Proceedings Paper:
Zhao, Shuai, Dai, Xiaotian orcid.org/0000-0002-6669-5234, Lesage, Benjamin Michael Jean-Rene et al. (1 more author) (2023) 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. , pp. 177-187.

https://doi.org/10.1145/3575757.3593642

Cache-Aware Allocation of Parallel Jobs on Multi-cores based on Learned Recency

Shuai Zhao
Sun Yat-sen University, China
zhaosh56@mail.sysu.edu.cn

Xiaotian Dai
University of York, UK
xiaotian.dai@york.ac.uk

Benjamin Lesage
Onera, France
benjamin.lesage@onera.fr

Iain Bate
University of York, UK
iain.bate@york.ac.uk

ABSTRACT

Scheduling of tasks on multi- and many-cores benefits significantly from the efficient use of caches. Most previous approaches use the static analysis of software in the context of the processing hardware to derive fixed allocations of software to the cache. However, there are many issues with this approach in terms of pessimism, scalability, analysis complexity, maintenance cost, etc. Furthermore, with ever more complex functionalities being implemented in the system, it becomes nearly impracticable to use static analysis for deriving cache-aware scheduling methods. This paper focuses on a dynamic approach to maximise the throughput of multi-core systems by benefiting from the cache based on empirical assessments. The principal contribution is a novel cache-aware allocation for parallel jobs that are organised as directed acyclic graphs (DAGs). Instead of allocating instruction and data blocks to caches, the proposed allocation operates at a higher abstraction level that allocates jobs to cores, based on the guidance of a predictive model that approximates the execution time of jobs with caching effects taken into account. An implementation of the predictive model is constructed to demonstrate that the execution time approximations can be effectively obtained. The experimental results, including a real-world case study, prove the concept of the proposed cache-aware allocation approach and demonstrate its effectiveness over the state-of-the-art.

CCS CONCEPTS

• **Computer systems organization** → **Real-time system architecture**; **Processors and memory architectures**.

KEYWORDS

Cache-Aware Allocation, Direct Acyclic Graphs, Predictive Execution Time Model, Multi-core Systems

RTNS 2023, June 7–8, 2023, Dortmund, Germany

ACM ISBN 978-1-4503-9983-8/23/06…$15.00
https://doi.org/10.1145/3575757.3593642

ACM Reference Format:
Shuai Zhao, Xiaotian Dai, Benjamin Lesage, and Iain Bate. 2023. Cache-Aware Allocation of Parallel Jobs on Multi-cores based on Learned Recency. In *The 31st International Conference on Real-Time Networks and Systems (RTNS 2023), June 7–8, 2023, Dortmund, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3575757.3593642

1 INTRODUCTION

In multi-core real-time systems, memory and cache are the significant barriers toward timing-predictability. Traditionally, cache replacement policies have been designed to improve cache performance and thus reduce task execution time, such as *Least Recent Used* (LRU) [1, 5]. This is achieved by utilising the instruction and data locality, which ensures that the cache has the most relevant content. However, such low-level information is often not accessible at the system level, which makes it difficult to improve cache performance from the perspective of task scheduling and allocation.

During the last few decades, the metric of *cache reuse distance* has been proposed [1, 5]. The cache reuse distance describes the distance (with respect to the number of cache lines) between two consecutive accesses to the same instruction/data. Since then, previous work has demonstrated the effectiveness of this metric for dynamic cache replacement policies, *e.g.*, Vietri et al. [30]. Following the same philosophy, existing methods apply a simplified timing model that tries to schedule instances of the same task spatially closer, w.r.t. core and cluster allocation, to reduce cache misses and cache miss latency, e.g., [10]. However, this approach relies on the information of the instructions and data usage of each task, which is hard to analyse or measure given complex systems. Other approaches to improve cache performance have included cache colouring [9] and cache locking [22] techniques. However, these methods are challenging to apply in real-world systems without additional support or modifications in the underlying hardware.

Contribution: This paper looks at a different mean to achieve efficient use of the cache at the job level, which assigns jobs to cores to optimise the cache performance, and hence, the throughput of the system. To represent the *spatial* and *temporal* relationship of mapping a job to a core in terms of cache performance, we define the term *cache recency*. The spatial relation refers to the allocation (i.e., cores) of the job and its previous instance. There are direct benefits in terms of cache when the task is executed on the same core on which it was last executed, and indirect benefits where the task is executed on a different core but can gain advantages through a shared resource (*e.g.*, the L2 cache). The temporal relation

refers to the time elapsed since that job was last executed. A longer elapsed time often leads to higher cache misses due to the cache usage of other tasks.

The principal contribution of the paper is a novel cache-aware Allocation of parallel Jobs based on Learned cache Recency (AJLR). The AJLR focuses on systems where jobs are organised as directed acyclic graphs (DAG) to reflect potential execution dependency, which can be found in many real-world systems [26, 29]. In contrast to existing methods that rely on static analysis or hardware facilities, the AJLR utilises a predictive model that approximates speedup on the execution time of jobs based on their cache recency distances, namely the Cache Recency Profile (CRP). With the CRP, the AJLR operates at the scheduling level with the fundamental principle of always allocating a job to the core with the maximum cache speedup. An implementation of the CRP is constructed to illustrate how the speedup approximations required by the AJLR can be effectively obtained in real-world systems.

To the best of the authors' knowledge, this is the first cache-aware allocation method for multi-DAGs systems that is decoupled from (i) the knowledge of instruction and data usage of the software and (ii) detailed settings as well as additional facilities of the underlying hardware. Extensive experimental results, including a real-world case study, validate the concept of the proposed cache-aware allocation approach and demonstrate the AJLR can outperform the existing methods even with errors existing in the CRP models.

Paper Organisation: The related work is described in Section 2. Then, based on the system model presented in Section 3, we construct the AJLR in Section 4, with an implementation of the CRP described in Section 5. Finally, we present the evaluation in Section 6 and conclude the paper in Section 7.

2 RELATED WORK

Scheduling DAGs on multi-core systems has been discussed in [19] and [34]. These works aim to reduce DAG makespan by proposing various priority assignments for DAGs with a schedulability test to bound the worst-case makespan, i.e., the time period from the release to the finish of a DAG. In addition, there exist extensive work on the allocation of DAG tasks on multi-core systems, including the fully partitioned [8, 15]; semi-partitioned [2, 25]; federated [3, 12]; and semi-federated [20, 27, 31] scheme.

While the above work does not consider cache in their scheduling and allocation decisions, there exists a large body of works that explicitly study the impact of cache in making scheduling decisions, e.g., [6, 7, 10, 17, 18, 21, 32]. In Chang et al. [10], the scheduling of tasks with consideration of cache performance is discussed in the application domain of the automotive industry. In this work, the scheduler fits job instances as continuous sequences so that cache reuse can be maximised. Then a non-uniform sampling controller based on this coarse-grained model is applied to improve system performance. The similar idea was applied to control systems in a later work [11].

In Guan et al. [18], a cache-aware non-preemptive schedule for multi-core systems is proposed. The schedule improves cache performance by partitioning the shared cache (e.g., the L2 cache) to avoid inter-core interference, and hence, improves predictability and performance.

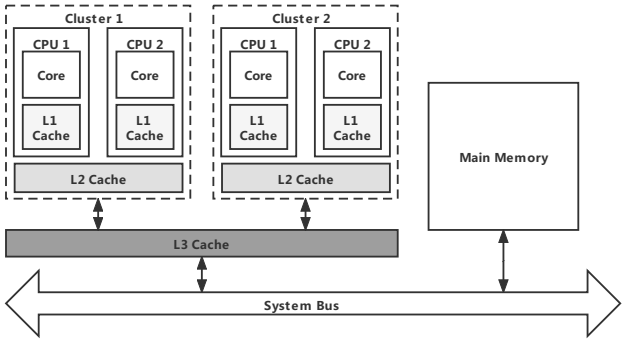


Figure 1: An example multi-core architecture with a three-layered cache hierarchy.

In Calandrino et al. [6], a scheduling policy is presented that encourages or discourages the co-scheduling of tasks based on their cache-related behaviours and releasing periods, to improve cache performance as well as scheduling results. Later in Calandrino et al. [7], a task profiling method is proposed to capture the cache-related characteristics of a task, and the efficiency of a scheduler implementation with the profiler applied is investigated.

A similar idea to CRP by building predictive models based on off-line profiling was explored by [17], where the mapping between the rate of progress against the given resources (memory bandwidth and cache capacity) are profiled and used for dynamic reassignment of resources at runtime. The authors in [16] used a profiler to investigate the relative importance of each memory page to the contribution of the overall timing of a target application.

Most existing cache-aware scheduling methods rely on static analysis of each task in the system and the support from the underlying platform (e.g., cache colouring [32] and cache partitioning [17]). In addition, existing cache-aware methods do not consider the DAG task model, i.e., all tasks are considered to be independent of one another. Such limitations impose significant barriers to these methods being applied to existing systems.

In this paper, we focus on the online allocation for DAG tasks on multi-core systems that uses a system-level execution time model to improve cache performance, and hence to reduce the execution time of DAG tasks. Instead of relying on static analysis of the system or modifications to the hardware, the proposed method aims to improve cache performance at the system level in an easy-to-practice and low-cost manner.

3 SYSTEM MODEL

Hardware Architecture. This work focuses on the hardware architecture that contains m homogeneous cores $\Lambda = \{\lambda_1, \lambda_2, ..., \lambda_m\}$ and a η-layered inclusive cache $\mathcal{L} = \{L_1, L_2, ..., L_\eta\}$, i.e., contents in the L_1 cache is also available in the L_2 and L_3 cache. Various cache hierarchies exist in commercial off-the-shelf (COTS) architectures, e.g., the three-layered cache in Intel i5-1145 Processor and the two-layered cache in ARM Cortex A72. Figure 1 presents an example multi-core system with a three-layered cache, where the level 1

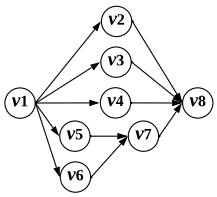


Figure 2: An example DAG task with eight nodes.

(L_1) cache is dedicated to each core, the level 2 (L_2) cache is shared among cores in a cluster, and the level 3 (L_3) cache is shared among all cores in the system. For all levels of cache, a cache replacement policy is applied to specify the cache blocks to be evicted when loading new instructions or data, e.g., the LRU [1, 5]. This work does not assume any specific cache settings, e.g., the size or associativity of the cache. That is, the proposed allocation method is decoupled from the underlying hardware and is generally applicable to commercial off-the-shelf architectures. In Section 5.2, COTS hardware is used as a case study to demonstrate the feasibility of the proposed approach.

Task Model. Tasks in the system are organised as n periodic DAG tasks based on their execution dependencies [19, 29, 33, 34], denoted as $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$. A DAG consists of a set of nodes and edges, as demonstrated by the example in Figure 2 with eight nodes. Each node represents a series of computations that must be executed in a sequential manner, e.g., a function routine [4]. An edge connecting two nodes indicates their execution dependency, e.g., node v_5 can start only if node v_1 has finished its execution.

A DAG τ_i is defined by $\tau_i = \{\mathcal{G}_i = (V_i, E_i), T_i, P_i, W_i\}$, in which (i) $\mathcal{G}_i = (V_i, E_i)$ defines the internal structure of τ_i with V_i and E_i denote the set of nodes and edges, respectively; (ii) T_i denotes the period of τ_i; (iii) P_i denotes the priority of τ_i; and (iv) W_i gives the total workload of τ_i. Each node $v_{i,j} \in V_i$ has a worst-case execution time (WCET), denoted as $C_{i,j}$. The total workload W_i is computed as the sum of WCETs of all nodes in the DAG, i.e., $W_i = \sum_{v_{i,j} \in V_i} C_{i,j}$. As with [19, 33, 34], we assume each DAG has one source and one sink node. The *makespan* of τ_i (denoted as R_i) is the worst-case response time of the DAG task under a given schedule, which is measured by the time interval from the start of the source node to the finish of the sink node. The priorities of the DAGs (i.e., P_i) are assigned according to the Rate Monotonic Priority Ordering (RMPO) algorithm [24]. At runtime, a DAG is released periodically and all nodes of the DAG will be executed once in each release. In this work, a job indicates a node instance in one release of a DAG, denoted by $v_{i,j,\nu}$ for the instance of node $v_{i,j}$ in the ν^{th} of DAG τ_i. Each job has an allocation $\alpha_{i,j,\nu}$ assigned by the AJLR. For simplicity, we omit the indexes i and ν in a $v_{i,j,\nu}$ and $\alpha_{i,j,\nu}$ (i.e., v_j and α_j) when they are not relevant in the allocation process.

Scheduling Model. With DAG tasks considered, a hierarchical scheduling scheme is applied on both the DAG level and the node level. At the DAG level, a global Fixed-Priority Scheduling (gFPS) scheme is applied, which always schedules the nodes of the ready DAG with the highest priority first. A node becomes ready to execute when all nodes it depends on have finished executions. At the node level, a non-preemptive work-conserving scheme is assumed, which always dispatches a node (if any) when a core becomes available. That is, preemption is only allowed between the execution of two nodes.

Given this system model, the proposed AJLR specifies the dispatch order and allocation of jobs based on the guidance of the predictive model. In the following sections, we present the AJLR and the implementation of the CRP in detail.

4 CACHE-AWARE ALLOCATION USING EXECUTION TIME APPROXIMATIONS

This section presents the proposed cache-aware allocation method, i.e., the AJLR. The AJLR utilises the CRP to approximate the speedup of the execution time of jobs with caching effects taken into account. Using the approximation as guidance, the method produces allocation decisions based on the principle of always allocating a job to the core with the maximum execution time speedup, to reduce the DAG makespan by benefiting from the cache. Below we first describe the preliminaries (Section 4.1) and the working mechanism (Section 4.2) of the AJLR, assuming that the speedup approximations are provided. Then, an implementation of the CRP is presented in Section 5 to demonstrate the required execution time speedup can be effectively estimated on a real-world system.

4.1 Preliminaries and Assumptions

As described above, the AJLR produces allocation decisions based on the CRP to reduce the DAG makespan by benefiting from the cache. To guide the AJLR, the CRP is required to approximate the speedup on the execution time of a job if it is allocated to a core with caching effects considered. The speedup is represented as the absolute reduction of the WCET of a job, denoted as $\mathcal{S}(v_j, \lambda_k, \mathbb{H}, CRP)$, in which v_j is the job (i.e., $v_{i,j,\nu}$ with i and ν omitted) to be dispatched, λ_k is the candidate allocation (i.e., core) of a job, $\mathbb{H}$ denotes the history table of dispatched jobs, and *CRP* gives the predictive model. We assume a higher value of $\mathcal{S}(\cdot)$ indicates a lower actual execution time for v_j if executed on λ_k, i.e., $C_j - \mathcal{S}(v_j, \lambda_k, \mathbb{H}, CRP)$.

The construction of AJLR requires the following assumptions: (i) the system maintains a history table $\mathbb{H}$ that contains the designated cores of the allocated jobs and (ii) the CRP is provided for approximating the speedup on the execution time of jobs. In Section 4.2, we first demonstrate that $\mathbb{H}$ can be effectively maintained without a high run-time cost along with the presentation of AJLR. Then in Section 5, we present the implementation of the CRP and the computation of $\mathcal{S}(\cdot)$ in detail.

4.2 Working Mechanism of AJLR

The AJLR is an online job-level allocation method that dispatches ready jobs(s) to the idle core(s) at a scheduling point, e.g., when a job arrives or finishes execution. At a scheduling point, the AJLR uses the approximated CRP to decide the dispatch order and the allocation target of the ready jobs so that their execution time can be reduced, and hence, the DAG makespan.

Dispatching Order. Following the basic gFPS scheme in Section 3, jobs in the ready queue (i.e., node instances) are first ordered by the priority of their DAGs, in which a job with a higher DAG priority will be dispatched first. For two jobs that have the same DAG priority, the one with a higher WCET will be dispatched first.

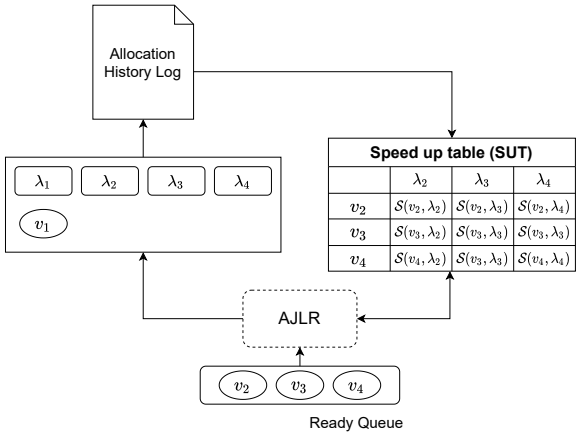


Figure 3: Overview of the AJLR-scheduled system.

Assuming δ idle cores are available at a scheduling point, at most δ ready jobs will be dispatched to execute, one on each idle core. The underlying intuition is, by dispatching the job with a higher WCET first, it has a higher number of idle cores as the candidate allocation targets, and hence, would obtain a higher speedup that can facilitate reducing the DAG makespan in the general case. In addition, we note that although the AJLR is not explicitly designed for hard real-time systems, offline scheduling orders can be supported in AJLR if DAGs have a hard deadline, in which the worst-case response time of DAGs can be obtained using the worst-case response time analysis in [19, 34].

Allocation Mechanism. Following the job order, the AJLR dispatches the first δ ready jobs to the available cores using two allocation rules: (1) the maximum speedup first (MSF); and (2) the least cache impact first (LCIF). For a job v_j, the MSF aims to reduce its execution time by allocating it to the core (say λ_k) with the highest speedup, i.e., $\mathcal{S}(v_j, \lambda_k, \mathbb{H}, CRP)$. If v_j has the same speedup on multiple cores (e.g., hits a cache that are shared among cores), the LCIF is then triggered to find an allocation with the least impact on other jobs in the system, in terms of the execution time speedup.

Maximum Speedup First. The MSF first introduces a speedup table (SUT) that contains the speedup approximations of the first δ jobs on every idle core. An illustrative example of the SUT is shown in Figure 3. In this example, three jobs will be dispatched on three idle cores at the current scheduling point, which leads to a SUT with a size of (3×3) that contains the speedup of each job on every idle core, i.e., $\mathcal{S}(v_j, \lambda_k)$ in the example SUT with $\mathbb{H}$ and CRP omitted. Based on the SUT, the AJLR always allocate the job v_j to an available λ_k with the highest $\mathcal{S}(\cdot)$. This in general provides a high reduction for the execution time of the jobs. Using Figure 3 as an example, assuming that $\mathcal{S}(v_2, \lambda_2)$ is the highest and $\mathcal{S}(v_3, \lambda_4) > \mathcal{S}(v_4, \lambda_3)$. The AJLR will dispatch v_2 to λ_2, and then allocate v_3 and v_4 to λ_4 and λ_3, respectively, based on the $\mathcal{S}(\cdot)$ approximations.

Least Cache Impact First. During allocation, a job may have the same speedup on multiple cores, e.g., when it hits a shared cache or misses all levels of cache completely. Thus, the allocation decision will not benefit the currently-examined job in terms of reducing the execution time. In this case, the objective is to reduce the potential impact on the cache benefits of other jobs caused by the current one. The underlying rationale is, when a job is allocated to a core, it could affect the speedup of the upcoming jobs where their previous jobs are also allocated to the same core. This generally holds as with the MSF, the AJLR tends to allocate jobs of the same node to the same core or cluster to increase the speedup. With this understanding, the LCIF is constructed as follows.

Algorithm 1: The online AJLR method

Input: Q_{ready}, Λ^*, $\mathbb{H}$, CRP

$Q_{sched} = sort(Q_{ready}).first(||\Lambda^*||)$;
$\mathbb{S} = init\_SUT()$;
for $v_j \in Q_{sched}$ **do**
  for $\lambda_k \in \Lambda^*$ **do**
    $\mathbb{S}(v_j, \lambda_k) = \mathcal{S}(v_j, \lambda_k, \mathbb{H}, CRP)$;
  end
end
while $Q_{sched} \neq \emptyset$ **do**
  $(v_j, \Lambda^\neg) = \underset{j,\ \lambda_k \in \Lambda^*}{\text{argmax}} \left\{ \mathbb{S}(v_j, \lambda_k) \mid \forall v_j \in Q_{sched} \right\}$;
  if $|\Lambda^\neg| == 1$ **then**
    $\lambda_k = \Lambda^\neg(1)$;
  else
    $\lambda_k = \underset{k}{\text{argmin}} \left\{ Imp(v_j, \lambda_k) \mid k \in \Lambda^\neg \right\}$;
  end
  $\alpha_j = \lambda_k$;
  $Q_{sched}.remove(v_j)$;
  $\mathbb{S}.remove(v_j, \lambda_k)$;
  $\mathbb{H}.add(v_j, \lambda_k)$;
end

For a job that has the same speedup on multiple candidate cores, the LCIF iterates each of such cores (say λ_k), computes the impact of the allocated jobs on λ_k caused by v_j, and finally chooses the core with the minimal impact as the allocation of v_j. Equation 1 computes the impact of v_j on jobs that are allocated to λ_k, denoted as $Imp(v_j, \lambda_k)$. Notations $\mathbb{H}'$ and $\mathbb{H}$ indicate the allocation history table with and without job v_j being added on core λ_k, respectively; and $\mathbb{H}(\lambda_k)$ returns the jobs that are executed on core λ_k.

$$Imp(v_j, \lambda_k) = \sum_{\forall v_x \in \mathbb{H}(\lambda_k)} \Big(\mathcal{S}(v_x, \lambda_k, \mathbb{H}, CRP) - \mathcal{S}(v_x, \lambda_k, \mathbb{H}', CRP) \Big) \quad (1)$$

We note that job v_x is taken from $\mathbb{H}(\lambda_k)$ by the Last In First Out order. In addition, the computation can finish if $\mathcal{S}(v_x, \lambda_k, \mathbb{H}, CRP) = 0$. In this case, the following jobs in $\mathbb{H}(\lambda_k)$ will not benefit from the cache regardless of whether v_j is allocated on λ_k.

Working Mechanism. The complete algorithm of the AJLR method is given in Algorithm 1. The algorithm takes jobs in the ready queue (Q_{ready}), a set of idle cores (Λ^*), the allocation history table ($\mathbb{H}$), and the CRP as the input. The algorithm allocates jobs in Q_{ready} on Λ^* using allocation rules MSF and LCIF. At a scheduling point, the AJLR first identify jobs to be dispatched (denoted as Q_{sched}) based on the dispatching order and the number of idle

cores, i.e., $||\Lambda^*||$ in line 1. Then, the SUT is constructed in lines 2-7 by computing $\mathcal{S}(\cdot)$ for each $v_j \in Q_{sched}$ on every $\lambda_k \in \Lambda^*$. Then, the MSF is applied in line 9 that returns an allocation decision (i.e., a job v_j and its candidate allocation set $\Lambda^\neg$) that has the highest $\mathcal{S}(\cdot)$. If more than one jobs have the same $\mathcal{S}(\cdot)$, the job will be selected based on the dispatching order. However, if v_j has the same $\mathcal{S}(\cdot)$ on multiple cores (i.e., $||\Lambda^\neg|| > 1$), the LCIF is applied to identify the λ_k with the least $Imp(v_j, \lambda_k)$ (lines 12-14). With both rules, the allocation of v_j is decided in line 15. Then, the $\mathbb{S}$, $\mathbb{H}$ and Q_{sched} are updated accordingly for the next iteration (lines 16-18). The algorithm returns when all jobs in Q_{sched} are assigned with an allocation.

The time complexity of the AJLR is cubic. When the AJLR is invoked with m cores being idle, at most m jobs will be dispatched so that the SUT is constructed with at most m^2 access to the CRP. As for the LCIF, the algorithm will identify the λ_k with the least $Imp(v_j, v_k)$ by at most $m * (m * n)$ computations, where n denotes the largest number of allocated jobs being examined on a core. For a given node, the $Imp(\cdot)$ will be invoked at most m (number of idle cores) times, and each time at most n nodes will be examined to compute the impact, i.e., $(m * n)$. Such computations will repeat at most m times to assign one node to each idle core. Therefore, the time complexity of the AJLR is $O(m^2 + m \times (m \times n))$. In addition, we note that n can be effectively limited as only jobs that can benefit from cache (based on the CRP) will be considered. Therefore, the run-time efficiency of AJLR can be obtained by: (i) utilising a pre-defined predictive model rather than performing complicated online computations; and (ii) limiting the number of jobs to be examined (and the associated computations) at each scheduling point.

5 IMPLEMENTION OF PREDICTIVE EXECUTION TIME MODEL

In this section, an execution time model based on the cache recency distance is proposed, namely the Cache Recency Profile (CRP). The CRP produces execution time approximations of a job based on the recency (i.e. both spatial and temporal) distance between the current and the last executed job of the same task. The CRP is a time-driven learnt model where the correlation between the recency and the execution time is derived based on data profiled from the execution platform. We note that the scheduling and allocation algorithm does not rely on the CRP being exactly accurate. Later in Section 6.5 it is shown that the approach does work better when the CRP is more accurate, however even with considerable inaccuracies the AJLR can still outperform the state-of-the-art method.

5.1 Cache Recency Profile

Assuming that the predictive execution time model is constructed using the static analysis, we need to understand every detail of a system, including the processor model, hardware model, memory model and interference model. In addition, some of the hardware architecture is often inaccurately described and sometimes details are not disclosed. In multi-core computing, a number of sources could interfere with the task of interest and consequently affect the execution time. Consequently, the complexity associated with this approach is extremely high, and the propagation error from constructing a high-level model using a compositional approach with low-level components could make it pessimistic or even unusable.

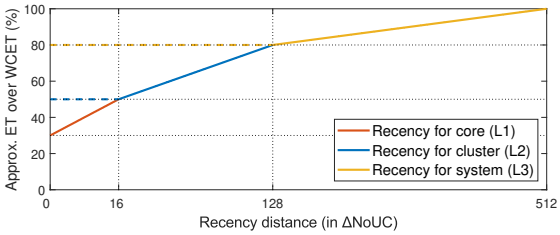


Figure 4: An illustrative cache recency profile.

Recency Distance: As an alternative approach, in this work we introduce the *recency distance* (r) of an executable entity (using node v_j as an example) which is defined as follows:

$$r(v_j, L_x) = \Delta\text{NoUC}(v_{j,v}, v_{j,v-1}, L_x) \tag{2}$$

where L_x indicates the Level-x cache, $v_{j,v}$ is the v^{th} job of node v_j, and $\Delta\text{NoUC}(\cdot)$ represents the *number of unique cache lines* accessed by jobs of other tasks between the two jobs of v_j. Note there exist dependency between L_x, L_{x+1} and L_{x+2}, for inclusive cache. This is because for an inclusive cache, if there is a L_x cache hit, then the higher-level cache will also hit, but not vice versa. That is, a hit of the L_1 cache implies a hit of the L_2 and the L_3 caches. In contrast, an L_2 cache hit does not guarantee there is also an L_1 cache hit. Due to the nature of a data-driven approach, the CRP model is able to capture these effects.

Recency Approximation with time: Given the number of new instructions is often a linearly increasing function of time [28], the recency distance can be approximated by Equation 3. This equation approximates the *recency distance* as the sum of the execution time of jobs from other tasks executed between $t(v_{j,v})$ and $t(v_{j,v-1})$, denoted by the function $\mathbf{g}(\cdot)$.

$$r(v_j, L_x) = \mathbf{g}\Big(t(v_{j,v}) - t(v_{j,v-1}), L_x\Big) \tag{3}$$

We note that there are cases where the number of cache accesses is not strictly correlated to time, e.g., when the task is formed of a *for* loop or has a complicated *if-then-goto* structure. However, as later shown in Section 6.5, this does not fundamentally break the benefits of the proposed method as long as the relationship described in Equation 3 generally holds.

To make the cache recency useful for system-level scheduling and allocation algorithms, instead of an accurate but slow method, we construct a CRP to fulfil this purpose. A CRP describes speedup over the WCET against the recency distance. Such a profile exists for each schedulable entity and is *learned* through the measurement of the actual execution time against different recency distance values. In addition, as it is based on the measurement of actual execution times, the CRP has a combinational effect on all cache levels, pre-fetching, and bus contention for the specific target of interest. This highlights the key difference between CRP and traditional compositional modelling, which is often impractical to model each of the considered effects explicitly.

The expected properties of the CRP include the following. Again, we note this is a model used for allocation and it is not essential that these properties always hold:

- *Property 1*: The execution time approximation produced by the CRP increases monotonically with recency distance;
- *Property 2*: The CRP can be modelled with a piece-wise linear function with n continuous lines.

Figure 4 illustrates a synthesised recency profile, in which the execution time (ET) of a job is approximated as a speedup (in percentage) over its WCET based on a given recency distance. In the profile, the relation between recency distance and the speedup is represented by three different trends (recency for core, cluster and system, respectively), indicating the impact on execution time caused by the gradually increased cache miss from L_1 to L_3 cache, with the increase of recency distance. For example, the first line in Figure 4 (recency for core) indicates the process where the job gradually misses the L_1 cache with an increased recency distance. For recency distances greater than 10, the job will completely miss the L_1 cache, which leads to a higher execution time.

With the CRP constructed, the speedup approximation (i.e., $\mathcal{S}(\cdot)$ described in Section 4.1) can be computed. For job $v_{j,v}$ and a candidate core λ_k, $\mathcal{S}(v_{j,v}, \lambda_k, \mathbb{H}, CRP)$ is computed by examining the recency distance of $v_{j,v}$ at each cache level $L_x \in \mathcal{L}$, as each cache level is shared among a different number of cores (and the associated jobs). Starting from the L_1 cache, we determine whether job $v_{j,v}$ can have a cache hit based on the following two conditions:

- a previous instance (denoted as $v_{j,v-1}$) is found on λ_k, i.e., $v_{j,v-1} \in \mathbb{H}(\lambda_k)$, and
- the recency distance $r(v_{j,v}, L_1)$ is less than the threshold of the recency for core, e.g. 16 in Figure 4.

If v_j can hit the L_1 cache, a speedup over the WCET of v_j can be obtained based on the $r(v_j, L_1)$ and the CRP (more specifically, the curve of the recency for core in Figure 4), denoted as $CRP(r(v_j, L_1))$. Accordingly, the absolute execution time reduction of v_j on P_k can be obtained, as shown in Equation 4.

$$\mathcal{S}(v_j, \lambda_k, \mathbb{H}, CRP) = \Big(1 - CRP\left(r(v_j, L_1)\right)\Big) \times C_j \tag{4}$$

If the above conditions are not met, v_j misses the L_1 cache so that the recency distance of v_j on the L_2 and then L_3 cache is computed and examined based on the CRP using the same approach. We note that for the cache level that is shared among cores, the allocation history of all these cores will be included when computing the recency distance of v_j to take the competition for cache from other cores into account.

5.2 Real-world Example

We now use a real-world example to (i) demonstrate the construction of a CRP for a given task, and (ii) validate the fundamental hypothesis of the CRP model, i.e. *tasks do benefit from the content remained in cache following the prior instance, resulting in execution speed-ups*. More importantly, through this case study, we show that the CRP exists for tasks in real systems. Later in Section 6.2 a proof of concept is provided to show the CRP can lead to better scheduling results using the same case study.

The targeted tasks are taken from the *TacleBench* real-time benchmarks suite [14], which covers a wide variety of computational and cache-related behaviours of tasks in the real world. The considered platform is a general-purpose quad-core Intel Core i5-6500 with a standard Linux Operating System. The processor is equipped with a 3-level cache hierarchy, where the L_1 (64KB) and the L_2 (256KB) cache are dedicated to each core, and the L_3 (6144KB) cache is shared among all four cores. The size of the cache line is 64 bytes. The CPU frequency is fixed to 800MHz.

Table 1: Profile of the *TacleBench* tasks.

Task	*NoUC*	Task	*NoUC*
ndes	194	*h264_dec*	648
adpcm_dec	151	*adpcm_enc*	148

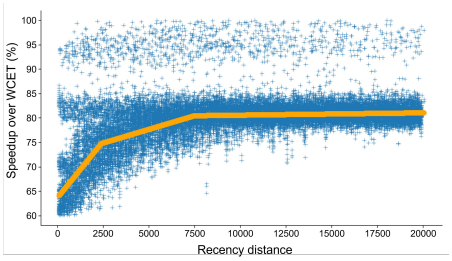


Figure 5: Observed execution time (in blue) and the modelled CRP (in yellow) of *ndes*. This model aggregates three CRPs.

In total, the CRPs of four tasks from the *TacleBench* are modelled. Table 1 presents the modelled tasks with their number of unique cache line accesses (*NoUC*) in one release. The *NoUC* is measured in the Valgrind profiling tool, by setting the cache size to a large value and accounting for the number of cache misses with a cold cache. That is, each cache miss under this cache setting means an access to a unique cache line, during one release of a task. As described in Equation 2, *NoUC* indicates the contribution of a task to the recency distance between two consecutive jobs of the modelled task τ_i. For a job $J_{i,k}$ released by τ_i, its recency distance is the sum of *NoUC* of all jobs from other tasks executed between $J_{i,k-1}$ and $J_{i,k}$. Note, the constructed allocation does not rely on the *NoUC* of each node being precisely accurate. An example of the profiled data (of *ndes* in *TacleBench*), and the learned model based on it is given in Figure 5. The profiled data is collected as a percentage of the WCET. For example, 50% means the execution time on measurement is 50% of the WCET. The recency distance (x-axis) is calculated based on Equation 2. Finally, a segmented linear function (the yellow lines) is fitted to the data with error minimised using piecewise linear regression. In section 6.5 we evaluate the effectiveness of the proposed method when considering different amounts of error in the CRP model, and show that the method remains effective when errors exist.

6 EXPERIMENTAL RESULTS

In this section, we evaluate the constructed AJLR against the state-of-the-art in scheduling DAG tasks. The experimental setup is described and justified in Section 6.1. Experimental results are presented to address the following objectives:

i. to provide proof-of-concept evidence that AJLR is effective on a real system (Section 6.2);
ii. to demonstrate AJLR reduces a single DAG makespan by improving cache performance (Section 6.3);
iii. to demonstrate the AJLR effectively reduces the makespan of concurrent DAGs in the general case (Section 6.4); and
iv. to show that the AJLR remains effective when deviations exist in a recency profile (Section 6.5).

Objective (i) validates that the AJLR is beneficial in a real-world system. Then, objectives (ii) to (vi) provide a comprehensive evaluation of AJLR under a wide range of system configurations in a simulation environment with a synthesised CRP, in which recency is approximated by execution time of nodes, i.e., Equation 3.

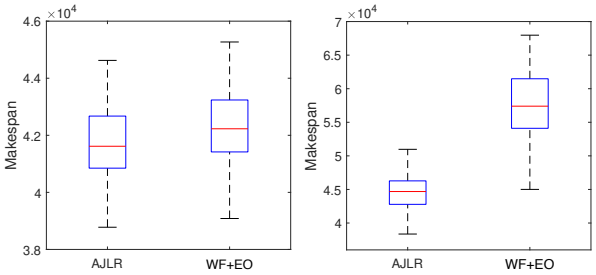
6.1 Experimental Setup

We first describe the experimental setup of the simulation environment, including the cache structure, the CRP profile used for simulation, the generation of DAG tasks, and the competing method. As the CRP abstracts the detailed cache settings (e.g., cache line size and the number of associative ways), such information is not provided without jeopardising the reproducibility of the results.

Cache and recency profile setup. Unless stated otherwise, the system has $M = 8$ symmetric cores organised into two clusters of four cores and a 3-level cache hierarchy. Each core has a dedicated L1 cache, each four-cores cluster shares an L2 cache, and all cores share the L3 cache. In addition, a global CRP shown in Figure 4 is used for evaluation, rather than different ones for each node. This reduces the number of factors changed in experiments so the results are more straightforward to interpret and analyse.

DAG and system setup. Each DAG under evaluation is randomly generated as follows. Starting from a source node, nodes are generated layer by layer, with the number of layers randomly decided in the range of $[5, 8]$. In each layer, the number of generated nodes is uniformly distributed in the range of $[2, 10]$. Each node in the newly-generated layer has a probability of 50% to be connected by randomly-chosen nodes in the previous layer. After all nodes are generated, the ones that are not connected will be linked directly to the source and/or the sink node to ensure that each DAG has one source node and one sink node. Finally, the WCETs of nodes are generated using a uniform random distribution within a range based on the total workload. We assume in general the $NoUC$ of a node is proportional to its WCET, where a larger $NoUC$ often leads to a higher WCET. This setting does not include corner cases such as a *for* loop that has a high WCET but a low $NoUC$. However, the constructed method does not require an exactly accurate CRP and is still effective when errors exist in the CRP (see Section 6.5). Nodes will execute up to their WCET if no speedup is obtained based on the recency profile.

The number of DAGs in the system is controlled by parameter $N \in [1, 4]$. The utilisation U_i of DAGs is generated by the UUnifast-Discard algorithm [13] based on a total utilisation U. DAG periods T_i are randomly generated in a uniform distribution from values that can lead to a hyperperiod of 144 units of time. The workload of a DAG is computed by $W_i = T_i \times U_i$. For multi-DAG systems (i.e., when $N > 1$), the Rate Monotonic Priority Ordering is applied to assign each DAG a priority. Nodes are ordered as well as allocated by the evaluated methods. A work-conserving non-preemptive scheme is applied for scheduling nodes.



(a) Benefits from the real system (b) Benefits from the simulation

Figure 6: The makespan (in microseconds) of AJLR and WF+EO on (a) the real system and (b) the simulator.

Competing method. For DAG tasks, the proposed AJLR is evaluated against the Worst-Fit allocation with the scheduling method in [34] (denoted as WF+EO hereinafter), which provides the state-of-the-art in scheduling and allocation of DAGs. At a scheduling point, the WF-EO takes the first node in the ready queue (ordered by the scheduling method) and allocates it to the idle core with the least utilisation. This process repeats until each idle core is assigned with a ready node (if there exists any). Compared to other heuristics (e.g. Best-Fit and First-Fit), the WF can achieve a shorter DAG makespan in more cases and AJLR outperforms all these heuristics.

Each data point plotted represents the results of 1000 trials unless stated otherwise. In each trial, all DAGs will keep executing until at least ten instances of every DAG are completed.

6.2 Proof of Concept

The first experiment aims to validate the performance of AJLR on a real-world system as the proof of concept of the proposed allocation approach. The experiment is conducted based on the case study described in Section 5.2, i.e. a 4-core Intel i5 with a 3-Level cache hierarchy. Cores 1 to 3 are used to allocate and execute tasks while core 0 is preserved for system activities. The five tasks shown in Table 1 are used for the evaluation. Each task has a unique CRP model, which is constructed based on the method in Section 5.2. Each task is assigned with a utilisation of 10%. The makespan (*i.e.* the total time spent for *executing* all tasks in the system, excluding scheduler overheads and idle periods) is reported.

Figure 6(a) presents the makespan collected from the testbed under both AJLR and WFD. As shown by the results, the AJLR demonstrates better performance than WFD in terms of achieving a lower makespan. However, the performance of AJLR on the testbed is limited as only three cores are available, i.e., a limited number of candidate allocations for each job. In the following section, a more pronounced performance of AJLR can be observed with a higher number of cores available. In addition, it is worth noting that the AJLR has been evaluated by our industrial partner on the 5G base station system with eight cores and has obtained an improvement of around 14%, i.e., the makespan of DAGs in the system is reduced by 14% on average. This suggests that the AJLR is effective in real-world systems, i.e., objective (i).

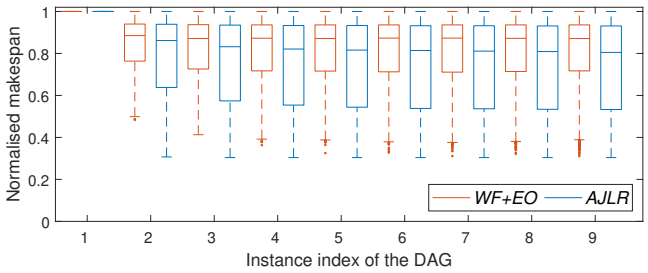


Figure 7: Normalised makespan of one DAG task.

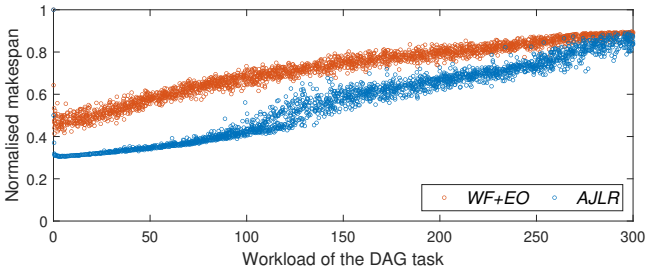


Figure 8: Relationship between the median value of the normalised makespan in 10 instances and recency distance.

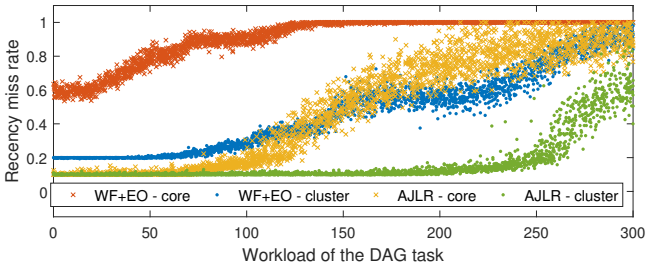


Figure 9: Relationship between recency miss and recency distance.

In addition, a simulator that adopts the same system settings of the testbed is constructed with the performance of AJLR and WFD obtained in Figure 6(b). From the Figure, we can observe that the AJLR still outperforms WFD, but the results demonstrate a large deviation compared to the testbed, in which the AJLR obtained a much lower makespan than that of the WFD. This observation reveals that in real-world systems, the potential prediction errors in the CRP model can affect the effectiveness of the AJLR (see Section 6.5 for more details). This leads to an ongoing work that applies adaptive learning techniques in AJLR to mitigate the impact of prediction errors of CRP models and to improve allocation decisions when deviations exist in the CRP [23].

6.3 Relationship between DAG Makespan and Cache Misses

With the effectiveness of AJLR validated in a real-world system, this section uses the simulator to demonstrate AJLR can achieve a shorter makespan for a single DAG due to better cache performance, compared to the state-of-the-art.

Setup. In this experiment, 5000 trials with $N = 1$ (i.e., one DAG in each trial) are generated as 1000 did not give a clear trend. For the DAG in a trial, a fixed period of 144 units of time is assigned and the utilisation is generated within a range of $(0\%, 50\%]$ for the eight-core system. The uniformly generated utilisation provides a wide range of DAG workloads, where a higher workload in general gives a higher recency distance. The DAG in each trial is then executed 10 times (i.e. 10 instances) under both methods.

Results. Figure 7 presents the normalised makespan of each instance for these DAGs. For the first instance (i.e. when the cache is cold), the normalised makespan under our method is identical to that of WF+EO, as no benefit from the cache can be obtained by different allocation decisions. For later instances, our method becomes effective and leads to a lower normalised makespan than WF+EO in general. In addition, we observe that our method becomes stable (i.e. the makespan does change less) after only a few instances. This indicates our method warms up the cache both quicker and better than WF+EO in a general case. However, we also observe that a similar 75% percentile is obtained for each instance with both methods. To understand this observation, we investigate the relationship between recency distance (represented by workload) and the resulting DAG makespan.

Figure 8 shows the median value of the normalised makespan among ten instances of DAGs with $W_i \in (0, 300]$ generated in Figure 7. For DAGs with $W_i > 300$, a similar trend is obtained for both methods. As shown in this figure, our method outperforms WF+EO in most cases by achieving a lower makespan. In particular, with $W_i \leq 100$, our method demonstrates the most pronounced advantage over WF+EO, where the normalised makespan is close to 30% of the worst case at first while maintaining a slower increasing trend compared to WF+EO. In such cases, the average workload on each core is relatively low (i.e. a short recency distance) so that most nodes can hit the L1 cache, and hence, obtains a high execution speedup. When $W_i > 100$, nodes under our method are less likely to hit the L1 cache, and hence, leading to an increased makespan. When $W_i = 300$, the performance of WFD and AJLR become similar as most of the nodes can hardly obtain any benefits from the cache.

To explain the results in Figure 8 and to provide evidence that the lower makespan of our method is achieved by improving cache performance, Figure 9 is presented to demonstrate the recency miss rate at both the core and cluster levels when executing the same set of DAGs 10 times. A miss of the recency for core means the job has a recency longer than the threshold for hitting the L1 cache (e.g., 16 in Figure 4), and hence, can only obtain a lower speedup on its execution time. As explained in Section 5 (also see Figure 4), a higher miss rate on a recency trend indicates a higher cache miss rate on the corresponding cache level. From Figure 9 we can observe that the recency miss rate of WF+EO is much higher than our method for both recency. With $W_i < 100$, our method maintains a miss rate close to 10% for both recency trends. This is the lowest miss rate possible as the first instance of each DAG is executed with a cold cache. In contrast, WF+EO demonstrates a much higher miss rate for both recency trends. For $W_i \geq 100$, the miss rate on recency for core under our method starts to increase due to longer recency

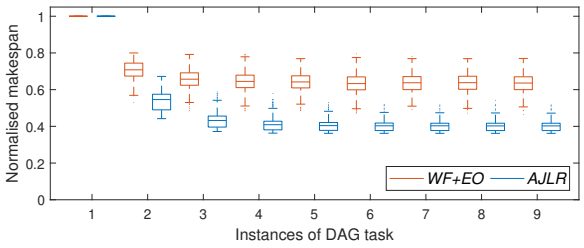


Figure 10: Normalised makespan with $N = 1$.

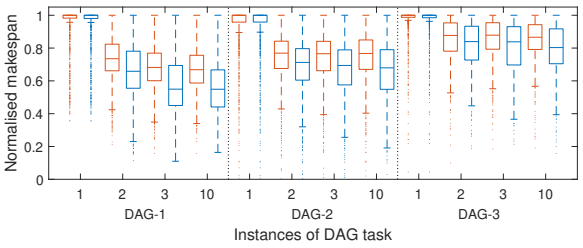


Figure 12: Normalised makespan with $N = 3$.

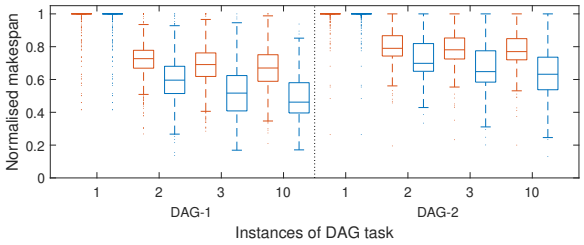


Figure 11: Normalised makespan with $N = 2$.

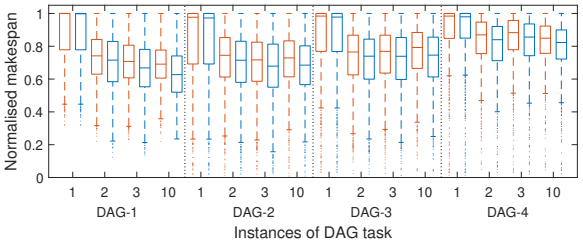


Figure 13: Normalised makespan with $N = 4$.

distance (i.e. an average distance higher than 10), as suggested by the recency profile (Figure 4). For a similar reason, our method demonstrates an increasing L2 cache miss rate when $W_i > 250$, and becomes similar to WF+EO when $W_i = 300$.

By cross-comparing Figures 8 and 9, we can observe that the recency miss has a major impact on DAG makespan. With $W_i \leq 100$ our method demonstrates its best performance with a very low miss rate on both recency trends. When $100 < W \leq 250$, our method the normalised makespan tends to increase as well as an increase in the L1 cache misses. Then, with $W_i > 250$ the normalised makespan under both methods becomes similar with our method being slightly better, with a similar trend observed from the miss rate of recency for the cluster. Based on the above results, we can observe that the AJLR achieves a lower makespan due to a general reduction in the number of cache misses.

Summary. From the experiments, we have established the understanding that the lower makespan achieved by AJLR is due to better cache performance. In addition, the experiment demonstrates the relationship between the DAG workload and makespan, as well as the situations where the AJLR has its best performance.

6.4 Makespan of Each DAG in the System

With the above understanding, this section demonstrates that the proposed AJLR can achieve a lower DAG makespan than the WF+EO, due to better cache performance. To achieve this, two sets of experiments are conducted. The first provides an in-depth evaluation of the makespan for each DAG in the system with an increasing number of DAGs. The second investigates the performance of AJLR on a common-seen setup in industrial systems.

Setup. Figure 10 to 13 provides the normalised makespan of the corresponding instances (the x-axis) of each DAG in the system, with $N = [1, 4]$ and a total system utilisation $U = 0.25 \times N$. We note that the DAG makespan after the tenth instance becomes similar and does not have an observable trend. To avoid too many boxes reducing readability, with $N \geq 2$ the makespan of the first three and the tenth instances (i.e., instance 1, 2, 3, 10) of each DAG are presented. This provides the trend in makespan from a cold cache to a stable makespan when the cache is warmed up.

Results. Figure 10 presents the normalised makespan of the first ten instances when the system contains a DAG ($N = 1$). In this figure, our method shows dominating results by achieving a constantly lower makespan than WF+EO, except for the first instance with a cold cache. As described by Figure 8 and 9, DAGs with $U_i = 0.25$ and $T_i = [10, 144]$ can have a relatively short recency distance which leads to fewer cache misses and/or cache misses with lower latency in general, and hence, a lower makespan.

For $N = 2$ (Figure 11 with two DAGs: DAG-1 and DAG-2), our method still maintains an obvious advantage over the WF+EO for each DAG. However, we can observe that the differential in normalised makespan between the two methods is decreased compared to that in Figure 10. The reason is that, by introducing another DAG it will inevitably increase the recency distance between the executions of nodes, and hence, leads to a higher cache miss rate. In addition, the competition for cores is also increased simultaneously with more tasks in the system, where a node cannot execute on the core with the shortest recency distance if the core is busy executing another node.

The trend of the differential described above becomes more obvious when further increasing N (see Figure 12 and 13), where the normalised makespan of the methods are becoming similar with a higher N. As shown by Figure 12, our method can outperform the WF+EO for the first two DAGs yet only demonstrates a slightly lower normalised makespan for the third DAG. The reason for this observation is similar to that of Figure 11, where a DAG with a

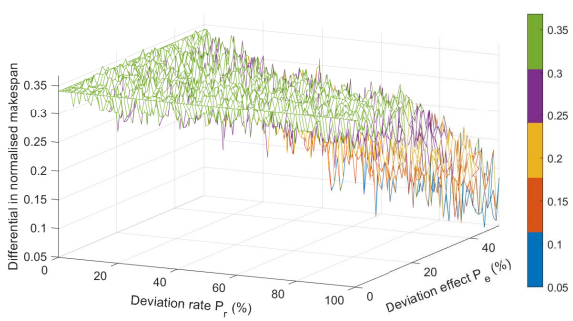


Figure 14: Differential in normalised makespan of WF+EO and our method for a DAG with $U = 20\%$.

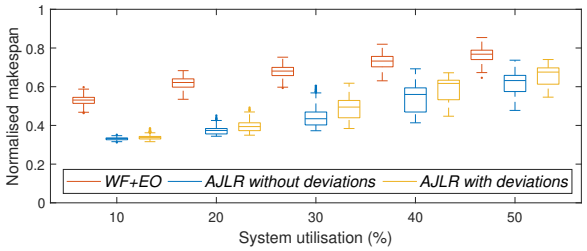


Figure 15: Normalised makespan with $P_r = 50\%$ and $P_e = 25\%$.

longer period in a busy system is more difficult to obtain benefits from the cache. In particular, for the third DAG which has the longest period, the recency distance of its nodes will be prolonged significantly by the first two DAGs with frequent releases. Thus, both methods lead to a high cache miss rate and achieve similar DAG makespan with our method being slightly lower.

In Figure 13, the normalised makespan of all DAGs becomes similar under both methods because of the ever-long recency distance and high cache miss rates. However, even for fully-loaded systems, our method still performs better than the WF+EO and can achieve a lower normalised makespan for all DAGs in general with the help of the recency profile. In addition, we can observe the proposed method is generally more effective for DAGs with shorter periods (e.g., the $10th$ instance of DAG-1 and DAG-4 where DAG-1 always has a shorter period than DAG-4). The more frequent releases mean the potential gains from reducing cache misses increase.

Summary. This section evaluates the performance of AJLR against WF+EO for systems (1) with a different number of DAGs and (2) with an increasing utilisation of the first DAG, where the latter highlights the performance of AJLR on a common-seen setup in industrial applications. Based on the results, we conclude that the AJLR in general outperforms WF+EO with a different number of DAG in the system and performs better in the system where the foreground DAG has a varied utilisation, in terms of reducing makespan for each DAG in the system.

6.5 Recency Profile with Deviations

The above experiments are conducted assuming an accurate recency profile, i.e., the execution time provided by the profile exactly matches the real case in all scenarios. In this section, we evaluate the performance of the AJLR against the WF+EO when deviations exist in the values returned by the recency profile.

Setup. We define the deviation in a recency profile as a pair of parameters (P_r, P_e) with $P_r \in [0\%, 100\%]$ and $P_e \in [0\%, 50\%]$. P_r controls the probability of deviation occurrence for each access to the recency profile. When a deviation occurs, the execution time suggested by a specific entry in the CRP will be replaced from S to $\mathrm{S} \pm \mathrm{S} \times P_e$, where S is the execution time obtained from the CRP without deviations (see Section 4.2).

Figure 14 presents the differential of WF+EO and our method for one DAG with $U = 20\%$, under varied P_r and P_e values. The differential is computed by $(\alpha_{WF+EO} - \alpha_{AJLR})/\alpha_{WF+EO}$, where α_x is the median of normalised makespan among the first ten instances of the DAG under method x. A higher positive differential indicates better performance of the AJLR. In addition, the results of WF+EO are not affected by the deviation.

Results. As shown in this figure, our method can outperform WF+EO and provide a relatively stable makespan of the DAG for a wide range of P_r and P_e values. For instance, in most cases when $P_r \leq 40\%$ and $P_e \leq 25\%$ our method outperforms WF+EO with a makespan differential around 35%, which is the differential of the two methods without deviations, i.e. $P_r = 0\%$ or $P_e = 0\%$. This observation indicates that our method remains effective when deviations exist in the recency profile. By further increasing P_r and P_e, our method starts to demonstrate large variations on makespan with a clear decreasing trend in the makespan differential, e.g. the differential is around 0.2 with $P_r = 60\%$ and $P_e = 40\%$. However, such cases only occur with large $P_r\%$ and $P_e\%$, which can fundamentally change the CRP to an opposite trend.

The above analysis indicates P_e can impose a more significant impact on the performance of AJLR than P_r by affecting the recency trend. This is justified by comparing the differential in makespan under 1) $P_r = 100, P_e \in [0\%, 50\%]$ and 2) $P_r \in [0\%, 100\%], P_e = 50\%$, where case 2) leads to much larger variations on the differential. The reason is our method does not rely on an exactly accurate value from the CRP. Instead, the correct recency trend is more important for our method to make effective allocation decisions.

Figure 15 presents the median value of the normalised makespan among ten instances of one DAG, with an increasing system utilisation and fixed values of P_r and P_e. Similar to Figure 14, our method demonstrates strong performance when deviations exist in the recency profile and outperforms WF+EO in all cases. However, the impact of deviations on makespan becomes more observable when the system has a higher system utilisation. This is expected as for DAGs with a larger workload, deviations can impose a higher impact on the execution time returned by the recency profile, which directly affects the online allocation decisions within MSF.

Summary. In this section, we evaluate the performance of AJLR under a recency profile with deviations. Within limits, we demonstrate that our method is effective with a relatively stable performance. Future work will look to make the approach more robust to deviations and actively learn the CRP from run-time measurements to reduce errors. A further analysis verifies this by showing that P_e imposes a higher impact on the performance of our method when

affecting the trend of the profile. In addition, we demonstrate that DAGs with a larger workload are more likely to be affected by the deviations due to a higher impact on the speedup computation (see Section 4.2 for details), and hence the allocation decisions.

7 CONCLUSIONS

In this paper, an approach is presented for scheduling and allocating nodes of a DAG to cores such that the cache miss rate and the cost of those misses are reduced. This is based on a realistic model with few assumptions, in which the change of execution time is based on how recently the node was executed and on where it is executed. This allows decisions to be taken to allocate nodes to cores to reduce the execution time of nodes, and hence, the DAG makespan. The evaluation shows that both on a real physical platform and in the simulation, the makespan of DAGs is in general shorter compared to a state-of-the-art algorithm. The evaluation then shows the potential negative effects of deviations between the model and the actual system. The proposed approach still outperforms the start-of-the-art algorithm with reasonably large deviations, however, understandably there are limits.

Future work will present an extension of the work that supports the off-line ordering and schedulability analysis of timing-critical DAGs, an evaluation of whether different recency profiles may further reduce the makespan of DAGs, and evidence that AJLR improves the predictability of the system. As mentioned, we are also looking at applying adaptive learning methods, e.g., with a Digital Twin, to improve the accuracy of CRP models by reducing prediction errors where exists a large deviation, from the real observations of the recency while the system is in operation.

ACKNOWLEDGMENTS

This work is supported by the Key-Area Research and Development of Guangdong Province (Grant No. 2020B0101650001).

REFERENCES

[1] Javanshir Farzin Alamdari and Kamran Zamanifar. 2012. A reuse distance based precopy approach to improve live migration of virtual machines. In *International Conference on Parallel, Distributed and Grid Computing*. IEEE, 551–556.

[2] Benjamin Bado, Laurent George, Pierre Courbin, and Joël Goossens. 2012. A semi-partitioned approach for parallel real-time scheduling. In *International Conference on Real-Time and Network Systems*. 151–160.

[3] Sanjoy Baruah. 2015. Federated scheduling of sporadic DAG task systems. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 179–186.

[4] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. 2012. A generalized parallel task model for recurrent real-time processes. In *Real-Time Systems Symposium*. 63–72.

[5] Kristof Beyls and Erik D'Hollander. 2001. Reuse distance as a metric for cache behavior. In *Conference on Parallel and Distributed Computing and Systems*, Vol. 14. 350–360.

[6] John M Calandrino and James H Anderson. 2008. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Euromicro Conference on Real-Time Systems*. IEEE, 299–308.

[7] John M Calandrino and James H Anderson. 2009. On the design and implementation of a cache-aware multicore real-time scheduler. In *21st Euromicro conference on real-time systems*. 194–204.

[8] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. 2018. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *IEEE Real-Time Systems Symposium*. IEEE, 421–433.

[9] Jichuan Chang and Gurindar S Sohi. 2007. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. 402–412.

[10] Wanli Chang, Dip Goswami, Samarjit Chakraborty, Lei Ju, Chun Jason Xue, and Sidharta Andalam. 2016. Memory-aware embedded control systems design. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 4 (2016), 586–599.

[11] Wanli Chang, Debayan Roy, Xiaobo Sharon Hu, and Samarjit Chakraborty. 2018. Cache-aware task scheduling for maximizing control performance. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 694–699.

[12] Jian-Jia Chen. 2016. Federated scheduling admits no constant speedup factors for constrained-deadline DAG task systems. *Real-Time Systems* 52, 6 (2016), 833–838.

[13] Robert I Davis and Alan Burns. 2011. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems* 47, 1 (2011), 1–40.

[14] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis*. 2:1–2:10.

[15] José Fonseca, Geoffrey Nelissen, Vincent Nelis, and Luís Miguel Pinho. 2016. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *IEEE Symposium on Industrial Embedded Systems*. IEEE, 1–10.

[16] Golsana Ghaemi, Dharmesh Tarapore, and Renato Mancuso. 2021. Governing with insights: towards profile-driven cache management of Black-Box applications. In *33rd Euromicro Conference on Real-Time Systems*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[17] Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. 2021. DNA: Dynamic Resource Allocation for Soft Real-Time Multicore Systems. In *Real-Time and Embedded Technology and Applications Symposium*. IEEE, 196–209.

[18] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-aware scheduling and analysis for multicores. In *ACM international conference on Embedded software*. 245–254.

[19] Qingqiang He, Xu Jiang, Nan Guan, and Zhishan Guo. 2019. Intra-Task Priority Assignment in Real-Time Scheduling of DAG Tasks on Multi-Cores. *Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2283–2295.

[20] Xu Jiang, Nan Guan, Xiang Long, and Wang Yi. 2017. Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *IEEE Real-Time Systems Symposium*. IEEE, 80–91.

[21] Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. 2013. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *25th Euromicro Conference on Real-Time Systems*. IEEE, 80–89.

[22] David B Kirk. 1989. SMART (strategic memory allocation for real-time) cache design. In *Real-Time Systems Symposium*. IEEE, 229–230.

[23] Benjamin Lesage, Xiaotian Dai, Shuai Zhao, and Iain Bate. 2023. Reducing Loss of Service for Mixed-Criticality Systems through Cache- and Stress-Aware Scheduling. In *Proceedings of the 31th International Conference on Real-Time Networks and Systems*.

[24] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20, 1 (1973), 46–61.

[25] Cláudio Maia, Patrick Meumeu Yomsi, Luís Nogueira, and Luis Miguel Pinho. 2015. Semi-partitioned scheduling of fork-join tasks using work-stealing. In *International Conference on Embedded and Ubiquitous Computing*. IEEE, 25–34.

[26] Salah Eddine Saidi, Nicolas Pernet, and Yves Sorel. 2017. Automatic parallelization of multi-rate fmi-based co-simulation on multi-core. In *Symposium on Theory of Modeling and Simulation* .

[27] Masoud Shariati, Mahmoud Naghibzadeh, and Hamid Noori. 2018. Semi-Federated Scheduling of Multiple Periodic Real-Time DAGs of Non-Preemptable Tasks. In *International Conference on Computer and Knowledge Engineering*. IEEE, 84–91.

[28] Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. 2007. Locality approximation using time. *ACM SIGPLAN Notices* 42, 1 (2007), 55–61.

[29] Micaela Verucchi, Mirco Theile, Marco Caccamo, and Marko Bertogna. 2020. Latency-Aware Generation of Single-Rate DAGs from Multi-Rate Task Sets. In *Real-Time and Embedded Technology and Applications Symposium*. 226–238.

[30] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving cache replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems*.

[31] Tao Yang, Yue Tang, Xu Jiang, Qingxu Deng, and Nan Guan. 2019. Semi-Federated Scheduling of Mixed-Criticality System for Sporadic DAG Tasks. In *International Symposium on Real-Time Distributed Computing*. IEEE, 163–170.

[32] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. Coloris: a dynamic cache partitioning system using page coloring. In *23rd International Conference on Parallel Architecture and Compilation Techniques*. IEEE, 381–392.

[33] Shuai Zhao, Xiaotian Dai, and Iain Bate. 2022. DAG Scheduling and Analysis on Multi-core Systems by Modelling Parallelism and Dependency. *IEEE Transactions on Parallel and Distributed Systems* (2022).

[34] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. 2020. DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *IEEE Real-Time Systems Symposium*. IEEE, 128–140.

Received 20 January 2023; revised 28 March 2023; accepted 16 April 2023