Local-Deadline Assignment for Distributed Real-Time Systems

Shengyan Hong, Thidapat Chantem, *Member, IEEE,* and Xiaobo Sharon Hu, *Senior Member, IEEE*

Abstract—In a distributed real-time system (DRTS), jobs are often executed on a number of processors and must complete by their end-to-end deadlines. Job deadline requirements may be violated if resource competition among different jobs on a given processor is not considered. This paper introduces a distributed, locally optimal algorithm to assign local deadlines to the jobs on each processor without any restrictions on the mappings of the applications to the processors in the distributed soft real-time system. Improved schedulability results are achieved by the algorithm since disparate workloads among the processors due to competing jobs having different paths are considered. Given its distributed nature, the proposed algorithm is adaptive to dynamic changes of the applications and avoids the overhead of global clock synchronization. In order to make the proposed algorithm more practical, two derivatives of the algorithm are proposed and compared. Simulation results based on randomly generated workloads indicate that the proposed approach outperforms existing work both in terms of the number of feasible jobs (between 51% and 313% on average) and the number of feasible task sets (between 12% and 71% on average).

Index Terms—Real-time and embedded systems, Real-time distributed, Sequencing and scheduling, Optimization, Performance of systems.

1 Introduction

Distributed soft real-time systems are widely used in cyber-physical applications such as the multimedia [14], [18], telecommunication [23], and automatic control and monitoring systems [20]. Since such systems often experience large variations in terms of their operating environments, a number of task deadlines may be missed without severely degrading performance. The scale of these distributed soft real-time systems often prohibits a centralized resource management approach. Designing low-overhead, distributed scheduling solutions is critical to a reliable operation of such systems.

A DRTS contains a set of tasks periodically or aperiodically releasing jobs which typically have end-to-end deadlines. Each job is composed of a set of sub-jobs that are executed on different processors. Since different tasks may require execution on different sets of processors, there may be high resource competition among sub-jobs on a given processor, which could severely increase job response times, potentially resulting in end-to-end deadline misses. Although the distributed soft real-time system allows some jobs to miss their end-to-end deadlines, frequent deadline misses can degrade the Quality of Service (QoS) of the system. Therefore, it is important to properly assign local sub-job priorities in order to meet as many job deadlines as possible.

1.1 Related Work

A number of recent papers investigated the sub-job priority assignment problem for DRTSs. Most of the local-deadline assignment approaches [5], [17], [28] divide the end-to-end deadline of a job into segments to be used as local deadlines by the processors that execute the sub-jobs. The division may depend on the number of processors on which the sub-job is executed [28] or the execution time distribution of the job among the processors [5], [17]. The local deadlines then dictate sub-job priorities according to the earliest-deadline-first (EDF) scheduling policy [6], [22]. While efficient, such approaches [5], [17], [28] do not consider resource competition of different sub-jobs on a processor, which may lead to local deadline misses and eventually end-to-end deadline violations.

To ensure the schedulability of the tasks on each processor, some work combines the local-deadline assignment problem with feasibility analysis so that the resulting deadline assignment is guaranteed to be schedulable. The approaches proposed in [13], [26] assign local deadlines to the sub-jobs on-line by considering the schedulability of sub-jobs on each processor in a distributed manner. The approach [13] is based on a strong assumption that each processor knows the local release times and upper bounds on the local deadlines of all the future sub-jobs, which may be impractical for real-world applications. In [26], the absolute local deadline of each sub-job is derived on-line based on the sub-job completion time on the preceding processors and the given relative local deadline of each subtask. However, the work can not handle the situation where the relative local deadlines of subtasks are not given off-line. In contrast, the works in [19], [24], [27] assign intermediate deadlines to subtasks and consider resource contention among subtasks off-line. The schedulability condition used in work [19] (from [21]) utilizes the ratio of subtask execution time over subtask local deadline in the schedulability analysis. According to [6], this condition can be very pessimistic in testing the schedulability of subtask set when the subtask period is not equal to the subtask local deadline or the subtask is not periodic. The

- *S. Hong and X.S. Hu are with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, 46556. E-mail: {shong3, shu}@nd.edu*
- *T. Chantem is with the Deparment of Electrical and Computer Engineering, Utah State University, Logan, UT, 84322. E-mail: tam.chantem@usu.edu*

This work is supported in part by NSF under grant numbers CPS-0931195, CNS-0702761, CSR-1319718 and CSR-1319904.

work in [24] employs the feasibility condition from [2] to assign local deadlines to subtasks on each processor in an off-line, iterative manner. The drawback of the approach is that it is time consuming and cannot adapt to dynamic changes in applications. In addition, the analysis assumes that all the periodic subtasks are synchronized, which is pessimistic in testing the schedulability of subtask set. The authors in [27] proposed a local-deadline assignment scheme to minimize processor resource requirements for a single task, yet many DRTSs need to execute multiple tasks.

1.2 Contributions

To address the shortcomings of existing work, we present an on-line distributed approach which combines local-deadline assignment with feasibility analysis to meet as many applications' end-to-end deadline requirements as possible in a distributed soft real-time system. Since the proposed approach is targeted towards soft real-time systems, it supports possibly infeasible applications. By extending our previous work [13], our local-deadline assignment algorithm supports soft real-time applications which can be modeled as a directed acyclic graph (DAG) and partitioned onto processors by whichever means. Our general application model covers a wide range of cyber-physical systems, e.g., multimedia system, data processing back-end systems, signal processing systems, control systems and wireless network systems.

In order to efficiently solve the local-deadline assignment problem, we formulate the local-deadline assignment problem for a given processor as a mixed integer linear programming (MILP) problem. We further introduce a locally optimal algorithm that can solve the MILP based local-deadline assignment problem in $O(N^4)$ time, where N is the number of sub-jobs executed by the processor. We should point out that the locally optimal solution may not be a globally optimal solution for the DRTS. Given the algorithm's distributed nature, the proposed algorithm avoids the overhead of global clock synchronization. In addition, the observations made in the proofs reveal several interesting properties (such as when a busy time interval occurs) for some special sub-job subsets used in our algorithm and can be applied to similar feasibility studies.

Although the local-deadline assignment problem can be solved efficiently and effectively by the canonical version of our algorithm, it is based on the strong assumption that each processor knows the local release times and the local deadline upper bounds of all the future sub-jobs as in [13]. Fundamentally, the canonical version of our algorithm is an off-line algorithm. To relax this assumption and make our algorithm practical for real-world applications, we propose two derivatives of our algorithm. In the first derivative, each processor only considers the currently active local sub-jobs, which are released (ready to be executed) but have not been finished. In the second derivative, the processor employs a prediction mechanism to estimate the timing information of future sub-jobs in order to further exploit the capability of the ideal algorithm to improve system performance. We prove that the first derivative can find the same solution to that generated by the canonical version of our algorithm if both solve the same set of sub-jobs. Additionally, we discuss other practical considerations such as communication among processors and investigate the time overhead on the performance of the system. Since our algorithm needs to be run upon each release of a sub-job, the time overhead of the proposed algorithm grow relatively quickly as the number of sub-jobs on each processor increases. Thus, our algorithm is suitable for DRTSs with the number of active sub-jobs in the order of tens. Such DRTSs often appear in avionics and automotive applications.

1.3 Organization

The rest of the paper is organized as follows. Section 2 provides the system model and motivations for our work. Section 3 describes our general approach as well as the MILP formulation for local-deadline assignment. Section 4 presents the canonical version of our algorithm to solve the local-deadline assignment problem. Section 5 presents two derivatives of our algorithm. Section 6 discusses the communication mechanism employed by DRTSs to support OLDA and the influence of time overhead by OLDA on the system performance. Experimental results are presented and analyzed in Section 7. Section 8 concludes the paper.

2 Preliminaries

Below, we first introduce the system model and scheduling properties. We then provide motivations for our work.

2.1 System Model

We consider a DRTS where a set of real-time tasks arrive either periodically or aperiodically and require execution on an arbitrary sequence of processors. Each task T_n is composed of a set of subtasks $T_{n,k}$ and has a relative end-to-end deadline $\mathscr{D}_n$. Since our focus is on an on-line distributed local-deadline assignment method, we only consider individual task and subtask instances, i.e., jobs and sub-jobs, respectively, without any assumption on task periodicity. Job J_i is composed of M_i sub-jobs $J_{i,k}$, $k = 1, ..., M_i$, where i and k are the index numbers of job J_i and subtask $T_{n,k}$, respectively[1]. Figure 1 shows a DRTS containing 2 jobs, J_1 and J_2, and each has 5 sub-jobs.

The precedence relationship among the sub-jobs of J_i is given by a directed acyclic graph (DAG). If sub-job $J_{i,k'}$ cannot begin its execution until sub-job $J_{i,k}$ has completed its execution, $J_{i,k}$ is a *predecessor* of $J_{i,k'}$, and $J_{i,k'}$ is the *successor* of $J_{i,k}$ (denoted as $J_{i,k} \prec J_{i,k'}$). $J_{i,k'}$ is an *immediate successor* of $J_{i,k}$ and $J_{i,k}$ is an *immediate predecessor* of $J_{i,k'}$ (denoted as $J_{i,k} \preccurlyeq J_{i,k'}$) if $J_{i,k}$ is the predecessor of $J_{i,k'}$ and no job $J_{i,h}$ satisfies $J_{i,k} \prec J_{i,h} \prec J_{i,k'}$. After all the immediate predecessors of a sub-job $J_{i,k}$ have finished their execution, sub-job $J_{i,k}$ is released and can start executing. A sub-job without any predecessor is called an *input sub-job* and a sub-job without any successor is called an *output sub-job*. A sub-job path $P_{i,k,k'}$ is a chain of successive sub-jobs starting with sub-job $J_{i,k}$ and ending with an output sub-job $J_{i,k'}$. A sub-job may belong to multiple paths. We let $P_{i,k}$ be the set of paths $P_{i,k,k'}$'s starting from $J_{i,k}$. See Figure 1 for examples of these definitions.

1. We omit a task's index number n when referring to a sub-job $J_{i,k}$ because we only consider jobs and sub-jobs.

TABLE 1
Summary of Key Notations Used

Symbol	Definition	Symbol	Definition
V_x	A processor in the system	T_n, $T_{n,k}$	The nth task, and its kth subtask
$\Psi(V_x)$	Set of subtasks to be executed on processor V_x	$\Omega(V_x)$	Set of sub-jobs to be scheduled
J_i, $J_{i,k}$	The ith job, and sub-job $J_{i,k}$	$\mathscr{D}_n$	Relative end-to-end deadline of task T_n
D_i,	Absolute end-to-end deadline of job J_i	R_i	Release time of job J_i
$d_{i,k}$	Local deadline of sub-job $J_{i,k}$	$r_{i,k}$	Release time of sub-job $J_{i,k}$
$UB_{i,k}$	Local deadline upper bound of sub-job $J_{i,k}$	$s_{i,k}$	Time slack of sub-job $J_{i,k}$
$P_{i,k,k'}$, $\mathscr{P}_{i,k}$	A sub-job path starting from $J_{i,k}$ and ending with output sub-job $J_{i,k'}$, and the set of all the $J_{i,k}$'s paths	$C_{i,k}$	The worst-case execution time of sub-job $J_{i,k}$
		$C_{i,k}^{cri}$	Critical execution time of sub-job $J_{i,k}$ (See (1))

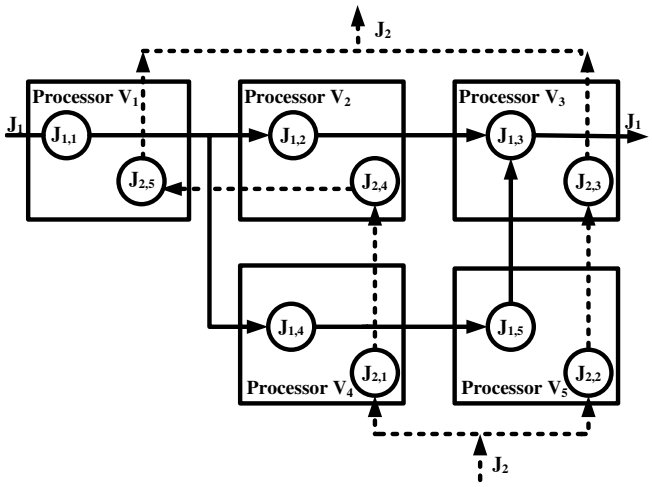


Fig. 1. An example system containing two jobs, each with 5 sub-jobs being executed on 5 processors. In the example, $J_{1,1} \preceq J_{1,2} \preceq J_{1,3}$, $J_{1,1} \preceq J_{1,4} \preceq J_{1,5} \preceq J_{1,3}$, $J_{2,1} \preceq J_{2,4} \preceq J_{2,5}$, $J_{2,2} \preceq J_{2,3}$, $J_{1,1}$, $J_{2,1}$, $J_{2,2}$ are input sub-jobs, and $J_{1,3}$, $J_{2,5}$, $J_{2,3}$ are output sub-jobs.

Job J_i is released at time R_i, and must be completed by its absolute end-to-end deadline, D_i, which is equal to $R_i + \mathscr{D}_n$. All the input sub-jobs of J_i are released at time R_i, and all the output sub-jobs of J_i must be completed by time D_i. The worst-case execution time of $J_{i,k}$ is $C_{i,k}$, and $J_{i,k}$ is associated with an absolute release time $r_{i,k}$ and absolute local deadline $d_{i,k}$, both of which are to be determined during the local-deadline assignment process. (We adopt the convention of using upper letters to indicate known values and lower letters for variables.)

We consider a multiprocessor system, where each processor V_x has a set $\Omega(V_x)$ of sub-jobs. We use $J_{i,k} \in \Omega(V_x)$ to indicate that sub-job $J_{i,k}$, an instance of subtask $T_{n,k}$, is executed on processor V_x. Subtask $T_{n,k}$ belongs to set $\Psi(V_x)$ of subtasks that reside on V_x, i.e., $T_{n,k} \in \Psi(V_x)$. Note that we do not assume any execution order among the processors in the distributed system. That is, processor V_x may appear before processor V_y in a sub-job's path while the order of the two processors may be reversed in another sub-job's path. In Figure 1, we have 5 processors, where $J_{1,1}, J_{2,5} \in \Omega(V_1)$, $J_{1,2}, J_{2,4} \in \Omega(V_2)$, $J_{1,3}, J_{2,3} \in \Omega(V_3)$, $J_{1,4}, J_{2,1} \in \Omega(V_4)$, and $J_{1,5}, J_{2,2} \in \Omega(V_5)$.

One way to meet the jobs' end-to-end deadlines is to assign local deadlines such that all the sub-jobs on every processor are schedulable and that the local deadlines of all the output sub-jobs are less than or equal to the respective end-to-end deadlines. In order to ensure that end-to-end deadlines are not violated, it is important for predecessor sub-jobs not to overuse their shares of slacks and to leave enough time for successor sub-jobs. We define the critical execution time $C_{i,k}^{cri}$ as the longest execution time among all the paths in $\mathscr{P}_{i,k}$, i.e.,

$$C_{i,k}^{cri} = \max_{P_{i,k,k'} \in \mathscr{P}_{i,k}} \sum_{\substack{\forall J_{i,h} \in P_{i,k,k'} \\ J_{i,k} \prec J_{i,h}}} C_{i,h}. \quad (1)$$

Using the definition of the critical execution time $C_{i,k}^{cri}$, we define the time slack of $J_{i,k}$ as the difference between D_i relative to $d_{i,k}$ and $C_{i,k}^{cri}$, i.e.,

$$s_{i,k} = D_i - d_{i,k} - C_{i,k}^{cri}. \quad (2)$$

The time slack provides information on the longest delay that a job can endure after the execution of sub-job $J_{i,k}$ for all of the respective output sub-jobs to meet their end-to-end deadlines. By using the end-to-end deadline and the critical execution time of the sub-job $J_{i,k}$, we define the upper bound on the local deadline of $J_{i,k}$ as

$$UB_{i,k} = D_i - C_{i,k}^{cri}, \quad (3)$$

which gives the maximum allowable value for the local deadline of $J_{i,k}$. According to (2) and (3), we see that maximizing the time slack of each sub-job on any processor provides the best opportunity for each sub-job to meet its local deadline, and for each job to satisfy its end-to-end deadline requirement. Table 1 presents the notations and definitions of the parameters and variables used throughout the paper.

We assume that EDF [6] is used on each processor since it is optimal in terms of meeting job deadlines for a uniprocessor[2]. A necessary and sufficient condition for schedulability under EDF on a uniprocessor is restated below with the notation introduced earlier.

Theorem 1. [8], [9] *Sub-job set $\Omega(V_x)$ can be scheduled by EDF if and only if $\forall J_{i,k}, J_{j,h} \in \Omega(V_x)$, $r_{i,k} \le d_{j,h}$,*

$$d_{j,h} - r_{i,k} \ge \sum_{\substack{\forall J_{p,q} \in \Omega(V_x), \\ r_{p,q} \ge r_{i,k}, \\ d_{p,q} \le d_{j,h}}} C_{p,q}. \quad (4)$$

2.2 Motivations

We use a simple DRTS to illustrate the drawbacks of existing approaches in terms of satisfying the real-time requirements. The example application contains 2 jobs, J_1 and J_2, and both jobs are composed of a chain of four sub-jobs, which are

2. This does not imply that EDF is optimal for distributed systems.

TABLE 2
A Motivating Example Containing Two Jobs that Traverse Four Processors

Processor Name	Job J_1		Job J_2		Local-Deadline Assignment BBW / OLDA		Response Time BBW / JA / OLDA	
	Execution Time	End-to-End Deadline	Execution Time	End-to-End Deadline	Job J_1	Job J_2	Job J_1	Job J_2
Processor V_1	100	N/A	70	N/A	111 / 100	90 / 170	170 / 170 / 100	70 / 70 / 170
Processor V_2	200	N/A	430	N/A	331 / 300	663 / 730	370 / 700 / 300	700 / 500 / 730
Processor V_3	100	N/A	100	N/A	441 / 400	797 / 830	470 / 800 / 400	800 / 600 / 830
Processor V_4	600	1100	100	930	1100 / 1100	930 / 930	1170 / 1400 / 1100	900 / 700 / 930

sequentially executed on four processors, V_1, V_2, V_3 and V_4. The sub-jobs' execution times and the jobs' end-to-end deadlines are shown in columns 2 to 5 in Table 2.

We consider two representative priority assignment methods: JA [15], [16] and BBW [5]. JA is a job-level fixed-priority based approach, employed by Jayachandran and Abdelzaher in [15], [16]. BBW, proposed by Buttazzo, Bini and Wu, is an end-to-end deadline partitioning based method. In [5], the local-deadline assignment by BBW is used as an input to partitioning hard real-time tasks onto multiprocessors. However, BBW can also be utilized to assign local deadlines to sub-jobs in a distributed soft real-time system when tasks have been partitioned onto different processors since it efficiently decomposes the job's end-to-end deadline in proportion to the sub-job's execution times on different processors.

In the motivating example, the local deadlines assigned by BBW (and the resultant sub-job response times) at each processor are indicated by the first value in columns 6 and 7 (and the first value in columns 8 and 9) of Table 2. The resultant sub-job response times at each processor obtained by JA are shown in the second value in columns 8 and 9. For example, under BBW, the local absolute deadline of sub-job $J_{1,1}$ on processor V_1 is 111 time units and the response time is 170 time units. BBW causes job J_1 to complete its execution on processor V_4 at time 1170 and miss its end-to-end deadline by 70 time units. The reason for J_1's end-to-end deadline miss is that BBW ignores the resource competition on individual processors and does not make the best use of the given resources, which results in an idle time interval $[470, 700]$ on processor V_3. JA performs much worse than BBW in reducing the response time of job J_1, and causes sub-job $J_{1,1}$ to complete its execution on processor V_4 at time 1400. This is because job J_1 is assigned a lower priority by JA and is preempted by J_2 on both processors V_1 and V_2. As a result, J_1 fails to meet its end-to-end deadline when its sub-job $J_{1,4}$ has a large execution time of 600 on processor V_4.

If there exists an alternative local-deadline assignment method that can consider both the workloads on a job's execution path and resource competition among different sub-jobs on a shared processor, adopting such a method may result in meeting the deadline requirements for both jobs J_1 and J_2. We will present one such method, OLDA (Omniscient Local-Deadline Assignment), in the subsequent sections. The new local deadlines obtained by OLDA are shown as the second values in columns 6 and 7 and the resultant response times are as given by the third values in columns 8 and 9 in Table 2. It is clear that this local-deadline assignment allows both jobs to meet their end-to-end deadlines.

3 Approach

In this section, we provide a high-level overview of our approach and present the detailed MILP formulation for finding a locally optimal local-deadline assignment. Since our objective is to assign local deadlines to a set of sub-jobs on-line, we will only use the concepts of jobs and sub-jobs from now on.

3.1 Overview

As shown in the last section, the probability that jobs meet their end-to-end deadlines can be greatly increased if appropriate local deadlines are assigned to the sub-jobs on different processors. Although it is possible to accomplish local sub-job deadline assignment in a global manner using mathematical programming or dynamic programming, such approaches incur high computation overhead and are not suitable for on-line use.

We adopt a distributed, on-line approach to determine local sub-job deadlines on each processor. At Algorithm 1, every time a new sub-job arrives at processor V_x, new deadlines are assigned for both the newly arrived sub-job and current active sub-jobs which are already in V_x and may have been partially executed (Section 5.1). Upon the completion of a sub-job at V_x, V_x sends a message to those downstream processors which are to execute the immediate successors of the completed sub-job. The downstream processors utilize the information contained in the message to release new sub-jobs. Consider the example shown in Figure 1. Suppose at time t, $J_{2,4}$ arrives at V_2 which is executing $J_{1,2}$. Processor V_2 suspends the execution of $J_{1,2}$ and assigns new local deadlines to $J_{2,4}$ and $J_{1,2}$. If at the same time, $J_{2,3}$ arrives at V_3, V_3 simultaneously assigns the new local

Algorithm 1 Distributed On-Line Approach in Processor V_x

1: Upon completing sub-job $J_{j,h}$ in V_x:
2: Send a message to V_y's that are to execute $J_{j,h'}$'s which satisfy $J_{j,h} \preccurlyeq J_{j,h'}$ and $J_{j,h'} \in \Omega(V_y)$
3: $\Omega(V_x) = \Omega(V_x) - \{J_{j,h}\}$
4: Execute $J_{j',h'}$ which satisfies $d_{j',h'} = \min_{J_{i,k} \in \Omega(V_x)}\{d_{i,k}\}$

5: Upon receiving a message on the completion of sub-job $J_{j,h}$ from V_y:
6: Suspend the currently executing sub-job
7: Release $J_{j,h'}$'s which satisfy $J_{j,h} \preccurlyeq J_{j,h'}$ and $J_{j,h'} \in \Omega(V_y)$, and calculate $UB_{j,h'}$'s of $J_{j,h'}$'s
8: Re-assign $d_{i,k}$'s to $J_{i,k}$'s in V_x
9: Update the dropped job record in V_x
10: Send an acknowledgement message to V_y
11: Execute $J_{j',h'}$ which satisfies $d_{j',h'} = \min_{J_{i,k} \in \Omega(V_x)}\{d_{i,k}\}$

12: Upon receiving an acknowledgement message from V_y:
13: Update the dropped job record in V_x

14: Upon dropping a subset of sub-jobs in V_x:
15: Update the dropped job record in V_x

deadlines to $J_{2,3}$ and $J_{1,3}$. If a feasible deadline assignment is not found, a sub-job dropping policy (supplemental material) is followed to remove a job from further processing. The drop information is propagated to the subsequent processors using some communication mechanism (Section 6.1).

The key to making the above distributed approach effective lies in the design of an appropriate local-deadline assignment algorithm to be run on each processor such that some specific QoS metric for the DRTS is achieved, e.g., the number of jobs dropped is minimized. In our framework, each processor determines the local-deadline assignment to maximize the minimum time slack of sub-jobs on the corresponding processor. (Readers can see the explanation of this objective in our previous work [13]). To achieve this goal, we formulate an MILP problem to capture the local-deadline assignment on each processor (Section 3.2). Then, we devise an exact off-line algorithm that can solve the MILP problem in polynomial time (more details in Section 4). The MILP problem and the off-line algorithm provide a theoretical foundation for the practical on-line local-deadline assignment algorithms (Section 5). It is important to note that our overall framework is a heuristic since the objective used by each processor to determine the local-deadline assignment does not guarantee to always lead to a globally optimal solution. (The local-deadline assignment problem for DRTS is an NP-hard problem [3].) Below, we present the MILP formulation for local-deadline assignment as it forms the basis for our off-line algorithm.

3.2 Mathematical Programming Formulation

Assuming that the release times and upper bounds on the local deadlines of all the sub-jobs are known, we capture the problem as a constrained optimization problem given below:

$$\max: \min_{J_{i,k}\in\Omega(V_x)} \quad \{D_i - d_{i,k} - C^{cri}_{i,k}\} \tag{5}$$

$$\text{s.t.} \quad r_{i,k} + C_{i,k} \le d_{i,k} \le UB_{i,k} = D_i - C^{cri}_{i,k}, \quad \forall J_{i,k} \in \Omega(V_x) \tag{6}$$

$$\max_{J_{i,k}\in\omega(V_x)} \{d_{i,k}\} - \min_{J_{i,k}\in\omega(V_x)} \{r_{i,k}\} \ge \sum_{\forall J_{i,k}\in\omega(V_x)} C_{i,k}, \forall \omega(V_x) \subseteq \Omega(V_x). \tag{7}$$

Readers can refer to our previous work [13] to see the explanation of the optimization problem formulation. The details of transforming (5), (6) and (7) to expressions in an MILP form are presented in the supplemental material.

If the release times of sub-jobs are known when computing the local deadlines, the resulting problem specified by (5), together with (6) and (7), can be solved by an MILP solver. However, such a solver is too time consuming for on-line use (see Section 7). In the next section, we introduce a polynomial time algorithm to solve the MILP problem exactly.

4 Omniscient Local-Deadline Assignment

In this section, we present the Omniscient Local-Deadline Assignment (OLDA), the canonical version of our local-deadline assignment algorithm, which solves the optimization problem given in (5), (6) and (7) in $O(N^4)$ (where N is the number of sub-jobs), assuming that the release times of all the existing and future sub-jobs are known a priori. Although OLDA is an off-line algoirthm, it forms the basis of the desired on-line algorithms (see Section 5). There are multiple challenges in designing OLDA. The most obvious difficulty is how to avoid checking the combinatorial number of subsets of $\Omega(V_x)$ in constraint (7). Another challenge is how to maximize the objective function in (5) while ensuring sub-job schedulability and meeting all jobs' end-to-end deadlines.

Below, we discuss how our algorithm overcomes these challenges and describe the algorithm in detail along with the theoretical foundations behind it. Unless explicitly noted, the deadline of a sub-job in this section always means the local deadline of the sub-job on the processor under consideration.

4.1 Base Subset and Base Sub-job

One key idea in OLDA is to construct a unique subset from a given sub-job set $\Omega(V_x)$. Using this sub-job subset, OLDA can determine the local deadline of at least one sub-job in $\Omega(V_x)$. This local deadline is guaranteed to belong to an optimal solution for the problem given in (5), (6) and (7). We refer to this unique sub-job subset of $\Omega(V_x)$ as the *base subset* of $\Omega(V_x)$ and define it as follows. We first describe sub-job subsets that are candidates for the base subset of $\Omega(V_x)$. Subset $\omega^c(V_x)$ is a candidate for the base subset of $\Omega(V_x)$ if

$$d_{c,k^c} = \min_{J_{i,k}\in\omega^c(V_x)} \{r_{i,k}\} + \sum_{\forall J_{i,k}\in\omega^c(V_x)} C_{i,k} \ge \min_{J_{i,k}\in\omega(V_x)} \{r_{i,k}\} + \sum_{\forall J_{i,k}\in\omega(V_x)} C_{i,k}, \forall \omega(V_x) \subseteq \Omega(V_x),$$

where d_{c,k^c} is the earliest completion time of $\omega^c(V_x)$. We now formally define the base subset of $\Omega(V_x)$. Let $\{\omega^c(V_x) | \forall \omega^c(V_x) \subseteq \Omega(V_x)\}$ contain all the candidates of the base subset.

Definition 1. *$\omega^*(V_x)$ is a base subset of $\Omega(V_x)$, if it satisfies*

$$\min_{J_{i,k}\in\omega^*(V_x)} \{r_{i,k}\} > \min_{J_{i,k}\in\omega(V_x)} \{r_{i,k}\}, \omega^*(V_x) \in \{\omega^c(V_x) | \forall \omega^c(V_x) \subseteq \Omega(V_x)\}, \forall \omega(V_x) \in \{\omega^c(V_x) | \forall \omega^c(V_x) \subseteq \Omega(V_x)\}.$$

The definition of the base subset of $\Omega(V_x)$ simply states that the completion time of the sub-jobs in the base subset is no less than that in any other sub-job subset in $\Omega(V_x)$ (such a property of the base subset will be proved in Lemma 4). If the completion times of all the sub-jobs in multiple subsets are the same, the base subset is the subset which has the latest released sub-job in $\Omega(V_x)$.

For a given base subset, determining which sub-job to assign a deadline to and what value the deadline should have constitutes the other key idea in OLDA. Recall that our optimization goal is to maximize the sub-job time slacks. Hence, we select this sub-job based on the local deadline upper bounds of all the sub-jobs in the base subset. Let $J_{c,k^c} \in \omega^*(V_x)$ be a candidate for the base sub-job if

$$UB_{c,k^c} \ge UB_{i,k} \quad \forall J_{i,k} \in \omega^*(V_x).$$

TABLE 3
A Sub-job Set Example

Sub-job	$r_{i,k}$	$C_{i,k}$	$UB_{i,k}$
$J_{1,1}$	0	2	35
$J_{2,1}$	4	2	42
$J_{3,1}$	5	2	39
$J_{4,1}$	6	1	35

$\omega(V_x)$	$\min_{J_{i,k}\in\omega(V_x)}\{r_{i,k}\}+\sum_{\forall J_{i,k}\in\omega(V_x)} C_{i,k}$
$\{J_{1,1},J_{2,1},J_{3,1},J_{4,1}\}$	7
$\{J_{2,1},J_{3,1},J_{4,1}\}$ $(\omega^*(V_x))$	9
$\{J_{3,1},J_{4,1}\}$	8
$\{J_{4,1}\}$	7

Let $\{J_{c,k^c}|\forall J_{c,k^c}\in\Omega(V_x)\}$ contain all the candidates for the base sub-job in $\omega^*(V_x)$. We refer to the selected sub-job as the base sub-job and define it as follows.

Definition 2. *$J_{*,k^*}\in\{J_{c,k^c}|\forall J_{c,k^c}\in\Omega(V_x)\}$ is a base sub-job for sub-job set $\Omega(V_x)$, if it satisfies*

$$(*>i)\quad or\ (*=i\ and\ k^*>k)\quad \forall J_{i,k}\in\{J_{c,k^c}|\forall J_{c,k^c}\in\Omega(V_x)\}.$$

The base sub-job has the largest local deadline upper bound among all the sub-jobs in the base subset. Ties are broken in favour of the sub-job with the largest job identifier and then in favour of the sub-job with the largest subtask identifier.

We use a simple example to illustrate how to find base subset and base sub-job. Consider a sub-job set $\Omega(V_x)$ with its timing parameters as shown in the top part of Table 3. It is easy to verify that subset $\{J_{2,1},J_{3,1},J_{4,1}\}$ is the base subset $\omega^*(V_x)$ (see the bottom part of Table 3), where d^* is 9. Among the three sub-jobs in $\omega^*(V_x)$, sub-job $J_{2,1}$ is the base sub-job according to Definition 2 since it has the largest local deadline upper bound of 42. OLDA uses the base subset and base sub-job to accomplish local-deadline assignment. The details of OLDA is given in the next subsection.

4.2 OLDA Algorithm Design

Given a sub-job set $\Omega(V_x)$, OLDA first constructs the base subset for the sub-job set. It then finds the base sub-job and assigns a local deadline to that base sub-job. The base sub-job is then removed from the sub-job set and the process is repeated until all the sub-jobs have been assigned deadlines.

Algorithm 2 summarizes the main steps in OLDA. (Recall that this algorithm is used by each processor in a distributed manner, so the pseudocode is given for processor V_x.) The inputs to OLDA are the sub-job set $\Omega(V_x)$ and the variable *Max_Allowed_Drop_Num*. $\Omega(V_x)$ contains all the active and future sub-jobs $J_{i,k}$'s. Without loss of generality, a sub-job is always associated with its local release time, execution time, and local deadline upper bound, and the local deadline upper bound of sub-job is computed before the call of OLDA. Thus, we do not use the local release time, execution time and local deadline upper bound as the input variables in OLDA. The variable *Max_Allowed_Drop_Num* is used in Function *Drop_Sub_Jobs*(), which will be discussed in the supplemental material. OLDA starts by initializing the set of sub-job deadlines (Line 1) and sorting the given sub-jobs in a non-decreasing order of their release times (Line 2), which breaks ties in favour of the sub-job with the largest job identifier and then in favour of the sub-job with the largest

Algorithm 2 OLDA($\Omega(V_x)$, *Max_Allowed_Drop_Num*)

```
1:  d = ∅
2:  Ω(V_x) = Sort_Sub_Jobs(Ω(V_x))
3:  while (Ω(V_x) ≠ ∅) do
4:      ω(V_x) = Ω(V_x)
5:      ω*(V_x) = Ω(V_x)
6:      Max_Deadline = 0
7:      Temp_Deadline = 0
8:      while ω(V_x) ≠ ∅ do
9:          Temp_Deadline = min_{J_{i,k}∈ω(V_x)}{r_{i,k}} + Σ_{J_{i,k}∈ω(V_x)}{C_{i,k}}
10:         if Temp_Deadline ≥ Max_Deadline then
11:             Max_Deadline = Temp_Deadline
12:             ω*(V_x) = ω(V_x)
13:         ω(V_x) = Remove_Earliest_Released_Sub_Job(ω(V_x))
14:     J_{*,k*} = Find_Base_Sub_Job(ω*(V_x)) //Find base sub-job J_{*,k*}
        according to Definition 2
15:     if (UB_{*,k*} ≥ Max_Deadline) then
16:         d_{*,k*} = Max_Deadline
17:         d = d ⋃ {d_{*,k*}}
18:         Ω(V_x) = Ω(V_x) − J_{*,k*}
19:     else
20:         J^drop = Drop_Sub_Jobs(ω*(V_x), Max_Allowed_Drop_Num)
            //Remove a subset of sub-jobs from ω*(V_x) according
            to some sub-job dropping policy, and return the subset
            containing the dropped sub-jobs
21:         d = ∅
22:         break
23: return d //d = {d_{i,k}}
```

subtask identifier. Then, the algorithm enters the main loop spanning from Line 3 to Line 22. The first part in the main loop (Lines 4–13) constructs the base subset $\omega^*(V_x)$ for the given sub-job set and computes the desired deadline value (*Max_Deadline*) according to Definition 1. (*Max_Deadline* is in fact the completion time of all the sub-jobs in the base subset, as will be shown in the next subsection.) The second part of the main loop (Line 14) applies Definition 2 to find the base sub-job in the base subset.

If the desired deadline value is smaller than or equal to UB_{*,k^*} of the base sub-job J_{*,k^*} (Line 15), the third part of the main loop (Lines 15–22) assigns the desired deadline value to the base sub-job as its local deadline (denoted by d_{*,k^*}) (Line 16), adds d_{*,k^*} to the set of sub-job deadlines (Line 17), and removes J_{*,k^*} from $\Omega(V_x)$ (Line 18). This process is repeated in the main loop until each sub-job in $\Omega(V_x)$ obtains a local deadline. In the case where the desired deadline value is larger than UB_{*,k^*} (Line 19), at least one sub-job will miss its deadline and a subset of sub-jobs $\mathbf{J^{drop}}$ are removed from the subset $\omega^*(V_x)$ based on some sub-job dropping policy (Line 20).(The discussion on the sub-job dropping policy is provided in the supplemental material.) Then, the set of sub-job deadlines is set to be empty (Line 21) and OLDA exits (Line 22). OLDA either returns the set of sub-job deadlines to be used by the processor in performing EDF scheduling or an empty set to processor V_x. In the latter case, V_x calls OLDA repeatedly until a feasible solution is found or all the sub-jobs in $\Omega(V_x)$ have been dropped. The time complexity of OLDA is $O(|\Omega(V_x)|^3)$, which is proved in Theorem 2, and a processor takes $O(|\Omega(V_x)|^4)$ time to solve a set $\Omega(V_x)$ using OLDA.

We use the example in Table 3 to illustrate the steps taken by OLDA to assign local deadlines given sub-job set

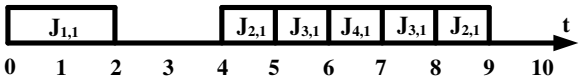


Fig. 2. The example of executing sub-jobs with local deadlines assigned by OLDA

$\{J_{1,1}, J_{2,1}, J_{3,1}, J_{4,1}\}$. In the first iteration of the main loop, OLDA finds the base subset $\{J_{2,1}, J_{3,1}, J_{4,1}\}$ and selects the base sub-job $J_{2,1}$. OLDA assigns the completion time of all the sub-jobs in the base subset, 9, to $J_{2,1}$ as its local deadline. In the next iteration, OLDA works on sub-job set $\{J_{1,1}, J_{3,1}, J_{4,1}\}$ and the process is repeated until all the sub-jobs have been assigned local deadlines. In the example, the local deadlines for sub-jobs $J_{1,1}$, $J_{2,1}$, $J_{3,1}$ and $J_{4,1}$ are 2, 9, 8 and 7, respectively. The base subset and base sub-job in each iteration are shown in Table 4. A possible schedule for sub-jobs $J_{1,1}$, $J_{2,1}$, $J_{3,1}$ and $J_{4,1}$ is shown in Figure 2.

It is worth noting that we assume that a processor knows the release times and local deadline upper bounds of all the future sub-jobs in OLDA (This assumption will be relaxed in Section 5). Thus, OLDA only requires information known upon a sub-job's release (such as the maximum allowed response time of a sub-job at the completion time of the sub-job's intermediate predecessor), which can be relayed between processors with the support of a specific distributed communication mechanism (Section 6). Therefore, OLDA does not require global clock synchronization.

4.3 Optimality of OLDA Algorithm

We claim that OLDA solves the optimization problem given by (5), (6) and (7). That is, if there exists a solution to the problem, OLDA always finds it. Furthermore, if there is no feasible solution to the problem, OLDA always identifies such a case, i.e., drop a job following some sub-job dropping policy. To support our claim, we first show that the local-deadline assignment made by OLDA (when no sub-jobs are dropped) satisfies the constraints in (6) and (7). This is given in Lemmas 1 and 2, respectively.

Lemma 1. *Given sub-job set $\Omega(V_x)$, let $d^*_{i,k}$ be the local deadline assigned by OLDA to $J_{i,k} \in \Omega(V_x)$. Then*

$$r_{i,k} + C_{i,k} \le d^*_{i,k} \le D_i - C^{cri}_{i,k} \quad \forall J_{i,k} \in \Omega(V_x). \tag{8}$$

Lemma 2. *Given sub-job set $\Omega(V_x)$, let $d^*_{i,k}$ be the local deadline assigned by OLDA to $J_{i,k} \in \Omega(V_x)$. We have*

$$\max_{J_{i,k} \in \omega(V_x)} \{d^*_{i,k}\} - \min_{J_{i,k} \in \omega(V_x)} \{r_{i,k}\} \ge \sum_{\forall J_{i,k} \in \omega(V_x)} C_{i,k}, \forall \omega(V_x) \subseteq \Omega(V_x). \tag{9}$$

TABLE 4
Base Subset and Base Sub-job in Each Iteration

Iter. Number	Sub-job Set	Base Subset	Base Sub-job
1	$\{J_{1,1}, J_{2,1}, J_{3,1}, J_{4,1}\}$	$\{J_{2,1}, J_{3,1}, J_{4,1}\}$	$J_{2,1}$
2	$\{J_{1,1}, J_{3,1}, J_{4,1}\}$	$\{J_{3,1}, J_{4,1}\}$	$J_{3,1}$
3	$\{J_{1,1}, J_{4,1}\}$	$\{J_{4,1}\}$	$J_{4,1}$
4	$\{J_{1,1}\}$	$\{J_{1,1}\}$	$J_{1,1}$

To show that OLDA always identifies the case where there is no feasible solution to the optimization problem, we observe that OLDA always finds a local-deadline assignment without dropping any job if there exists a feasible solution that satisfies constraints (6) and (7). This is stated in the following lemma.

Lemma 3. *Given sub-job set $\Omega(V_x)$, if there exists $d_{i,k}$ for every $J_{i,k} \in \Omega(V_x)$ that satisfies (6) and (7), OLDA always finds a feasible local-deadline assignment for every $J_{i,k} \in \Omega(V_x)$.*

Proving that the local-deadline assignment made by OLDA indeed maximizes the objective function in (5) requires analyzing the relationship among the sub-jobs' time slacks. Since OLDA assigns sub-job local deadlines by identifying the base sub-job in each base subset, a special property that the base subset possesses greatly simplifies the analysis process. Lemma 4 below summarizes this property.

Lemma 4. *Let $\omega^*(V_x)$ be a base subset of sub-job set $\Omega(V_x)$ and $r^* = \min_{J_{i,k} \in \omega^*(V_x)} \{r_{i,k}\}$. Under the work-conserving EDF policy, processor V_x is never idle once it starts to execute the sub-jobs in $\omega^*(V_x)$ at r^* and before it completes all the sub-jobs in $\omega^*(V_x)$. In addition, the busy interval during which the sub-jobs in $\omega^*(V_x)$ are executed is $[r^*, r^* + \sum_{\forall J_{i,k} \in \omega^*(V_x)} C_{i,k}]$. Furthermore, there is at least one sub-job unfinished at any time instant within $[r^*, r^* + \sum_{\forall J_{i,k} \in \omega^*(V_x)} C_{i,k})$.*

Based on Lemma 4, it can be proved that the local-deadline assignment made by OLDA maximizes the objective function (5), which is stated in Theorem 2.

Theorem 2. *Given sub-job set $\Omega(V_x)$, let $d^*_{i,k}$ be the local deadline assigned to each $J_{i,k} \in \Omega(V_x)$ by OLDA. Then $d^*_{i,k}$ maximizes the minimum time slack, $\{D_i - d_{i,k} - C^{cri}_{i,k}\}$, among all the sub-jobs executed on V_x, i.e.,*

$$\textit{max:} \min_{J_{i,k} \in \Omega(V_x)} \left\{D_i - d_{i,k} - C^{cri}_{i,k}\right\}. \tag{10}$$

Based on Theorem 2, we conclude that the solution found by OLDA maximizes the objective function (10). Note that the found solution may not be globally optimal for DRTS.

Based on Lemmas 1, 2, 3 and Theorem 2, we have the following theorem.

Theorem 3. *In $O(|\Omega(V_x)|^3)$ time, OLDA returns a set of local deadlines if and only if there exists a solution to the optimization problem specified in (5), (6) and (7). Furthermore, the returned set of local deadlines is a solution that maximizes the objective function (5).*

The importance of Theorem 3 is that the deadline assignment problem can be solved exactly by OLDA in polynomial time even though the original MILP formulation contains $(|\Omega(V_x)| + 2^{|\Omega(V_x)|})$ constraints. Note that processor V_x needs $O(|\Omega(V_x)|^4)$ time to solve $\Omega(V_x)$ using OLDA since V_x may call OLDA for at most $|\Omega(V_x)|$ number of times due to dropping sub-jobs.

5 More Practical Versions of OLDA

OLDA assumes that a processor knows the release times and upper bounds on the local deadlines of all the future sub-jobs on a given processor. For general DRTSs, this assumption may

be too strong and hence OLDA may not be directly applicable in certain real-world applications. In order to make OLDA more practical, we propose below two derivatives of OLDA.

5.1 Active Local-Deadline Assignment

In this section, we present a local-deadline assignment algorithm in which each processor considers only the active sub-jobs. We refer to this new algorithm as ALDA. In ALDA, sub-job set $\Omega^a(V_x)$ contains only the active sub-jobs on the processor when ALDA is invoked. Whenever a sub-job is completed, it is removed from $\Omega^a(V_x)$. Every time a new sub-job arrives at the processor, the processor stops its current execution and calls ALDA to determine the deadlines of all the active sub-jobs. The remaining execution times of the active sub-jobs, which is maintained by the processor, are used by ALDA instead of the original execution times. Note that ALDA only returns the solution of $\Omega^a(V_x)$. For a given sub-job set $\Omega(V_x)$ containing sub-jobs with different release times, a sub-job can be assigned local deadlines by ALDA multiple times from its release to its completion. This is because multiple sub-jobs may be released during such a time interval. Hence, the solution of $\Omega(V_x)$ by ALDA is a set of local deadlines, $\{d_{i,k}\}$, where $d_{i,k}$ is the last local deadline assigned to each sub-job $J_{i,k} \in \Omega(V_x)$ by ALDA.

ALDA actually is very similar to OLDA in that both algorithms need to find the base subset and the base sub-job and then assign the local deadline to the base sub-job. However, ALDA only considers all the active sub-jobs on the processor, which possesses a special property. The property can greatly reduce the time complexity of OLDA and is summarized in Lemma 5.

Lemma 5. *Given sub-job set $\Omega^a(V_x)$, if all the sub-jobs are ready for execution, the base subset $\omega^*(V_x)$ is just $\Omega^a(V_x)$.*

Based on Lemma 5, it costs ALDA a lower time overhead to identify the base subset than that of OLDA.

The steps of ALDA are summarized in Algorithm 3. We briefly discuss the key steps and omit the ones that are similar to OLDA. The inputs to ALDA are the newly released sub-job $J_{j,h}$ and the sub-job set $\Omega^a(V_x)$ that contains all the active sub-jobs that are already in V_x before the current invocation of ALDA. The sub-jobs in $\Omega^a(V_x)$ are sorted in the non-decreasing order of the upper bound on the local deadline of each sub-job in $\Omega^a(V_x)$. Ties are broken in favour of the sub-job with the largest job identifier and then in favour of the sub-job with the largest subtask identifier. Since the sub-job set $\Omega^a(V_x)$ is the base subset according to Lemma 5, the sorting of sub-jobs in $\Omega^a(V_x)$ makes the tail sub-job in $\Omega^a(V_x)$ the base sub-job according to Definition 2. In addition, ALDA directly calculates the desired local deadline value, *Max_Deadline*, for $\Omega^a(V_x)$ according to Lemma 4 (Lines 4–6).

The algorithm then enters the main loop spanning from Line 7 to Line 22. ALDA finds the base sub-job J_{*,k^*}, which is the last sub-job of the sub-job set in $\Omega^a(V_x)$ according to Lemma 5 (Line 8). If the desired local deadline value is smaller than or equal to UB_{*,k^*}, ALDA updates *Max_Deadline* for the next iteration (Line 14). If the desired local deadline value is larger than UB_{*,k^*} of the base sub-job J_{*,k^*}, the total execution time of the removed sub-jobs, *Temp_C*, is calculated (Lines 17–19). Since the sub-jobs in $\mathbf{J^{drop}}$ are removed from the base subset $\Omega^a(V_x)$, *Temp_C* is reduced from *Max_Deadline* (Line 20). According to the local-deadline assignment in ALDA (Lines 4–14), a sub-job which is assigned its local deadline earlier will have a longer local deadline than a sub-job being assigned its local deadline later. This implies that a sub-job that has been moved to $\Omega^{'}(V_x)$ will be completed after the sub-jobs currently still in $\Omega^a(V_x)$. Thus, after removing the sub-jobs in $\mathbf{J^{drop}}$ from $\Omega^a(V_x)$, each sub-job in $\Omega^{'}(V_x)$ can be completed earlier by *Temp_C* and each previously assigned local deadline is reduced by *Temp_C* (Lines 21–22).The above process is repeated until each sub-job in $\Omega^a(V_x)$ either receives a deadline or is dropped. ALDA eventually returns the set of sub-job deadlines to be used by the processor in performing EDF based scheduling.

Algorithm 3 ALDA($\Omega^a(V_x)$, $J_{j,h}$)

1: $\Omega^a(V_x)$ = *Insert_by_Non_Dec_Local_Deadline_UB*($\Omega^a(V_x)$, $J_{j,h}$) //Insert $J_{j,h}$ into the sub-job set in the non-decreasing order of the upper bound on the local deadline of each sub-job in $\Omega^a(V_x)$
2: $\mathbf{d} = \emptyset$
3: $\Omega^{'}(V_x) = \emptyset$
4: $Max\_Deadline = 0$
5: **for** ($J_{i,k} \in \Omega^a(V_x)$) **do**
6: $Max\_Deadline = Max\_Deadline + C_{i,k}$
7: **while** ($\Omega^a(V_x) \neq \emptyset$) **do**
8: $J_{*,k^*} = Tail(\Omega^a(V_x))$ //Select the base sub-job which is the last sub-job of the sub-job set in $\Omega^a(V_x)$
9: **if** ($UB_{*,k^*} \geq Max\_Deadline$) **then**
10: $d_{*,k^*} = Max\_Deadline$
11: $\mathbf{d} = \mathbf{d} \bigcup \{d_{*,k^*}\}$
12: $\Omega^a(V_x) = \Omega^a(V_x) - J_{*,k^*}$
13: $\Omega^{'}(V_x) = \Omega^{'}(V_x) \bigcup \{J_{*,k^*}\}$
14: $Max\_Deadline = Max\_Deadline - C_{*,k^*}$
15: **else**
16: $\mathbf{J^{drop}} = Drop\_Sub\_Jobs(\Omega^a(V_x))$ // Remove a subset of sub-jobs from $\Omega^a(V_x)$ according to some sub-job dropping policy, and return the subset containing the dropped sub-jobs
17: $Temp\_C = 0$
18: **for** ($J_{i,k} \in \mathbf{J^{drop}}$) **do**
19: $Temp\_C = Temp\_C + C_{i,k}$
20: $Max\_Deadline = Max\_Deadline - Temp\_C$
21: **for** ($J_{i,k} \in \Omega^{'}(V_x)$) **do**
22: $d_{i,k} = d_{i,k} - Temp\_C$
23: return $\mathbf{d}$ //$\mathbf{d} = \{d_{i,k}\}$

Since ALDA simplifies OLDA by only considering the active sub-jobs on a local processor, all the lemmas and theorems in Section 4.3 still hold for ALDA except for the time complexity of ALDA. The time complexity of ALDA is dominated by the main `while` loop starting at Line 7. (Refer to Algorithm 3.) Every time a subset of sub-jobs are to be removed from $\Omega^a(V_x)$ (Line 16), OLDA needs to traverse $|\Omega^a(V_x)|$ number of sub-jobs in Function *Drop_Sub_Jobs*(). Hence, the time complexity of ALDA when handling $|\Omega^a(V_x)|$ number of sub-jobs on processor V_x is $O(|\Omega^a(V_x)|^2)$, where $|\Omega^a(V_x)|$ is the number of active sub-jobs on processor V_x. Processor V_x calls ALDA every time a new sub-job from $\Omega(V_x)$

is released at V_x. Since $|\Omega^a(V_x)| \leq |\Omega(V_x)|$, processor V_x takes $O(|\Omega(V_x)|^3)$ time to solve a set $\Omega(V_x)$ using ALDA. Compared with OLDA, ALDA is much more efficient in solving the local-deadline assignment problem.

We show next that ALDA is equivalent to OLDA. First, we have the following lemma to show that ALDA can solve the sub-job set if and only if the sub-job set is schedulable.

Lemma 6. *Let $\Omega(V_x)$ contain all the sub-jobs to be scheduled by OLDA. If and only if there exists a schedulable solution for $\Omega(V_x)$, ALDA can find a feasible solution for $\Omega(V_x)$.*

Second, we have the following lemma to show that the equivalence of the solutions found by ALDA and OLDA.

Lemma 7. *The solutions found by ALDA and OLDA are the same if $\Omega(V_x)$ is schedulable.*

Since ALDA and OLDA are equivalent for schedulable sub-job sets and we have proved the optimality of OLDA, ALDA is also an optimal algorithm to solve the proposed problem.

Based on Theorem 3, Lemma 6 and Lemma 7, we have the following theorem.

Theorem 4. *In $O(|\Omega(V_x)|^2)$ time, ALDA returns a set of local deadlines if and only if there exists a solution to the optimization problem specified in (5), (6) and (7). Furthermore, the returned set of local deadlines is a solution that maximizes the objective function (5).*

Theorem 4 shows that ALDA is able to solve the local-deadline assignment problem exactly in polynomial time, which demonstrates the same performance to that of OLDA.

5.2 Workload Prediction Based Local-Deadline Assignment

Although ALDA is extremely efficient in assigning local deadlines to the active sub-jobs, it does not consider any future sub-job when judging the schedulability of a sub-job subset. That is, the sub-job set which is deemed schedulable by ALDA at the current time may not actually be schedulable after additional sub-jobs arrive at the processor. If a processor considers the workload of some future sub-jobs when assigning local deadlines, an infeasible sub-job subset can be detected ahead of time, and a subset of sub-jobs can be dropped as early as possible without wasting valuable resources. To achieve this, we propose another derivative of OLDA, WLDA, which leverages workload prediction during local-deadline assignment to improve resource utilization and avoid dropping jobs. In WLDA, a processor can estimate the release times and local deadline upper bounds of future sub-jobs and then apply OLDA directly to all the active sub-jobs and predicted future sub-jobs on the processor.

In order to predict future sub-jobs, we define the next release interval $\Delta r\_next(J_{j,h})$ and the next local deadline upper bound difference $\Delta UB\_next(J_{j,h})$ of sub-job $J_{j,h}$ to be the average release interval and the average difference between local deadline upper bounds, respectively, of two consecutive instances of subtask $T_{m,h}$. Figure 3 depicts the main steps in WLDA. $\Omega^w(V_x)$ contains all the active sub-jobs that have been released but not finished and all the future sub-jobs that have been considered by WLDA. The prediction time window W is first calculated based on the current time and the active sub-jobs on a local processor. If the newly arriving sub-job, $J_{j,h}$, is the second or a later instance of subtask $T_{m,h}$, the next release interval ($\Delta r\_next(J_{j,h})$) and the next local deadline upper bound difference ($\Delta UB\_next(J_{j,h})$) are estimated based on past time data. Then, $\Delta r\_next(J_{j,h})$ and $\Delta UB\_next(J_{j,h})$ are used to update release times and local deadline upper bounds of future instances of $T_{m,h}$, respectively.

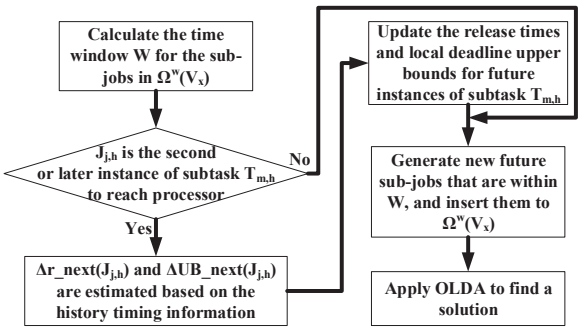


Fig. 3. WLDA flow to determine future release times and upper bounds on the local deadlines of future sub-jobs and assign local deadlines to the newly released sub-job as well as the active and future sub-jobs in $\Omega^w(V_x)$.

If $J_{j,h}$ is the first instance of a subtask to arrive at processor V_x, the release times and local deadline upper bounds of future instances of a subtask cannot be simply calculated unless subtasks are periodic. In this case, WLDA does not predict future sub-jobs for this subtask. For the newly constructed sub-job set, OLDA is to assign local deadlines to the newly released sub-job and the sub-jobs in $\Omega^w(V_x)$. The details of WLDA can be found in the supplemental material. The time complexity of WLDA is $O(|\Omega^w(V_x)|^4)$, where $|\Omega^w(V_x)|$ is the number of active and future sub-jobs on V_x. Since processor V_x calls WLDA every time a new sub-job from $\Omega(V_x)$ is released at V_x, processor V_x takes $O(|\Omega(V_x)| \cdot |\Omega^w(V_x)|^4)$ time to solve a set $|\Omega(V_x)|$ using WLDA.

Since WLDA considers some future sub-jobs, it may be able to detect an unschedulable sub-job subset earlier and hence drop sub-jobs earlier than ALDA. Doing so avoids wasteful execution of sub-jobs whose successors would be dropped later, and results in a possible decrease in the total number of dropped sub-jobs. However, if ALDA cannot find a feasible local-deadline assignment for a task set, neither can WLDA. This observation is summarized in Lemma 8.

Lemma 8. *If ALDA drops any sub-jobs due to an infeasible deadline assignment, WLDA must also drop some sub-jobs.*

6 Discussions

In this section, we discuss two important issues applicable to all versions of OLDA. Specifically, we present a communication mechanism to support timely release of a sub-job whose predecessors, possibly on different processors, have finished execution. In addition, we discuss the influence of time overhead by OLDA on the performance of the DRTSs.

6.1 Communication Mechanism

OLDA and its derivatives all rely on the following generic communication scheme. A processor completing a sub-job sends a message to downstream processors which are to execute the immediate successors of the completed sub-job. The message contains the identifier of the completed sub-job and the identifier of the subtask that the completed sub-job belongs to, which downstream processors utilize to release new sub-jobs. In addition, the maximum allowed response time of job comprising the completed sub-job is included in the message, based on which downstream processors calculate the upper bound on the local deadline of the newly released sub-job and compute the necessary local deadlines. We define the maximum allowed response time $resp_i(t)$ of job J_i at time t to be the difference between the relative end-to-end deadline $\mathscr{D}_m$ of task T_m that J_i belongs to and the total delay that J_i has experienced up to time t. Moreover, the message includes the dropped job identifiers which have been recorded in the local processor but never been told to downstream processors. The downstream processors utilize such information to drop sub-jobs for which other sub-jobs composing the same job have been dropped in other processors. Notice that whenever a sub-job is dropped, the job that this dropped sub-job belongs to cannot meet its end-to-end deadline and all the sub-jobs belonging to the job needs to be dropped. To support our proposed algorithms, we can employ a low-cost communication mechanism implemented in a bus-based network similar to those discussed in [7], [11], [12], [29].

To reduce network traffic, we do not require global clock synchronization when implementing our algorithms. The main challenge lies in how to calculate the local deadline upper bound of a newly released sub-job without requiring global clock synchronization. For any newly released sub-job, its execution time, release time and local deadline upper bound is required in OLDA. The execution time and release time of a sub-job are known locally. In contrast, the local deadline upper bound of a sub-job is determined by the end-to-end deadline of the corresponding job comprising the sub-job according to (3), which may be different on different processors in an asynchronous DRTS. Therefore, downstream processors cannot directly use the end-to-end deadline value of a job delivered from the local processor. We employ a distributed method to calculate the local deadline upper bound of a newly released sub-job, which leverages the relative end-to-end deadline of a *task*. Below, we illustrate how to accomplish this without requiring global clock synchronization.

Assume that sub-job $J_{i,k}$ belonging to subtask $T_{m,k}$ is assigned to processor V_y. To calculate $UB_{i,k}$ of $J_{i,k}$, processor V_y needs to obtain the end-to-end deadline D_i according to (3), where D_i can be calculated by using the following equation,

$$D_i = r_{i,k} + resp_i(r_{i,k}). \tag{11}$$

When all the immediate predecessors of $J_{i,k}$ have finished execution, $J_{i,k}$ is immediately released. Therefore, $resp_i(r_{i,k})$ of J_i at time $r_{i,k}$ is calculated by

$$resp_i(r_{i,k}) = \min_{\forall J_{i,h} \preccurlyeq J_{i,k}} \{resp_i(d_{i,h})\}. \tag{12}$$

Without loss of generality, suppose an immediate predecessor $J_{i,h}$ of $J_{i,k}$ is finished on processor V_x at time $d_{i,h}$. Then, processor V_x can obtain $resp_i(d_{i,h})$ by using equation

$$resp_i(d_{i,h}) = resp_i(r_{i,h}) - (d_{i,h} - r_{i,h}). \tag{13}$$

If $J_{i,k}$ is an input sub-job, the maximum allowed response time $resp_i(r_{i,k})$ of J_i is equal to the relative end-to-end deadline $\mathscr{D}_m$ of task T_m. Therefore, to calculate $UB_{i,k}$ of $J_{i,k}$, a message triggered by the completion of $J_{i,h}$ at $d_{i,h}$ is sent from V_x to V_y to inform V_y about the completion of $J_{i,h}$. Then, processor V_y reads $r_{i,k}$ directly and uses (3) to calculate the local deadline upper bound $UB_{i,k}$ of $J_{i,k}$ without global clock synchronization.

Furthermore, every time a processor drops a sub-job or overhears that a sub-job has been dropped by other processors, the local processor adds the drop information to a list of jobs whose sub-jobs have been dropped. When a message is sent from V_x to V_y to inform V_y about the completion of $J_{i,h}$, the message will include the dropped jobs' identifiers which have been recorded in V_x's list but never been told to V_y by V_x. Similarly, when V_y receives a message that a sub-job finished execution at processor V_x, V_y will send V_x an acknowledgement message. The acknowledgement message includes the dropped jobs' identifiers that have been recorded in V_y's list and have never been told to V_x by V_y. The list of dropped jobs kept by each processor is then updated and the corresponding sub-jobs are dropped when they arrive at the local processor.

In all, the message transmitted upon the completion of a sub-job contains a small amount of information, which can be supported by the bus-based network platforms, e.g., Controller Area Network (CAN). There exist some communication delays due to message transmissions among processors. When the on-line derivatives of OLDA, ALDA and WLDA, are implemented in a DRTS, the communication delays between the processors can increase the maximum allowed response time of jobs on the downstream processors and reduce the local deadline upper bounds of sub-jobs on the upstream processors. If the communication delays along the downstream paths of sub-jobs can be estimated, these delays can be readily incorporated into the maximum allowed response time of jobs and the local deadline upper bounds of sub-jobs during runtime. Hence, though ALDA and WLDA cannot precisely handle communication delays along the downstream paths, they can indirectly account for such delays.

6.2 Influence of Time Overhead by OLDA

The time overhead associated with OLDA may cause some sub-jobs to miss their local deadlines. There are two factors that determine the effects of time overhead of OLDA on the performance of the distributed system. The first factor is the density level of a task T_n, i.e., $\frac{C_n}{\mathscr{D}_n}$. A job of a task with a high density level has a higher probability of missing its end-to-end deadline when it is delayed due to the execution of OLDA. The second factor is the ratio of the time overhead over the relative end-to-end deadline of task T_n, i.e., $\frac{Overhead}{\mathscr{D}_n}$. If the relative end-to-end deadline of a job is not large enough to accommodate the time overhead of OLDA, the job will violate its end-to-end deadline. The time overhead is determined by the time complexity of OLDA and the frequency of the call

to OLDA. Since the time complexity of OLDA's derivatives is at least quadratic in the number of sub-jobs and OLDA needs to be run each time a sub-job enters a local processor, our algorithm is suitable for DRTSs where dozens of active sub-jobs are to be executed, e.g., avionics and automotive control applications. We discuss quantitatively the effect of the time overhead of OLDA in Section 7.4.

7 Evaluation

In this section, we analyze the performance and efficiency of our proposed algorithms using generated task sets. We start by evaluating ALDA with a specific sub-job dropping policy for ALDA, and then compare ALDA against WLDA. Note that we do not evaluate OLDA since it is not practical in real settings, as explained in Section 5. To determine how our proposed algorithms fare against existing techniques, we select one derivative with a better performance out of ALDA and WLDA and compare this derivative against JA and BBW for different types of workloads. Notice that ALDA performs better than WLDA for the ST workloads while WLDA performs better than ALDA for the GT workloads. Below, we describe the simulation setup and then discuss simulation results.

7.1 Simulation Setup

The distributed system consists of 8 processors. We use two different types of workloads, the stream-type (ST) workloads and the general-type (GT) workloads to emulate different kinds of application scenarios. For the ST workloads, each task is composed of a chain of subtasks. Such tasks can be found in many signal processing and multimedia applications. In contrast, the GT workloads are more general where (i) a sub-job may have multiple successors and predecessors, and (ii) there is no fixed execution order in the system.

The ST workloads consist of randomly generated task sets in order to evaluate two different processor loading scenarios. For the first set of workloads, the execution time of a job is randomly distributed along its execution path. As a result, processor loads tend to be balanced. As a stress test, the second set of workload represents a somewhat imbalanced workload distribution among the processors. The workloads were generated in such a way that the first few subtasks as well as the last few subtasks are more heavily loaded. This set of workloads was designed to test the usefulness in considering severe resource competition among different jobs on a given processor in meeting end-to-end deadlines. (Note that imbalanced workload scenarios may occur in real life if an originally balanced design experiences processor failures and the original workload must be redistributed.)

Both sets of the ST workloads contain 100 randomly generated task sets of 50 tasks each for 10 different system utilization levels ($400\%, 425\%, \ldots, 625\%$), for a total of 1,000 task sets. Each task is composed of a chain of 4 to 6 subtasks. Each subtask is assigned to a processor such that no two subtasks of the same task run on a common processor. Task periods were randomly generated within the range of from 100,000 to 1000,000 microseconds and the end-to-end deadlines were set to their corresponding periods. We used the UUnifast algorithm [4] to generate the total execution time of each task since UUnifast provides better control on how to assign execution times to subtasks than a random assignment. After the call to the UUnifast algorithm, the set of processors used by task T_i was randomly selected based on the actual number of subtasks M_i for each task T_i and the execution time of each subtask was determined. Each task set was generated with the guarantee that the total utilization at each processor is no larger than 1.

TABLE 5
Selection of Sub-job Dropping Policies for Different Types of Workloads by ALDA and WLDA.

Workload Type	GT	Balanced ST	Imbalanced ST
ALDA	MRET	MRET	MRET
WLDA	MLET	MLET	MLET

Similar to the ST workloads, the GT workloads also contain 1000 task sets, but each set only has between 25 and 100 subtasks. The GT workloads were generated using TGFF [10]. Task periods were generated using uniform distribution and can take any value between $[10,000, 150,000]$ microseconds. The end-to-end deadline of each job was set to be equal to the release time of the job plus the period of the corresponding task. The execution time of a subtask was randomly generated and was within $[1, 10,000]$ microseconds. After the task set was generated, the execution time of each sub-job was uniformly scaled down so that the total utilization of the task set is equal to the desired utilization.

To ensure a fair comparison of the different algorithms under consideration, we made some modifications to JA and BBW. The original versions of JA and BBW require global clock synchronization. We removed this requirement by implementing JA and BBW on-line on each processor. We implemented our proposed algorithms (ALDA and WLDA) as well as two sub-job dropping policies. The first policy (denoted as MLET, for Maximum Local Execution Time) abandons a job with the largest execution time on the processor first. The second policy (denoted as MRET, for Maximum Remaining Execution Time) drops the job with the largest remaining execution time on the processor first. The selection of the sub-job dropping policies for different workloads by ALDA and WLDA is summarized in Table 5. More details on comparing two sub-job dropping policies and selecting one of them are introduced in the supplemental material. All algorithms were implemented in C++. Experimental data were collected on a computer cluster, which is composed of 8 quad-core 2.3 GHz AMD Opteron processors with Red Hat Linux 4.1.2-50. Each task set was simulated for the time interval $[0, 100 \cdot max\_period]$, where *max_period* is the maximum period among the periods of all the tasks in the task set.

We measure the performance of each algorithm with three metrics. The first metric is the *job drop rate*, i.e., the ratio between the number of jobs dropped and the number of jobs released in the system. This metric measures the algorithm's dynamic behavior in a soft real-time system. The second metric is the *number of schedulable task sets*. This metric indicates each algorithm's ability in finding feasible solutions (i.e., static behavior). The third metric is the *running time* of each algorithm (averaged on each processor) to solve a task

TABLE 6
Comparison of ALDA and WLDA in terms of the Three Metrics for Different Types of Workloads.

Metrics		Job Drop Rate (%)	Solved Set Number	Running Time (μs)
GT	ALDA	1.70	844	4549292
	WLDA	1.62	834	86812928
Balanced ST	ALDA	0.0195	861	6604004
	WLDA	0.0203	859	68431633
Imbalanced ST	ALDA	0.0257	745	6924910
	WLDA	0.0279	739	70330281

set. This metric shows the time overhead of each algorithm.

7.2 Comparing OLDA Derivatives

We now discuss the comparison results for our proposed algorithms, ALDA and WLDA for the ST and GT workloads. Since the performance of WLDA depends on some input parameters (e.g., *Max_Allowed_Drop_Num* and α), we set these parameters to optimal values in order to fully exploit the potential of WLDA. (Please refer to the supplemental material for more information.) Table 6 shows the total job drop rates, total solved task set numbers and total running times of solving all the 1000 task sets by ALDA and WLDA for different types of workloads. It is found that WLDA drops 5.21% fewer jobs (up to 9.09%) on average than ALDA for the GT workloads. In contrast, WLDA drops 4.49% and 8.80% more jobs (up to 7.54% and 120%) on average than ALDA for the balanced and imbalanced ST workloads, respectively. Our results show that ALDA can solve 2, 6 and 10, more task sets (out of 1000 task sets) than WLDA for the balanced ST workloads, imbalanced ST workloads and GT workloads, respectively. WLDA requires 11, 11, and 19 times more cycles on average than ALDA for the balanced ST workloads, imbalanced ST workloads and GT workloads, respectively. More details of comparing ALDA and WLDA are provided in the supplemental material.

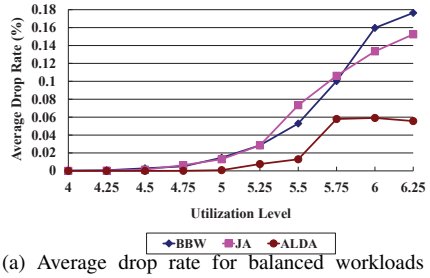
Ideally, WLDA can use its prediction mechanism to reduce the number of dropped jobs. However, WLDA may drop a schedulable job by mistake due to a mis-prediction of future sub-jobs. Moreover, the time overhead caused by the prediction mechanism can greatly degrade the performance of WLDA. This time overhead is caused by several maintenance operations (such as the update of the timing information of the future sub-jobs already considered in the previous assignment, the addition and removal of some new and obsolete future sub-jobs, respectively, etc.) in the prediction mechanism. The time overhead caused by the prediction mechanism in WLDA makes some jobs not only miss their assigned local deadlines but also violate their local deadline upper bounds. In summary, our results indicate that ALDA performs better than WLDA for the ST workloads while WLDA outperforms ALDA for the GT workloads. The higher time overhead of WLDA makes it unsuitable for ST workloads, as a sub-job $J_{i,k}$ can delay the execution of all its successors $J_{i,k'}$'s since a job is composed of a chain of sub-jobs. In contrast, for the GT workloads, the time overhead incurred due to scheduling will most likely not delay the execution of all the other active sub-jobs $J_{i,k'}$'s in J_i since some active sub-jobs in J_i are not successors of $J_{i,k}$. Therefore, we focus on ALDA for the ST workloads and WLDA for the GT workloads in the discussion below.

7.3 Performance of OLDA against Other Algorithms

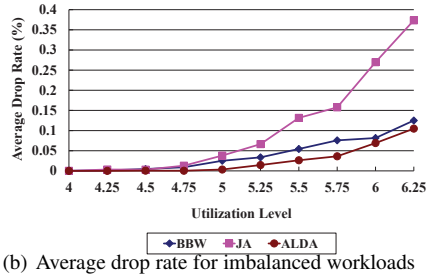
We compare the performance of OLDA with JA and BBW, the two representative priority assignment methods. We use ALDA and WLDA to test the performance of OLDA in the ST and GT workloads, respectively. More details on the sub-job dropping policy is provided in the supplemental material. In the first experiment, we compare the average job drop rates of infeasible task sets when using different algorithms for balanced ST workloads, imbalanced ST workloads and GT workloads. A job is dropped either because no local-deadline assignment can be found for the sub-job set on a processor using OLDA or the job's end-to-end deadline is missed using BBW and JA. The job drop rates for the three algorithms for the balanced ST workloads, imbalanced ST workloads, and GT workloads are shown in Figures 4(a), 4(b), and 4(c), respectively. It is clear that OLDA drops much fewer jobs than the other two methods. Specifically, for balanced ST workloads, BBW and JA drop 179% and 165% more jobs on average than OLDA, respectively. For imbalanced ST workloads, the averages are 61% and 313%, respectively. For GT workloads, 160% and 51% more jobs are dropped by BBW and JA than those by OLDA on average, respectively.

In the second experiment, we compare the percentage of feasible task sets (over the 100 task sets at each utilization level) found by our algorithm, with those found by JA and BBW for the three sets of the workloads. The results are summarized in Figures 4(d), 4(e) and 4(f), respectively. The data shows that OLDA finds far more feasible sets than the other two methods. Specifically, for balanced ST workloads, using OLDA leads to 71% and 22% on average (and up to 2,250% and 124%) more feasible task sets than using BBW and JA, respectively. For imbalanced ST workloads, using OLDA results in 60% and 48% on average (and up to 338% and 2300%) more feasible task sets than BBW and JA, respectively. For GT workloads, the number of solutions found by OLDA is 13% and 12% on average (and up to 100% and 200%) more than that found by BBW and JA. Observe that OLDA performs much better than existing techniques at high utilization levels where there are more jobs in the system. We would also like to point out that sometimes OLDA may not be able to find a feasible solution even though such solutions indeed exist, since OLDA finds local sub-job deadlines for each processor independently instead of using a global approach. For balanced ST workloads, OLDA can find on average 98.81% and 99.86% of those found by BBW and JA, respectively. For imbalanced ST workloads, OLDA can find on average 95.27% and 99.40% of the feasible task sets found by BBW and JA, respectively. For GT workloads, OLDA can find on average 99.86% and 99.60% of the feasible task sets found by BBW and JA, respectively. These results demonstrate that OLDA not only finds more feasible task sets than BBW and JA, but also solves most of the problems that BBW and JA can solve.

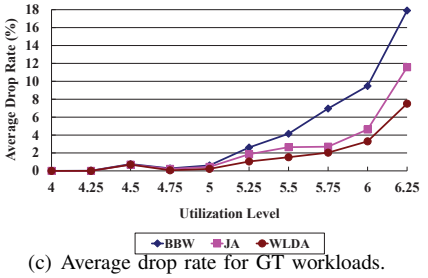
To see how well OLDA fares compared to an MILP solver, we randomly selected 3 workloads containing 4, 20 and 26



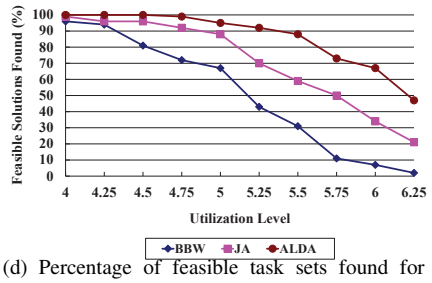
(a) Average drop rate for balanced workloads (ST workloads).



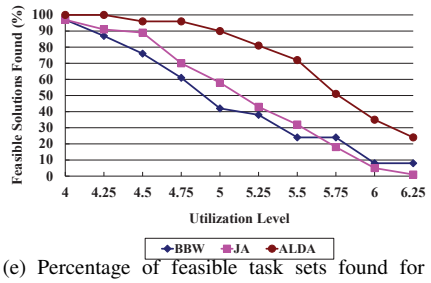
(b) Average drop rate for imbalanced workloads (ST workloads).



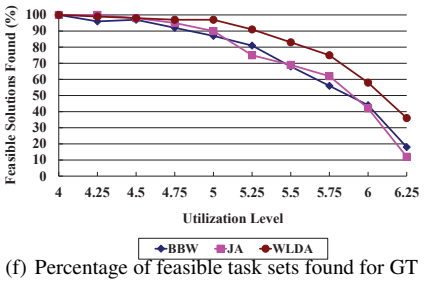
(c) Average drop rate for GT workloads.



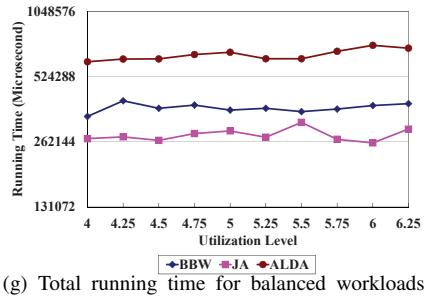
(d) Percentage of feasible task sets found for balanced workloads (ST workloads).



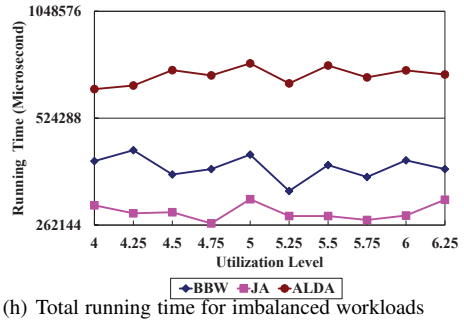
(e) Percentage of feasible task sets found for imbalanced workloads (ST workloads).



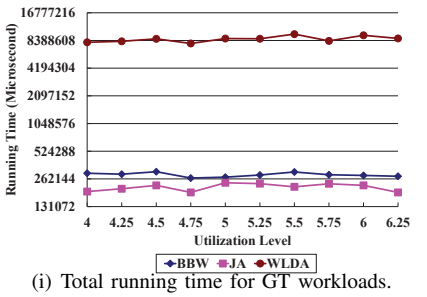
(f) Percentage of feasible task sets found for GT workloads.



(g) Total running time for balanced workloads (ST workloads).



(h) Total running time for imbalanced workloads (ST workloads).



(i) Total running time for GT workloads.

Fig. 4. Comparison of OLDA, JA and BBW.

tasks, respectively, and compared the solutions obtained by OLDA and lp_solve [1], an MILP solver. For the workload with 4 tasks, both OLDA and lp_solve find the same solution. For the workload with 20 tasks, OLDA and lp_solve find two different feasible solutions, however, the objective function values by the two solutions are the same. For the workload with 26 tasks, OLDA is able to solve the problem containing 10 sub-jobs within 6 *ns* while lp_solve fails to find a solution after running for 48 hours. The comparisons support our earlier claim that OLDA always finds an optimal solution to the problem stated in (5), (6) and (7) whenever a feasible solution exists. Furthermore, the execution time of OLDA is more suitable for on-line use than that of lp_solve.

7.4 Time Overhead of OLDA

In our evaluations, we consider the time overhead due to OLDA when simulating a task set. That is, every time OLDA is called by a local processor, our simulator records the running time of OLDA and postpones the execution of all the active local sub-jobs for that time duration to simulate the influence of the time overhead due to OLDA. To examine whether OLDA is suitable for on-line local-deadline assignments, we show the total running time overheads of OLDA, BBW and JA for the balanced ST workloads, imbalanced ST workloads and GT workloads in Figures 4(g), 4(h) and 4(i), respectively. We still use ALDA and WLDA to test the performance of OLDA in the ST and GT workloads, respectively. Based on the results, we compare the number of cycles required by OLDA against those of JA and BBW. For the balanced ST workloads, OLDA requires on average 1.76 and 2.34 times more cycles per task set (with 50 tasks) than BBW and JA, respectively. For the imbalanced ST workloads, OLDA needs about 1.81 and 2.43 times more cycles per task set than BBW and JA, respectively. For the GT workloads, OLDA requires on average 30 and 41 times more cycles per task set than BBW and JA. Although OLDA has a longer running time than both BBW and JA, the average numbers of cycles required to run OLDA for once are about 292, 295 and 4507 cycles for the balanced ST workloads, imbalanced ST workloads and GT workloads, respectively, while the average number of the sub-jobs handled by an activation of OLDA is 3, 3 and 8 for the balanced ST workloads, imbalanced ST workloads and GT workloads, respectively. Such runtime overhead is tolerable in DRTSs executing computationally demanding real-time jobs, e.g. in avionics and automotive control applications [15], [25], where dozens of active sub-jobs are to be executed.

8 Summary and Future Work

This paper presented a novel distributed local-deadline assignment approach to guarantee job end-to-end deadlines in a DRTS. Our algorithms have the following features: (i) they are guaranteed to find a feasible deadline assignment on

each processor locally if one exists, (ii) the local-deadline assignment solution always minimizes the maximum slack among all the local sub-jobs on the processor, and (iii) they do not require global synchronization. Our algorithms have been shown to be very effective and general in that they support general tasks, each of which can be represented as a DAG and partitioned to processors by any arbitrary method. Furthermore, our algorithms are efficient enough for on-line use, and thus can quickly adapt to dynamic changes in the system. In order to further validate the advantages of our algorithms, we plan to implement them in a real-time operating system and apply them to some real-world applications. Our algorithms can be improved by employing an appropriate bus-based network platform, incorporating the communication delays and applying different QoS criteria.

REFERENCES

[1] http://lpsolve.sourceforge.net/5.5/.

[2] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Syst.*, vol. 2, no. 4, pp. 301–324, Oct. 1990.

[3] R. Bettati and J. W.-S. Liu, "End-to-end scheduling to meet deadlines in distributed systems," in *Proceedings of the 12th International Conference on Distributed Computing Systems,*, June 1992, pp. 452–459.

[4] E. Bini and G. C. Buttazzo, "Biasing effects in schedulability measures," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, Jul. 2004, pp. 196–203.

[5] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning parallel applications on multiprocessor reservations," in *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, Jul 2010, pp. 24–33.

[6] G. C. Buttazzo, "Hard real-time computing systems: Predictable scheduling algorithms and applications." Springer, 2005.

[7] S. Cavalieri, "Meeting real-time constraints in CAN," *IEEE Transactions on Industrial Informatics*, vol. 1, no. 2, pp. 124–135, May 2005.

[8] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems*, vol. 2, no. 3, pp. 181–194, Sep. 1990.

[9] H. Chetto and M. Chetto, "Scheduling periodic and sporadic tasks in a real-time system," *Information Processing Letters*, vol. 30, no. 4, pp. 177–184, Feb. 1989.

[10] 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*, Mar. 1998, pp. 97–101.

[11] S. Gopalakrishnan, L. Sha, and M. Caccamo, "Hard real-time communication in bus-based networks," in *Proceedings. of 25th IEEE International Real-Time Systems Symposium*, Dec. 2004, pp. 405–414.

[12] A. Hagiescu, U. D. Bordoloi, S. Chakraborty, P. Sampath, P. V. V. Ganesan, and S. Ramesh, "Performance analysis of FlexRay-based ECU networks," in *Proceedings of the 44th annual Design Automation Conference*, June 2007, pp. 284–289.

[13] S. Hong, T. Chantem, and X. S. Hu, "Meeting end-to-end deadlines through distributed local deadline assignments," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, Dec. 2011, pp. 183–192.

[14] S. Hua, G. Qu, and S. S. Bhattacharyya, "Probabilistic design of multimedia embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, Jul. 2007.

[15] P. Jayachandran and T. Abdelzaher, "Transforming distributed acyclic systems into equivalent uniprocessors under preemptive and non-preemptive scheduling," in *Proceedings of the 20nd Euromicro Conference on Real-Time Systems*, Jul. 2008, pp. 233–242.

[16] ——, "Delay composition in preemptive and non-preemptive real-time pipelines," *Real-Time Systems*, vol. 40, no. 3, pp. 290–320, Dec. 2008.

[17] J. Jonsson and K. G. Shin, "Robust adaptive metrics for deadline assignment in distributed hard real-time systems," *Real-Time Systems*, vol. 23, no. 3, pp. 239–271, Nov. 2002.

[18] H. Kopetz, "Real-time systems design principles for distributed embedded applications." Springer, 2011.

[19] J. Lee, I. Shin, and A. Easwaran, "Convex optimization framework for intermediate deadline assignment in soft and hard real-time distributed systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2331–2339, 2012.

[20] C. Lin, T. Kaldewey, A. Povzner, and S. A. Brandt, "Diverse soft real-time processing in an integrated system," in *27th IEEE International Real-Time Systems Symposium, 2006.*, Dec. 2006, pp. 369–378.

[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[22] J. W. Liu, "Real-time systems." Prentice Hall, 2000.

[23] S. Manolache, P. Eles, and Z. Peng, "Optimization of soft real-time systems with deadline miss ratio constraints," in *10th IEEE Real-Time and Embedded Technology and Applications Symposium.*, May 2004.

[24] D. Marinca, P. Minet, and L. George, "Analysis of deadline assignment methods in distributed real-time systems," *Computer Communications*, vol. 27, no. 15, pp. 1412–1423, Jun. 2004.

[25] S. Samii, P. Eles, Z. Peng, and A. Cervin, "Quality-driven synthesis of embedded multi-mode control systems," in *Proceedings of the 46th Annual Design Automation Conference*, July 2009, pp. 864–869.

[26] N. Serreli, G. Lipari, and E. Bini, "The distributed deadline synchronization protocol for real-time systems scheduled by EDF," in *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2010, pp. 1–8.

[27] ——, "Deadline assignment for component-based analysis of real-time transactions," in *2nd Workshop on Compositional Real-Time Systems*, Dec. 2009.

[28] Y. Zhang and R. West, "End-to-end window-constrained scheduling for real-time communication," in *IEEE 10th International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug. 2004, pp. 143–152.

[29] W. Zheng, Q. Zhu, M. Di Natale, and A. Vincentelli, "Definition of task allocation and priority assignment in hard real-time distributed systems," in *28th IEEE International Real-Time Systems Symposium*, Dec. 2007.

Shengyan Hong is a Ph.D. candidate in University of Notre Dame. He received his Master's degrees from the Pennsylvania State University and Fudan University in 2008 and 2006, respectively. He received his Bachelor's degree from Fudan University in 2003. His research interests include Real-Time Scheduling in Cyber-Physical Systems and Computer Architecture.

Thidapat Chantem (S'05-M'11) received her Ph.D. and Master's degrees from the University of Notre Dame in 2011 and her Bachelor's degrees from Iowa State University in 2005. She is an Assistant Professor of Electrical and Computer Engineering at Utah State University. Her research interests include Intelligent Transportation Systems, Cyber-Physical System Design, Hardware-Software Co-design of Embedded Real-Time Systems, and Energy-Aware and Thermal-Aware System-Level Design.

Xiaobo Sharon Hu (S'85-M'89-SM'02) received her B.S. degree from Tianjin University, China, M.S. from Polytechnic Institute of New York, and Ph.D. from Purdue University, West Lafayette, Indiana. She is Professor in the department of Computer Science and Engineering at University of Notre Dame. She has published more than 200 papers in the related areas. She served as Associate Editor for IEEE Transactions on VLSI, ACM Transactions on Design Automation of Electronic Systems, and ACM Transactions on Embedded Computing.