Scheduling Periodic Task Graphs for Safety-Critical Time-Triggered Avionic Systems

Menglan Hu, Jun Luo, *Member, IEEE*, Yang Wang, and Bharadwaj Veeravalli, *Senior Member, IEEE*

***Abstract*—Time-triggered communication protocols, such as Time-Triggered Protocol (TTP) and FlexRay, have potential to solve many system integration and concurrent engineering issues in the aerospace industry. This paper investigates the scheduling of periodic applications on time-triggered systems. A novel scheduling problem is formulated to capture a unique feature commonly existing in the safety-critical time-triggered systems, i.e., in task graphs running in such systems, some nodes (i.e., tasks and messages) are strictly periodic while others are not. To address the problem, a novel scheduling algorithm called *Synchronized Highest Level First* (SHLF) algorithm is presented. Moreover, to further improve schedulability, this paper also proposes two rescheduling and backtracking approaches, namely *Release Time Deferment* (RTD) procedure and *Backtracking and Priority Promotion* (BPP) procedure. Performance evaluation results are presented to demonstrate the effectiveness and competitiveness of our approaches when compared to existing algorithms.**

***Index Terms*—Avionic systems, time-triggered systems, real-time scheduling, list scheduling, end-to-end-delays, task graphs.**

I. Introduction

Time-triggered communication protocols, such as Time-Triggered Protocol (TTP) [9], Time-triggered CAN (TTCAN) [4] and FlexRay [5], have the capabilities to solve many system design and integration problems in the avionics industry. They have considerably influenced the design of a variety of modular system architectures of modern aerospace vehicles. For example, TTP has been applied to critical modular systems for distributed power generation, engine and flight controls in aircrafts including Airbus A380, Boeing 787, Bombardier CSeries, Embraer Legacy, fighter jets and other regional and business airplanes [1].

Due to the wide deployment of the time-triggered protocols (i.e., TTP, TTCAN, and FlexRay), the scheduling of *applications* on time-triggered systems becomes a critical issue for offering quality-of-service (QoS) guarantees to safety-critical systems, here, an application is referred to a set of tasks with timing constraints to perform a well-defined function in the system. Each task in the application running in a specified electronic control unit (ECU) is triggered by one or more messages from other tasks and sends messages to its downstream tasks. Consequently, the application can be abstracted as a *directed acyclic graph* (aka task graph) where nodes represent either the tasks or the messages and edges specify the precedence constraints among the tasks. For instance, TTP-based modular aerospace control (MAC), which is a part of the F110 full authority digital engine control (FADEC) system of General Electric, is integrated into the Lockheed Martin F-16 fighter aircraft. The FADEC is a system consisting of a collection of ECUs, and its related accessories control all aspects of the aircraft engine performance. The FADEC works by receiving multiple input variables of the current flight condition, including air density, throttle lever position, engine temperatures, engine pressures, and many other parameters. The inputs are periodically received by the ECUs and then analyzed up to 70 times per second. Accordingly, an applications in the FADEC can be abstracted as a periodic task graph.

Unlike traditional embedded systems, the complexity of the avionic systems is dramatically increased to fulfill diverse functionalities in a time-bounded manner, rendering the scheduling of both tasks and messages at the same time to be commonplace. However, many existing research efforts only concentrate on the message scheduling in the communication channels between ECUs, lacking the notion of its task counterpart [10], [11], [27], [12], [13]. Such isolated message-only scheduling may severely compromise the overall performance and feasibility of the application. A handful of studies have investigated the holistic scheduling problem of both tasks and messages on time-triggered systems [18], [15], [16], [17], [31]. However, most of them suffer from the limited scalability as their solutions are usually relied on mathematical programming techniques, which cannot be scaled to a large-scale system. To deal with the sharply increasing amount of software functionality in the avionic systems, it is desirable to design efficient and scalable heuristic approaches. Of course, many heuristics have been proposed in literature to deal with the scheduling of task graphs in multiprocessors [34], [33], [32], [20], [25], [24], [23]. Nevertheless, these algorithms were not developed for the safety-critical time-triggered systems. Further, most of these studies even neglect the contention among the underlying communication resources.

Further, in most previous studies [21], [22], [19], [25], [26], a task graph is only periodic in terms of its release time and deadline while the nodes in the task graph are not necessary to be strictly periodic. That is, the comparative

Menglan Hu is with the School of Electronic Information and Communications, Huazhong University of Science and Technology, Wuhan, China 430074. E-mail: humenglan@gmail.com.

Jun Luo is with the School of Computer Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore 639798. E-mail: junluo@ntu.edu.sg.

Yang Wang is with the Faculty of Computer Science, University of New Brunswick, Fredetricton, Canada, E3B 5A3. E-mail: ywang8@unb.ca.

Bharadwaj Veeravalli is with the Department of Electrical and Computer Engineering, National University of Singapore, 4 Engineering Drive 3, Singapore 117576. E-mail: elebv@nus.edu.sg.

start time (to the release time of the task graph) of different invocations of a task can be different provided that their deadlines and precedence constrains can be met. However, such an assumption is untenable for safety/time-critical applications (i.e., task graphs) in time-triggered systems. For such an application, besides release time and deadline, the start time of different invocations of the same time-critical task (e.g., data collection task) need also be periodic. In this case, some prior studies on the time-triggered systems which used mathematical programming techniques simply assumed that all nodes in a task graph are strictly periodic. Nevertheless, since in fact other nodes (e.g., messages and data processing nodes) in the application are unnecessarily periodic, this oversimplified assumption may impose excessive constraints and undermines the schedulability of the whole system. Therefore, a scheduling algorithm that is aware of the periodicity of specific nodes should correctly address both periodic and aperiodic nodes for periodic time-triggered applications.

Motivated by these needs, this paper investigates the problem of scheduling periodic task graphs consisting both periodic and aperiodic tasks and messages for the safety-critical time-triggered systems. To the best of our knowledge, this study is the first attempt to consider this problem. The contributions are multi-fold. Firstly, this study proposes a novel scheduling algorithm, referred to as *Synchronized Highest level First* (SHLF) algorithm. The SHLF algorithm simultaneously assigns all instances of each periodic node to guarantee the periodicity of the periodic nodes. Meanwhile, it separately schedules each instance of an aperiodic node to utilize the flexibility of those aperiodic nodes. Also, the contention on communication resources in the time-triggered systems is explicitly addressed.

Additionally, to further improve the schedulability, this paper also proposes two rescheduling and backtracking approaches, called *Release Time Deferment* (RTD) method and *Backtracking and Priority Promotion* (BPP) method, respectively. The essence of the RTD algorithms is to defer the release time of conflicted task graphs for conflict resolution, which overcomes the downside of many prior studies (e.g., [21], [22], [25]), where the release time of task graphs is simply fixed as zero, leading to poor performance (i.e., schedulability) under complex timing constraints. Once the conflicts cannot be eliminated by deferring the release time, BPP backtracks a number of previously scheduled applications to create space for the conflicted applications and reschedules them with promoted priorities.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 explains the motivation of the paper via an example. Section 4 introduces mathematical models, assumptions, and problem formulation. Section 5 describes the proposed algorithms in great detail. Section 6 presents simulation results to evaluate the algorithm, with conclusions following in Section 7.

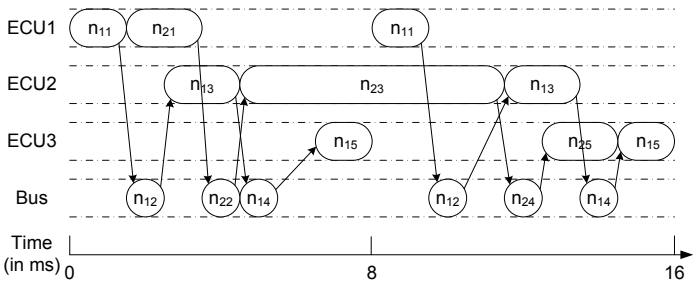
II. Related Work

The scheduling problem in time-triggered systems has received considerable attention in recent years. Park et al. [14] designed a FlexRay network parameter optimization approach that can derive the durations of the static slot and the communication cycle. Another two papers [11] and [10] studied the message scheduling problem via nonlinear integer programming (NIP) techniques for the static and dynamic segments, respectively. Specifically, [11] utilized a frame packing technique that packs multiple frames into one message, while [10] reserved time slots for aperiodic messages to assure QoS guarantees and flexible scheduling of the dynamic segment. Martin et al. [12] transformed the message scheduling problem on the static segment into a bin packing problem and then utilized ILP to address it. Nevertheless, these studies only concern on message scheduling and neglected its task counterpart. Such isolated message-only scheduling may seriously compromise the overall performance and feasibility of the applications which comprise both tasks and messages. In contrast, this study holistically schedules both tasks and messages on time-triggered avionic systems.

A number of studies on the holistic scheduling of both tasks and messages in time-triggered systems have also been reported in the literature [18], [15], [16], [17]. Pop et al. [18] employed constraint logic programming (CLP) to deal with the scheduling and voltage scaling issue of low-power fault-tolerant hard real-time applications. Davare et al. [16] applied geometric programming (GP) for scheduling tasks and messages for distributed vehicle systems. Another work [17] proposed scheduling analysis methods for hybrid event-triggered and time-triggered systems. All of these studies relied on mathematical programming techniques and the high computation complexity of solving mathematical programming thus limits the applicability and scalability of their methods.

The scheduling of task graphs in multiprocessor systems has received much attention in past decades [36], [37], [38], [40], [39]. One category of classic algorithm that has been widely applied is *list scheduling*. A list scheduling algorithm typically maintains a list of tasks according to their assigned priorities. It has two phases: the task selection phase for selecting the highest-priority ready task and the processor selection phase for selecting a suitable processor that minimizes a predefined cost function. Some examples include the Modified Critical Path (MCP) [34], Dynamic Level Scheduling (DLS) [33], Highest Level First (HLF) [32], and Heterogeneous Earliest-Finish-Time (HEFT) [20] algorithms. Since list scheduling heuristics are generally more practical and provide better performance at a lower scheduling time than the other categories, our paper presents algorithms based on list scheduling techniques. Specifically, our proposed SHLF algorithm attempts to use highest level first policy for node selection, which is similar to some of the known algorithms in the literature [32], [20]. Nevertheless, the design of node assignment in SHLF, as well as RTD and BPP methods, are the novel contributions of this paper.

Another type of heuristic is clustering [25], [24], [23]. In this category, tasks are pre-clustered before allocation begins to reduce the size of the problem. Task clusters (instead of individual tasks) are then assigned to individual processors. Dave et al. [25] presented a clustering-based co-synthesis algorithm, which schedules periodic task graphs for hardware



Node	n_{11}	n_{12}	n_{13}	n_{14}	n_{15}	n_{21}	n_{22}	n_{23}	n_{24}	n_{25}
Host	ECU1	Bus	ECU2	Bus	ECU3	ECU1	Bus	ECU2	Bus	ECU3
Size	1.5	1	2	1	1.5	2	1	7	1	2

Fig. 1. Motivating example.

and software co-synthesis on heterogeneous distributed embedded systems. [24] proposed a period-based approach to the problem of workload partitioning and assignment for large distributed real-time systems. [23] designed a static algorithm for allocating and scheduling components of periodic tasks across sites in distributed systems. In the problem discussed in this paper each task must be processed in a specific ECU; therefore, clustering may be useless since the tasks have been naturally clustered by their functionality. Accordingly, when handling the problem discussed in this paper, these clustering approaches may lose their benefits.

Prior researchers have also proposed optimal approaches for some class of real-time task graph scheduling problems. Peng et al. [21] designed an optimal Branch and Bound (B&B) approach for scheduling periodic task graphs to heterogeneous distributed real-time systems. Another work [22] proposed an optimal B&B algorithm for allocating tasks on multiprocessors subject to precedence and exclusion constraints. Nevertheless, these optimal solutions are applicable to only small task graphs consisting of tens or so tasks.

Energy-efficient requirements have also been considered in real-time task graph scheduling problems [19], [18], [28]. [19] presented a power-conscious algorithm for jointly scheduling multi-rate periodic task graphs and aperiodic tasks in distributed real-time embedded systems. [28] proposed static and dynamic power management schemes that save energy by exploring the idle periods of processors.

III. Motivating Example

Consider two task graphs g_1 and g_2. The period of g_1 is 8ms and the period of g_2 is 16ms. Accordingly in one hyper period two instances of g_1 and one instance of g_2 are scheduled together. g_1 has five nodes $\{n_{11}, n_{12}, ..., n_{15}\}$ and g_2 also has five nodes $\{n_{21}, n_{22}, ..., n_{25}\}$. These nodes are running in 3 ECUs (ECU1, ECU2, and ECU3) and a bus. The table in Fig. 1 shows the details of all nodes. As ECU1 collects periodic data, tasks running on it (e.g., n_{11} and n_{21}) are periodic. For example, to guarantee the correctness of data collection, the intervals between the start times of two consecutive instances of n_{11} must be 8ms. On the other hand, as other ECUs and the bus only process and transmit data, other nodes running on the ECUs and the bus are not periodic, i.e., there is no constraint regarding the intervals between the start times of two consecutive invocations of such a node.

In previous studies on multiprocessor scheduling, [21], [22], [19], [25], [26], there is no constraint on whether an individual node is periodic and they commonly assume that a task graph is only periodic in terms of release time and deadlines. Accordingly, their models do not capture the timing requirements of periodic nodes such as n_{11} and n_{21} and may cause erroneous scheduled on safety-critical time-triggered systems. To guarantee the correctness of periodic nodes (i.e., n_{11} and n_{21}), some prior papers (e.g., [15], [16]) simply assumed that all nodes are periodic and they formulated mathematical programming to solve their scheduling problems. Based on this oversimplified assumption, the intervals between the start times of two consecutive instances of n_{13} is 8ms and thus n_{23} cannot be scheduled in a hyper period since the size of n_{13} is 2ms and the size of n_{23} is 7ms. Nevertheless, this example problem actually has a feasible schedule, which is shown in Fig. 1. In the schedule, two instances of n_{13} start at 2.5ms and 12.5ms, respectively, and thus n_{23} can be scheduled at 4.5ms. Therefore, it is desirable to design an algorithm that is aware of the periodicity of specific nodes and can correctly schedule both periodic and aperiodic nodes for periodic time-triggered applications.

IV. Problem Formulation

We consider a FlexRay cluster consisting of a set of ECUs (i.e., hosts) connected via the FlexRay bus. Fig. 2 shows an example of such a system on an airplane where the dark boxes on the plane represent ECUs. The ECUs can process specific tasks that exchange data via messages transmitted on the bus. The FlexRay protocol [5] is a time-triggered protocol. Its operation is based on a repeatedly executed FlexRay cycle with a fixed duration. Fig. 3 shows the timing hierarchy of 64 FlexRay cycles. A FlexRay cycle comprises a static segment (SS), a dynamic segment (DS), a symbol window (SW), and the network idle time (NIT). This paper studies the scheduling of periodic applications, which only utilizes the static segment. The organization of the static segment is based on a time-division multiple access (TDMA) scheme. It consists of a fixed number of equal size static slots (i.e., communication slots). Each static slot in each channel of each cycle can only be uniquely assigned to one ECU to transfer one frame (i.e., message), but each static slot can be assigned to different ECUs in different channels or cycles. Fig. 3 illustrates the time hierarchy of the static segment. The lengths of static slot T_s, static segment T_{ss}, and the communication cycle T_c are assumed to be known to the scheduler beforehand as previous papers have thoroughly studied how to choose proper values for these parameters [14], [11].

This paper assumes a periodic real-time task model in which $G = \{g_1, g_2, ..., g_J\}$ is a set of J applications to be processed in the cluster. Let $p(g_j)$ be the period of application $g_j \in G$ and P be the least common multiple of all $p(g_j)$s. The interval $[0, P)$ is called the hyper period. In one hyper period an application invokes $K(g_j) = \frac{P}{p(g_j)}$ times. Also, one hyper

Fig. 2. A cluster of ECUs on a plane.

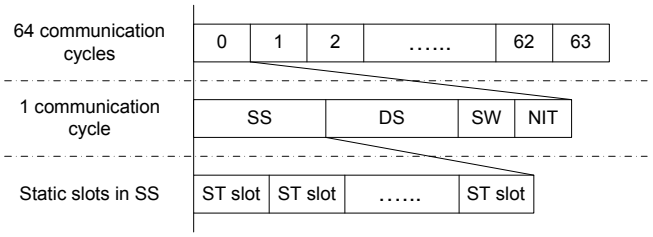


Fig. 3. FlexRay timing hierarchy.

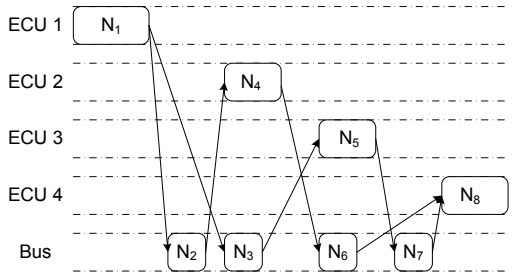


Fig. 4. An example of task graph running on ECUs.

$CPL_{(g_i)}$	the critical path length of g_j
$d(g_j)$	deadline of g_j
$EST(n_i)$	the earliest start time of n_i
g_j	j-th task graph
g_j^k	k-th instance of j-th task graph
$K(g_j)$	the number of instances of g_j
$LST^{max}(n_i)$	the maximum latest start time of n_i
n_i	i-th node
n_i^k	k-th instance of i-th node
$o(g_j)$	the offset of g_j
P	duration of a hyper period
$p(g_j)$	period of g_j
$priority(g_j)$	priority of g_j
$rank(g_j)$	the rank of g_j
$rank_u(n_i)$	the upward rank of n_i
$ST(n_i)$	the start time of n_i
$w(n_i)$	time cost of n_i

TABLE I
NOTATIONS AND TERMINOLOGY

period spans multiple communication cycles. It suffices to analyze the behavior of the whole system only in one hyper period, since it will repeat for all hyper periods [21].

As shown in Fig. 3, an application can be modeled by a directed acyclic graph (DAG) comprising multiple nodes (i.e., vertexes), which are the smallest units to be scheduled, and edges, which specify precedence constraints. *Task graph*, *DAG* and *application* terms are interchangeably used in this paper. A node can be either a task (i.e., computation module) running on a particular ECU (e.g., a sensor, or an actuator) or a message (i.e., communication module) exchanged on the communication bus. Associated with each node n_i is its time cost, denoted by $w(n_i)$ which indicates the execution time on a ECU if the node is task, or the message transmission time on the bus if the node is a message. In addition, since each ECU is different and has its specific functions, the hosted ECU of each task is known beforehand and processor selection appearing in conventional work is not needed. The host of node n_i is specified as $H(n_i)$. Further, messages nodes should be fit into communication slots on the bus. The transmission of a message cannot span two or more communication slots.

In a DAG, an edge e_{ij} linking two nodes n_j and n_i represents the precedence constraint between the nodes, i.e., n_i should complete its execution before n_j starts. The edges have no time cost. The source node n_i and the destination node n_j of the edge e_{ij} are named as the *parent* node and the *child* node, respectively. A node which has no parent is called an entry node while a node which has no child is called an exit node. Again, once n_i is scheduled onto $H(n_i)$, the node scheduled immediately before n_i on $H(n_i)$ is denoted as $prev(n_i)$ and the node scheduled immediately after n_i on $H(n_i)$ is denoted as $next(n_i)$. Further, a node n_p that must finish computation before the start of another node n_q is called an *ancestor* of n_q and n_q is called an *offspring* of n_p. Accordingly, the ancestors of n_i's parents and $prev(n_i)$ are n_i's ancestors. Similarly, the offspring of n_i's children as well as $next(n_i)$ are also n_i's offspring.

An application g_j has an offset $o(g_j)$ indicating the start time of each instance of g_j in one hyper period, i.e., the start time of the k-th invocation of g_j is $o(g_j)+k*p(g_j)$ where $k = 0, 1, ..., K(g_j) - 1$. The offset of each application is initially set as zero and will be decided by the scheduling algorithm. Also, each application g_j may have a relative deadline $d(g_j)$. The k-th instance of g_j cannot finish after a time $d(g_j) + o(g_j) + k * p(g_j)$. Notice that possibly $d(g_j) + o(g_j) + k * p(g_j) > P$, i.e., the execution of an application is allowed to span multiple hyper periods. Moreover, each application g_j is scheduled $K(g_j)$ times in a hyper period. Thus each node n_i has $K(g_j)$ invocations in a hyper period. Let n_i^k denote k-th invocation of n_i. Let the start time of n_i^k be $ST(n_i^k)$. If n_i is periodic, $ST(n_i^k) = ST(n_i^0) + k * p(g_j)$. In other words, the interval between the start times of two consecutive instances of n_i is $p(g_j)$ where n_i belongs to task graph g_j. This constraint should be respected by the scheduler. Otherwise, n_i is aperiodic and there is no constraint regarding the interval between the start times of two consecutive instances of n_i.

The objective is to schedule all nodes in all instances (i.e., to allocate $ST(n_i^k)$ for all nodes) of all applications in one hyper period to guarantee that all instances of all applications can meet their respective deadlines. Finally, notations and terminology are listed in Table I.

V. THE PROPOSED ALGORITHM

The algorithm first orders the graphs and picks each graph for scheduling. Then the algorithm schedules all nodes in

each selected task graph. If the algorithm fails to schedule a node in a task graph g_j, $o(g_j)$ is modified to a proper value and all nodes in the task graph are rescheduled. In case that rescheduling cannot help, the algorithm backtracks multiple previously scheduled graphs to create space for the failed graph.

A. Graph Selection

This procedure calculates the priorities of the graphs and ordering them according to their priorities. The rank of a graph g_j is defined as:

$$rank(g_j) = \frac{2CPL(g_j)}{p(g_j) + d(g_j)} \tag{1}$$

where $CPL(g_j)$ is the critical path length of the graph. The critical path is defined as a set of nodes and edges, forming a path from an entry node to an exit node, of which the sum of computation and communication costs is the maximum. To obtain $CPL(G_j)$, one can recursively calculate the upward rank of each node in the graph via:

$$rank_u(n_i) = w(n_i) + \max_{s \in child(n_i)} \{rank_u(n_s)\} \tag{2}$$

where $succ(n_i)$ is the set of n_i's children. Basically, $rank_u(n_i)$ is the longest distance from n_i to the exit node and $CPL(g_j)$ is the largest $rank_u(n_i)$ of all nodes in the graph. In other words, $CPL(g_j)$ is equal to the rank of the entry node. The idea to select $rank(g_j)$ as the priority is that a longer critical path length implies that the graph is likely to be more difficult to schedule within limited room while a longer period and a longer end-to-end deadline usually indicate larger space for the nodes in the application to be flexibly scheduled. The graph list is generated by sorting the graphs by the descending order of their ranks. The scheduler then selects the application with the largest rank for scheduling (line 6 of Algorithm 3).

B. Synchronized Highest Level First (SHLF) Algorithm

This subsection details the SHLF algorithm which schedules all nodes of the selected task graph g_j. A priority queue is maintained to contain all ready nodes at any given instant. A node becomes ready for scheduling if none of its parents is unscheduled. At each step, a node n_i with the largest $rank_u(n_i)$ in the queue is selected from the queue. The node is then scheduled by procedure *schedule(n_i)*(lines 9-10 of Algorithm 3).

Algorithm 2 elaborates *schedule(n_i)*, which deals with the node in two cases: the node is periodic, and the node is aperiodic. To achieve this, the procedure needs to call another procedure *findStartTime(t, n_i^k)* (shown in Algorithm 1) that finds a feasible start time after a given time instant t for node n_i. If the node n_i is periodic (lines 2-22 of Algorithm 2), this procedure searches for $K(g_j)$ periodic time slots on $H(n_i)$ to concurrently accommodate $K(g_j)$ instances of the node. Since the scheduling of nodes should preserve precedence constraints of the task graph, the search of an appropriate idle time slot for n_i^k on $H(n_i^k)$ starts from the earliest start time of n_i^k (denoted as $EST(n_i^k)$), i.e., the time when all n_i^k's

Algorithm 1 findStartTime(t, n_i^k)

```
1: round ← 0
2: ST(n_i^k) ← t
3: SST ← 0
4: if L(H(n_i^k)) is not empty then
5:   if ST(n_{M-1}) + w(n_{M-1}) − P > 0 then
6:     SST ← ST(n_{M-1}) + w(n_{M-1}) − P
7:   end if
8:   outerLoop:
9:   while ST(n_i^k) < LST^max(n_i^k) do
10:    m ← 0
11:    while m ≤ M do
12:      if m < M then
13:        SET ← ST(n_m) + round * P
14:      else
15:        SET ← ST(n_0) + (round + 1) * P
16:      end if
17:      if ST(n_i^k) ≥ SST and ST(n_i^k) + w(n_i) ≤ SET
         then
18:        break outerLoop
19:      end if
20:      if m < M then
21:        SST ← ST(n_m) + w(n_m) + round * P
22:      else
23:        SST ← ST(n_0) + w(n_0) + (round + 1) * P
24:      end if
25:      if ST(n_i^k) < SST then
26:        ST(n_i^k) ← SST
27:        ST(n_i^k) is mapped to next idle communication
           slot if n_i is a message
28:      end if
29:      m ← m + 1
30:    end while
31:    round ← round + 1
32:  end while
33: end if
34: if ST(n_i^k) > o(g_j) + k * p(g_j) + d(g_j) − rank_u(n_i) then
35:   δ_o ← ST(n_i^k) − o(g_j) − k * p(g_j) − d(g_j) + rank_u(n_i)
36:   return -1
37: end if
38: return ST(n_i^k)
```

parents are finished (line 4 in Algorithm 2). The "while" loop (line 5 in Algorithm 2) then repeatedly searches until $K(g_j)$ periodic slots are found. In Algorithm 2, cnt, x, and y are local variables: cnt is the number of feasible periodic (i.e., synchronized) slots that have been found, i.e., totally a number of cnt equaling to $K(g_j)$ synchronized slots should be found for accommodating the $K(g_j)$ periodic instances; x is the index of the instance for which (i.e., n_i^x) the first feasible slot is found and thus the scheduled time of other instances should be synchronized to this instance; y is the index of another instance for which (n_i^y) the slot should be synchronized to n_i^x. Accordingly, line 6 in Algorithm 2 attempts to find a feasible slot for an instance n_i^y such that the slot is synchronized to that of n_i^x. In other words, line 6 is repeatly used to search for

$K(g_j)$ synchronized time slots. If the node is aperiodic (lines 23-31 in Algorithm 2), the algorithm separately schedules the $K(g_j)$ instances of the node via the for loop. Line 25 again calls $findStartTime(EST(n_i^k), n_i^k)$ to separately determine the start time of the instances.

Algorithm 1 returns the earliest feasible start time that can be allocated to n_i^k, i.e., $ST(n_i^k)$. If no feasible start time can be found, -1 is returned. Suppose totally M nodes $\{n_0, ..., n_{(M-1)}\}$ have been scheduled on $H(n_i)$ in one hyper period. Within one hyper period, a selected node n_i^k may be scheduled into $(M+1)$ possible time slots i.e., slot $m \in [0, M]$ between two consecutively scheduled nodes $n_{(m-1)}$ and n_m. Virtual nodes n_{-1} and n_M are used for the convenience of denoting the first and the last slots. The search of a feasible slot starts from the slot immediately before n_0 of the current hyper period, i.e., the slot immediately after $n_{(M-1)}$ of the last hyper period (lines 5-6 of Algorithm 1). For a time slot m being examined, the start time of the slot, recorded by local variable SST (line 21 of Algorithm 1), is indeed the finish time $n_{(m-1)}$; the end time of the slot, recorded by local variable SET (line 13 of Algorithm 1), is the start time of n_m. The last slot (i.e., slot M) of the last hyper period is indeed the first slot of the current hyper period. Accordingly, the virtual node n_M of the last hyper period is indeed the n_0 of the current hyper period. SET and SST for this special slot are obtained in lines 15 and 23, respectively. The algorithm searches for the first feasible idle slot that can hold n_i^k. That is, the duration of the time slot should be no shorter than the computation (or transmission) cost of the node, as shown in line 17 of Algorithm 1. Since the execution of an application is allowed to span multiple hyper periods, the procedure searches for multiple hyper periods until the overall latest start time of n_i, denoted as $LST^{max}(n_i^k)$, has been reached, as shown in the outer "while" loop, line 9 of Algorithm 1. $LST^{max}(n_i^k)$ is defined as:

$$LST^{max}(n_i^k) = d(g_j) + (k+1) * p(g_j) - rank_u(n_i). \quad (3)$$

In this equation, $LST^{max}(n_i^k)$ is obtained by assuming $o(g_j) = p(g_j)$; therefore, no feasible start time of n_i^k can be greater than $LST^{max}(n_i^k)$. Then, for each hyper period that is being searched, the inner "while" loop (line 11 of Algorithm 1) searches the $(M+1)$ possible time slots. Further, if n_i is a message, $ST(n_i^k)$ is mapped to the start time of the next idle communication slot. Further, δ_o obtained in line 35 of Algorithm 1 will be used by RTD described in the next subsection.

The worst case time complexity of Algorithm 1 is $O(N)$ where N is the number of nodes to be scheduled in a hyper period due to the inner while loop (line 11 in Algorithm 1). The outer while loop (line 9 in Algorithm 1) will only repeat for constant (usually very small) times as the maximum value of $LST^{max}(n_i^k)$ is a constant at runtime (usually very small). The worst case time complexity of Algorithm 2 is also $O(N)$ due to the while loop of Algorithm 2. For scheduling N nodes in a hyper period, the worst case time complexity of SHLF is thus $O(N^3)$ as Algorithm 3 will run N times to schedule N nodes. However, normally Algorithm 1 and Algorithm 2 will not run $O(N)$ times and thus the average time complexity of SHLF should be much lower than $O(N^3)$.

Algorithm 2 schedule(n_i)

```
1: δ_o ← 0
2: if n_i is periodic then
3:    cnt ← 1, x ← 0, y ← 1
4:    ∀k ∈ K(g_j), ST(n_i^k) ← call findStartTime(EST(n_i^k), n_i^k)
5:    while cnt < K(g_j) do
6:       ST(n_i^y) ← call findStartTime(ST(n_i^x)+(y−x)∗p(g_j), n_i^y)
7:       if ST(n_i^y) = −1 then
8:          return false
9:       end if
10:      if ST(n_i^y) = ST(n_i^x) + (y − x) ∗ p(g_j) then
11:         cnt ← cnt + 1
12:      else
13:         cnt ← 1
14:         x ← y
15:      end if
16:      if y = K(g_j) − 1 then
17:         y ← 0
18:      else
19:         y ← y + 1.
20:      end if
21:   end while
22:   ∀k ∈ K(g_j) schedule n_i^k at ST(n_i^k)
23: else
24:   for k ∈ K(g_j) do
25:      ST(n_i^k) ← call findStartTime(EST(n_i^k), n_i^k)
26:      if ST(n_i^k) = −1 then
27:         return false
28:      end if
29:      schedule n_i^k at ST(n_i^k)
30:   end for
31: end if
32: return ture
```

C. Release Time Deferment (RTD)

As periodic nodes have more strict timing constraints than aperiodic nodes, conflict may become common in node assignment, especially when the offset (release time/start time) of task graphs is fixed as zero. Notice that in many previous papers (e.g., [21], [22], [25]), offsets were simply assumed to be zero, which may suffer poor performance under complex timing constraints. If different applications are allowed to start work from different offsets such that the time assignments of different task graphs can be easily staggered, the schedulability may be enhanced even under sophisticated time constraints. Therefore, this study develops the RTD method, which reschedules conflicted applications with adjusted offsets to prevent conflictions.

The offset of each task graph is initialized as zero. If Algorithm 2 fails to schedule n_i, conflicts in time allocation have appeared. To remove the conflicts, RTD reschedules

Algorithm 3 The SHLF Algorithm

1: compute $rank_u(n_i)$ for all nodes and compute $rank(g_j)$ for all graphs
2: for all graphs, $priority(g_j) \leftarrow rank(g_j)$
3: sort the applications in a list by decreasing order of $priority(g_j)$ values
4: outerLoop:
5: **while** any application is unscheduled **do**
6: pick next application g_j from the application list for scheduling
7: initialize ready nodes
8: **while** any node in g_j is unscheduled **do**
9: select a node n_i with the highest $rank_u(n_i)$ for scheduling among all ready nodes
10: call $schedule(n_i)$
11: **if** it fails to schedule n_i **then**
12: **if** $cnt_{res} > max_{res}$ or $\delta_o = 0$ **then**
13: $cnt_{res} \leftarrow 0$
14: **if** exit conditions are fulfilled **then**
15: exit without a feasible solution
16: **end if**
17: $cnt_{back} \leftarrow cnt_{back} + 1$
18: set $back_{limit}$
19: execute long backtrack for g_j
20: $priority(g_j) \leftarrow 2 * priority(g_j)$
21: continue outerLoop:
22: **end if**
23: $cnt_{res} \leftarrow cnt_{res} + 1$
24: $o(g_j) \leftarrow o(g_j) + \delta_o$
25: backtrack all scheduled nodes of g_j
26: continue outerLoop.
27: **end if**
28: **end while**
29: **end while**

g_j to a new offset $o(g_j)$ such that n_i can be inserted into feasible slots on $H(n_i)$. The offset of g_j is updated (line 24 of Algorithm 3):

$$o(g(n_i)) = o(g(n_i)) + \delta_o \tag{4}$$

where δ_o is obtained in Algorithm 1. As shown in step 35 of Algorithm 1, if $ST(n_i^k) > o(g_j) + k * p(g_j) + d(g_j) - rank_u(n_i)$, the deadline will be violated. Since $ST(n_i^k)$ is the start time of the earliest feasible slot for n_i^k that can be found by Algorithm 1, the only way to satisfy the deadline is that the offset of g_j must be delayed so that the entry node can start at a later time:

$$\delta_o \leftarrow ST(n_i^k) - o(g_j) - k * p(g_j) - d(g_j) + rank_u(n_i). \tag{5}$$

Afterwards, all scheduled nodes of the task graph are backtracked, i.e., they are now unscheduled, and the whole task graph will be rescheduled by SHLF (lines 25-26 of Algorithm 3).

D. Long Backtrack

If RTD cannot help to eliminate the conflicts, BPP (lines 11-21 in Algorithm 3) will backtrack previously scheduled applications to create space for the failed application. It may be noticed that the backtracking is widely applied in exhaustive approaches such as branch and bound (B&B) [21], [22]. Nevertheless, these approaches are merely useful for small task graphs. In this case, some previous studies [30], [29] imposed restrictions on the times or levels of backtracks to reduce time costs. Such limitations would dramatically undermine the effectiveness of their methods when the problem scale grows up.

The proposed BPP policy makes a trade-off between time and performance in the sense that it may backtrack multiple task graphs rather than just one node each time. Once the total number of rescheduling operations (by RTD) for the failed application g_j (denoted as cnt_{res}) reaches a given limit (denoted as max_{res}) or δ_o is zero, the scheduler runs a long-hop backtrack, which backtracks not only g_j, but also multiple previously scheduled task graphs to create space for g_j. The number of task graphs backtracked, denoted as $back_{limit}$, is initialized as 1. It will be doubled if g_j has ever failed previously, and otherwise be reset. All nodes of a backtracked application are backtracked. A list L_S containing all scheduled task graphs is maintained. The scheduler repeatedly backtracks scheduled task graphs from the tail of L_S until L_S is empty or totally $back_{limit}$ task graphs have been backtracked. Also, the priority of g_j is updated as:

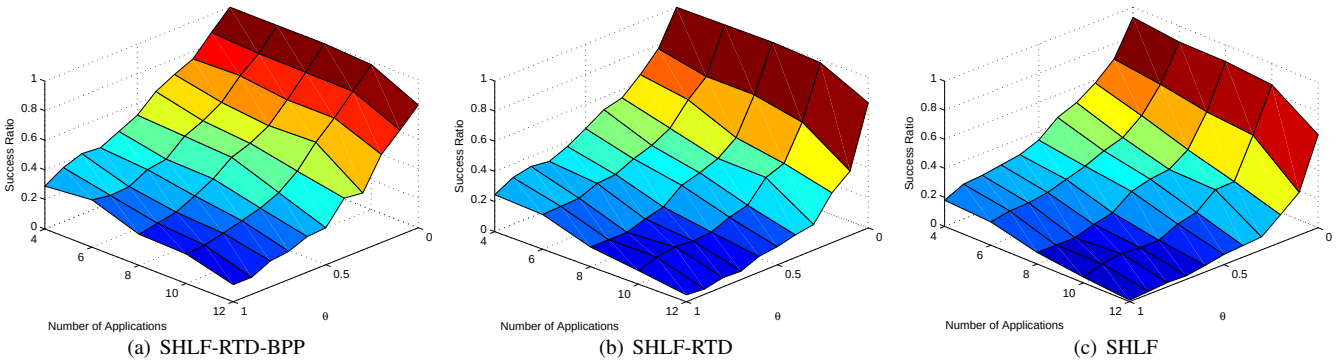
$$priority(g_j) = 2 * priority(g_j). \tag{6}$$

The intuition is that since g_j is probably hard to schedule, promoting its priority can help to schedule it in the next run. The algorithm then continues to select applications and scheduling nodes via SHLF. This method finally ends up with the cases either a feasible solution is derived or the following conditions are satisfied: all recently N_F failed task graphs have ever failed in previous runs or the total number of long-hop backtracks cnt_{back} reaches max_{back}. In our experiments, we set N_F as 5 and max_{back} as 20.

VI. Performance Evaluation

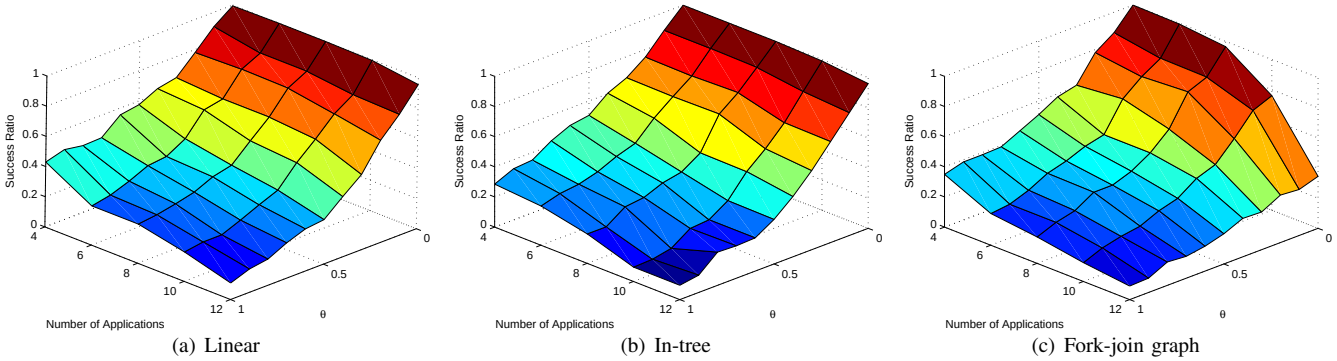
In order to assess the effectiveness of the proposed scheduling algorithm, this section presents a performance evaluation study for scheduling a number of real-time applications (i.e., task graphs). The major performance metric is success ratio, which is defined as the number that an algorithm successfully schedules all applications to the number of total experiments.

Following [15], the system configurations are set as follows: Synthetic applications with random DAG topologies are generated for the experiments. The period of each application is varied among [5ms, 10ms, 20ms, 40ms] to cover a spectrum of periods in reality. The length of one hyper schedule period is thus 40ms. The communication bus is configured with a cycle duration of 5 ms and the duration of the available segment per cycle is 3.75ms. The duration of a communication slot is set as 0.0625ms. The average cost of a task is 0.5ms. In each experiment, a number of applications are scheduled. The average number of nodes per application is 15. By default, the DAG topology of applications is randomly mixed with four topologies, *chain*, *in-tree*, *out-tree*, and *fork-join* graphs. The



(a) SHLF-RTD-BPP (b) SHLF-RTD (c) SHLF

Fig. 5. Results for random topologies.



(a) Linear (b) In-tree (c) Fork-join graph

Fig. 6. Results for specific topologies.

number of applications is varied among [4, 12] and the number of ECUs is set as twice of the number of the applications. The probability (denoted as θ) that a task node is periodic is varied from [0, 1]. When θ is zero, no task is periodic and when θ is one, all tasks are periodic.

Since the proposed algorithms comprise three major parts, SHLF, RTD, and BPP, three algorithm combinations are therefore generated for evaluation. The first algorithm, denoted as SHLF-RTD-BPP, enables all three parts. The second algorithm, denoted as SHLF-RTD, enables SHLF and RTD and disables BPP. The third algorithm, denoted as SHLF, only enables SHLF and disables both RTD and BPP. Hence the later two algorithms SHLF-RTD and SHLF can be deemed as baselines for understanding the merits of RTD and BPP. Fig. 5 plots success ratio versus number of applications and θ for the three algorithm combinations, respectively. In Fig. 5(c), when θ is equal to one the results of SHLF is equivalent to the results of HLF [32] with the oversimplified assumption that all nodes are periodic.

From Fig. 5 one can observe that SHLF-RTD-BPP outperforms SHLF and SHLF-RTD. Also, SHLF-RTD outperforms SHLF. These demonstrate the effectiveness of RTD and BPP, respectively. In addition, one can observe that the success ratio decreases as θ increases or as the number of applications increases. This is due to the fact that as θ increases or as the number of applications increases, the problem becomes more complex and the probability that conflicts on node assignment occur becomes larger. The results show that the proposed algorithms can effectively utilize the flexibility in scheduling offered by aperiodic nodes. Without these, the naive peer heuristics (e.g., HLF) driven by the oversimplified assumption that all nodes are periodic will cause poor performance, which corresponds to the results that θ is equal to one.

Since one may be interested in the performance of the algorithm on various DAG topologies, for three topologies (linear chain, in-tree, and fork-join graphs), a number of DAGs are generated and the corresponding results of SHLF-RTD-BPP are shown in Fig. 6. The results in Fig. 6 are quite similar to those of Fig. 5(a), showing that the performance of the proposed algorithm is influenced little by the DAG topology of applications.

VII. Conclusions

This study has investigated the problem of scheduling a set of periodic applications with both periodic and aperiodic tasks on the time-triggered systems. Novel models have been formulated to capture the unique features of the problem. The SHLF algorithm has been proposed to address the problem. To further improve schedulability, this study has also presented the RTD and BPP procedures. Upon a confliction in time allocation, RTD reschedules the conflicted application with an adjusted offset (i.e., release time) such that the scheduling of different applications can be staggered to avoid conflicts. Once RTD cannot help to eliminate conflics, BPP promotes the priority of the conflicted application and backtracks previously scheduled applications to create space for the conflicted applications. Moreover, the communication models for the typical time-triggered systems are considered and solutions

for bandwidth optimization are proposed. To the best of our knowledge, the study is the first effort to deal with the periodic applications with both periodic and aperiodic nodes on the time-triggered systems. Extensive simulation results with various test configurations have demonstrated the effectiveness and competitiveness of our algorithm. The simulation results have shown that the proposed approaches significantly outperform the previous algorithms under various settings.

VIII. Acknowledgement

This work is started when Menglan Hu was a research fellow at Nanyang Technological University. It was supported in part by AcRF Tier 1 Grant RGC5/13.

References

[1] Avionics Tech Report: Deterministic Networks for Advanced Integrated Systems. http://www.aviationtoday.com/Assets/AVS_120112_TechReport_Final.pdf.

[2] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in Automotive Communication Systems," *Proc. IEEE*, vol. 93, pp. 1204-1224, 2005.

[3] CAN in automation. http://www.can-cia.org/can.

[4] TTCAN. http://www.can-cia.org/can/ttcan.

[5] "The flexray communication system specification, version 3.0.1," http://www.flexray.com.

[6] A. Albert, "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems," in *Embedded World*, 2004.

[7] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proc. IEEE*, vol. 93, pp. 1204-1224, 2005.

[8] BMW brake system relies on FlexRay, "http://www.automotivedesignline.com/news/218501196," July 2009.

[9] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proc. IEEE*, vol. 91, no. 1, pp. 1120-126, 2003.

[10] E. Schmidt and K. Schmidt, "Message scheduling for the flexray protocol: The dynamic segment,"*IEEE Trans. Vehicular Technology*, vol. 58, no. 5, pp. 2160-2169, 2009.

[11] K. Schmidt and E. Schmidt, "Message scheduling for the flexray protocol: The static segment," *IEEE Trans. Vehicular Technology*, vol. 58, no. 5, pp. 2170-2179, 2009.

[12] M. Lukasiewycz, M. Glab, J. Teich, and P. Milbredt, "Flexray schedule optimization of the static segment," *Proc. CODES+ISSS*, 2009.

[13] B. Tanasa, U.D. Bordoloi, P. Eles, and Z. Peng, "Scheduling for fault-tolerant communication on the static segment of FlexRay," In *Proc. RTSS* 2010.

[14] I. Park and M. Sunwoo, "FlexRay network parameter optimization method for automotive applications," *IEEE Trans. Industrial Electronics*, vol.58, no. 4, pp. 1449-1459, APR. 2011.

[15] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty, "Modular scheduling of distributed heterogeneous time-triggered automotive systems," In *Proc. ASP-DAC*, 2012.

[16] A. Davare, Q. Zhu, M.D. Natale, C. Pinello, S. Kanajan, and A.S. Vincentelli, "Period Optimization for Hard Real-time Distributed Automotive Systems," In *Proceedings of the 44-th annual Design Automation Conference*, 2007.

[17] T. Pop, P. Eles, and Z. Peng, "Schedulability Analysis for Distributed Heterogeneous Time/Event Triggered Real-Time Systems," *Proceedings of 15th Euromicro Conference on Real-Time Systems*, 2003.

[18] P. Pop, K.H. Poulsen, V. Izosimov, and P. Eles, "Scheduling and Voltage Scaling for Energy/Reliability Trade-offs in Fault-Tolerant Time-Triggered Embedded Systems," In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2007.

[19] J. Luo, and N.K. Jha, "Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems," *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, 2000.

[20] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260-274, MAR. 2002.

[21] D. Peng, K.G. Shin, and T.F. Abdelzaher, "Assignment and scheduling communicating periodic tasks in distributed real-time systems," *IEEE Trans. Software Engineering*, vol. 23, no. 12, pp. 745-758, DEC. 1997.

[22] T.F. Abdelzaher, and K.G. Shin, "Combined task and message scheduling in distributed real-time systems," Parallel and Distributed Systems, *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 11, pp. 1179-1191, NOV, 1999.

[23] K. Ramamritham, "Allocation and Scheduling of Precedence-Related Periodic Tasks," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412-420, Apr. 1995.

[24] T.F. Abdelzaher, and K.G. Shin. "Period-based load partitioning and assignment for large real-time applications," *IEEE Trans. Computers*, vol. 49, no. 1 pp. 81-87, Jan. 2000.

[25] B.P. Dave, G. Lakshminarayana, and N.K. Jha, "COSYN: HardwareC-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 92-104, Jan. 1999.

[26] R.P. Dick, and N.K. Jha, "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems," *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, 1998.

[27] H. Zeng, , M.D. Natale, A. Ghosal, and A. Sangiovanni-Vincentelli. "Schedule optimization of time-triggered systems communicating over the FlexRay static segment," *IEEE Trans. Ind. Informat.*, 7(1), pp. 1-17, 2011.

[28] R. Mishra, N. Rastogi, D. Zhu, D. Moss, and R. Melhem, "Energy Aware Scheduling for Distributed Real-Time Systems," In *Proceedings. International Parallel and Distributed Processing Symposium*, 2003.

[29] K. Ramamritham, J.A. Stankovic, and P.F. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 2, pp. 184- 194, Apr. 1990.

[30] G. Manimaran, C. Murthy, "An efficient dynamic scheduling algorithm for multiprocessor real-time systems," *IEEE Trans. Parallel Distrib. Systems.* vol. 9 pp. 312C319, Mar. 1998.

[31] H. Jaouani, R. Bouhouch, W. Najjar, and S. Hasnaoui, "Hybrid task and message scheduling in hard real time distributed systems over FlexRay bus," *Proc. ICCIT*, pp. 21-26, 2012.

[32] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Oper. Research*, vol. 19, no. 6, pp.841-848, Nov. 1961.

[33] G.C. Sih, and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures." *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175-187, Feb. 1993.

[34] M. Wu and D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330-343, 1990.

[35] S. Thiel and A. Hein, "Modeling and Using Product Line Variability in Automotive Systems", *IEEE Software*, 2002.

[36] Y. Xu, K. Li, J. Hu, and K. Li, "A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues," *Information Sciences*, 270 255-287, 2014.

[37] K. Li, X. Tang, and K. Li, "Energy-efficient stochastic task scheduling on heterogeneous computing systems," *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[38] A.E. Gil, K.M. Passino, S. Ganapathy, and A. Sparks, "Cooperative task scheduling for networked uninhabited air vehicles," *IEEE Transactions on Aerospace and Electronic Systems*, 44, no. 2, pp. 561-581, 2008.

[39] Suresh, Sundaram, C. Run, H.J. Kim, T.G. Robertazzi, and Y. Kim, "Scheduling second-order computational load in master-slave paradigm." *IEEE Transactions on Aerospace and Electronic Systems*, vol. 48, no. 1, pp. 780-793, 2012.

[40] J.T. Hung and T.G. Robertazzi, "Scheduling nonlinear computational loads," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 44, no. 3 pp. 1169-1182, 2008.

Menglan Hu received the B.E. degree in Electronic and Information Engineering from Huazhong University of Science and Technology, China (2007), and the Ph.D. degree in Electrical and Computer Engineering from the National University of Singapore, Singapore (2012). He is currently an Associate Professor at the School of Electronic Information and Communications, Huazhong University of Science and Technology, China. His research interests includes cloud computing, parallel and distributed systems, scheduling and resource management, as well as wireless networking.

Jun Luo received the B.S. and M.S. degrees in electrical engineering from Tsinghua University, Beijing, China, in 1997 and 2000, respectively, and the Ph.D. degree in computer science from the Swiss Federal Institute of Technology in Lausanne (EPFL), Lausanne, Switzerland, in 2006. From 2006 to 2008, he has worked as a Post-Doctoral Research Fellow with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada. In 2008, he joined the faculty of the School of Computer Engineering, Nanyang Technological University, Singapore, where he is currently an Assistant Professor. His research interests include wireless networking, mobile and pervasive computing, distributed systems, multimedia protocols, network modeling and performance analysis, applied operations research, and network security. He is a member of the IEEE.

Yang Wang received the BS degree in applied mathematics from the Ocean University of China in 1989 and the MS and PhD degrees in computing science from Carleton University and the University of Alberta in 2001 and 2008, respectively. He is currently at IBM Center for Advanced Studies (CAS), Atlantic, University of New Brunswick, Fredericton, Canada. Before joining CAS Atlantic in 2012, he was a research fellow at the National University of Singapore from 2010 to 2012. Before that, he was a research associate at the University of Alberta, Canada, from August 2008 to March 2009. His research interests include scientific workflow computation and virtualization in Clouds and resource management algorithms.

Bharadwaj Veeravalli received his BSc in Physics, from Madurai-Kamaraj University, India in 1987, Master's in Electrical Communication Engineering from Indian Institute of Science, Bangalore, India in 1991 and PhD from Department of Aerospace Engineering, Indian Institute of Science, Bangalore, India in 1994. He did his post-doctoral research in Concordia University, Montreal, Canada, in 1996. He is currently with the Department of Electrical and Computer Engineering at the National University of Singapore, Singapore, as a tenured Associate Professor. His research interests include Cloud/Grid/Cluster Computing, Scheduling in Parallel and Distributed Systems, Bioinformatics & Computational Biology, and Multimedia Computing. He is one of the earliest researchers in Divisible Load Theory (DLT). He had secured several externally funded projects and published over 120 papers in high-quality journals and conferences. He has co-authored three research monographs in the areas of PDS, Distributed Databases, and Networked Multimedia Systems, in 1996, 2003, and 2005, respectively. He is currently serving the Editorial Board of IEEE Transactions on SMC-A, Multimedia Tools & Applications (MTAP) and Cluster Computing, as an Associate Editor. Until 2010 he had served as an AE for IEEE Transactions on Computers. More information can be found in http://cnl-ece.nus.edu.sg/elebv/. He is a senior member of the IEEE and the IEEE computer society.