Tensity-Aware Optimized Scheduling of Parallel Real-Time Tasks on Multiprocessors

Anway Mukherjee[†], Tanmaya Mishra[†], Thidapat Chantem[†], Nathan Fisher[‡]
[†]*Department of Electrical and Computer Engineering, Virginia Tech, USA.*
Email: {anwaym, tanmayam, tchantem} @vt.edu
[‡]*Department of Computer Science, Wayne State University, USA.*
Email: fishern@wayne.edu

***Abstract*—The federated scheduling framework is a popular multicore scheduling policy for parallel periodic real-time tasks that are often modeled as directed acyclic graphs (DAGs). However, it often over-estimates the processing requirements of parallel task execution, resulting in acute resource under-utilization of available processing capacity in an already resource-constrained system. In this work, we aim to reduce resource under-utilization by proposing HL-DAGs, where compatible DAG tasks are fused and transformed into a fork-join DAG task model to opportunistically reclaim the usable utilization of the system. HL-DAGs, however, may fail to meet a task's timing requirements and impact the schedulability of the system. To tackle this challenge, we present a technique to enforce both the logical and timing correctness requirements of a HL-DAG task. In addition, we discuss a fixed-priority partitioned scheduling algorithm (HL-FED) to schedule HL-DAGs, along with other DAG tasks, on multicore systems. Simulation results indicate that HL-FED can improve the usable system utilization by 27% on average, and up to 33%, over existing DAG scheduling frameworks. In addition, our proposed solution can also tighten the processing capacity by up to 11% when compared to the state-of-the-art federated scheduling framework.**

I. Introduction

Multicore architectures can often be used to improve the performance by utilizing all the available processor cores. In parallel real-time applications, there is an additional need to meet the strict timing constraints, while simultaneously improving the usable system utilization. Directed acyclic graphs (DAGs) [1] elegantly capture the parallel execution patterns in such applications while providing highly deterministic timing guarantees. The DAG task model can be used to express both *inter-task* and *intra-task* parallelism. Several existing DAG-based scheduling policies have been shown to provide improved schedulability, alongside higher performance (for instance, reduced resource utilization), over the utilization-based task scheduling algorithms [2], [3].

The federated scheduling algorithm [4] is a popular DAG task scheduling framework that targets the *parallelizable* sections of a task by scheduling them on multiple processors for concurrent execution. In federated scheduling, a DAG task is categorized as either a *heavy* task which has a utilization greater than 1, or a *light* task with utilization less than or equal to 1. Each heavy task is assigned a dedicated number of cores to fully utilize their scope for parallel processing without competing for shared processors. Conversely, the light tasks are grouped together as tasks with strictly sequential execution and scheduled on the rest of the available core(s). While the federated scheduling framework exploits the high intra-task, and inter-task, parallelism to improve the overall system performance, it often over-estimates the processing requirements of heavy tasks, and the scheduler ends up dedicating up to almost double the cores to a heavy task than their actual utilization demand [5]. This can result in an acute under-utilization of usable resources in an already resource-constrained embedded system.

Recent works by Jiang et.al. [5], [6] addresses the resource over-utilization problems associated with federated scheduling by computing more tightly the processor demands of heavy tasks, albeit with certain caveats. Given a heavy DAG task with utilization $U + \epsilon$, a federated scheduler assigns at least $U + 1$ dedicated processors to feasibly schedule the task. In contrast, the work in [5] presents a *semi-federated* scheduling policy which partitions the heavy task τ_h into two sub-tasks, τ_a and τ_b, with utilization values U and ϵ, respectively, before assigning U cores to τ_a, and one or more core(s) (previously assigned to τ_h) to schedule τ_b along with rest of the tasks in the task set. However, partitioning DAG tasks using the heuristic technique in [5] may result in sub-optimal solutions. Conversely, optimal solvers are expensive and time consuming. Moreover, the advantages of a semi-federated scheduling policy decreases with increasing system core count.

Another recent work [6] extends federated scheduling to a *mixed* scheduling framework where tasks are scheduled based on their *tensity* values. A tensity value [6] of a DAG is the ratio of its *critical path* length to its assigned deadline. A new DAG scheduling algorithm was proposed that heuristically categorizes tasks into groups and scheduled using different scheduling strategies such that (1) heavy and light DAG tasks with *low* tensity values are scheduled using the global earliest deadline first (G-EDF) [7] algorithm, while (2) the rest of the DAG tasks in the task set (i.e., tasks with *high* tensity values) are scheduled using the federated scheduling approach. While this approach greatly improves the DAG task schedulability bounds, its major drawback is that it still suffers from the over-estimation problem for DAG tasks with relatively high tensity values as will be shown in Section III. Moreover, no strict *tensity* bound within which this algorithm [6] can improve the

This work was supported in part by the National Science Foundation under grant numbers CNS-1618979 and CNS-1618185.

978-1-5386-5541-2/18/$31.00 ©2020 IEEE

schedulability of DAG tasks was given.

A sustainable DAG task scheduling algorithm, therefore, must find a judicious trade-off between (1) improved schedulability, while (2) assigning cores to fully utilize their available processing capacity. We aim to improve upon the existing work by presenting a novel task transformation technique, and a multicore scheduling policy (HL-FED) to feasibly schedule DAG tasks while improving resource utilization. Our approach is based on the key observation that the tensity of a DAG task is a major indicator of its existing intra-task parallelism. Tensity is also a major bottleneck in fully utilizing the available parallelism on a federated scheduling framework as is evident in [6]. Contrary to the task partitioning policy in [5], our solution proposes a novel technique that fuses a heavy DAG task with a compatible light task, irrespective of their tensity values (as opposed to [6]), and transforms the fused task into a *fork-join* [8] DAG task (HL-DAGs). We also present a technique to enforce the correct timing and logical requirements of the individual tasks that constitute a fused task. Moreover, we leverage the efficient fork-join scheduling algorithm in [8] to devise a multicore task assignment and scheduling algorithm ((HL-FED) to feasibly schedule the transformed HL-DAG tasks.

The advantages of our novel DAG transformation technique are manifold; first, the fork-join DAG task model requires simple scheduling policies, with negligible time overhead due to reduced concurrency. Second, a fork-join DAG task can limit the existing intra-task parallelism in the original DAG tasks as long as the assigned tensity and deadline are met. This allows us to accurately compute the processor demands of a DAG without over-estimating the task utilization demand. Third, the fork-join DAG task model finds relevance as a popular industrial standard for parallel task execution in high-performance computing (HPC), and is recently being used for embedded real-time applications that support parallel programming (e.g., OpenMP, JVM-based Scala etc. [9]–[11]). Our main contributions are as follows.

1) We present an approach to fully utilize the processing capacity of processing cores in DAG task scheduling by forming HL-DAGs, which involves fusing together a heavy DAG task with a compatible light DAG task, and transforming the resultant fused DAG into a fork-join DAG task. We also describe a task model to represent the HL-DAG task, and discuss a technique to enforce logical and timing correctness.
2) We propose an offline fixed-priority task partitioning and scheduling technique called HL-FED for multicore systems. We also validate our approach and assess its performance in a custom-built simulated environment. We show that our approach never performs worse, and in fact often outperforms the widely used most relevant scheduling approaches [2], [5], [6] for parallel applications modeled as DAGs with high tensity values in a multicore system.

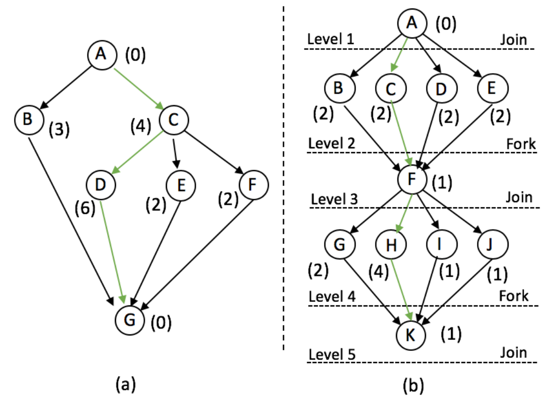

Fig. 1. (a) A simple example of a normal DAG task; (b) Example of a special case of a DAG task, also called a fork-join DAG task. The number in parentheses associated with each vertex denotes the worst-case execution time of each sub-task. The complete paths highlighted in green denote the critical path of individual DAGs.

II. Preliminaries

This section presents the relevant background details on two task models that can efficiently capture the inter- and intra-task parallelism that exists in real-time applications, along with their precedence constraints.

A. Directed Acyclic Graphs (DAG)

A synchronous real-time DAG task is defined as $\tau_g = (\boldsymbol{V}, \boldsymbol{E}, T_g)$. The variables $\boldsymbol{V}$ and $\boldsymbol{E}$ are the set of vertices and edges of the DAG τ_g, respectively. T_g is the strict time period of execution for the overall task τ_g. Since, in this work, we consider real-time tasks with implicit deadlines, we will assume that task deadline equals task period T_g. We also assume data independent DAGs to solely concentrate on intra-task parallelism. However, our proposed model can also be easily extended to DAGs with inter-task parallelism.

Each *vertex* $v \in \boldsymbol{V}$ represents a *sub-task* which delimits the sequential execution. Henceforth, we will use the terms sub-task and vertex interchangeably unless otherwise specified. Each vertex $v \in \boldsymbol{V}$, is associated with a weight $C(v)$ which represents the worst-case execution time (WCET) of the sub-task. We assume that the DAG task τ_g has a unique start (or the source vertex) vertex $v_s \in \boldsymbol{V}$ which has no predecessor, and a unique end (or a sink vertex) vertex $v_e \in \boldsymbol{V}$ which has no successor. Any DAG task with multiple start (or end) vertices can be transformed to adhere to the said assumption by adding in an *pseudo* start (or end) vertex that becomes the immediate predecessor (or successor) of the existing start (or end) vertices. Each edge $(u, v) \in \boldsymbol{E}$ captures the precedence relationship between the execution of two vertices u and v. Therefore, u is the immediate *predecessor* of v, and similarly, v is the immediate *successor* of u. We assume that each edge $(u, v) \in \boldsymbol{E}$ of the DAG task τ_g does not have an associated weight. Next, we revisit a set of properties associated with the DAG task model.

We use $\boldsymbol{\pi}$ to denote a set of sequential paths of execution in the DAG τ_g. A sub-set of paths $\boldsymbol{\pi_c}$ represents all *complete* paths wherein each path's first vertex is the source vertex

v_s and last vertex is the sink vertex v_e. The length of a path denoted by $span(\tau_g)$ represents the length of the longest complete path in the task τ_g such that,

$$span(\tau_g) = \max_{\boldsymbol{\pi_c} \in \boldsymbol{\pi}} \left\{ \sum_{v \in \boldsymbol{\pi_c}} C(v) \right\} \tag{1}$$

The complete path that contributes to $span(\tau_g)$ is also called the *critical path*. We assume that each DAG task has one and *only* one critical path. We use $work(\tau_g)$ to represent the aggregate WCETs of all the vertices of a DAG task τ_g such that,

$$work(\tau_g) = \sum_{v \in \boldsymbol{V}} C(v) \tag{2}$$

For a DAG task τ_g, its overall utilization $U(\tau_g)$, and assigned tensity $\omega(\tau_g)$ are given by,

$$U(\tau_g) = \frac{work(\tau_g)}{T_g} \tag{3}$$

$$\omega(\tau_g) = \frac{span(\tau_g)}{T_g} \tag{4}$$

The DAG task τ_g can be categorized as either a heavy task when $U(\tau_g) > 1$, or a light task when $U(\tau_g) \leq 1$. In this work, we consider both heavy and light DAGs to be independent synchronous tasks with implicit deadlines. The minimum number of cores $p \in \boldsymbol{P}$ required to feasibly schedule a DAG task τ_g using *any* work-conserving algorithm is given by [4],

$$p = \left\lceil \frac{work(\tau_g) - span(\tau_g)}{T_g - span(\tau_g)} \right\rceil \tag{5}$$

Note that the fraction inside the ceiling function in Equation 5 increases proportionally with increasing $span(\tau_g)$ values for a fixed $work(\tau_g)$ and T_g in a DAG [1]. Figure 1(a) shows a normal DAG task structure where A is the *source* vertex (v_s) and G is the *sink* vertex (v_e). The complete path $\pi_c = \{A, C, D, G\}$ is the critical path of the DAG. Therefore, the values $span(\tau_g)$ and $work(\tau_g)$ for the DAG are 10 and 17 respectively.

B. Fork-Join DAG Task Model

The fork-join DAG task model is a *unique* subset of the basic DAG task model. A fork-join DAG task combines together two classical tree structures, namely, the *in-tree* structure and the *out-tree* structure, thereby conforming to a hybrid DAG task structure. The fork-join DAG finds use in many parallel programming applications, and can successfully capture the timing parameters of complex inter- and intra-task execution patterns [8]. Fork-join DAGs have been frequently used to model parallel execution in the OpenMP framework, and provides excellent schedulability and timeliness guarantees [9], [11].

Figure 1(b) shows a fork-join DAG task, and is defined as $\tau_g = (\boldsymbol{V}, \boldsymbol{E}, T_g)$. The terms $\boldsymbol{V}$, $\boldsymbol{E}$, and T_g have the same meanings as described in Section II-A. In addition, a fork-join DAG task τ_g has a *fork* phase of execution that spawns a number of parallel sub-tasks from a single predecessor sub-task, and a *join* phase of execution where the output edges of said parallel sub-tasks spawned in the fork phase converge to a single successor sub-task. In Figure 1(b), A is the source vertex and K is the sink vertex. The complete path $\pi_c = \{A, C, F, H, K\}$ is a critical path of the DAG τ_g. Therefore, the values $span(\tau_g)$ and $work(\tau_g)$ for the DAG are *8* and *18* respectively.

The sub-tasks of a fork-join DAG task can also be categorized into levels $l \in \boldsymbol{L}$ where each level l strictly maintains the following rules [12], [13]; (1) exactly one sub-task of the critical path can exist in each level l of the DAG, (2) all other sub-tasks in level l that are not in the critical path are data independent from one another, and (3) every sub-task of the DAG must be assigned to a level. For instance, the fork-join task in Figure 1(b) has $l = 5$ levels, where each level corresponds to either a *fork* phase or a *join* phase of execution.

[1] For all $a, b \in \mathbb{Z}_{>0}$, $\frac{a}{b} < \frac{a-1}{b-1}$ when $a > b$.

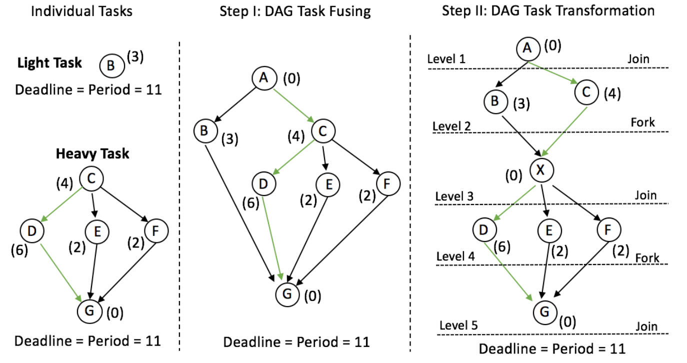

Fig. 2. A motivating example where two DAG tasks, heavy and light, are fused together and transformed into a fork-join DAG task resulting in improved schedulability.

III. Motivations

The effectiveness of the federated scheduling approach in fully utilizing the given processing capacity, i.e., available processors, is heavily dependent on the degree of *intra-task* parallelism that may exist in a *heavy* DAG task. In the worst-case scenario, a federated schedule can lead to more than half the system resources being wasted on heavy DAG tasks. The problem is further aggravated when high *tensity* heavy DAG tasks are scheduled using a federated scheduling algorithm. That is, we observe that for any given heavy DAG task with implicit deadline, if we increase the length of its critical path, and in turn its tensity, then the minimum number of cores required to feasibly schedule the DAG using a federated scheduling policy also increases as reported in the explanation for Equation 5 (in Section II-A). Jiang et.al. [6] proposed a utilization-tensity (UT) schedulability bound to amortize the overall resource wastage in federated scheduling. However, Jiang's approach categorically fails to achieve any improvement on heavy DAGs with considerably high tensity values as shown in our results in Section VI-C.

For our motivating example, consider a task set consisting of a heavy DAG task τ_h, along with a light DAG task τ_l

as shown in Figure 2. Both tasks have an assigned period of 11 time units. Also assume that the task set is being scheduled using the federated scheduling algorithm. Therefore, the utilization $U(\tau_h)$, and tensity $\omega(\tau_h)$ values of the DAG task τ_h are 1.27 and 0.9 respectively, and the minimum number of cores required to feasibly schedule task τ_h using the federated scheduling framework is calculated to be 4 using Equation 5. Similarly, the light task τ_l is assigned to a separate core. Thus, we require at least 5 processor cores to successfully schedule the given example task set. However, in reality only 3 cores are needed. Such a scheduling approach, therefore, can lead to more than *half* of the total processing capacity being wasted. Moreover, the same example task set when scheduled using the UT scheduling approach, fails to reduce any utilization wastage since τ_h is a heavy task with high tensity value,

We tackle such a resource under-utilization problem by proposing a novel DAG task fusion-and-transformation technique to improve the usable utilization of the system within the federated scheduling framework. Given a set of DAG tasks, our proposed approach, first, groups a heavy DAG task τ_h with a compatible light task τ_l to form a task tuple (τ_h, τ_l). Then we combine the DAGs in said tuple to improve the intra-task parallelism of the resultant fused DAG. Finally, we transform the fused DAG into a HL-DAG task that conforms to a fork-join DAG task model to limit the required number of processing cores for a feasible schedule while still improving the achievable task parallelism in the task set. For our example task set in Figure 2, fusing the heavy DAG task τ_h with the light task τ_l may inadvertently result in a DAG task (in Step I) that will require considerably more processing cores (7 processors in our case) for a feasible federated schedule. This is due to an increased parallelism in the resultant fused DAG. Hence, there is a need to transform our fused DAG task into a fork-join DAG task model (Step II of Figure 2) which has the following advantages; (1) effectively limit the intra-task parallelism in DAGs, and (2) leverage existing scheduling approaches [8] to derive a tighter processor demand bound. For example, the transformed DAG task in Step II of Figure 2 will require *only* 3 processing cores using the partitioned scheduling policy in [8] while still adhering to the federated scheduling framework.

While DAG task fusion-and-transformation results in reduced resource wastage, such a modification, however, may change the collective period of the fused tasks under consideration, and increase the total utilization of the task set. This can potentially result in violating the timing requirements of the individual tasks. We address these challenges next.

IV. OFFLINE HL-DAG TASK TRANSFORMATION

We construct a heavy-light (HL)-DAG task to amortize the grave under-utilization of processing capacity in federated scheduling while maintaining logical and timing correctness. A HL-DAG is defined as a real-time DAG task formed by (1) fusing together a heavy DAG task with a compatible light DAG task, and (2) transforming the fused task into a fork-join DAG task. We exploit the existing works that provide high timeliness guarantees during parallel task execution in fork-join DAGs to tightly bound the desired processing capacity of HL-DAG to improve resource utilization in a federated scheduling framework. In this section, we first introduce a HL-DAG task and its transformation process, followed by its representation using a real-time task model, and discuss its timing and logical correctness. Finally, we present a technique to select which tasks to fuse, and transform into HL-DAGs in order to maximize the schedulability of the system. Note that for a given task set, the process of creating HL-DAG tasks is carried out once offline.

A. Overview and Requirements of Task Fusion

In a federated scheduling framework, light tasks cannot utilize the resources dedicated exclusively to heavy tasks. This may lead to acute resource wastage if heavy tasks cannot fully utilize the resources assigned to them. We tackle this problem by fusing a heavy DAG task with a light DAG task so that both tasks can improve the overall resource utilization. In this section, we discuss how to fuse a heavy DAG task τ_h with a compatible light DAG task τ_l together. The question on which two DAG tasks to fuse together will be addressed later in section IV-D. We focus on independent DAG tasks with no inter-task data dependency for fused DAG task construction in this work.

To help us ensure logical correctness during the process of task fusion, we list the following set of constraints. First, as discussed in Section II-A, we consider a synchronous periodic DAG task set consisting of both heavy tasks and light tasks with implicit deadlines. Second, all sub-tasks in the resultant fused DAG task must preserve the precedence relationship as observed in the original *heavy* DAG task, thereby maintaining the original DAG task structure. We only need to preserve the precedence constraints of each heavy task since all light tasks are considered to have sequential execution in federated scheduling algorithm. Third, the fused DAG task must maintain a unique start vertex and a unique end vertex to adhere to the DAG task model assumptions described in Section II-A. Finally, the length of the critical path of the fused DAG task *must* be equal to the critical path length of the original heavy DAG task. This ensures timing correctness in that the light DAG task's execution context starts and finishes within the execution context of the heavy task. We plan to redress this restriction in future work. Step I of Figure 2 shows an example process of task fusing, where a heavy DAG task τ_h with implicit deadline of 11, is fused with a light DAG task τ_g with the same deadline.

However, fusing two DAG tasks and executing them together within a common execution context may lead to an increased contention of processing capacity shared between the two tasks. Federated scheduling would tackle such an increase in computational interference by dedicating additional processors to feasibly schedule the fused DAG as shown in our motivating example in Section III. Conversely, we propose a structural transformation technique that can limit the additional

parallelism introduced into the fused DAG due to task fusion process as shown next.

B. Maintaining Logical Correctness

In the previous section, we showed that a naive introduction of additional sub-tasks within the execution context of a heavy DAG task through task fusion may increase the overall system utilization. To tackle this challenge, we present a structural transformation technique where a fused DAG task is modified into a HL-DAG task that conforms to the fork-join model of parallel execution as shown in Step II of Figure 2. It is based on the observation that the fork and join phases of execution in a fork-join DAG have different degrees of parallelism, and therefore have different processor utilization efficiency. The fork phase captures a highly parallel execution context, and may require multiple processors to fully accomplish the required parallelism. Conversely, the join phase consists of serialized execution on a single processor. We propose a heuristic transformation of DAG task structures that exploits said observation to find a judicious trade-off that can utilize the available parallelism with negligible increase in required processing capacity.

Step I in Figure 2 shows a fused DAG task before transformation, where the light task B is currently an independent sub-task in the DAG that requires parallel execution. Therefore, it can either (1) execute on a separate core, thus, increasing the required processing capacity, or (2) potentially interfere with the execution of the rest of the sub-tasks in the DAG by sharing the available processors, possibly leading to missed deadlines. The amount of interference that a critical path in a DAG can withstand, without missing its deadline, can be quantified by the available *slack* in the DAG. The slack is defined as the temporal difference between a DAG deadline and its span. For instance, our example fused DAG in Figure 2 has a slack of 1 time units. Our DAG transformation policy assigns the parallel execution within the fork phase of the HL-DAG to fully exploit the allocated processing capacity, while any contention in the critical path due to shared resources is assigned to the join phase, and predictably upper-bounded by the existing slack in the DAG. Step II in Figure 2 shows the construction of our HL-DAG task. We start with the fused DAG task in Step I. To construct a HL-DAG task that aligns with the fork-join model of execution, we insert a *pseudo* vertex 'X' which denotes a sub-task with WCET given by $C(X) = 0$. If the WCET of the light task was 3+σ, we could easily split it in two sequential sections, one with WCET 3 (added to the fork phase) and the rest of the execution context σ within the join phase, where the value of $sigma$ is upper-bounded by the available slack (in our case, 1) in the DAG reflected by $C(X) = \sigma$.

C. Maintaining Temporal Correctness

While the previous section focused on maintaining the logical correctness of the individual tasks after HL-DAG transformation, we now explore the timing correctness of such a transformation technique. Let us reconsider our motivating

TABLE I
TASK SET 1

Task (τ)	C	T
τ_l	3	21
τ_h	14	11

TABLE II
MODIFIED TASK SET 1

Task (τ)	C^{peak}	C^{nml}	T	l
τ_{hl}	17	14	11	1

example in Section III where the corresponding task set consists of two synchronous and independent tasks; a heavy DAG task τ_h and a light DAG task τ_l. Since both tasks have the same period, it is easy to run τ_l within the execution context of τ_h while meeting individual task deadlines. However, in reality, tasks can have different periods assigned to them. In such a case, there are two scenarios associated with scheduling the HL-DAG that may pose challenges for upholding the temporal correctness: (1) scheduling the HL-DAG with τ_l's period, say with larger period, may be logically incorrect since τ_h, with smaller period, needs to run more frequently, and similarly, (2) executing the HL-DAG with τ_h's period may result in unnecessary resource usage since τ_l, with larger period, needs to run less frequently. Therefore, it is important to ascertain the time intervals between τ_h and τ_l within which can execute the HL-DAG, while at other times we run only τ_h since it is the task with a lower period. We leverage a variant of the multi-frame task model [14] to model the execution of a HL-DAG task as $\tau_{hl} = (T_{hl}, C_{hl}^{peak}, C_{hl}^{nml}, l_{hl})$. The term T_{hl} is defined as the period of the HL-DAG task τ_{hl}, and is set as $\min\{T_h, T_l\}$. The execution time parameter C_{hl}^{peak} corresponds to the WCET of a *frame* when the fused-and-transformed execution context of both τ_h and τ_l is running, and is calculated using equation 6. Conversely, C_{hl}^{nml} corresponds to the WCET of a frame when the fused-and-transformed execution context is not running, and is calculated using the equation 7. The parameter l_{hl} captures the minimum inter-peak frame distance such that every l_{hl} consecutive frames contain at most *one* peak frame.

$$C_{hl}^{peak} = C(\tau_h) + C(\tau_l) \quad (6)$$

$$C_{hl}^{nml} = C(\tau_h) \quad (7)$$

$$l_{hl} = \left\lfloor \frac{T_l}{T_{hl}} \right\rfloor \quad (8)$$

Example #1: Let us consider two tasks, a heavy DAG task τ_h and a light DAG task τ_l, with time periods 11 and 21 respectively. The other real-time parameter(s) of the tasks in this task set are given in Table I. We will now use Equations 6–8 to generate the real-time parameters for the HL-DAG task τ_{hl}. Therefore, we have $C_{hl}^{peak} = 17$ and $C_{hl}^{nml} = 14$. Similarly, T_{hl} is set to 11 since $\min\{\mathrm{T}_h, \mathrm{T}_l\} = 11$. From Equation 8, we know that l_{hl} equals to 1, and the task HL-DAG maintains temporal correctness as long as each $(T_{hl} \cdot l_{hl})$ time interval contains at most *one* C^{peak} frame as shown in Figure 3(a). Note that even though the WCETs of each execution frame is greater than its time period, the illustration in Figure 3(a) assumes that a multicore scheduling algorithm can fully exploit the existing intra-task parallelism to ensure that the DAG frame completes within its deadline. At time 0,

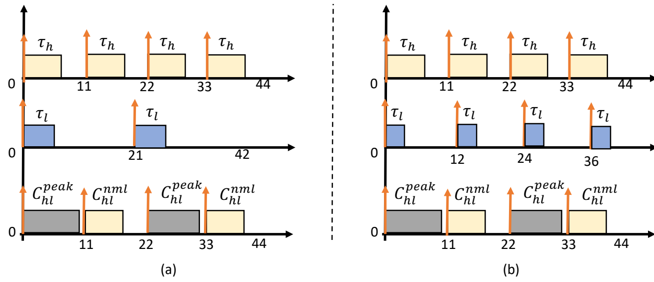

Fig. 3. Example execution instants of HL-DAG for (a) task parameters in Table II, and (b) task parameters in Table IV.

TABLE III
TASK SET 2

Task (τ)	C	T
τ_l	3	12
τ_h	14	11

TABLE IV
MODIFIED TASK SET 2

Task (τ)	C^{peak}	C^{nml}	T	l
τ_{hl}	17	14	11	1

both job of τ_h and τ_l have been released allowing HL-DAG to execute its C_{hl}^{peak} frame. In contrast, at time 11, the HL-DAG will have to run its C_{hl}^{nml} frame since a new job instance of τ_l is yet to be released. The next C_{hl}^{peak} frame can only be formed at time 22. The same pattern is repeated after every hyperperiod of τ_{hl} (the least common multiple of the periods of τ_h, τ_l). The modified task set is shown in Table II.

D. *Finding* HL-*DAG Candidates*

While the DAG transformation technique reduces the collective processor under-utilization, such a modification may change the collective period of the individual DAG tasks under consideration, thereby, leading to incorrect timeliness. For instance, in Example #1 (Section IV-C), even though a new job of task τ_l is released at time 21 (in Figure 3(a)), it can only start within the next frame of HL-DAG (C^{peak}) at time 22. Therefore, it is important to bound the delay between the *release* time and the *start* time for such a DAG task to maintain temporal correctness. Let us consider the task set in Table III where the modified task set (Table IV) after HL-DAG transformation is illustrated in Figure 3(b). While the job instance of τ_l is ready at time 12, it will only get to execute at a delayed time 22 within the corresponding C^{peak} frame, and will fail to meet the deadline set at 24 since $C(\tau_l)$= 3. Thus, we must have at least one job of τ_h completely contained within each period of τ_l to guarantee the timeliness of individual tasks τ_h and τ_l. The following lemma 1 [15] must be satisfied to select compatible DAG tasks for HL-DAG construction.

Lemma 1: ([15]). Let us consider a HL-DAG candidate, which consists of the task-tuple $\{\tau_h,\ \tau_l\}$. If $T_h \leq T_l$ and $2 \cdot T_h - gcd(T_h, T_l) \leq T_l$ where $gcd()$ denotes the *greatest common divisor*, then each job of the HL-DAG candidate with period T_h is guaranteed to meet the individual deadlines of tasks τ_h and τ_l.

For our HL-DAG candidates $\{\tau_h,\ \tau_l\}$, the period is set at $\min(T_h,\ T_l)$, and the $work(\tau_l) \leq span(\tau_h)$ (Section IV-C).

V. HL-FED: A FEDERATED SCHEDULING FRAMEWORK

Following a successful HL-DAG transformation process, any given task set in our proposed solution can have the following types of real-time DAGs; HL-DAG tasks, and the rest of heavy and light DAG tasks in the task set that could not be fused and transformed into HL-DAGs. Therefore, we propose a multicore scheduling framework that can feasibly schedule such mixed DAG task sets. In this work, we extend the existing federated scheduling framework to improve the overall usable system utilization while providing a tighter processor demand bound to ensure timeliness guarantees.

Figure 4(a) describes our proposed solution. We divide our approach into two phases. In the first phase, we leverage an existing deadline-based fixed-priority partitioned scheduling policy [8] to schedule HL-DAGs on a dedicated subset of available processing cores. We opt for partitioned scheduling since it has the advantage of reducing the multiprocessor scheduling problem to scheduling on individual processors. We focus on partitioned scheduling where each sub-task of the HL-DAG task is statically assigned to a processor core based on its deadline. Each sub-task deadline is assigned using our local deadline assignment policy described in Section V-A. While the scheduling algorithm in [8] uses the first-fit decreasing (FFD) to heuristically assign sub-tasks to processors, our approach uses an offline first-fit non-decreasing (FF) sub-task to core assignment policy. Our proposed algorithm works as follows; (1) all the sub-tasks in the critical path are allocated to a separate individual processor, while (2) the remaining sub-tasks are heuristically bin-packed into the rest of the processing cores based on FF task assignment policy. Finally, the sub-tasks in each processor are scheduled through a local deadline-based fixed-priority scheduling algorithm (DM-FF). Our choice of DM-FF is based on the observation that both dynamic- and static-priority scheduling perform similarly for fork-join model of DAG task execution. In the second phase, we use the utilization-tensity based mixed scheduling algorithm in [6] to schedule the remaining tasks, both heavy and light DAGs that could not be transformed into HL-DAGs, on the rest of the available processors in the system. Both phases of the scheduling algorithm must result in a successful schedule, otherwise, the task set as a whole is considered infeasible.

A. *Assigning Local Deadlines*

The two-phase HL-FED scheduling algorithm proposed in this work performs an offline first-fit task assignment, and a deadline-based fixed priority partitioned scheduling algorithm (DM-FF) to feasibly schedule HL-DAGs. However, since any DAG task is assigned a *single* overall deadline (or period since we assume real-time tasks with implicit deadlines in this work), we must allocate sub-task level deadlines to successfully realize deadline monotonic scheduling on individual processors. We solve this problem by proposing a slack-based local deadline (LD) assignment technique. First, we compute and distribute slack during the schedule such that it ensures that the independent light task can reclaim the under-utilized

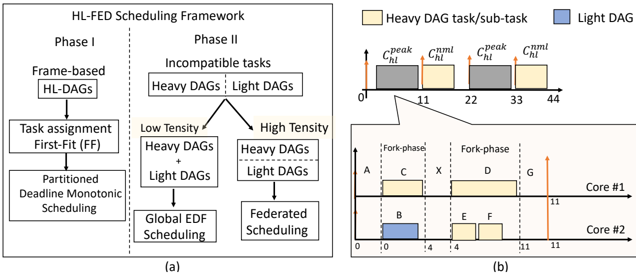

Fig. 4. (a) An overview of our proposed scheduling algorithm (HL-FED). Phase I schedules newly formed HL-DAGs, while phase II uses an existing work [6] to feasibly schedule the rest of DAGs consisting of incompatible HL-DAG candidates; (b) An example execution instant of the *peak* frame of HL-DAG (Table II), along with sub-task level assigned local deadlines.

processing capacity within the heavy DAG execution context. We can compute the worst-case response time $R(\tau_{hl}^{\upsilon})$ of all the sub-tasks in each level $\upsilon \in \Upsilon$ of the HL-DAG task τ_{hl} by modifying Equation 5 as follows.

$$R(\tau_{hl}^{\upsilon}) = \max\left\{span(\tau_{hl}^{\upsilon}), \left\lceil \frac{work(\tau_{hl}^{\upsilon}) - span(\tau_{hl}^{\upsilon})}{p-1} \right\rceil \right\} \tag{9}$$

where $work(\tau_{hl}^{\upsilon})$ and $span(\tau_{hl}^{\upsilon})$ represents the length of the longest complete path and the aggregate WCETs of all the sub-tasks of the DAG task τ_{hl}^{υ} in level υ, respectively, and p represents the number of processors assigned to this HL-DAG. Finally, we assign a local deadline $LD(\upsilon)$ to each level $\upsilon \in \Upsilon$ such that,

$$LD(\upsilon) = LD(\upsilon - 1) + R(\tau_{hl}^{\upsilon}) \tag{10}$$

Figure 4(b) shows an example multi-frame based scheduling instant of the HL-DAG task whose real-time parameters are given in Table II and its DAG task structure in illustrated in Step II of Figure 2. The computed local deadlines within the peak frame C^{peak} of the HL-DAG is shown at the end of each phase of execution, i.e., the fork phase and the alternating join phase as part of the execution timeline. Similarly, the normal frame C^{nml}, which is in fact the original heavy task, will have the same assigned local deadlines since excluding the light task does not change the overall response time analysis.

VI. Simulation

In this section, we evaluate the benefits of our proposed approach by scheduling synthetic real-time DAG tasks on a multicore system.

A. Setup

We validate our proposed scheduling approach (HL-FED) and assess its real-time performance in a simulated environment on randomly generated task sets. We designed a task set generator to create synthetic benchmark applications over a range of utilization levels (100%, 150%, 200%, ..., 1000%). The period (T_i) and worst-case execution time (C_i) of each task are randomly generated so long as the overall utilization of the task set remains within the corresponding utilization level. The worst-case execution time C_i of a DAG task τ_i is upper bounded by $\sum_{q=1}^{n} C(\tau_q)$, where $C(\tau_q)$ denotes the worst-case execution time of each sub-task within a given DAG task τ_i. The value of n denotes the number of vertices for each DAG task, and is chosen from a range (10, 15, 20,..., 100). The period T_i of each DAG task τ_i is computed using Equation 4 such that the fraction $\frac{span(\tau_i)}{\omega(\tau_i)}$ ranges from (0-1) for each DAG τ_i. Once the number of vertices, and their corresponding real-time parameters, are generated, we design the DAG task structure by modifying an existing user defined pseudo-random task graph generator framework called Task Graphs For Free (TGFF) [16]. This allows us to realize the HL-DAG transformation process. For a fixed utilization, we generate 100 DAG task sets, with heavy DAGs ranging between 30%-70% of the total number of tasks. Each simulation run is carried out for one hyperperiod, which is the least common multiple of the periods of all the tasks in a task set.

B. Evaluation Metrics

We compared our approach against three most closely related works; the first work is the most closely related DAG scheduling framework that extends the federated scheduling algorithm to schedule tasks based on their tensity values [6] (henceforth denoted by UT). Moreover, since the authors in [6] compare their work against (1) the classical federated scheduling algorithm (denoted by FED) [4], and (2) a semi-federated scheduling algorithm (henceforth referred to as SF) [5], it is only fair for us to evaluate our proposed policy against them as well.

Our objective is to evaluate the effectiveness of our proposed framework HL-FED with respect to the system-wide usable utilization. We also test the scalability of our approach by comparing the processor allocation strategy by comparing the required number of processing cores to ensure a feasible schedule for each of the above mentioned scheduling algorithms. Both of these evaluation parameters have been shown to be good indicators of schedulability analysis and system performance, and are highly sensitive to system DAG task tensity values [6].

C. Results

We now assess the performance of our proposed scheduling algorithm HL-FED. Figures 5-6 compare the simulation results to highlight the benefits of HL-FED over existing algorithms for scheduling DAG tasks. Figure 5(a) reports the simulation results comparing the performance of HL-FED against UT, SF, and FED in terms of the percentage of feasible task sets as a function of utilization demands where the DAG task tensity is set as low between 0.1-0.5. The number of heavy DAGs is set at 30% of the total number of tasks in the task set. We compute the number of cores saved by each scheduling algorithm by comparing the total cores actually used for scheduling against the worst-case theoretical processor requirement using Equation 5. The trend shows that UT improves the usable utilization 9% and 5% on average over SF and HL-FED, respectively, across all utilization levels. It also shows an improvement in the usable utilization value by 22% on average

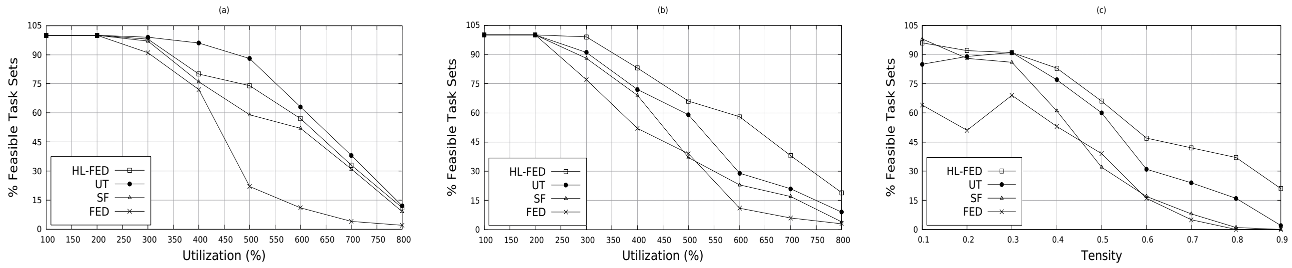

Fig. 5. Simulation results showing (a) the percentage of feasible task sets as a function of utilization for DAGs with low tensity values and low heavy DAG count in the task set, (b) the percentage of feasible task sets as a function of utilization for DAGs with high tensity values and high heavy DAG count in the task set, and (c) the percentage of feasible task sets as a function of scaling DAG tensity values.

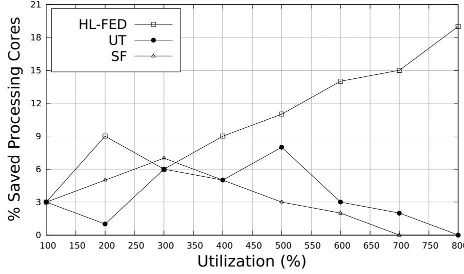

Fig. 6. Simulation results showing the percentage of saved processing cores as a function of utilization for DAGs with high tensity values and high heavy DAG count in the task set.

over FED across all utilization levels. HL-FED fails to improve on the usable utilization because (1) the task set does not have enough heavy DAG tasks to exploit the DAG transformation technique, and (2) low tensity DAGs show better schedulability results with G-EDF algorithm used in [6]. However, with high tensity values (0.7-0.9), and 50% heavy DAG tasks in the task set, Figure 5(b) shows that HL-DAG improves the usable utilization by 18%, 15% and 11% on average over FED, SF and UT across all utilization levels. This best-case scenario effectively underlines the improvement of our approach over existing schedules. Note that all algorithms policies fail to feasibly schedule task sets once the task utilization demand exceeds the 800% utilization mark.

Similarly, in the best-case scenario, Figure 5(c) reports the simulation results comparing the performance of HL-FED against FED, SF and UT in terms of percentage of feasible DAG task sets as a function of its scaling DAG task tensity values. The trend shows an improved usable utilization of 27%, 20% and 11% on average over FED, SF and UT, respectively, across all utilization levels. Figure 6 reports the simulation results comparing the scalability of our approach. It compares the performance of HL-FED against FED, SF, and UT in terms of percentage improvement in core allocation policy as a function of task set utilization. We report an average saving of up to 3%, 4% and 11% for SF, UT, and HL-FED algorithms, respectively, over the baseline FED scheduling algorithm across all utilization levels.

VII. Conclusion

In this work, we introduce a real-time DAG task model called HL-DAG with support for the fork-join execution model in a federated scheduling framework. We present a static DAG task transformation technique, and an offline task assignment and scheduling framework (HL-FED) which provides improved performance, while maintaining timing and logical correctness over existing DAG scheduling algorithms for multicore systems.

References

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

[2] J. Li, K. Agrawal, C. Lu, and C. Gill, "Outstanding paper award: Analysis of global edf for parallel tasks," in *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013, pp. 3–13.

[3] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic dag task model," in *2013 25th Euromicro conference on real-time systems*. IEEE, 2013, pp. 225–233.

[4] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 85–96.

[5] X. Jiang, N. Guan, X. Long, and W. Yi, "Semi-federated scheduling of parallel real-time tasks on multiprocessors," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 80–91.

[6] 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, 2019.

[7] S. Baruah and T. Baker, "Schedulability analysis of global edf," *Real-Time Systems*, vol. 38, no. 3, pp. 223–235, 2008.

[8] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *2010 31st IEEE Real-Time Systems Symposium*. IEEE, 2010, pp. 259–268.

[9] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[10] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Pearson Education, 2014.

[11] M. Odersky, L. Spoon, and B. Venners, *Programming in scala*. Artima Inc, 2008.

[12] R. P. Brent, "The parallel evaluation of arithmetic expressions in logarithmic time," *Complexity of sequential and parallel numerical algorithms*, pp. 83–102, 1973.

[13] ——, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 201–206, 1974.

[14] V. Lesi, I. Jovanov, and M. Pajic, "Security-aware scheduling of embedded control tasks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 188, 2017.

[15] A. Mukherjee, T. Mishra, T. Chantem, N. Fisher, and R. Gerdes, "Optimized trusted execution for hard real-time applications on cots processors," in *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, 2019, pp. 50–60.

[16] R. P. Dick, D. L. Rhodes, and W. Wolf, "Tgff: task graphs for free," in *Proceedings of the Sixth International Workshop on Hardware/Software Codesign.(CODES/CASHE'98)*. IEEE, 1998, pp. 97–101.