Article:
Yi, Huixuan, Zhang, Yuanhai, Lin, Zhiyang et al. (4 more authors) (2025) A cache-aware DAG scheduling method on multicores:Exploiting node affinity and deferred executions. Journal of systems architecture. 103372. ISSN 1383-7621

https://doi.org/10.1016/j.sysarc.2025.103372

A Cache-aware DAG Scheduling Method on Multicores: Exploiting Node Affinity and Deferred Executions

Huixuan Yi†, Yuanhai Zhang†, Zhiyang Lin†, Haoran Chen†, Yiyang Gao†, Xiaotian Dai§, Shuai Zhao†*
†Sun Yat-sen University, China §University of York, UK

***Abstract*—With increasingly complex functionalities being implemented in emerging applications, multicores are widely adopted with a layered cache hierarchy, and Directed Acyclic Graphs (DAGs) are commonly employed to model the execution dependencies between tasks. For such systems, scheduling methods can be designed to effectively leverage the cache to accelerate the system execution. However, the traditional methods either do not consider DAGs, or rely on sophisticated static analysis to produce fixed scheduling solutions that require additional hardware support (*e.g.*, cache partitioning and colouring), which undermines both the applicability and flexibility of these methods. Recently, an online cache-aware DAG scheduling method has been presented that schedules DAGs using an execution time model with caching effects considered, eliminating the need for static analysis and additional hardware support. However, this method relies on simple heuristics with limited considerations on both the allocatable cores and the competition between nodes, resulting in intensive inter-node contention that undermines cache performance. This paper proposes CADE, a cache-aware scheduling method for DAG tasks that leverages the cache to reduce DAG makespan. To achieve this, an affinity-aware priority assignment is first constructed that mitigates the competition among nodes for their preferred cores to hit the cache. Then, a contention-aware allocation mechanism is constructed, which (i) accounts for the impact of an allocation decision on the speed-up of other nodes; and (ii) includes the busy cores for allocation by enabling the deferred execution, effectively enhancing the cache performance to accelerate the DAG execution. Experiments show that compared to the state-of-the-art, the CADE significantly reduces the DAG makespan by 24.02% on average (up to 33%) with the cache miss rate reduced by 22.06% on average.**

***Index Terms*—Cache-Aware Scheduling, Direct Acyclic Graphs, Multi-core Systems**

I. Introduction

With the ever-growing complexity of emerging applications (*e.g.*, 5G telecommunications, autonomous driving, and smart manufacturing), multicore architectures are increasingly employed to meet computational demands, in which a layered cache hierarchy is often equipped to accelerate system execution [1]–[3]. In addition, to enable complex functionalities, parallel tasks in the system often require frequent communication and synchronisation, leading to pervasive execution dependencies between the tasks that can be modelled as a Directed Acyclic Graph (DAG) [4]. Fig. 1 presents an example DAG task with ten nodes, in which a node indicates a series of computations that must be executed sequentially, and an edge connecting two nodes gives their execution dependency. For instance, v_6 can start only when v_2 and v_3 have finished.

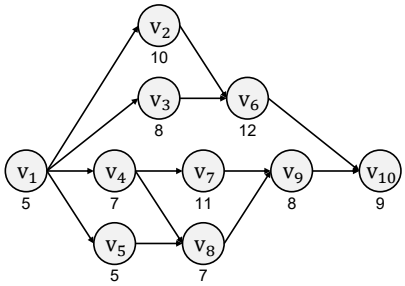


Fig. 1: An example DAG task with ten nodes *(numbers in black: the worst-case execution time of nodes)*.

For such systems, the complex dependency and parallelism relationships between the nodes pose significant challenges for DAG scheduling, which is proven as an NP-hard problem [5]. Furthermore, such challenges are greatly intensified when caching effects are considered, where the node ordering and allocation decisions would cause a substantial impact on the cache performance [6]. Unfortunately, most DAG scheduling methods [7]–[12] completely neglect the caching effects, which can cause frequent cache misses and impose substantial memory latency that significantly affects the execution of the DAG task, undermining the performance of the system with prolonged DAG makespan [1].

In addition, most existing cache-aware approaches do not address DAG tasks [13]–[15], which rely on an over-simplified task model where all tasks are independent of each other. For DAGs, the strict precedence constraints would significantly affect the scheduling order produced by these methods, leading to a cache speed-up much lower than expected [16]. In addition, some methods enhance cache performance by either applying static code analysis or mandating additional support from hardware platforms and compilers to manipulate the cache, *e.g.*, cache partitioning [15], [17] and cache colouring [14]. However, for large systems on COTS (commodity-off-the-shelf) architectures, these approaches face significant challenges in applicability and effectiveness due to the reliance on static analysis and additional hardware support.

Focusing on DAGs, an online cache-aware scheduling method (named the AJLR) is proposed in [6]. Unlike most cache-aware methods, the AJLR utilises an execution time approximation model with caching effects to guide the scheduling of DAGs, leveraging the cache to accelerate the execution. The use of the approximation model eliminates the need for static analysis or additional hardware support. However, it

This work is supported by National Key R&D Program of China under Grant 2023YFB4503703, Guangdong Basic and Applied Basic Research Foundation under Grant 2024A1515010240, Guangzhou Fundamental Research Funds under Grant SL2023A04J00996/2024A04J3903, and FoShan NanHai key area research under Grant 2230032004606.
*Corresponding author: Shuai Zhao, zhaosh56@mail.sysu.edu.cn.

relies on simple and restricted heuristic rules for node ordering and allocation, which neglects the competition of nodes for favourable cores and considers only the idle cores as the allocation candidates. This results in intensive node contention for cores, leading to less effective scheduling solutions that fail to fully leverage the cache to enhance system performance.

Contributions: This paper presents a novel Cache-aware DAG scheduling method that exploits node Affinity and Deferred Executions to enhance the cache performance, namely the CADE. To achieve this, an affinity-aware node priority ordering is constructed, which assigns node priorities by taking the competition of nodes over the favourable cores into account, effectively mitigating the node contention that improves the overall speed-up from the cache. In addition, a contention-aware allocation mechanism is proposed with the deferred execution enabled, which dispatches nodes to cores by (i) examining the impact of allocation decisions on the speed-up of other nodes and (ii) considering the busy cores as candidates via deferred executions, enhancing the speed-up gained from the cache that effectively accelerates the DAG execution. The evaluation shows that the proposed CADE outperforms the AJLR by 24.48% on average (up to 33%) in terms of the DAG makespan, by effectively leveraging the cache (reducing the cache miss rate by 22.06% on average).

Organisation: The rest of the paper is organised as follows: Sec. II presents the related work. Sec. III gives the system and task model. Sec. IV presents the working process of CADE. The constructed priority assignment and the allocation are described in Sec. V and Sec. VI, respectively. Finally, Sec. VII presents the experiments and Sec VIII concludes the paper.

II. Related Work and Limitations

There are a large number of works proposed in recent years focusing on cache-aware task scheduling and allocation [13]–[18]. In [13], a non-preemptive fixed-priority scheduling approach is proposed with the shared cache partitioned for each task, eliminating the inter-core cache interference to enhance system predictability. In [14], a dynamic cache page allocation for tasks is proposed based on cache colouring. This approach continuously monitors cache performance to enable adaptive cache repartitioning for tasks. In [15], an execution profile is proposed to analyse resource usage and cache behaviour, enabling the dynamic allocation of tasks on cache partitions. However, these methods often rely on complex static analysis (*e.g.*, cache-related behaviours analysis [15]) or instruction-level code analysis [16]; or mandate additional hardware support, *e.g.*, cache partitioning [15], [17], colouring [14], [18] or locking [13]. This significantly undermines the generality and applicability of these methods [6].

In addition, the above methods are designed for independent tasks without considering any execution dependencies with them. For the parallel tasks modelled as DAGs, existing works can be categorised by the following scheduling schemes: (i) the *global* scheme defines that a node can be executed on any cores in each release [7], [19], [20]; (ii) the *partitioned* scheme applied specifies that a node is designated with a fixed allocation for all its releases [8], [21]–[23]; (iii) the *federated* scheme is a hybrid of the global and partitioned schemes where certain nodes are assigned with fixed allocations [9], [24], [25]; and (iv) the *semi-partitioned* scheme provides a finer-grained mapping where each release of a node is assigned with a specific allocation, providing an effective balance between the global and partitioned schemes [10], [26], [27]. However, most DAG scheduling methods assume that nodes will always execute with their worst-case execution times (WCETs), which completely neglect the impact of caching effects on the execution time of nodes when scheduling the nodes. This can cause extensive memory access latency due to frequent cache misses, leading to the underutilisation of the cache with prolonged DAG makespan.

The state-of-the-art cache-aware scheduling method for DAG tasks (named AJLR) is proposed in [6]. The AJLR is an online scheduling method based on the semi-partitioned scheme, which consists of a priority assignment algorithm and two allocation rules that orders and dispatches the ready nodes to idle cores at each scheduling point. For the node ordering, the AJLR assigns priorities to ready nodes based on the rule of the highest node WCET first (HWF). By prioritising the node with a higher WCET, the node is dispatched first with more cores available for allocation, and hence, is more likely to have a high speed-up from the cache. However, the AJLR neglects the potential competition of the high-priority nodes for the same preferred cores, at which they can obtain a high cache speed-up. If multiple high-priority nodes compete for the same core, only one node can be allocated to that core. This can cause frequent cache misses due to the contention for cores, leading to the following limitation.

Limitation 1. *AJLR orders nodes solely based on their WCETs without considering the contention among nodes for preferred cores, leading to intensive core competition that significantly undermines the cache performance.*

With the ready nodes ordered by their priorities, the AJLR decides their allocations by estimating the execution time with caching effects of a node on a given core based on a Cache Reuse Profile (CRP) constructed by a measurement-based approach (See Sec. III-B). This timing model allows AJLR to operate without the need for complex static analysis or additional hardware support. At a scheduling point, AJLR first constructs a speed-up table that contains the absolute speed-up for all possible allocations based on the timing model, in which the one with the maximum speed-up value is always chosen as the allocation decision. Intuitively, this in general improves the cache performance that can reduce the DAG makespan.

However, once an allocation decision is made, the selected core becomes unavailable for other nodes, potentially resulting in allocations with significantly reduced cache speed-up for those nodes. In addition, AJLR restricts its consideration to the idle cores only, which can be limited at a scheduling point, especially for systems with a high workload. In such cases, the cores that are better suited for the ready nodes may not be available at the current scheduling point, yet are completely ignored by the allocation process of the AJLR. Hence, such a restriction would significantly undermine the effectiveness of the method. Based on the above, AJLR suffers

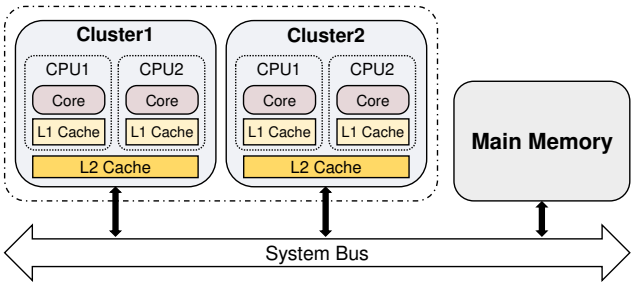


Fig. 2: System architecture with a two-layered cache.

from Limitations 2 and 3 that undermine its effectiveness.

Limitation 2. *AJLR focuses on maximising the speed-up of individual nodes without considering the core competition of other ready nodes, resulting in high contention with undermined cache performance.*

Limitation 3. *The AJLR only considers the idle cores during the allocation, which fails to fully leverage the cache to accelerate node execution, leading to underutilised caches and prolonged makespan.*

Motivation. To address the above limitations, this paper proposes a cache-aware DAG scheduling method (named CADE) that enhances the cache performance of DAG tasks to reduce the makespan. To achieve this, an affinity-aware node priority ordering is proposed which accounts for the node competition over favourable cores when assigning priorities, effectively mitigating the core competition with improved overall speed-up for the ready nodes. Then, an inter-core contention-aware allocation mechanism with deferred executions is constructed, which (i) considers the speed-up gain and loss collaboratively when examining a potential allocation and (ii) takes the busy cores into account to identify the most appropriate allocation of a node, effectively leveraging the cache to accelerate the node execution. Below we first describe the system model and preliminaries of the constructed method (Sec. III), and detail its overall working process (Sec. IV). Then, the proposed priority assignment and the allocation mechanisms are described in Sec. V and Sec. VI, respectively.

III. System Model and Preliminaries

This section describes the system and task model (Sec. III-A), and the preliminaries for constructing the cache-aware DAG scheduling method (Sec. III-B). Notations introduced in this section are summarised in Tab. I.

A. System and Task Model

System Model. The system considered contains m symmetric cores $\mathcal{P} = \{p_1, p_2, ..., p_m\}$, where the k^{th} core is denoted as p_k. Cores in the system are grouped into clusters, with each cluster containing a fixed number of cores. The system employs a two-layered cache hierarchy, as shown in Fig. 2. Such a two-layered cache hierarchy is commonly applied in many COTS architectures, such as the ARM big.LITTLE architecture [28] and Cortex A72 [29], etc. Under this setup, the L_1 cache is dedicated to one individual core, whereas the L_2 cache is shared among cores within a cluster. Both L_1 and L_2 caches are inclusive, in which any data stored in the L_1 cache is also present in the L_2 cache [28]. The Least Recently Used algorithm [30] is applied as the cache replacement policy.

TABLE I: Notations introduced in Sec.III

Notation	Description
$\mathcal{P}$	The set of cores in the system.
m	The number of cores in the system.
p_k	The k^{th} core among m symmetric cores.
L_η	The η level of cache with $\eta \in \{1, 2\}$.
τ / τ_l	A periodic DAG task / the l^{th} job of τ.
V / E	The set of the nodes / edges in the DAG τ.
T	The period of τ.
W	The workload of τ.
v_i / v_i^l	A node from a DAG task / the l^{th} release of v_i.
C_i	The worst-case execution time (WCET) of v_i.
$t(v_i^l)$	The time at which the node v_i^l starts executing.
ρ_i^l / α_i^l	The priority and allocation of node v_i^l.
$\mathcal{Q}^r$	The set of the ready nodes at the scheduling point.
$\mathcal{Q}^*$	The set of nodes to be examined for dispatching.
$\mathcal{P}^*$	The set of idle cores at a scheduling point.
$\lvert \cdot \rvert$	The size of a given set.
$\mathbb{H}$	The node allocation history table for all cores.
$\mathbb{H}(p_k)$	The list of nodes that are executed on p_k.
$\mathcal{AT}$	The list of earliest available time for all cores.
$\mathcal{AT}(p_k)$	The earliest available time of p_k.
Ω_i	The CRP model of node v_i.
$\Omega_i(L_\eta)$	The reuse threshold for the L_η cache provided by Ω_i.
$S(v_i, p_k)$	The execution time speed-up approximated by the CRP for node v_i on the given core p_k.

Task Model. This work focuses on the scheduling of a single periodic DAG task. A DAG is defined as $\tau = \{V, E, W, T\}$, where V represents the set of nodes in τ, $E \subseteq (V \times V)$ represents the set of directed edges between nodes, W denotes the total workload of τ, and T denotes the period of τ. For a node $v_i \in V$, its worst-case execution time (WCET) is denoted as C_i. An edge $(v_i, v_j) \in E$ represents the execution dependency between v_i and v_j, where v_j can only start after v_i has finished. In a DAG, the node without any predecessor is defined as the source node, and the node without any successor is considered as the sink node. As with [6], [31], we assume that a DAG has a sole source node and a sole sink node in this paper. For the DAG in Fig. 1, v_1 and v_{10} are the source node and the sink node, respectively. For DAGs with multiple source (resp. sink) nodes, a dummy source (resp. sink) node with zero WCET can be added so that this assumption still holds. The workload W of τ is calculated as $W = \sum_{v_i \in V} C_i$.

At runtime, τ is released periodically with the period T, where each release is referred to as a job. The l^{th} job of τ is denoted as τ_l. During the execution of τ_l, node v_i^l is assigned with a priority ρ_i^l and a specific allocation α_i^l by the proposed method. A higher numeric value of ρ_i^l indicates a higher priority. The time duration from the release of a job τ_l to its completion is defined as its *makespan*.

Scheduling Model. The Non-Preemptive Fixed-Priority Scheduling scheme is applied [32]. At a scheduling point, the ready nodes (*i.e.*, nodes where all the predecessors have finished execution) are listed in a ready queue $\mathcal{Q}^r$, sorted by node priorities in decreasing order. The set of idle cores is

denoted as $\mathcal{P}^*$. The number of such cores ($|\mathcal{P}^*|$) indicates the maximum number of nodes in $\mathcal{Q}^r$ that can be dispatched for execution at the current scheduling point, with $\mathcal{Q}^*$ denotes the set of nodes prioritised to be examined for dispatching.

Starting from the highest-priority node, the scheduler dispatches the ready nodes to the idle cores based on the proposed allocation method. Once a node starts executing, it runs to completion without being preempted by other nodes on the designated core. Note, for different releases of the same node (*e.g.*, v_i^l and v_i^{l-1}), they can be dispatched to different cores based on the allocation method.

As with the method in [6], the proposed scheduling method requires the system to maintain an allocation history table $\mathbb{H}$ that records the nodes being dispatched on each core. Function $\mathbb{H}(p_k)$ returns the list of nodes that are executed on p_k, in which the most recent node is always placed at the head of the list. Notation $\mathcal{AT}(p_k)$ represents the earliest available time of p_k, where $\mathcal{AT} = \{\mathcal{AT}(p_1), ..., \mathcal{AT}(p_m)\}$ contains the $\mathcal{AT}(p_k)$ of all cores.

B. Cache Reuse Profile

To account for the caching effects in DAG scheduling methods, the Cache Reuse Profile (CRP) constructed in [6] is applied in this work to approximate the execution time of a node with caching effects. The CRP is constructed for each node $v_i \in V$, denoted as Ω_i for v_i. For a release of v_i (say v_i^l), its CRP approximates the cache miss effects and the resulting execution time based on the *cache reuse distance (RD)* between v_i^l and its previous release v_i^{l-1}, which indicates the number of unique cache line accesses of other nodes during the two releases of v_i [6]. The computation of the reuse distance and the construction of CRP for v_i is described below.

In general, the computation of the exact cache reuse distance of two consecutive releases of a node (*i.e.*, the number of unique cache line accesses by others, denoted as $\Delta NoUC$ in Fig. 3) heavily relies on static code analysis and simulations, which suffer from severe scalability issues that undermine the applicability for large-scale applications. However, as demonstrated in [6], the number of unique cache line accesses is often closely correlated to the execution time of these nodes in a general case. Accordingly, the reuse distance between v_i^{l-1} and v_i^l on a given level of cache L_η can be estimated as the total execution time of other nodes that accesses the $L\eta$, as shown in Eq. 1 [6]. Function $t(\cdot)$ denotes the time that a node starts its execution and $\mathbf{g}(\cdot)$ provides the sum of the execution time of nodes that (i) are executed between the given time interval and (ii) access the cache L_η. This significantly reduces the complexity compared to the static analysis. More importantly, as demonstrated, the CRP provides effective approximations that can guide the allocation process, enhancing the cache performance. Detailed justifications of the CRP are referred to [6].

$$RD(v_i^l, L_\eta) = \mathbf{g}\Big(t(v_i^l) - t(v_i^{l-1}), L_\eta\Big) \tag{1}$$

Based on the reuse distance, the CRP is constructed for each node by learning from the measured execution time of the node under varied reuse distances. In general, as the distance grows, a node would experience gradual cache misses from L_1 to L_2, and eventually miss all cache levels in a general case (see Fig. 3). For a release of v_i (*e.g.* v_i^l), a designated core p_k, and its CRP Ω_i, the execution time is estimated by determining whether v_i^l can hit any of the two levels of cache. For a v_i^l and a candidate core p_k, the speed-up on its worst-case execution time (*i.e.*, C_i) due to the caching effects is denoted as $S(v_i^l, p_k)$, which is computed as follows. Starting with the L_1 cache, v_i^l hits the cache and obtains a speed-up if the following two conditions are satisfied:

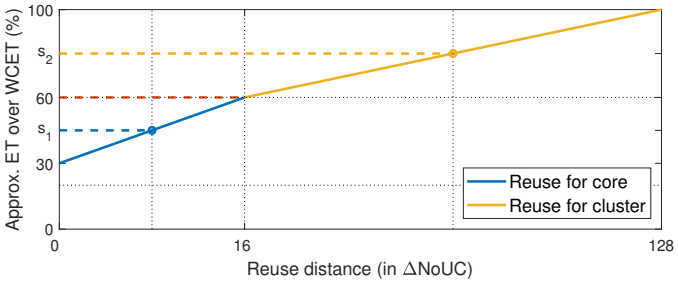


Fig. 3: An example CRP with a two-layered cache from [6].

C1. A previous instance of v_i^l is found on the same cache, *e.g.*, v_i^{l-1} is found on the allocation history of the cache.

C2. The reuse distance of v_i^l is less than the threshold of the cache in the CRP, which is denoted as $\Omega_i(L_\eta)$ for L_η hereafter, *e.g.*, 16 for L_1 cache in Fig. 3.

If v_i^l hits the L_1 cache, the speed-up over its WCET (*e.g.*, $s_1\%$ in Fig. 3) is obtained based on the cache reuse distance. Accordingly, the execution time of v_i^l on p_k is approximated by $C_i \times s_1\%$. Otherwise, the CRP examines whether v_i^l can hit the L_2 cache using the same process. If v_i cannot hit any level of cache, no speed-up is obtained so that execution time is approximated as the WCET.

The CRP provides the foundation for the proposed CADE, allowing effective allocation decisions to be made that better utilise the cache to improve the system performance (see Sec VII). Without the CRP, complex static analysis would be required to assess the impact of caching effects on the node execution time, such as instruction-level code analysis, leading to significant computational overhead and reduced efficiency in allocation algorithms. In addition, the use of CRP removes the need for the additional support of underlying hardware and compilers (*e.g.*, cache colouring [33] and locking [34]), enhancing the applicability of the proposed method [6].

IV. The Working Process of CADE

Before diving into the technique details, this section first presents the overall working process of CADE and its major components. The CADE is an online scheduling method for DAGs, which is executed at each scheduling point to produce a job-level schedule for nodes in $\mathcal{Q}^r$ (*i.e.*, the set of ready nodes). A scheduling point is triggered when a core becomes idle and there are nodes in Q^r waiting for execution. Note, as CADE focuses on scheduling nodes in one DAG release, we omit the index l for v_i^l hereafter in the proposed method for simplicity (*i.e.*, v_i is used where applicable).

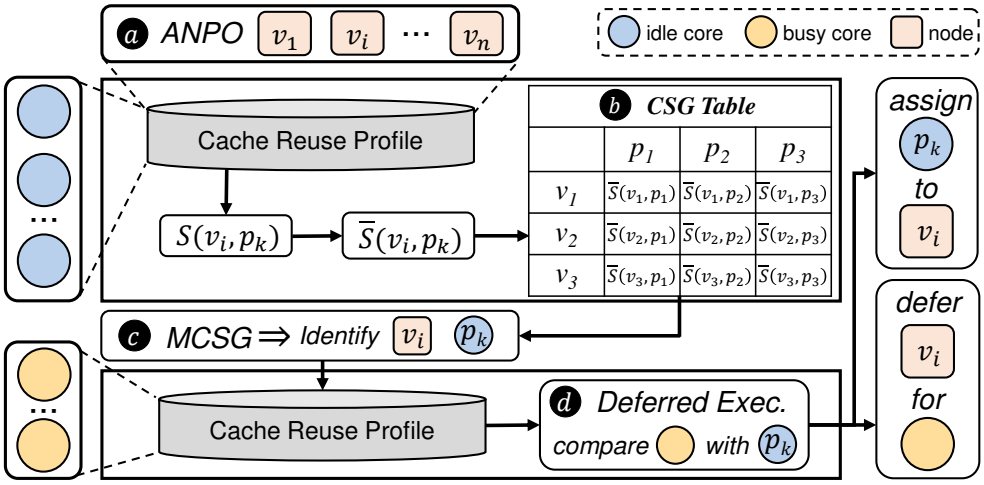


Fig. 4: The framework overview of a CADE-scheduled system.

The overall working process of CADE is presented in Alg. 1, with an illustration provided in Fig. 4. At a scheduling point, the algorithm is invoked and produces scheduling decisions for nodes in $\mathcal{Q}^r$. The algorithm takes the ready queue $\mathcal{Q}^r$, the set of idle cores $\mathcal{P}^*$ and the set of busy cores $\mathcal{P}^\neg$ as the inputs. In addition, two node queues $\mathcal{Q}^s$ and $\mathcal{Q}^d$ are initialised. The $\mathcal{Q}^s$ stores the nodes allocated by CADE at the current scheduling point. The $\mathcal{Q}^d$ contains nodes that are deferred by CADE, which will be examined at the next scheduling point for a higher speed-up and earlier completion.

At a scheduling point, CADE starts by prioritising the ready nodes Q^r using an affinity-aware node priority assignment (line 2), *i.e.*, the ANPO in Fig. 4a. The ANPO considers the core affinity of each node (*i.e.*, cores where it can hit the cache), and prioritises the nodes based on the contention over their preferred cores. The working mechanism and construction of ANPO are detailed in Sec. V. Given the set of idle cores $\mathcal{P}^*$, at most $|\mathcal{P}^*|$ nodes will be dispatched for execution at the current scheduling point (line 4). The notation $\mathcal{Q}^*$ denotes the set of nodes to be examined for dispatching with $|\mathcal{Q}^*| \leq |\mathcal{P}^*|$, which contains $|\mathcal{Q}^*|$ nodes with the highest priority in $\mathcal{Q}^r$.

Given $\mathcal{Q}^*$ and $\mathcal{P}^*$, the CADE then produces node allocations that leverage cache to speed up the node execution (lines 6-17). To approximate the cache speed-up gain of different allocation decisions, a contention-aware speed-up gain (CSG) table (Fig. 4b) is constructed for all possible allocations (line 6). For a given v_i and a p_k, the CSG estimates the speed-up of a node v_i on a given core p_k, with the potential loss of other nodes considered p_k would become unavailable for their allocations. The CSG table provides a comprehensive view of the speed-up gain and loss of all nodes for each allocation, providing effective guidance for CADE to produce allocations that fully exploit the cache to enhance the system performance.

Based on the CSG, two allocation rules are constructed: (i) the *Maximum CSG First* (MCSG), as shown in Fig. 4c; and (ii) the *Conditional Deferred Execution* (CDE), as shown in Fig. 4d. The MCSG identifies the allocation (*i.e.*, a pair of v_i and p_k) with the maximum CSG value (line 9). For the identified v_i and p_k, the CDE further compares the speed-up of v_i on p_k with the speed-up expected on the busy cores, determining whether it is beneficial to postpone the schedule of v_i for a certain busy core that can provide a higher speed-up with earlier completion of v_i's execution (line 11).

Based on MCSG and CDE, the CADE examines every node $v_i \in \mathcal{Q}^*$, in which the node with the highest CSG value is always considered first. For a given node v_i, the algorithm determines whether v_i should be dispatched or deferred to obtain a higher speed-up. If v_i is deferred by CDE, v_i is placed in $\mathcal{Q}^d$, in which the nodes will be postponed to the next scheduling point for allocation (lines 11-12). Otherwise, it is dispatched to the core with the highest CSG value (*i.e.*, MCSG) and is added to $\mathcal{Q}^s$, which contains all the allocated nodes with their α_i decided (line 14). After v_i is examined, the algorithm updates $\mathcal{Q}^r$ and $\mathcal{Q}^*$ to remove v_i (line 16). Finally, the CADE terminates when all the idle cores are assigned with a node or every node in $\mathcal{Q}^r$ is examined, where the allocation (*i.e.*, α_i) for each v_i in $\mathcal{Q}^s$ and the set of nodes deferred for the next scheduling point $\mathcal{Q}^d$ are returned (line 19).

In addition, the CADE maintains the allocation history table $\mathbb{H}$ and tracks the earliest available time $\mathcal{AT}(p_k)$ for each p_k in the system. The $\mathbb{H}$ is required by the CRP to approximate the speed-up for the given node and core, providing critical guidance for the scheduling decisions made by CADE. The $\mathcal{AT}$ provides the times at which the busy cores become available, which is adopted in the implementation of the deferred execution mechanism in Sec. VI. Below we first introduce the proposed node priority assignment ANPO (Sec. V). Then, we explain the constructed node allocation mechanism (Sec. VI), including the CSG table and the two allocation rules, forming the complete CADE method that fully leverages the cache to accelerate DAG execution.

V. Affinity-aware Node Priority Ordering

This section describes the construction of the ANPO described in Sec. IV (*i.e.*, Fig. 4a). The existing methods often schedule nodes simply based on their WCETs, *e.g.*, the *highest node WCET first (HWF)* [6], [35], [36]. However, these methods fail to consider the core contention among the ready nodes in terms of their preferred cores at which cache hits

Algorithm 1: Overall working process of CADE;

Input: $\mathcal{Q}^r$, $\mathcal{P}^*$, $\mathcal{P}^\neg$;
Initialise: $\mathcal{Q}^s = \varnothing$, $\mathcal{Q}^d = \varnothing$;

```
/* (a) Prioritises the ready nodes by ANPO.*/
sort Q^r by ρ_i, ∀v_i ∈ Q^r;
while Q^r ≠ ∅ ∧ P^* ≠ ∅ do
    Q^* = Q^r.first(min{|Q^r|, |P^*|});
    /* (b) Construct CSG table for all Q^* and P^*.*/
    S̄ =CSG(Q^*, P^*);
    while Q^* ≠ ∅ do
        /* (c) Rule 1: Maximum CSG first by Alg. 3.*/
        (v_i, p_k) = MCSG(S̄, Q^*, P^*);
        /* (d) Rule 2: Deferred Execution by Alg. 4.*/
        if CDE(v_i, p_k, P^*, P^¬) then
            Q^d = Q^d ∪ v_i;
        else
            α_i = p_k;   P^* = P^* \ p_k;   Q^s = Q^s ∪ v_i;
        end
        Q^r = Q^r \ v_i;   Q^* = Q^* \ v_i;
    end
end
return α_i, ∀v_i ∈ Q^s, Q^d;
```

can be achieved to accelerate the execution, *i.e.*, the affinity. Such an approach can lead to extensive contention if the nodes selected for execution can only benefit from cache on the same core, where only one node can be allocated on the preferred core. This significantly undermines the cache performance, thereby leading to prolonged DAG makespan.

To address this, the ANPO (affinity-aware node priority assignment) is constructed to assign priorities for ready nodes by considering (i) the core affinity of the ready nodes; and (ii) the contention over the preferred cores among the ready nodes. To achieve this, the affinity set (*i.e.*, the set of preferred cores) of each ready node is identified on which the node achieves cache hits. Based on this, a metric *Conflict Factor (CF)* is introduced to depict the degree of contention between a node and other ready nodes for a given idle core. Leveraging CF, the ANPO method assigns higher priorities to nodes with lower contention with other ready nodes. Below we first present the construction of the Conflict Factor. The notations introduced in the ANPO are summarised in Tab. II.

Computation of CF. The CF quantifies the severity of contention for the idle cores ($\mathcal{P}^*$) between a node v_i and other ready nodes, given the affinity set of each node. To calculate the value of CF, the affinity set where v_i can hit the cache is first derived, denoted as $\mathcal{P}(v_i)$ in Eq. 2. For each p_k in $\mathcal{P}^*$, the cache hit status of v_i is represented by $h_i^k \in \{L_1, L_2, \epsilon\}$, where L_1 and L_2 denote cache hits at the corresponding cache levels whereas ϵ indicates v_i achieves no cache hits on p_k. The h_i^k can be derived by the CRP following the approach outlined in Sec III-B. As defined in Eq. 2, p_k is incorporated into the affinity set of v_i only if v_i hits the caches on p_k, *i.e.*, $h_i^k \neq \epsilon$.

$$\mathcal{P}(v_i) = \{p_k \mid h_i^k \neq \epsilon, \forall p_k \in \mathcal{P}^*\} \tag{2}$$

With affinity set $\mathcal{P}(v_i)$ determined for each ready node v_i, the set of nodes that could contend for a specific core p_k is identified by Eq. 3. The function $H(p_k)$ returns the set of nodes that can hit the cache on p_k (*i.e.*, $p_k \in \mathcal{P}(v_i)$). If such nodes are dispatched at the same scheduling point, they might compete for p_k to accelerate the execution via cache.

$$H(p_k) = \{v_i \mid p_k \in \mathcal{P}(v_i), \forall v_i \in \mathcal{Q}^r\} \tag{3}$$

Based on $\mathcal{P}(v_i)$ and $H(p_k)$, the degree of contention that v_i can incur for being dispatched to p_k is computed in Eq. 4, denoted as $CF(v_i, p_k)$. Note, Eq. 4 only considers the case where p_k is one of its preferred cores (*i.e.*, $v_i \in H(p_k)$). Otherwise, it has no contention with other nodes on p_k. In addition, because the $CF(v_i, p_k)$ is applied as the divisor in Eq. 5, $CF(v_i, p_k) = 1$ if v_i incurs no contention on p_k.

$$CF(v_i, p_k) = 1 + \sum_{\forall v_j \in H(p_k) \setminus v_i} \frac{1}{|\mathcal{P}(v_j)|} \tag{4}$$

As shown in the equation, the $CF(v_i, p_k)$ is computed by examining the size of the affinity set of the other nodes that also prefer p_k, *i.e.*, the $|\mathcal{P}(v_j)|, \forall v_j \in H(p_k) \setminus v_i$. The intuition is that, if v_j has a large size of $\mathcal{P}(v_j)$, this indicates v_j has more candidate cores at which it can benefit from the cache, hence, leading to a lower contention with v_i on p_k. For instance, if $|\mathcal{P}(v_j)| = 5$, v_j has a number of allocation candidates at which it can still hit the cache, regardless of whether p_k is occupied. In contrast, if $\mathcal{P}(v_j) = \{p_k\}$, allocating v_i to p_k would result in v_j missing the cache without any execution speed-up. This generally leads to a more severe contention for core p_k between the ready nodes.

The Construction of ANPO. With $CF(v_i, p_k)$ computed, the affinity-aware node priority assignment (ANPO) is constructed that always assigns a higher priority value for a ready node with a lower contention on the preferred cores, where a node with a higher priority is always scheduled first. For a $v_i \in \mathcal{Q}^r$, its priority ρ_i is computed by Eq. 5, which examines the potential benefits of v_i and the resulting contention for other ready nodes if v_i is allocated on $p_k \in \mathcal{P}(v_i)$.

$$\rho_i = \sum_{\forall p_k \in \mathcal{P}(v_i)} \begin{cases} \frac{|\mathcal{C}(p_k)|}{CF(v_i, p_k)}, & \text{if } h_i^k = L_1 \\ \frac{1}{CF(v_i, p_k)}, & \text{if } h_i^k = L_2 \end{cases} \tag{5}$$

For a given p_k, the equation examines whether v_i can hit the L_1 or L_2 cache. If so, a reward is assigned as $|\mathcal{C}(p_k)|$ (*i.e.*, the number of cores in the cluster) and 1 for hitting the L_1 and L_2 cache, respectively, where a higher reward can lead to a higher priority. In addition, the $CF(v_i, p_k)$ is applied to consider the potential contention that v_i can impose on other ready nodes if it occupies p_k, in which a low $CF(v_i, p_k)$ results in a high ρ_i. By considering both the benefits of v_i and the contention

TABLE II: Notations introduced in ANPO.

Notation	Description
h_i^k	The cache hit status of node v_i on core p_k.
ϵ	A value of h_i^k indicating no cache hits.
$\mathcal{P}(v_i)$	The affinity set of v_i.
$H(p_k)$	The set of nodes that can hit the cache on p_k.
$\mathcal{C}(p_k)$	The set of cores within the cluster of p_k.
$CF(v_i, p_k)$	The contention degree of v_i for dispatching to p_k.

TABLE III: Notations introduced in CADE.

Notation	Description
$\mathcal{Q}_i^*$	Nodes to be examined excluding v_i, *i.e.*, $\mathcal{Q}^* \setminus v_i$.
$\mathcal{P}_k^*$	The set of idle cores excluding p_k, *i.e.*, $\mathcal{P}^* \setminus p_k$.
$S^\dagger(\mathcal{Q}_i^*, p_x)$	The maximum speed-up that the core p_x can offer for the given set of nodes $\mathcal{Q}_i^*$.
$\mathcal{P}^\diamond(v_j)$	The set of cores that provides v_j with the speed-up equals to $S^\dagger(\mathcal{Q}_i^*, p_x)$ for each $p_x \in \mathcal{P}_k^*$.
$S^\ddagger(v_j, \mathcal{P}_k^*)$	The maximum speed-up that $v_j \in \mathcal{Q}_i^*$ can obtain on the cores in $\mathcal{P}_k^*$.
$L(v_i, p_k)$	The highest speed-up loss among $v_j \in \mathcal{Q}_i^*$ if v_i is allocated on p_k.
$\overline{S}(v_i, p_k)$	Contention-aware speed-up gain if v_i is allocated on p_k.
$\mathcal{V}'$	The set of nodes with the highest CSG value in $\overline{S}$.
$\mathcal{P}'$	The set of cores with the highest CSG value for the node identified by MCSG.
$\mathcal{V}^*(p_k)$	The set of nodes executed on p_k that can hit the cache if they are dispatched on p_k again.
$R^2D(p_k)$	The minimum distance between reuse distance and the threshold $\Omega_j(L_2)$ among nodes in $\mathcal{V}^*(p_k)$.
t	The current time at the scheduling point.
$\overline{\mathcal{P}}(v_i)$	The set of cores that the node v_i has considered at previous scheduling points.
$\mathcal{P}^\neg$	The set of busy cores at the current scheduling point.

it imposes on other nodes for occupying a given core p_k, the priority of v_i is computed by iterating through every p_k on which v_i can hit the cache, *i.e.*, $\forall p_k \in \mathcal{P}(v_i)$.

With the ANPO applied, nodes that can hit the cache while causing a low contention with others are in general assigned a higher priority, which will be dispatched first at a scheduling point. At a scheduling point, the nodes in $\mathcal{Q}^r$ are ordered by their priorities in non-increasing order, in which the first $|\mathcal{P}^*|$ nodes (*i.e.*, $\mathcal{Q}^*$) will be considered by the allocation process for dispatching. When multiple nodes have the same priority value (*i.e.*, ρ_i), the node with a higher WCET is selected first. Compared to the existing methods, the ANPO can mitigate the competition between nodes for certain cores while still leveraging the cache to accelerate the node execution, effectively reducing the resulting DAG makespan. This is further justified by the experimental results in Sec. VII. The Characteristic 1 describes the key feature of ANPO that addresses Limitation 1 in Sec. II.

Characteristic 1. *The ANPO considers the node competition over the preferred cores when assigning priorities, effectively mitigating the core contention among the ready nodes with enhanced cache performance. This addresses Limitation 1.*

VI. Contention-aware Allocation Mechanism with Deferred Execution

This section presents the allocation process of CADE. As described in Sec. II, the allocation mechanism of AJLR (*i.e.*, the SOTA) suffers from two major limitations: (i) it fails to account for the potential impact of an allocation on the cache speed-up of other ready nodes and (ii) it restricts the consideration of allocation to the idle cores only, neglecting potentially more suitable allocations that will be available in near future. These limitations result in less effective scheduling solutions for DAGs with high inter-core cache contention, which significantly undermines the effectiveness of AJLR, hence, leading to prolonged DAG makespan.

TABLE IV: Speed-up of each possible allocation by the CRP.

$S(v_i, p_k)$	p_1	p_2	p_3
v_1	510	500	500
v_2	400	600	100
v_3	500	400	200

TABLE V: The contention-aware speed-up gain table.

$\overline{S}(v_i, p_k)$	p_1	p_2	p_3
v_1	210	300	500
v_2	300	600	100
v_3	490	200	200

To address these limitations, the CADE allocates nodes by (i) considering the impact on cache speed-up caused by inter-core cache contention and (ii) taking into account the potential benefits of waiting for nearly released cores, with a deferred execution mechanism to enable improved allocation decisions, thereby enhancing cache performance and accelerating execution. To achieve this, a contention-aware speed-up gain table (CSG) is constructed that considers the impact of inter-core contention on the speed-up of ready nodes (Sec. VI-A). Based on the CSG, two allocation rules are designed for enhanced cache utilisation, forming the complete CADE method that schedules tasks with immediate allocation or deferred execution (Sec. VI-B). The notations introduced in this section are summarised in Tab. III.

A. The Construction of the CSG Table

This section describes the construction of CSG based on the execution time approximations produced by the CRP (*i.e.*, Fig. 4b). As described in Sec. II, the AJLR produces allocations solely based on the independent cache speed-up of each individual node, *i.e.*, $S(v_i, p_k)$. This ignores the potential impact of an allocation on the cache speed-up of other ready nodes, leading to limited cache performance. To address this, the CSG is constructed, which incorporates the potential speed-up loss resulting from other ready nodes that cannot be executed on a specific core.

Intuitions and Benefits. Before presenting the construction process of the CSG, we first illustrate the key intuition of the CSG and the benefits it brings to CADE. Consider a scheduling point in which there exist three ready nodes $\{v_1, v_2, v_3\}$ and three idle cores $\{p_1, p_2, p_3\}$, in which the $S(v_i, p_k)$ of every possible allocation decision is presented in Tab.IV. Below we explain (i) the necessity of the CSG by illustrating the limitation of the AJLR and (ii) the benefits brought by the CSG using two examples.

Example 1. *For the example system in Table IV, as the allocation with the highest $S(v_i, p_k)$ is always chosen in AJLR, v_2 is first dispatched to p_2 as $S(v_2, p_2) = 600$ is the highest speed-up among all possible allocations. Then, v_1 is allocated on p_1 following the same approach, where $S(v_1, p_1) = 510$. Finally, v_3 is left with p_3 as p_1 and p_2 are occupied. This schedule obtains a total speed-up of $510 + 600 + 200 = 1310$ for all three nodes.*

Example 1 illustrates the schedule produced by the AJLR, which obtains a total speed-up of 1310. However, it is not

difficult to find a better solution in which v_1, v_2, v_3 can be allocated to p_3, p_2 and p_1 respectively, achieving a higher speed-up of $500 + 600 + 500 = 1600$. From the example, the AJLR is limited to the local optimisation of each individual node during the scheduling process, overlooking the contention for nodes where multiple nodes can hit the same cache. Therefore, this method often leads to sub-optimal scheduling decisions that fail to effectively leverage the cache to accelerate node execution, leading to prolonged DAG makespan.

However, as an online scheduling method, global optimisation through extensive search would be impractical due to its high complexity. To address this issue, a more informative speed-up approximation with the node contention (denoted as $\overline{S}(v_i, p_k)$) is constructed, which considers both (i) the speed-up for dispatching a node to a core and (ii) the potential loss for other nodes as this core is no longer available. For v_1 in Tab. IV, if it is allocated on p_1, it would obtain a speed-up of 510 as described above. However, for v_2 and v_3, they cannot be allocated on p_1 again, in which v_2 incurs no loss if being allocated on p_2 whereas v_3 incurs a speed-up loss of 300 if it is allocated on p_3. Therefore, considering both the speed-up of v_1 and the loss of v_3, the final $\overline{S}(v_1, p_1)$ is $510 - 300 = 210$. Following a similar approach, Tab. V presents the contention-aware speed-up gain for the nodes in Tab. IV.

Example 2. *With CSG in Tab. V, v_2 would be scheduled first with the highest $\overline{S}(v_2, p_2)$. However, in contrast to Example 1, v_1 and v_3 are allocated to p_3 and p_1, respectively, addressing the issue of the low cache speed-up of v_3. Guided by the CSG, this schedule achieves a total absolute speed-up of $500{+}600{+}500 = 1600$ as suggested by Tab. IV, outperforming the schedule produced by AJLR in Example 1.*

Example 2 illustrates the resulting schedule based on the new speed-up metric, demonstrating its effectiveness in mitigating the impact on the speed-up due to the node contention on cores. This effectively improves the overall speed-up of the ready nodes rather than focusing on certain individual nodes.

Construction of CSG Table. As mentioned above, the CSG table provides approximations of the speed-up considering node contention for each ready node on every idle core. Instead of focusing solely on the speed-up of an individual node, the CSG table accounts for both the speed-up gained by dispatching a node to a core and the potential loss incurred by other nodes when the core becomes unavailable. By capturing the impact of inter-core contention, the allocation based on the CSG table mitigates the negative effects of node competition, contributing to improved overall speed-up.

We now describe the construction of the CSG table, as shown in Alg. 2. First, following the priority order based on ANPO, $\mathcal{Q}^*$ is obtained by taking the first $|\mathcal{P}^*|$ ready nodes in $\mathcal{Q}^r$ (if they exist), which will be examined for dispatching. In addition, the $S(v_i, p_k), \forall v_i \in \mathcal{Q}^*, \forall p_k \in \mathcal{P}^*$ are computed based on the allocation history $\mathbb{H}$ (see Sec. III-B), which provide the foundation for the constructed CSG table. Then, the algorithm starts by computing the contention-aware speed-up gain for each possible allocation, *i.e.*, $\overline{S}(v_i, p_k), \forall v_i \in \mathcal{Q}^*, \forall p_k \in \mathcal{P}^*$. For each $v_i \in \mathcal{Q}^*$ and every $p_k \in \mathcal{P}^*$ (line 1), the algorithm calculates $\overline{S}(v_i, p_k)$ by considering both the benefits of assigning v_i on p_k and the potential loss for other nodes if p_k becomes unavailable, based on the following computations. Below we detail the computations of each term.

Algorithm 2: CSG($\mathcal{Q}^*, \mathcal{P}^*$): the construction of $\overline{S}$.

```
1  for each v_i ∈ Q* and each p_k ∈ P* do
2      Q*_i = Q* \ v_i;   P*_k = P* \ p_k;
3      /* Maximum speed-up on each p_x by Eq. 6 */
4      for each p_x ∈ P*_k do
5          compute S†(Q*_i, p_x) by Eq. 6;
6      end
7      /* Speed-up estimation for v_j ∈ Q*_i on P*_k */
8      for each v_j ∈ Q*_i do
9          compute P◇(v_j) based on Eq 8;
10         compute S‡(v_j, P*_k) based on Eq. 7;
11     end
12     /* Speed-up loss estimation by Eq. 9*/
13     compute L(v_i, p_k) based on Eq. 9;
14     /* CSG value assuming v_i allocated to p_k */
15     S̄(v_i, p_k) = S(v_i, p_k) − L(v_i, p_k);
16 end
17 return S̄;
```

- $S^\dagger(\mathcal{Q}_i^*, p_x)$: the maximum speed-up of other ready nodes (*i.e.*, nodes in $\mathcal{Q}_i^* = \mathcal{Q}^* \setminus v_i$) on p_x (*i.e.*, a given core in $\mathcal{P}_k^* = \mathcal{P}^* \setminus p_k$) (lines 4-6);
- $S^\ddagger(v_j, \mathcal{P}_k^*)$: the maximum speed-up that v_j can obtain on the idles cores except p_k, *i.e.*, $\mathcal{P}_k^*$ (lines 8-11);
- $L(v_i, p_k)$: the highest speed-up loss of $v_j \in \mathcal{Q}_i^*$ if they are not being allocated on p_k (line 13);
- $\overline{S}(v_i, p_k)$: the CSG value of v_i if it is allocated on p_k based on $L(v_i, p_k)$ (line 15).

First, $S^\dagger(\mathcal{Q}_i^*, p_x)$ is computed in Eq. 6, which takes the highest speed-up value among the $S(v_j, p_x)$ of all the $v_j \in \mathcal{Q}_i^*$.

$$S^\dagger(\mathcal{Q}_i^*, p_x) = \max_{\forall v_j \in \mathcal{Q}_i^*} S(v_j, p_x) \tag{6}$$

Then, the maximum speed-up of v_j on cores in $\mathcal{P}_k^*$ (*i.e.*, $S^\ddagger(v_j, \mathcal{P}_k^*)$) can be obtained by Eq. 7. For a given $v_j \in \mathcal{Q}_i^*$, function $\mathcal{P}^\diamond(v_j)$ denotes the set of cores on which the speed-up of v_j equals the maximum speed-up that $p_x \in \mathcal{P}_k^*$ can offer for all nodes in $\mathcal{Q}_i^*$, *i.e.*, $S(v_j, p_x) = S^\dagger(\mathcal{Q}_i^*, p_x)$, as shown in Eq. 8. If $\mathcal{P}^\diamond(v_j) \neq \varnothing$, $S^\ddagger(v_j, \mathcal{P}_k^*)$ is computed as the maximum value of $S^\dagger(\mathcal{Q}_i^*, p_x), \forall p_x \in \mathcal{P}^\diamond(v_j)$. For $p_x \in \mathcal{P}^\diamond(v_j)$, v_j can obtain the highest speed-up compared to any other node in $\mathcal{Q}_i^*$, so that v_j are more likely to be allocated on such cores. Hence, the associated speed-up is used to determine $S^\ddagger(v_j, \mathcal{P}_k^*)$. Otherwise (*i.e.*, $\mathcal{P}^\diamond(v_j) = \varnothing$), the value of $S^\ddagger(v_j, \mathcal{P}_k^*)$ is computed as the maximum speed-up $S(v_j, p_x)$ of v_j among on every $p_x \in \mathcal{P}_k^*$.

$$S^\ddagger(v_j, \mathcal{P}_k^*) = \begin{cases} \max\limits_{p_x \in \mathcal{P}^\diamond(v_j)} S^\dagger(\mathcal{Q}_i^*, p_x), & \mathcal{P}^\diamond(v_j) \neq \varnothing \\ \max\limits_{\forall p_x \in \mathcal{P}_k^*} S(v_j, p_x), & \text{otherwise} \end{cases} \tag{7}$$

$$\mathcal{P}^\diamond(v_j) = \{p_x \mid S(v_j, p_x) = S^\dagger(\mathcal{Q}_i^*, p_x), \forall p_x \in \mathcal{P}_k^*\} \tag{8}$$

Algorithm 3: MCSG(): Maximum CSG First.

Input: $\overline{S}$, $\mathcal{Q}^*$, $\mathcal{P}^*$;
Output: (v_i, p_k);
Parameters: $\mathcal{V}' = \mathcal{P}' = \varnothing$;
1 $\mathcal{V}' = \mathcal{V}' \cup \underset{v_i}{\text{argmax}}\{\overline{S}(v_i, p_k) \mid \forall v_i \in \mathcal{Q}^*, \forall p_k \in \mathcal{P}^*\}$;
2 $v_i = \underset{v_i}{\text{argmax}}\{\rho_i \mid v_i \in \mathcal{V}'\}$;
3 $\mathcal{P}' = \mathcal{P}' \cup \underset{p_k}{\text{argmax}}\{\overline{S}(v_i, p_k) \mid \forall p_k \in \mathcal{P}^*\}$;
4 $p_k = \underset{p_k}{\text{argmax}}\{R^2D(p_k) \mid \forall p_k \in \mathcal{P}'\}$;
5 **return** v_i, p_k;

Algorithm 4: CDE(): Conditional Deferred Execution.

Input: v_i, p_k, $\mathcal{P}^*$, $\mathcal{P}^\neg$;
Parameters: $\overline{P}(v_i)$;
1 **for** each $p_x \in \mathcal{P}^\neg \setminus \overline{\mathcal{P}}(v_i)$ **do**
2 **if** $S(v_i, p_x) - S(v_i, p_k) > \mathcal{AT}(p_x) - t$ **then**
3 $\overline{\mathcal{P}}(v_i) = \overline{\mathcal{P}}(v_i) \cup \mathcal{P}^*$;
4 **return** true;
5 **end**
6 **end**
7 $\overline{P}(v_i) = \varnothing$;
8 **return** false;

To this end, the disparity between the speed-up of v_j on p_k (*i.e.*, $S(v_j, p_k)$) and on other idle cores (*i.e.*, $S^\ddagger(v_j, \mathcal{P}_k^*)$) can be calculated as $S(v_j, p_k) - S^\ddagger(v_j, \mathcal{P}_k^*)$ for every $v_j \in \mathcal{Q}_i^*$, as shown in Eq. 9. If a negative value is obtained, it indicates that v_j would not incur a speed-up loss if p_k is not available. In such cases, the value of speed-up loss for v_j is zero. Finally, the maximum speed-up loss among all nodes in $\mathcal{Q}_i^*$ is obtained as $L(v_i, p_k)$, indicating the potential impact for other ready nodes if v_i is allocated on p_k.

$$L(v_i, p_k) = \max_{\forall v_j \in \mathcal{Q}_i^*} \begin{cases} 0, & \text{if } S(v_j, p_k) - S^\ddagger(v_j, \mathcal{P}_k^*) < 0 \\ S(v_j, p_k) - S^\ddagger(v_j, \mathcal{P}_k^*), & \text{otherwise} \end{cases} \tag{9}$$

Following this, the $\overline{S}(v_i, p_k)$ can be determined as $\overline{S}(v_i, p_k) = S(v_i, p_k) - L(v_i, p_k)$ (line 15), highlighting the benefits gained by v_i against the potential impacts on other ready nodes if v_i is dispatched to p_k. Finally, Alg. 2 terminates after the $\overline{S}(v_i, p_k)$ of every v_i and p_k are computed, with the CSG table $\overline{S}$ returned at line 17.

Time Complexity. It is worth noting that a number of optimisations (*e.g.*, via dynamic programming) can be performed to achieve a time complexity of $\mathcal{O}(n^2 \times \log n)$ for constructing the CSG table. First, the $\mathcal{S}^\dagger(\cdot)$ and $\mathcal{S}^\ddagger(\cdot)$ can be computed and stored as look-up tables with a time complexity of $\mathcal{O}(n^2)$. Based on the look-up tables, time complexity of $\mathcal{O}(\log n)$ can be achieved for computing $L(\cdot)$, where the maximum value of $S(\cdot) - S^\ddagger(\cdot), \forall v_j \in \mathcal{Q}_i^*$ can be obtained by utilising a balanced tree structure to accelerate the computation. Therefore, $\overline{S}$ can be constructed with a complexity of $\mathcal{O}(n^2 \times \log n)$.

B. Construction of Allocation Rules

Based on the CSG table, the CADE produces allocation decisions based on two major rules: (i) Maximum CSG First (*i.e.*, MCSG in Fig. 4c); and (ii) Conditional Deferred Execution (*i.e.*, the CDE in Fig. 4d). The MCSG identifies the allocation (*i.e.*, a pair of v_i and a p_k) with the maximum $\overline{S}(v_i, p_k)$ value. Then, CDE determines whether to dispatch v_i according to MCSG or postpone the schedule of v_i for a more suitable core. Below we detail the construction of two rules.

Maximum CSG First. The working process of MCSG is presented in Alg. 3, which takes the $\overline{S}$, $\mathcal{Q}^*$ and $\mathcal{P}^*$ as the inputs. The algorithm starts by identifying the v_i that can achieve the maximum CSG value (lines 1-2). If multiple nodes are identified (*i.e.*, $|\mathcal{V}'| > 1$), the one with the highest priority ρ_i is selected for consideration, and in cases of equivalent priority, the node with larger WCET (c_i) is chosen.

Then, for the identified v_i, the algorithm identifies the core where v_i has the highest CSG value (lines 3-4). However, v_i may obtain the same maximum CSG value on multiple cores, *i.e.*, $|\mathcal{P}'| > 1$. In this case, MCSG analyses the allocation history of each $p_k \in \mathcal{P}'$ (denoted as $\mathbb{H}(p_k)$), and identifies the p_k where allocating v_i has the least impact on nodes previously executed on p_k. For such nodes, they are likely to be dispatched to the same core again to achieve a high cache speed-up. To achieve this, Eq. 10 provides the set of nodes that can hit the cache if they are dispatched again on p_k (denoted as $\mathcal{V}^*(p_k)$) based on $\mathbb{H}(p_k)$, in which $\Omega_j(L_2)$ gives the maximum reuse distance of v_j for hitting the cache.

$$\mathcal{V}^*(p_k) = \{v_j \mid RD(v_j, L_2) < \Omega_j(L_2), \forall v_j \in \mathbb{H}(p_k)\} \tag{10}$$

Then, for a p_k and the nodes executed on it (*i.e.*, $\mathcal{V}^*(p_k)$), the algorithm identifies the minimum distance between the reuse distance and the threshold $\Omega_j(L_2)$ among all $v_j \in \mathcal{V}^*(p_k)$, denoted as $R^2D(p_k)$ in Eq. 11. If the execution time of v_i is higher than such a distance, this will lead to a direct cache miss of v_j if it is dispatched on p_k again. Following this intuition, the p_k with the maximum $R^2D(p_k)$ is selected for v_i (line 4), which is less likely to affect the cache speed-up of following releases of nodes in $\mathcal{V}^*(p_k)$.

$$R^2D(p_k) = \min_{v_j \in \mathcal{V}^*(p_k)} \{\Omega_j(L_2) - RD(v_j, L_2)\} \tag{11}$$

Finally, the rule MCSG is finished with a potential allocation (v_i, p_k) returned (line 5). As described above, the MCSG dispatches nodes to cores based on the CSG value instead of the speed-up of a single node, which considers and effectively mitigates the contention among the ready nodes competing for the same core. This provides the key for addressing the Limitation 2 in Sec. II, as described in Characteristic 2.

Characteristic 2. *With the CSG table and MCSG constructed, CADE explicitly considers the core contention between nodes during allocation, enhancing the cache performance for the scheduled nodes rather than certain individuals. This addresses the Limitation 2.*

Conditional Deferred Execution. With MCSG applied, a potential allocation (v_i, p_k) is identified. However, as the computation is limited to $\mathcal{P}^*$, p_k may not provide the highest speed-up for v_i, where a busy core that will be available soon

could offer a high speed-up for v_i with earlier completion. Based on this consideration, the CDE is constructed to determine whether to defer the schedule of v_i for a more suitable core or to dispatch v_i to p_k at the current scheduling point.

The CDE improves allocation decisions by determining whether deferring the execution of a node for a short period can lead to improved cache speed-up and reduced makespan. Unlike immediate dispatch to idle cores, CDE evaluates the potential benefits of deferring the execution of a node by comparing the speed-up from waiting for a busy core with the current allocation identified by MCSG. This rule avoids premature allocation to less suitable cores by considering the future availability of cores, helping to mitigate unnecessary inter-core contention and enhancing the utilisation of caches.

Alg. 4 presents the computation process of CDE, which takes the identified allocation (v_i, p_k), the set of idle cores $\mathcal{P}^*$, and the set of busy cores $\mathcal{P}^\neg$ as inputs. The algorithm starts by iterating over every core $p_x \in \mathcal{P}^\neg \setminus \overline{\mathcal{P}}(v_i)$ (line 1). The function $\overline{\mathcal{P}}(v_i)$ gives the set of cores that are considered for v_i in previous scheduling points. This ensures that the cores which are already examined for v_i are not reconsidered by the CDE, effectively preventing a scenario where v_i is repeatedly deferred without ever being executed. For each p_x, the difference in cache speed-up that v_i can achieve on p_k and p_x is calculated, and is then compared to the time duration that v_i must wait for p_x to become available (*i.e.*, $\mathcal{AT}(p_x) - t$), where $\mathcal{AT}(p_x)$ gives the next idle time of p_x and t is the current time (line 2). If $S(v_i, p_x) - S(v_i, p_k)$ outweighs $\mathcal{AT}(p_x) - t$, a more suitable core p_x is found that can provide a higher cache speed-up for v_i. In this case, the execution of v_i is deferred for p_x instead of being dispatched immediately, with $\overline{\mathcal{P}}(v_i)$ updated to include $\mathcal{P}^*$ and $CDE(\cdot)$ returns `true` (lines 3-4). Otherwise, deferring the execution of v_i is not beneficial, hence, it is dispatched to p_k as suggested by MCSG. In this case, $\overline{\mathcal{P}}(v_i) = \varnothing$ and the $CDE(\cdot)$ is terminated with `false` returned (lines 7-8).

Based on the above, the CADE extends the consideration of the candidate cores by accounting for busy ones, instead of focusing solely on the idle cores. This facilitates the allocation decision in terms of improving cache speed-up, effectively reducing the DAG makespan. Characteristic 3 summarises the key that addresses Limitation 3 as discussed in Sec. II.

Characteristic 3. *With CDE applied, the CADE effectively extends the consideration to the busy cores with effective allocation decisions, which fully leverages the cache to accelerate the DAG execution. This addresses Limitation 3.*

Time Complexity. The time complexity of MCSG and CDE is $\mathcal{O}(n^2)$. First, the MCSG identifies the maximum $\overline{S}(v_i, p_k)$ with a time complexity of $\mathcal{O}(|\mathcal{P}^*|^2)$, in which at most $|\mathcal{P}^*|$ nodes are examined for every p_k. For a candidate allocation (v_i, p_k), the complexity of the CDE is $\mathcal{O}(n)$ as at most all $m - |\mathcal{P}^*|$ cores will be examined. Hence, the complexity is $\mathcal{O}(|\mathcal{P}^*|^2 + n \times |\mathcal{P}^*|) = \mathcal{O}(n^2)$. Finally, given the complexity the CSG table ($\mathcal{O}(n^2 \times \log n)$), the time complexity of CADE is $\mathcal{O}(n^2 \times \log n)$, which is similar to that of the AJLR [6].

Discussion. It is worth noting that although the proposed CADE illustrated using a two-level cache hierarchy, it can be effectively adapted to more complex cache architectures with minor modifications. For instance, for a system with a three-layered cache, the corresponding CRP can be firstly constructed as described in [6]. Then, when computing ANPO, the rewards for prioritising nodes with cache hits at different levels can be effectively adjusted to m, $\mathcal{C}(k)$, and 1 for L_1, L_2, and L_3 cache hits, respectively. In addition, when applying MCSG, the maximum reuse distance of v_j for cache hits can be updated from $\Omega_j(L_2)$ to $\Omega_j(L_3)$. This highlights the generality of CADE in adapting to varying cache hierarchies while keeping the overall framework unchanged.

VII. Experiment

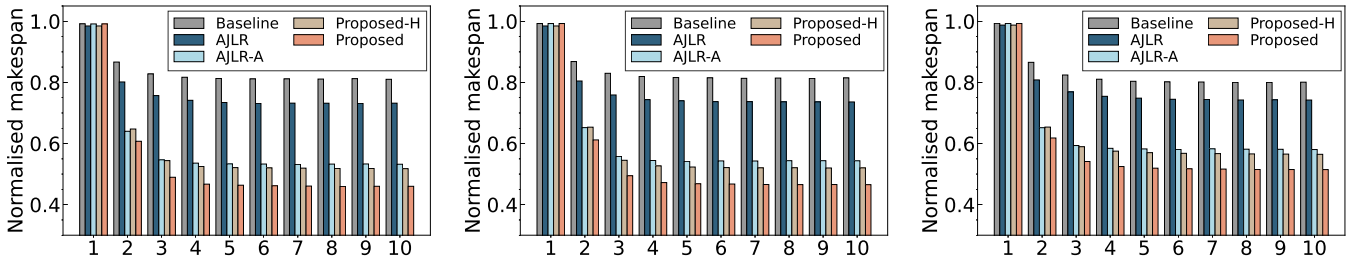
In this section, the effectiveness of the proposed CADE is evaluated against the following methods in terms of the resulting DAG makespan and cache performance.

- *Baseline*: The traditional method that schedules and allocates a DAG by the node eligibility ordering in [37] and the Worst-Fit allocation without considering the cache.
- *AJLR*: The SOTA in cache-aware DAG scheduling [6].
- *AJLR-A*: The variant of the AJLR with the proposed ANPO applied for node ordering.
- *Proposed-H*: The proposed CADE with the *highest node WCET first* constructed in [6] applied for node ordering.
- *Proposed*: The proposed CADE with ANPO applied.

A. Experimental Setup

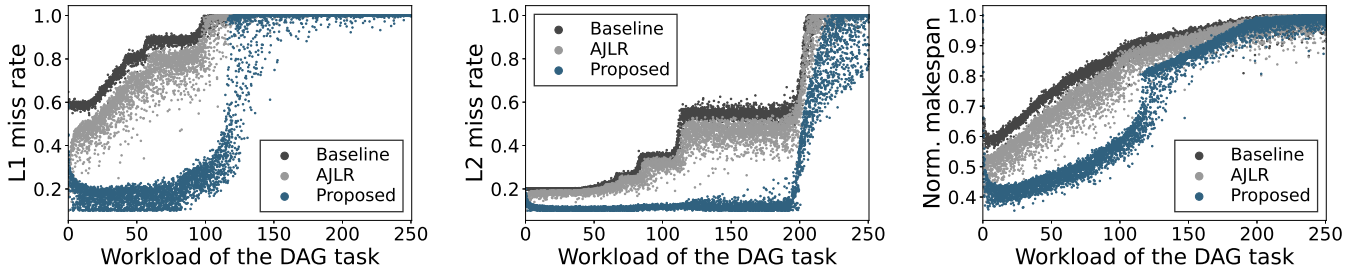
The evaluation is performed using the simulator constructed in [6] with extensive synthesised DAG tasks. The simulation is conducted on a desktop machine equipped with an Intel Core i7-13700KF processor. Unless stated otherwise, the simulated system contains $m = 8$ homogeneous cores, with each four cores organised into one cluster. The system is modelled with a two-level cache hierarchy (see Sec III-A), where the L_1 cache is dedicated to each core with a size of 64 KB, and the L_2 cache is shared among cores of a cluster with a size of 256 KB. The CRP model constructed in [6] is applied in this work, with modifications to adapt it to a 2-level cache model (*i.e.*, without the L_3 caching effects), as shown in Fig. 3. The caching effects are modelled by the CRP, which abstracts detailed cache configurations (*e.g.*, cache line size and associative ways). Hence, these specific details are omitted without jeopardising the reproducibility of the results. As with [6], [38], [39], the CRP is adopted as the global CRP for all nodes in the system, in which a longer reuse distance generally leads to a higher execution time. The objective is to evaluate the effectiveness of CADE against the state-of-the-art given a CRP, which can be effectively constructed for various hardware platforms using the measurement-based approach presented in [6].

Each simulated system consists of a single periodic DAG task. The DAGs are generated by the DAG generator in [37], which produces a DAG by generating its nodes layer by layer. The number of layers Ψ in the DAG is randomly decided in the range $[5, 15]$. In each layer, the number of nodes Φ is randomly decided in the range $[10, 40]$. Each newly generated node has a 20% probability of being connected to a random node in the previous layer to form execution dependencies.



(a) with $\Psi \in [5, 15]$, $\Phi = 25$ and $U = 25\%$ (b) with $\Psi = 10$, $\Phi \in [10, 40]$ and $U = 25\%$ (c) with $\Psi = 10$, $\Phi = 25$ and $U \in [20\%, 40\%]$

Fig. 5: The normalised makespan of the first ten instances of DAG tasks.



(a) Relationship between the L_1 miss rate and the DAG workload (b) Relationship between the L_2 miss rate and the DAG workload (c) Relationship between the normalised makespan and the DAG workload

Fig. 6: Evaluation of cache miss rates and makespan across varying workloads

After all layers are generated, all nodes without predecessors (resp. successors) are directly connected to the source (resp. sink) node to ensure one source (resp. sink) node within the DAG. The DAG period T is randomly generated in the range $[10, 144]$ units of time following a uniform distribution, with values chosen to ensure a hyperperiod of 144 units of time. The utilisation U of each DAG is randomly generated in the range $[20\%, 40\%]$, and its workload W is determined by $W = U \times T$. The WCET of each node in the DAG is then generated in a uniformly random distribution based on W. The total system utilisation is computed by $\overline{U} = U \times m$. For each configuration, 1000 DAGs are generated.

B. Comparison of the Makespan of the First Ten Jobs

This section evaluates the competing methods by comparing the normalised makespan of the first ten jobs released by 1000 DAG tasks generated under various settings of Ψ, Φ and U, as shown in Fig. 5. For each trial (*i.e.*, one DAG per trial), the makespan is normalised by the maximum value observed under all scheduling methods.

Obs 1. The proposed CADE outperforms the Baseline and AJLR by 35.09% and 29.98% on average, respectively.

The observation is derived from Fig. 5(a) to 5(c). In this experiment, the proposed CADE outperforms the Baseline by 37.27%, 36.86% and 31.15% on average, respectively. In addition, it outperforms the AJLR by 31.80%, 31.48% and 26.67% under different system settings, respectively. As observed in the figures, starting with a cold cache, the makespan for the first job across various methods shows almost no difference, as no cache speed-up could be achieved. Then, the advantages of cache-aware methods (*i.e.*, AJLR and CADE) become apparent for the later jobs compared to the Baseline for the following jobs. After the fourth job, the DAG makespan becomes stable due to the warmed cache. Among the examined jobs, the proposed CADE constantly dominates other competing methods (including the AJLR) by producing a lower DAG makespan under all settings, which justifies its effectiveness in scheduling DAG tasks.

Obs 2. The proposed node priority ordering outperforms the HWF by 15.34% on average in reducing the DAG makespan.

This observation can be obtained by comparing either AJLR-A with AJLR or Proposed-H and Proposed in Fig. 5(a) to 5(c). First, AJLR-A outperforms AJLR by 23.84%, 22.93% and 19.54%, respectively with different system settings. In contrast to the HWF, the constricted ANPO effectively mitigates such competition by considering the favourable cores of each ready node when assigning priorities, hence, outperforming the HWF by improving the overall speed-up of the ready nodes. In addition, similar observations can also obtained by comparing the Proposed and Proposed-H, where the Proposed reduces the DAG makespan by 9.39%, 8.80% and 7.53% on average, respectively. This justifies the Characteristic 1 described in Sec. V.

Obs 3. The proposed allocation mechanism outperforms the AJLR by 17.14% on average in reducing the DAG makespan.

This is observed from Fig. 5 by comparing either Proposed-H with AJLR or Proposed and AJLR-A. First, the Proposed-H outperforms the AJLR with a reduced makespan by 25.09%, 25.19% and 20.92% for varied system settings, respectively. This is expected as AJLR only considers the idle cores and produces allocations solely based on the independent speed-up values of individual nodes without considering the impact on others. In contrast, CADE dispatches nodes based on the CSG table, which accounts for both the speed-up gain and

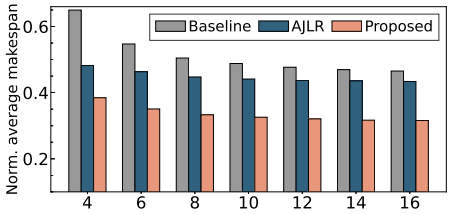


Fig. 7: under $\Phi = 18,\ m = 8,\ U = 25\%$ with varied Ψ.

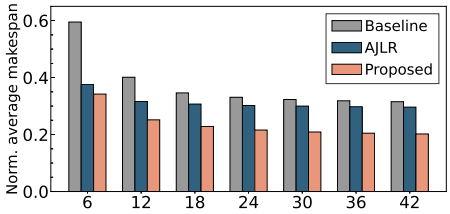


Fig. 8: under $\Psi = 8,\ m = 8,\ U = 25\%$ with varied Φ.

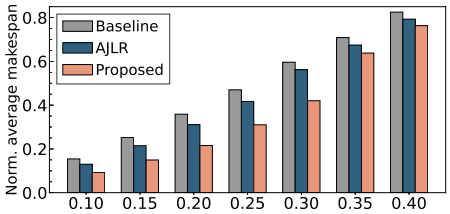


Fig. 9: under $\Psi = 8$, $\Phi = 18$, $m = 8$ with varied U.

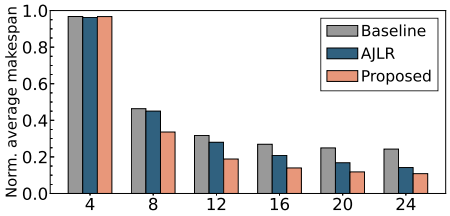


Fig. 10: under varied m when Ψ = 10, Φ = 30 and $\overline{U}$ = 30%∗8.

loss for allocating a node. In addition, with the deferred execution technique, the busy cores are also taken into account to obtain a higher speed-up, effectively exploiting the cache to accelerate the node execution. Similar observations are also obtained by comparing Proposed with AJLR-A, in which Proposed outperforms AJLR-A by 10.91%, 11.55%, 9.14%. This justifies the Characteristics 2 and 3 presented in Sec. VI.

C. Comparison of Cache Performance

This section compares the cache performance of the competing methods under varied DAG workloads, as shown in Fig. 6. This experiment demonstrates that the reduced makespan is achieved by reducing the cache miss rates, which is estimated by the CRP (Sec. III-B) based on the reuse distances.

Experimental Setup. In this experiment, 6,000 DAGs are randomly generated for evaluation. To obtain DAGs with varied workloads (*i.e.*, W), the T of each DAG is fixed at 144 units of time while $U \in (0, 40\%]$. As a similar trend is observed when $W > 250$, all 6000 DAGs are generated with $W \in (0, 250]$. Other setups remain consistent with Sec. VII-A.

Obs 4. CADE reduces the average miss rate by 24.45% on L_1 and 19.66% on L_2 compared to AJLR.

Fig. 6(a) and Fig. 6(b) present the average L_1 and L_2 miss rate of the first ten jobs of DAGs with a varied workload, respectively. As observed, CADE consistently achieves lower miss rates compared to the competing methods at both cache levels. When $W < 100$, the L_2 miss rate under CADE is around 10%. This indicates that under CADE, only the first job of most DAGs would miss the L_2 with a cold cache. With $100 < W < 150$, an obvious increase in the L_1 miss rate is observed, as large workloads generally result in prolonged reuse distances, causing most nodes to exceed the threshold of the L_1 cache reuse distance. The L_2 miss rate begins to increase after $W > 200$ due to the same reason. When $W = 250$, the miss rates at both levels under varied methods become similar, in which the high workload leads to larger nodes with very long cache reuse distances, preventing nodes from hitting any cache regardless of the scheduling methods.

Fig. 6(c) shows that CADE dominates other competing methods in terms of makespan in a general case, demonstrating its effectiveness under DAGs with varied workloads. By cross-comparing Fig. 6(a) to Fig. 6(c), we observe that the increase in makespan correlates closely with the rise in miss rates at L_1 and L_2. When $W < 100$, makespan is increased along with a gradual rise in the L_1 miss rate. A significant increase in makespan is observed when $100 < W < 150$, with a similar increase observed for the L_1 cache miss rate. As W approaches 200, the L_2 miss rate increases significantly, causing complete cache misses for most nodes, where the resulting DAG makespan is increased and becomes similar across all the competing methods. Based on the above, we observe that the resulting DAG makespan is closely associated with the cache miss rates. This justifies the impact of caching effects on the resulting DAG makespan, hence, the motivation of this work. Most importantly, it provides evidence demonstrating that the reduction in makespan under the proposed CADE is achieved by enhancing the cache performance.

D. Overall Comparison of the Average Makespan

This section explores the overall effectiveness of the proposed CADE under DAGs with varied structural parameters (*i.e.*, the number of layers Ψ, the number of nodes per layer Φ and the utilisation U) and system configurations (*i.e.*, the number of cores m), as shown in Fig. 7 to 10, respectively. For each configuration in a figure, 500 DAGs are generated and the normalised average makespan of the first ten jobs across all the DAGs is reported. The normalisation factor is set to the maximum makespan observed by all the methods among all configurations in each figure.

Obs 5. CADE outperforms AJLR by 24.02% on average in terms of DAG makespan across all configurations.

As shown in Fig. 7 to 10, CADE outperforms the competing methods in most cases in terms of DAG makespan. Compared to AJLR, the CADE reduces the makespan by 25.35%, 25.29%, 21.40% and 24.00% under varied Ψ, Φ, U and m, respectively. First, with an increased Ψ and Φ in Fig. 7 and 8, the makespan of DAGs decreases under all methods. The reason is with the same U, an increased number of nodes would cause a

reduction in node WCETs, hence, leading to a more balanced schedule with an earlier finish in a general case [37], [40]. As the number of nodes increases, the competition between nodes is intensified for the preferred cores. In this case, the advantages of CADE over the competing methods become obvious, which explicitly considers the affinity and contention between nodes, effectively mitigating the competition with enhanced cache performance. By contrast, the AJLR fails to consider such contention, leading to a high competition over certain cores with undermined speed-up effects, especially when the number of nodes is high, *e.g.*, with $\Psi = 14$ or $\Phi = 16$ in Fig. 7 and 8, respectively.

In addition, with an increased U (Fig. 5(c)), the makespan in general increases under all methods, which becomes similar with $U = 0.4$. This is also observed in Fig. 6, in which a higher U (and hence, the workload) in general prolongs the cache reuse distances of nodes, thereby leading to a higher cache miss rate under all competing methods. A similar observation is also obtained from Fig. 10 with $m = 4$, in which the high workload on each core results in ever-long cache reuse distances for nodes. However, as the number of cores increases (*i.e.*, $m \in [8, 16]$), the advantages of CADE over AJLR become pronounced (up to 33.00% in reducing DAG makespan) due to the constructed node priority ordering and the allocation mechanism. With more cores available, CADE can effectively mitigate contention between nodes and benefit from deferred executions. When m exceeds 20, the disparity between methods diminishes, as nodes can easily achieve a high cache hit rate regardless of the schedule, at which CADE still retains a slight advantage over competing methods.

VIII. Conclusion

This paper proposes CADE, a cache-aware scheduling and allocation method for DAG tasks on multicore systems. First, the ANPO is proposed, which assigns priorities to ready nodes by considering the node competition for preferred cores, effectively mitigating the core contention among the nodes. For the allocation process, the CSG table is constructed to account for the speed-up gain of an allocation decision of the node as well as the potential loss for other nodes. Based on the CSG, CADE dispatches nodes with the highest CSG value, which mitigates the impact of cache contention on the speed-up for other ready nodes, obtaining a high cache speed-up for the ready nodes instead of certain individuals. In addition, a deferred execution mechanism is adopted that enables nodes to defer their executions for cores with higher cache speed-up. Experimental results demonstrate that CADE outperforms the SOTA methods in terms of reducing DAG makespan by enhancing the cache performance. Future work includes the extension of CADE to systems with multiple DAG tasks exhibiting varying cache sensitivities, the exploration of its adaptation to heterogeneous architectures, and the optimising of CADE for further enhancement of the cache performance.

References

[1] Y. Lin, Q. Deng, M. Han, Z. Feng, S. Wang, and Q. Peng, "Lag-based schedulability analysis for preemptive global edf scheduling with dynamic cache allocation," *Journal of Systems Architecture*, vol. 147, p. 103045, 2024.

[2] L. Qian, Z. Qu, M. Cai, B. Ye, X. Wang, J. Wu, W. Duan, M. Zhao, and Q. Lin, "Fastcache: A write-optimized edge storage system via concurrent merging cache for iot applications," *Journal of Systems Architecture*, vol. 131, p. 102718, 2022.

[3] B. Agarwalla, S. Das, and N. Sahu, "Efficient cache resizing policy for dram-based llcs in chipmultiprocessors," *Journal of Systems Architecture*, vol. 113, p. 101886, 2021.

[4] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *IEEE 33rd Real-Time Systems Symposium*, 2012, pp. 63–72.

[5] J. D. Ullman, "Np-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[6] S. Zhao, X. Dai, B. Lesage, and I. Bate, "Cache-aware allocation of parallel jobs on multi-cores based on learned recency," in *Proceedings of the 31st International Conference on Real-Time Networks and Systems*, 2023, p. 177–187.

[7] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, "Global edf scheduling for parallel real-time tasks," *Real-Time Systems*, vol. 51, no. 4, pp. 395–439, 2015.

[8] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Partitioned fixed-priority scheduling of parallel tasks without preemptions," in *IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 421–433.

[9] S. Baruah, "Federated scheduling of sporadic dag task systems," in *IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 179–186.

[10] C. Maia, P. M. Yomsi, L. Nogueira, and L. M. Pinho, "Semi-partitioned scheduling of fork-join tasks using work-stealing," in *IEEE 13th International Conference on Embedded and Ubiquitous Computing*. IEEE, 2015, pp. 25–34.

[11] Q. He, N. Guan, Z. Jiang, and M. Lv, "On the degree of parallelism for parallel real-time tasks," *Journal of Systems Architecture*, vol. 156, p. 103286, 2024.

[12] S. Ben-Amor and L. Cucu-Grosjean, "Graph reductions and partitioning heuristics for multicore dag scheduling," *Journal of Systems Architecture*, vol. 124, p. 102359, 2022.

[13] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proceedings of the seventh ACM international conference on Embedded software*, 2009, pp. 245–254.

[14] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris: a dynamic cache partitioning system using page coloring," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014, p. 381–392.

[15] R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen, "Dna: Dynamic resource allocation for soft real-time multicore systems," in *IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 196–209.

[16] W. Chang, D. Goswami, S. Chakraborty, L. Ju, C. J. Xue, and S. Andalam, "Memory-aware embedded control systems design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 4, pp. 586–599, 2017.

[17] S.-W. Lo, K.-Y. Lam, W.-Y. Huang, and S.-F. Qiu, "An effective cache scheduling scheme for improving the performance in multi-threaded processors," *Journal of Systems Architecture*, vol. 59, no. 4, pp. 271–278, 2013.

[18] X. Pan and F. Mueller, "Numa-aware memory coloring for multicore real-time systems," *Journal of Systems Architecture*, vol. 118, p. 102188, 2021.

[19] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo, "Schedulability analysis of conditional parallel task graphs in multicore systems," *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 339–353, 2017.

[20] X. Jiang, J. Sun, Y. Tang, and N. Guan, "Utilization-tensity bound for real-time dag tasks under global edf scheduling," *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 39–50, 2020.

[21] Y. Gao, H. Yi, H. Chen, X. Fang, and S. Zhao, "A structure-aware dag scheduling and allocation on heterogeneous multicore systems," in *2024 IEEE 14th International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2024, pp. 26–33.

[22] S. Ben-Amor and L. Cucu-Grosjean, "Graph reductions and partitioning heuristics for multicore dag scheduling," *Journal of Systems Architecture*, vol. 124, p. 102359, 2022.

[23] X. Deng, A. H. Sifat, S.-Y. Huang, S. Wang, J.-B. Huang, C. Jung, R. Williams, and H. Zeng, "Partitioned scheduling with safety-performance trade-offs in stochastic conditional dag models," *Journal of Systems Architecture*, vol. 153, p. 103189, 2024.

[24] T. Yang, Y. Tang, X. Jiang, Q. Deng, and N. Guan, "Semi-federated scheduling of mixed-criticality system for sporadic dag tasks," in *IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2019, pp. 163–170.

[25] M. Han, T. Zhang, Y. Lin, and Q. Deng, "Federated scheduling for typed dag tasks scheduling analysis on heterogeneous multi-cores," *Journal of Systems Architecture*, vol. 112, p. 101870, 2021.

[26] M. Hatami, "Semi-partitioned scheduling hard real-time periodic dags in multicores," in *The Proceeding of First Work-in-Progress Session of CSI International Symposium on Real-Time and Embedded Systems and Technologies (WiP-RTEST 2018)*, 2018, p. 9.

[27] F. Li, R. Bi, J. Wang, J. Sun, Z. Sun, G. Tan, and M. Chen, "Vpss: A dag scheduling heuristic with improved response time bound," *Journal of Systems Architecture*, vol. 148, p. 103084, 2024.

[28] Holdings, Arm, "Arm cortex-a series programmer's guide for armv8-a-14.2. cache coherency," 2015.

[29] A. Holdings, "Cortex-a72 mpcore processor technical reference manual," 2015.

[30] K. Beyls and E. D'Hollander, "Reuse distance as a metric for cache behavior," in *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, vol. 14. Citeseer, 2001, pp. 350–360.

[31] Q. He, x. jiang, N. Guan, and Z. Guo, "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.

[32] A. Burns and A. J. Wellings, *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[33] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 2007, pp. 402–412.

[34] D. B. Kirk, "Smart (strategic memory allocation for real-time) cache design," in *Real-Time Systems Symposium*. IEEE Computer Society, 1989, pp. 229–230.

[35] J. Roeder, B. Rouxel, and C. Grelck, "Scheduling dags of multi-version multi-phase tasks on heterogeneous real-time systems," in *IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2021, pp. 54–61.

[36] P. Chen, W. Liu, X. Jiang, Q. He, and N. Guan, "Timing-anomaly free dynamic scheduling of conditional dag tasks on multi-core systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–19, 2019.

[37] S. Zhao, X. Dai, I. Bate, A. Burns, and W. Chang, "Dag scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency," in *IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 128–140.

[38] X. Dai, S. Zhao, B. Lesage, and I. Bate, "Using digital twins in the development of complex dependable real-time embedded systems," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2022, pp. 37–53.

[39] B. Lesage, X. Dai, S. Zhao, and I. Bate, "Reducing loss of service for mixed-criticality systems through cache-and stress-aware scheduling," in *Proceedings of the 31st International Conference on Real-Time Networks and Systems*, 2023, pp. 188–199.

[40] S. Zhao, X. Dai, and I. Bate, "Dag scheduling and analysis on multi-core systems by modelling parallelism and dependency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4019–4038, 2022.

Huixuan Yi is currently a student pursuing his master's degree at the School of Computer Science, Sun Yat-sen University, Guangzhou, China. His research interests mainly include the scheduling and allocation of DAG tasks.

Yuanhai Zhang received his Ph.D. degree in computer science in 2024, from Sun Yat-sen University, Guangzhou, China. He is currently a researcher at Sun Yat-sen University, China. His research interests include fault-tolerant scheduling, real-time systems and cyber-physical systems.

Zhiyang Lin is an undergraduate student at the School of Computer Science, Sun Yat-sen University, Guangzhou, China. His research interests mainly include ensuring temporal determinism in real-time systems and heuristic scheduling algorithms.

Haoran Chen is currently pursuing his master's degree at the School of Computer Science, Sun Yat-sen University, Guangzhou, China. His research interests mainly include the scheduling and allocation of DAG tasks in containers.

Yiyang Gao is currently a student pursuing his master's degree at the School of Computer Science, Sun Yat-sen University, Guangzhou, China. His research interests mainly include the scheduling of parallel tasks and the resource management.

Xiaotian Dai received the Ph.D. degree from the University of York, York, U.K., in 2019. He is a Lecturer with the Real-Time Systems Research Group, University of York. His research applies to high-performance embedded computing, robotic systems, and safety-critical autonomous systems. His research interest is mainly in real-time systems, including flexible and adaptive task scheduling, control scheduling co-design, and timing analysis of multicore systems.

Shuai Zhao received the Ph.D. degree in Computer Science from the University of York, UK., in 2018. He is an associate professor at the Sun Yat-sen University, China. His research interests include scheduling algorithms, multiprocessor resource sharing, schedulability analysis, and safety-critical programming languages. He can be reached at: zhaosh56@mail.sysu.edu.cn.