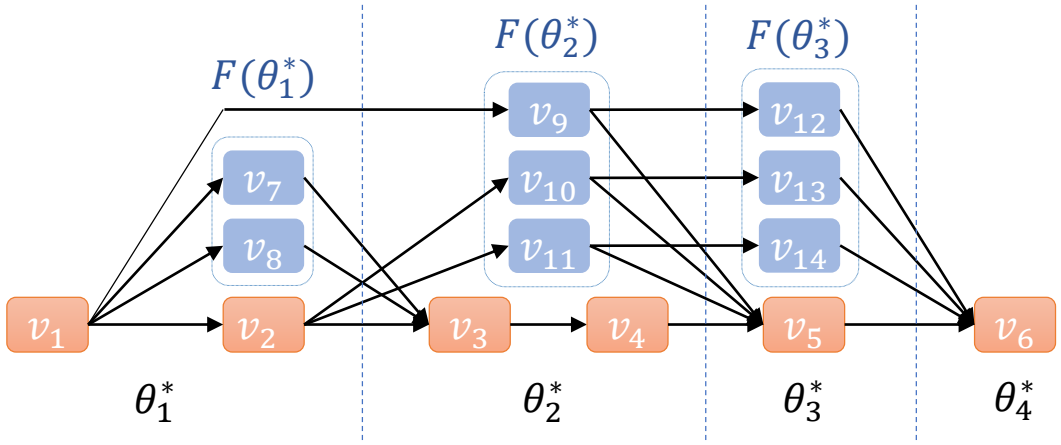
Starting from the first node in the critical path, a capacity provider θ_i^* is formed by taking the following nodes that cannot be delayed by non-critical nodes.

- Only the first node in a provider can incur a delay.

Then, for each provider θ_i^*, its consumer nodes $F(\theta_i^*)$ are given by:

- Nodes that execute concurrently with θ_i^*.
- Nodes that can delay the next provider θ_{i+1}^*.

Example:



This example has 4 providers:
$\theta_1^* = \{v_1, v_2\}$, $\theta_2^* = \{v_3, v_4\}$, $\theta_3^* = \{v_5\}$, $\theta_4^* = \{v_6\}$.

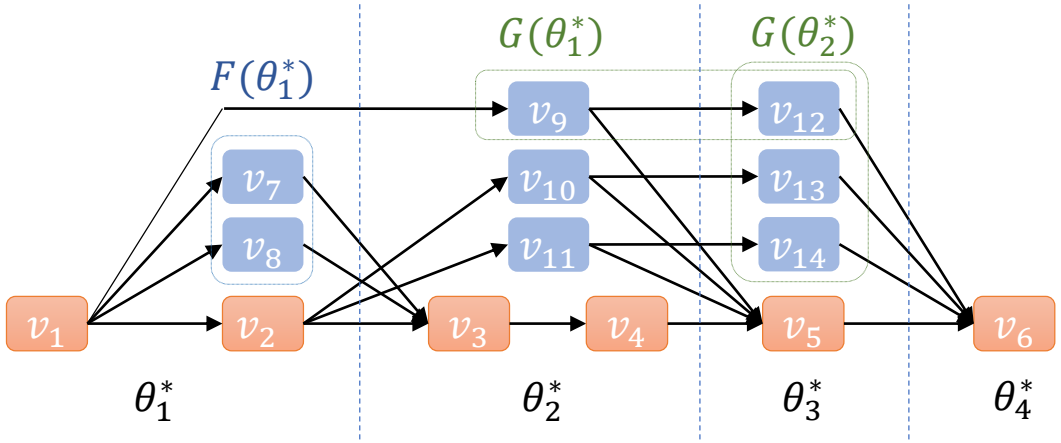
$F(\theta_1^*) = \{v_7, v_8\}$,
$F(\theta_2^*) = \{v_9, v_{10}, v_{11}\}$,
$F(\theta_3^*) = \{v_{12}, v_{13}, v_{14}\}$,
$F(\theta_4^*) = \emptyset$.

Concurrent provider and consumer model

In addition, there can be certain nodes that are not in $F(\theta_i^*)$, but can execute concurrently with θ_i^*, denoted as $G(\theta_i^*)$.

- Nodes in $G(\theta_i^*)$ can execute concurrently with $F(\theta_i^*)$.
- This imposes a delay to the finish of $F(\theta_i^*)$, and hence, the start of θ_{i+1}^*.

Example:



This example has 4 providers:
$\theta_1^* = \{v_1, v_2\},\ \theta_2^* = \{v_3, v_4\},\ \theta_3^* = \{v_5\},\ \theta_4^* = \{v_6\}.$

$F(\theta_1^*) = \{v_7, v_8\},$ $G(\theta_1^*) = \{v_9, v_{12}\},$
$F(\theta_2^*) = \{v_9, v_{10}, v_{11}\},$ $G(\theta_2^*) = \{v_{12}, v_{13}, v_{14}\},$
$F(\theta_3^*) = \{v_{12}, v_{13}, v_{14}\},$ $G(\theta_3^*) = \emptyset,$
$F(\theta_4^*) = \emptyset.$ $G(\theta_4^*) = \emptyset.$

A rule-based DAG schedule

Based on CPC, a rule-based scheduling method is proposed to minimize the makespan.

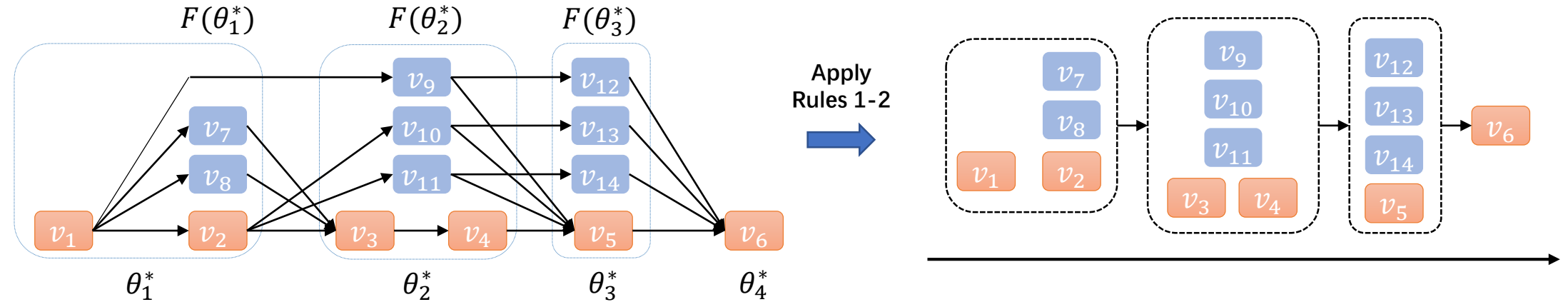
Rule 1. Provider nodes first.

★ This provides the maximum capacity for non-critical nodes to execute in parallel.

Rule 2. Early consumer group first.

★ This minimizes the interference to a consumer group $F(\theta_i^*)$, from nodes in $G(\theta_i^*)$.

Example:



A rule-based DAG schedule

Then, for each $F(\theta_i^*)$, the objective is to minimize its finish time. However, simply applying certain heuristic is not sufficient as each $F(\theta_i^*)$ can form a smaller local DAG $\mathcal{G}' = \{V_x', E_x'\}$.

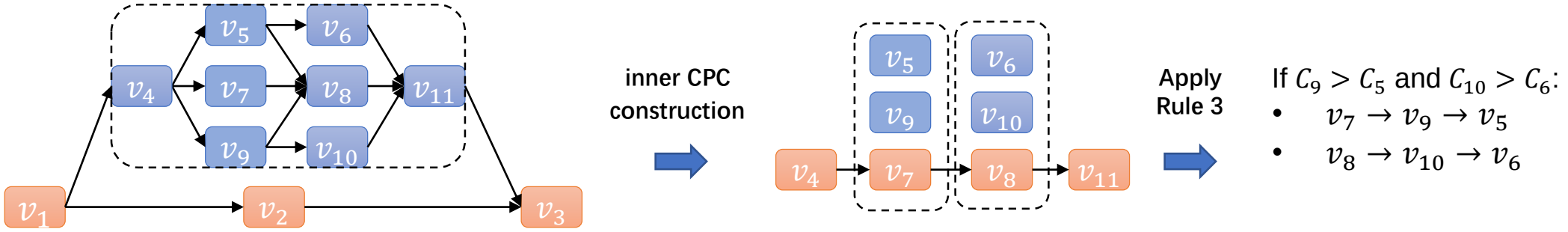
Therefore, the CPC is applied recursively to form inner CPC models, with Rules 1-2 applied on each inner CPC model. This process repeats until paths in a consumer group are independent.

Then, the final rule is applied to independent consumer paths in each inner-most CPC.

Rule 3*. Longer path in a consumer group first.

★ This maximizes node-level parallelism and minimizes the finish time of $F(\theta_i^*)$.

Example:

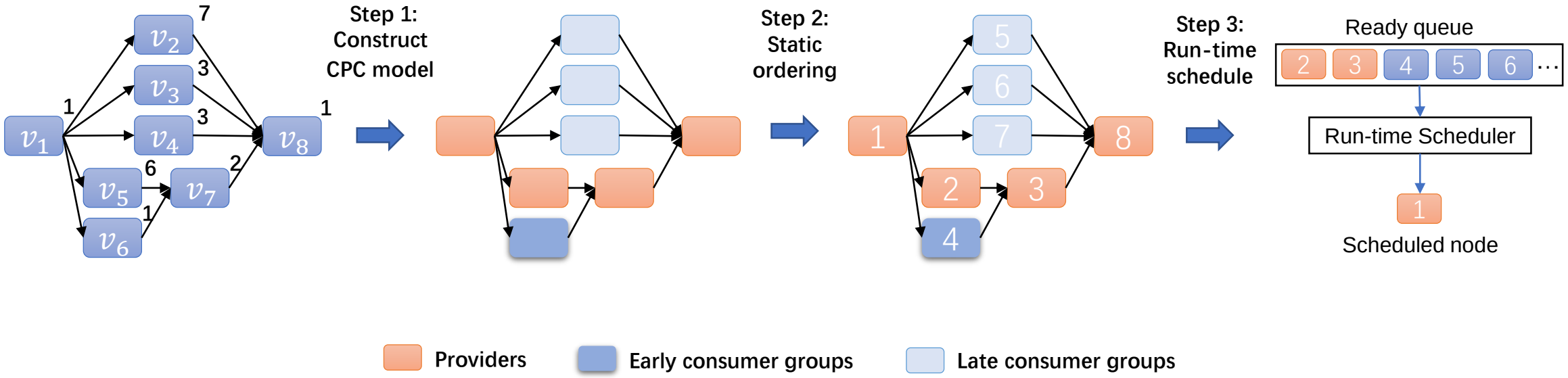


A rule-based DAG schedule

Different from existing schedule, the proposed method performs the following:

1. Transform the DAG to the proposed CPC model.
2. Statically order the execution of each node.
3. Schedule the nodes at run-time based on the pre-planned order.

Steps 1-2 can be performed offline if the input DAG is known before run-time, which effectively reduces its scheduling cost to that of the traditional Fixed-Priority Scheduling.



Response time analysis (brief)

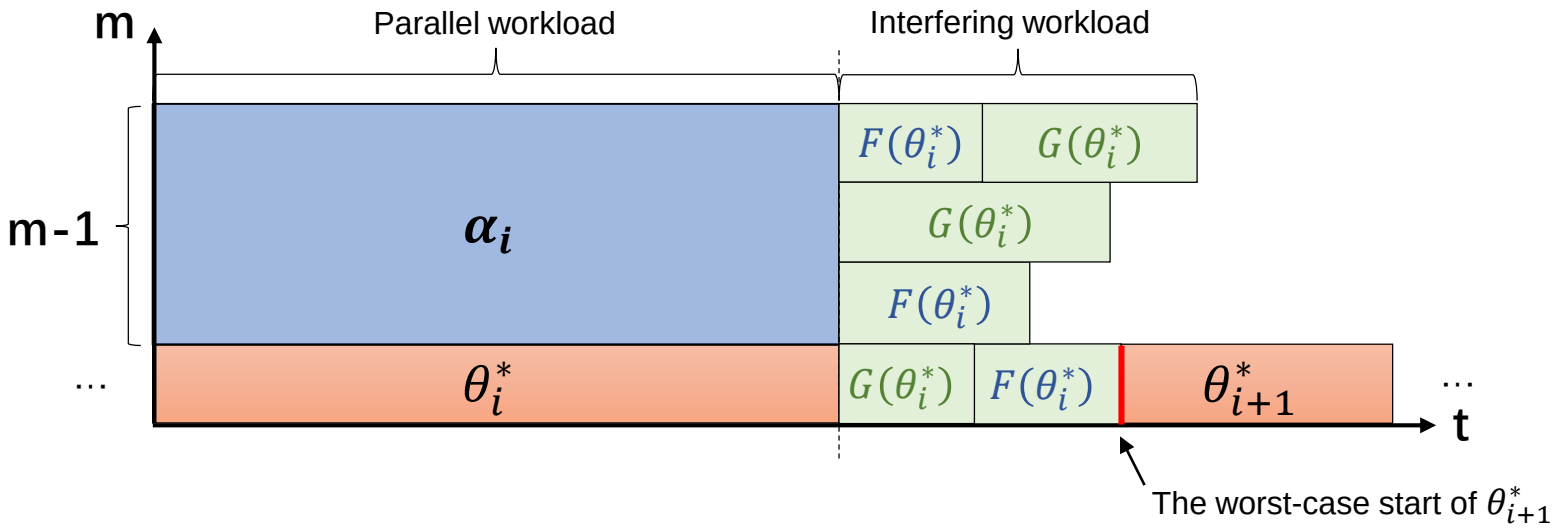
With CPC and schedule proposed, a new response time analysis is developed to achieve tighter bounds than existing methods.

The analysis provides two bounds:

- The first bound assumes critical path first with any execution order of non-critical nodes.
- The second bound assumes critical path first with an explicit execution order of non-critical nodes.

The analysis computes the worst-case finish of a DAG by:

1. bounds α_i for each provider $\theta_i^* \in \Theta^*$, and applies it as a safe reduction on the delay of critical path.
2. Bounds the delay caused by the interfering workload of each $\theta_i^* \in \Theta^*$, i.e., the worst-case finish of nodes in $F(\theta_i^*)$.



Experimental Results

The experiments aim to demonstrate the effectiveness of proposed schedule and analysis.

Evaluated methods:

- The proposed analysis that only assumes critical path first execution (*rta-cpf*).
- The proposed analysis for explicit execution order (*rta-cpf-eo*).
- The proposed rule-based execution order (*EO*).
- The classic analysis for any work-conserving schedule (*classic*).
- The state-of-art schedule and analysis methods (*He et al. 2019* [1]).

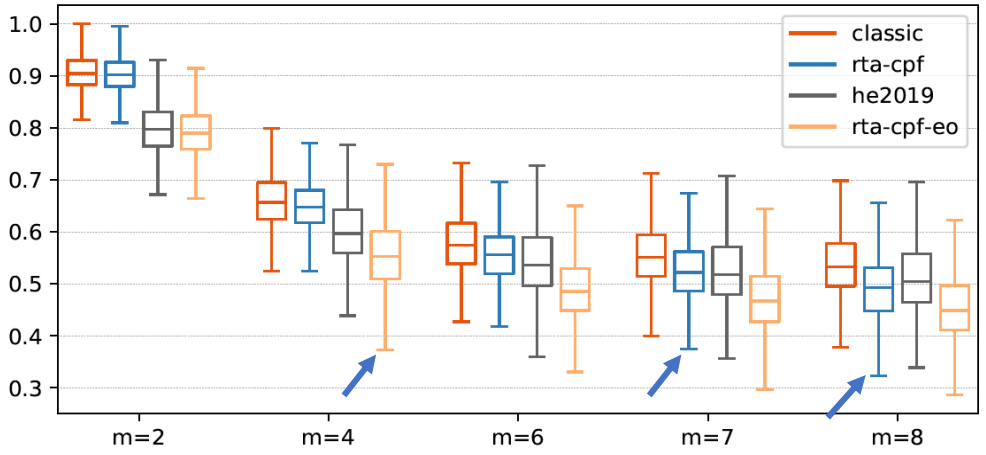
System Setting:

- Number of cores m.
- A parallelism indicator p.
- The length of critical path $\%L$.

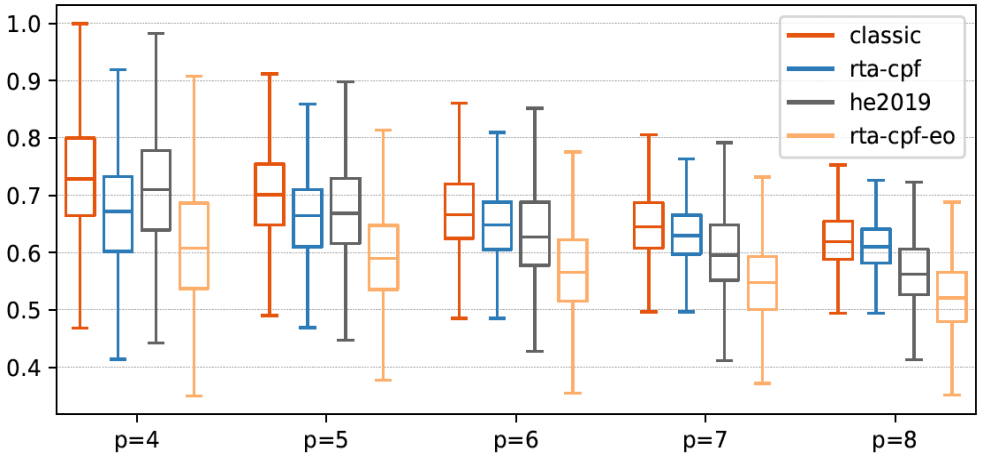
[1] Q. He, N. Guan, Z. Guo et al., *Intra-task priority assignment in real-time scheduling of DAG tasks on multi-cores*, IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 10, pp. 2283–2295, 2019.

Experimental Results

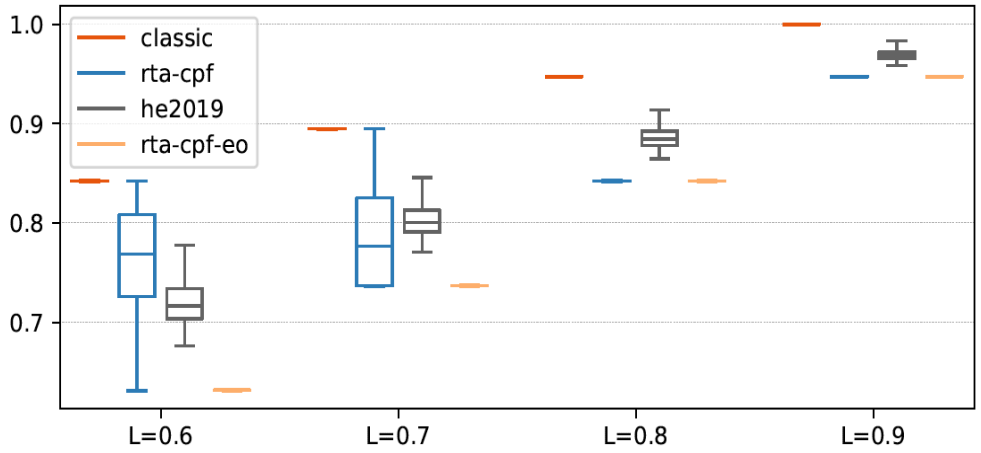
Evaluation of worst-case makespan:



With random p and $\%L$



With $m = 4$ and random $\%L$



With $m = 4$ and $p = 8$

The proposed methods leads to shorter (more accurate) makespan when:

1. The number of cores is relatively high.
2. The parallelism of the DAG is relatively low (for *rta-cpf* only).
3. The critical path is long.

Experimental Results

Effectiveness of the proposed schedule and analysis:

Table I: Percentage of improvement in advantage cases w.r.t. node ordering policy

	EO $\succ$ He2019							EO $\prec$ He2019						
	m=2	m=3	m=4	m=5	m=6	m=7	m=8	m=2	m=3	m=4	m=5	m=6	m=7	m=8
avg.	7.89	8.05	7.21	6.77	6.18	5.72	5.41	6.47	5.92	4.53	3.24	2.52	1.64	1.65
max.	30.63	36.18	33.39	34.17	30.65	27.75	25.27	30.68	27.19	23.83	21.59	24.09	16.76	19.26
min.	0.05	0.02	0.02	0.02	0.02	0.02	0.03	0.01	0.04	0.02	0.03	0.03	0.02	0.03

Table II: Advantage cases numbers and scientific significance in node-level priority assignment – EO and He2019 ordering both implemented in (α, β) analysis.

m	Dataset	# of data	Magnitude
2	EO $\succ$ He	668	medium
	He $\succ$ EO	261	medium
4	EO $\succ$ He	450	medium
	He $\succ$ EO	276	small
6	EO $\succ$ He	298	small
	He $\succ$ EO	255	negligible
8	EO $\succ$ He	192	small
	He $\succ$ EO	184	negligible

The proposed schedule has a better performance:

1. A higher percentage and significance of improvement.
2. A higher number of advantage cases.

Similar observations are also obtained for the proposed and existing analysis.

Conclusion

In this work, we proposed:

- A novel concurrent provider and consumer model.
  - Captures node parallelism and dependency.
  - Can be applied recursively to fully phrase a DAG.

- A rule-based scheduling method.
  - Improves node-level parallelism.
  - Reduces the makespan of the DAG.

- A response time analysis.
  - Fully exploits parallel execution.
  - Tightens the worst-case makespan approximations.

In future work, we aim to:

- Further optimizing the proposed methods.
- Supporting multi-DAGs with a work-conserving approach.

Thank you for your attention!