Flexible Static Scheduling of Software with Logical Execution Time Constraints

P. Derler, S. Resmerita

Technical Report
May 17, 2010

Software & systems Research Center (SRC)
C. Doppler Laboratory 'Embedded Software Systems'
Univ. Salzburg
5020 Salzburg
Austria, Europe

Flexible Static Scheduling of Software with Logical Execution Time Constraints

Patricia Derler, Stefan Resmerita University of Salzburg
{patricia.derler, stefan.resmerita}@cs.uni-salzburg.at

***Abstract*—Various programming models for embedded, time-triggered software employ the logical execution time (LET) abstraction in order to achieve time- and value-determinism. In these models, the application software is partitioned into tasks and a LET is associated with every task. In every execution, a task reads input values at the beginning of the LET and writes output values at the end of the LET. To achieve this behavior, existing implementations of LET models impose execution constraints where every physical execution of a task must take place between the corresponding LET bounds. In this paper we investigate a more efficient implementation paradigm in which LET-based execution constraints are relaxed by allowing tasks to execute outside of their LET bounds while preserving their I/O LET behavior. We present a modified runtime operational semantics where scheduling operations are decoupled from data transfer operations. Moreover, we propose a way to statically determine task release times that may precede LET start times, by using information about tasks connectivity (available in the LET model) and about task execution times (required for schedulability analysis). The consequences of using the relaxed constraints on the schedulability of the system are explored. We address sustainability of preemptive scheduling with respect to variations in release times and propose bounds on such variations that preserve schedulability for Fixed Priority (FP) scheduling. Moreover, we describe an application of Dual Priority scheduling which guarantees schedulability for any release time variation of a system that is originally FP-schedulable. Relaxing execution constraints leads to increased processor utilization. Some of the benefits thereof are illustrated on a typical control application.**[1]

I. Introduction

Developing complex real time applications requires programming disciplines that take into account suitable abstractions of execution and communication times. The Giotto programming model [7] employs the concept of logical execution time (LET) of a software component (a task), representing a fixed logical duration for one execution of the task. For a periodic task, the LET specifies the real time instants when task inputs and outputs are updated in every execution of the task. A runtime system performs these I/O actions at the right times, and also dispatches the task for execution, such that the execution uses the input values that are available at the beginning of the LET. Also, the task must issue its outputs to the runtime system before the end of the task's LET, when they are made available to the task's environment.

The LET concept is used in Giotto successors such as TDL [12] and HTL [6]. All of these LET-based approaches offer the same advantages for time-triggered software: First, the application development process benefits from platform independence and from a dual separation of concerns: timing versus functionality and reactivity versus scheduling [5]. Second, the I/O behavior of the application is time and value deterministic.

For an application with LET specifications and a given execution platform, a time-safety check must be performed to decide whether the LET requirements can be met. This check involves a schedulability analysis, which takes as inputs a scheduling policy, the worst case execution times (WCETs) of the application tasks and a set of constraints derived from the LET specifications. The analysis decides if the task set can be scheduled such that the LET constraints are satisfied. Existing implementations of the above mentioned programming model use conservative constraints, essentially requiring that the total physical execution of a task takes place within the LET interval. In other words, the beginning of the LET is the release time of the task, when the task becomes ready for execution, and the end of the LET is the latest termination time of the task.

Figure 1, which can be found in [5] and [12], illustrates the classical LET view with the I/O semantics and the execution constraints.

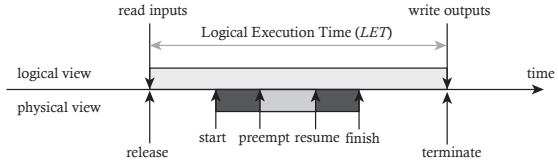


Fig. 1. Classical picture: LET defines a maximal physical execution interval

An embedded application (program) consisting of a set of time-triggered tasks with LET specifications is said to be *schedulable* if any run of the program satisfies the LET-based I/O semantics. Satisfaction of the classic execution constraints is a sufficient condition for schedulability of the application. Thus, if no schedule with these constraints can be found, then nothing can be said about time-safety of some runs of the program and in practice the application is declared unschedulable. Note that these constraints are

[1]This work was supported in part by the Embedded Software Research Center (SRC) of the University of Salzburg, which receives support from the Christian Doppler Laboratory and in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, Lockheed-Martin, National Instruments, and Toyota.

platform independent and reflect a black-box view for a task, since no information about the internal structure of tasks is used. Yet the constraints are directly used in a schedulability analysis that is specific to a scheduling policy and that uses WCET information, which is usually obtained from a detailed analysis of task implementations.

This paper describes the conditions under which a task can be released for execution before the beginning of its LET and the execution of a task can be completed after the end of the LET, thus enlarging the time frame for scheduling the task's execution, as depicted in Figure 2. Extending the execution window is based on statically available information about the tasks. One can release a task T for execution δ_r units of time earlier than the beginning of the LET if it is guaranteed that: (1) Each input port of T accessed during δ_r has been updated by the runtime system before being used, and (2) The value of the corresponding input source does not change from the moment of the update until the beginning of the LET. A particular case is when no input port is accessed in the part of the task executed during δ_r. Similarly, one can terminate the task δ_e units of time later than the end of the LET if it is guaranteed that no output port of the task is accessed after the end of the LET.

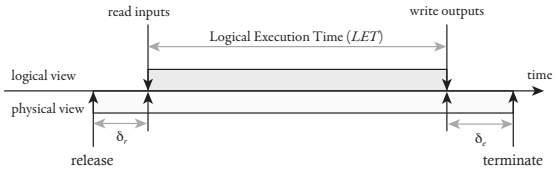


Fig. 2. Extended time frame for physical execution

The relaxed constraints are relevant for general time-triggered applications, where a periodic task can have an offset and the LET may be smaller than the invocation period. We assume a shared memory communication model between the runtime system that performs the LET-based I/O and the time-triggered tasks. Thus, a task has a set of (internal) input ports implemented as global variables, which are updated by the runtime system and accessed within the task's code. Also, a task has a set of (internal) output ports as global variables, which are updated by the task during an execution and read by the runtime system at the end of the LET.

Our approach enables opportunistic executions of time-triggered tasks, where a well-defined initial part of a task's code can be executed before the beginning of the task's LET, when the processor would otherwise be idle. This can lead to the following advantages:

(a) Enlarging the search space for feasible schedules that achieve time-safety. In particular, an application that is declared unschedulable under the classical constraints may become time-safe by using the relaxed constraints.
(b) If the system has high priority event-triggered tasks that may preempt time-triggered tasks, using the relaxed constraints reduces the risk of missing LET deadlines or reduces the number of missed LET deadlines.
(c) If the system has event-triggered tasks executed in the background, then the response times of such events can be shorter with the relaxed constraints.

We investigate the effects of using these constraints together with three preemptive scheduling algorithms: (1) Fixed-priority (FP) scheduling, (2) Earliest Deadline First (EDF), and (3) Dual-Priority (DP) scheduling, where every LET-based task has two fixed priorities: a nominal priority, with which the task is scheduled inside its LET and a reduced priority, with which the task is scheduled outside of its LET. An event-triggered task has one priority, as in FP. Some of the benefits mentioned above are illustrated by applying the DP scheduling with relaxed constraints on a standard control application, showing a 25% decrease in response times of event-triggered tasks on average, versus the classical constraints with FP scheduling.

II. Background

We consider an embedded software application consisting of a set of periodic tasks $\mathbf{T}$ where timing requirements for the tasks in $\mathbf{T}$ are expressed at the level of the software model by LETs, using one of the time-triggered languages originated in Giotto such as TDL, xGiotto, or HTL. In this paper, we focus on single-mode operation, where all tasks in T are active.

Each time-triggered task T has a period $\pi(T)$ and a logical execution time $LET(T)$, which can be placed anywhere within the period bounds. The beginning of the LET in a period is defined by an offset $\phi(T)$ such that $\phi(T)+LET(T) \leq \pi(T)$. The hyperperiod of the system is the least common multiple of all periods of tasks in $\mathbf{T}$. For every period of time $[i\cdot\pi(T), (i+1)\cdot\pi(T)]$ $(i = 0, 1, \ldots)$, the start and end points of the corresponding LET interval are represented by $t^i_{Ls}(T) = i \cdot \pi(T) + \phi(T)$ and $t^i_{Le}(T) = i \cdot \pi(T) + \phi(T) + LET(T)$. Note that $[t^i_{Ls}(T), t^i_{Le}(T)] \subseteq [i \cdot \pi(T), (i + 1) \cdot \pi(T)]$. Let $t^{-1}_{Le}(T) = 0$ by convention.

The application tasks are subject to the usual assumptions: Tasks do not have internal synchronization points, no task can suspend itself and any task can be preempted at any time. In particular, tasks do not communicate directly: every task reads inputs from designated internal variables called *(internal) input ports* and writes computed values to designated local variables called *(internal) output ports*.

A. Task Information

We consider that a task's code is available for structural and execution time analysis. The types of information that can be obtained by such analysis can be described by looking at common structures of code, as follows. Consider the example shown in Figure 3, where global variables $i1, i2$ are task input ports and o is a task output port. Assume that task T has an associated LET, being part of a timed program (e.g, a Giotto program, or a TDL program). In existing implementations of such timed programs, the software task is regarded as a black box - Figure 3(a): all its inputs are read at the beginning and all its outputs are updated at the end of every execution, leading to restrictive execution constraints: the task is released for execution no earlier than the beginning of the LET and it is terminated no later than the end of the LET.

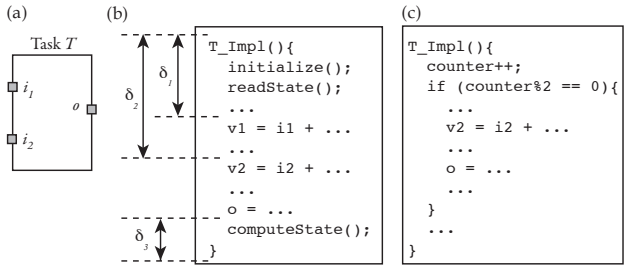


Fig. 3. Examples of task implementations

Figure 3(b) shows a possible task implementation where δ_1 is the minimum execution time between the beginning of the task and any access to any input port (assuming that variable i_1 is the first input port to be used in any execution of the task), δ_2 is the minimum execution time between the beginning of the task and the first access to i_2, and δ_3 is the minimum execution time between the last access to any output port and the end of the task. These minimum delays can be used to determine which parts of the task's code can be executed outside of the LET.

Figure 3(c) shows another example, where the code of the *if* block is executed every second invocation of the task. If the ports $i2$ and o are accessed only within that block, then we know that $i2$ need not be updated by the runtime system in invocations $1, 3, 5, \ldots$ of the task, and that o has a constant value during the even periods of the task.

In some cases one can statically determine time intervals within which input values for a task remain constant. In general, we distinguish among two types of predictable input sources for time-triggered tasks in a LET-based system: LET-based time-triggered tasks and event-triggered tasks. For LET-based time-triggered tasks the time when outputs are updated is the end of the LET. In the case of event-triggered tasks one can sometimes specify a minimum time period between consecutive occurrences of an event. If the event is triggered by computational tasks, this minimum inter-arrival time is based on timing predictability of software executions, as described above. If the event is triggered by a change in value of a physical parameter in a control system, one can use the model of the physical environment to predict the inter-arrival bound. For example, consider a heater/cooler control application, where a temperature sensor with hysteresis is used. The maximum rate of temperature change at the sensor's location in the controlled space and the sensor's hysteresis determine a minimum time period between consecutive changes in the sensor's output.

Another type of information that can be used to obtain flexible execution constraints at compile time refers to parameters of the scheduler. An example in this respect is a fixed-priority preemptive scheduling with deadline-monotonic prioritization, where task priorities are known at compile time.

B. LET-Based Operational Semantics

We assume the existence of a runtime system which executes three main types of operations related to the tasks in T (in the absence of multiple modes):

- Updating task outputs and actuators: A *visible output* of a task is a variable of the runtime system that corresponds to an internal output port of the task and is updated at the end of the task's LET with the value of the internal output port. An actuator update is a value transfer from a task output to an actuator device.
- Updating input ports: The runtime system updates a tasks' input ports with values from outputs of other tasks or from the physical environment. Values from the physical environment are obtained by using platform-specific components (drivers), which are called by the runtime system to retrieve sensor data. Thus, a task's input port has an associated *input source*, which is either a visible task output or a sensor.
- Releasing time-triggered tasks for execution: The runtime system is responsible for activating a time-triggered task, i.e., setting the task to be ready for execution. Once the task is released, the underlying scheduler is responsible for allocating the CPU to the task.

Each time-triggered programming language defines a discrete-time operational semantics, specifying the sequence of operations that have to be executed by the runtime system at every time step. This sequence, also called the timing program or top-level schedule is created by compiling the timing specifications. At runtime the timing program is executed every hyperperiod. A particular characteristic of existing LET-based operational semantics is grouping the update of a task's inputs with the release of the task into one common operation, sometimes called task invocation. This implies that the LET behavior is achieved only if every task execution is released exactly at $t^i_{Ls}(T)$, which is an execution constraint.

Our aim is to relax this constraint, by allowing an interval of release times for a task execution, that *ends* at $t^i_{Ls}(T)$. The start of this interval is computed based on various information about the application, platform and scheduler. We assume that the communication between the runtime system and each task is achieved by shared memory, i.e., task input and output ports are global variables. We consider each update of an input port as a distinct operation, which can be guarded. Thus, for the same task execution, the beginning of the LET, the time step when the task is released, and the moment when an input port is updated, can all be distinct.

We propose a modified operational semantics that specifies fixed time steps for data transfer in terms of the LET end-points, while replacing specification of precise time steps for task releases with a requirement expressed in terms of system behavior. The exact time steps for task releases are determined during compilation, such that the release requirement is satisfied. Thus, the release times are fixed in the timing program. The modified operational semantics presented below describes only operations that affect task scheduling; other operations (such as actuator updates, checking guards, etc.) are the same as in the classical semantics. For each time-triggered task T, the following operations are defined:

(O1) Every visible output port of T is updated with the value of the corresponding (internal) output port at $t^i_{Le}(T)$.

(O2) Every input port of T that is connected to a sensor is

updated at $t^i_{Ls}(T)$.

(O3) Each input port of T that is connected to an output of a task U is updated at $t^i_{Le}(T)$ if $t^i_{Ls}(T) \leq t^j_{Le}(U) \leq t^i_{Le}(T)$ and at $t^j_{Le}(U)$ otherwise ($\forall i, j = 0, 1, \ldots$). Thus, an input of T is updated each time its source value is modified, except when that time is within a LET interval of T, when the input must stay unchanged. If the source value is modified within the LET of T, the input update at the end of T's LET ensures that the last modification of the source is reflected in T's input.

(O4) The task T is released for its i-th execution at a time $t^i_r(T)$ with $t^{i-1}_{Le}(T) \leq t^i_r(T) \leq t^i_{Ls}(T)$ such that no input port connected to a sensor is accessed earlier than $t^i_{Ls}(T)$ and for any other input port used by T at any time $t < t^i_{Ls}(T)$ the value of the corresponding input source at time t is equal to the value of the same source at time $t^i_{Ls}(T)$. This allows for $t^i_r(T) < t^i_{Ls}(T)$ if sources of input values used before $t^i_{Ls}(T)$ remain constant at least until $t^i_{Ls}(T)$.

C. LET Compliant Runs

The set of input ports of task T is denoted by $\mathbf{P_{IN}}(T)$ and the set of output ports is denoted by $\mathbf{P_{OUT}}(T)$. Each output port q (which is a variable updated by the task) has a correspondent variable called *visible output port*, denoted by $\tilde{q}$, which is maintained by the runtime system. Every input port p is connected to a source, referred to as s_p, which can be a visible output port of some task or a sensor. The value of a variable a at time t is denoted by $v(a, t)$, where a can be an input port, an output port, a visible output port, or a sensor.

An execution trace of task T is a (possibly infinite) sequence of executions indexed by $i = 0, 1, 2, \ldots$, where the i^{th} execution corresponds to the period of time $[i \cdot \pi(T), (i+1) \cdot \pi(T)]$ and every execution starts after the termination of the previous execution. We use $V^i(p)$ to represent the set of all values read from input port p of T during the i^{th} execution of T. The termination time of the i^{th} execution of T is denoted by $t^i_t(T)$. A run of the embedded software application is a collection of execution traces, one for each task in $\mathbf{T}$.

Definition 2.1: A run of the application is LET-compliant, if for every task $T \in \mathbf{T}$, every execution $i = 0, 1, \ldots$, of the execution trace of T is such that:

$$(i) \forall p \in \mathbf{P_{IN}}(T), \qquad V^i(p) = \left\{ v\left(s_p, t^i_{Ls}(T)\right) \right\} \text{ and}$$

$(ii) \forall q \in \mathbf{P_{OUT}}(T),$

$$(a) \quad v(\tilde{q}, t^i_{Le}(T)) = v(q, t^i_{Le}(T))$$

$$(b) \quad \forall t \in \left[t^i_{Le}(T), t^{i+1}_{Le}(T)\right), v\left(\tilde{q}, t\right) = v\left(\tilde{q}, t^i_{Le}(T)\right)$$

(iii) if $t^i_t(T) \geq t^i_{Le}(T)$, then $\forall q \in \mathbf{P_{OUT}}(T)$,

$$\forall t \in \left[t^i_{Le}(T), t^i_t(T)\right), v\left(q, t\right) = v\left(q, t^i_{Le}(T)\right)$$

Point (i) means that all values read from an input port during an execution must be equal to the value of the input source at the time of the LET start. Relation $(ii)(a)$ holds if every visible output port is updated with the value of the corresponding output port at $t^i_{Le}(T)$, while $(ii)(b)$ says that the value of a visible output port must stay unchanged between consecutive LET ends. Point (iii) states that if an execution exceeds $t^i_{Le}(T)$, then no value of an output port may be updated during the part of the execution which takes place after $t^i_{Le}(T)$.

For simplicity of presentation, in the formal results of this paper we will keep the classical termination constraint - a task must finish execution by the end of its LET - and focus on the relaxation of the release time constraints.

One can show that the modified operational semantics defined in Section II-B leads to correct I/O timing behavior of the application.

Theorem 2.2: Any run of the embedded application in conjunction with a runtime system that implements the operations O1-O4 defined in Section II-B, where every execution of a task terminates before the corresponding LET end, is LET-compliant.

Proof: The proof is a simple check of the properties given in Definition 2.1. Taking them in reverse order, note that (iii) is eliminated by hypothesis and (ii) holds due to operation $(O1)$. It remains to show that (i) holds. To this end, let $t^i_r(T)$ be the release time of the i^{th} execution of T, with $t^{i-1}_{Le}(T) \leq t^i_r(T) \leq t^i_{Ls}(T)$ and take any input port p of T. If the source of p, denoted by s_p, is a sensor, then operation $(O2)$ ensures that p is updated only once in every execution of T, at $t^i_{Ls}(T)$, while $(O4)$ chooses a release time such that p is not accessed before $t^i_{Ls}(T)$. Thus $V^i(p) = \left\{ v\left(s_p, t^i_{Ls}(T)\right) \right\}$ for all $i = 0, 1, \ldots$

If s_p is a visible output port of some task $U \in \mathbf{T}$, then according to $(O1)$, s_p is updated only at the LET end of U. By $(O3)$, the value of p is updated with the value of s_p whenever s_p changes, except when this is included in a LET interval of T. At the end of T's LET, p receives the last value of s_p updated during T's LET. This implies that the value of p used by T during the LET of T is $v(s_p, t^i_{Ls}(T))$. Consider now that p is accessed at a time $t_p < t^i_{Ls}(T)$. By $(O4)$, $t^i_r(T)$ was chosen such that $v(s_p, t_p) = v(s_p, t^i_{Ls}(T))$. Note that $t_p \geq t^i_r(T) \geq t^{i-1}_{Le}(T)$ and operation $(O3)$ ensures that at t_p the port p is updated with the latest value of s_p: $v(p, t_p) = v(s_p, t_p)$. It follows that $v(p, t_p) = v(s_p, t^i_{Ls}(T))$ for any access time $t_p \in [t^i_r(T), t^i_{Le}(T)]$. ■

Corollary 2.3: Any run of a schedulable embedded application with classical LET-based operational semantics is LET-compliant.

Proof: It can be readily seen that the release operation in the classical semantics is a particular case of $(O4)$, when $t^i_r(T) = t^i_{Ls}(T)$, and that under this circumstance updating all inputs at $t^i_{Ls}(T)$ (before releasing the task) has the same effects as $(O3)$ and $(O2)$. Then the statement follows immediately from the above theorem. ■

For the remaining of this paper we assume existence of a runtime system that executes operations $(O1) - (O4)$ at statically determined times, and we investigate the situation where release times determined for step $(O4)$ are smaller than the corresponding LET start times. First, we describe one way of generating these release times such that the conditions of $(O4)$ are satisfied. Second, we explore the consequences of extending the execution window on the schedulability of the system. Third, we evaluate the proposed approach on a common control system.

III. LET-Based Static Release Times

This section describes a method for computing early release times based on information about task connectivity (available at the model level) and execution times of code segments (provided by some execution time analysis). The formal presentation is preceded by an illustration of the main ideas on simple examples.

A. Examples

Consider a LET-based task T and assume, for simplicity, that T has only one execution path. For an input port p of task T, we denote by $\delta(T,p)$ the minimum execution time of T's code segment from the beginning until the line of code where p is accessed for the first time. The input ports of T can be grouped in two categories, as follows.

1) Inputs from sensors: Consider the implementation sketched in Figure 4(a), where p is the first input port read by T. Assume that p is connected to a sensor. Notice that T can be released at time $t_r = t_{Ls}(T) - \delta(T,p)$, as depicted in Figure 4(b), without violating the I/O behavior given by the LET specification: at the time when p is accessed in the execution, p has the value of the sensor at the beginning of the LET interval.

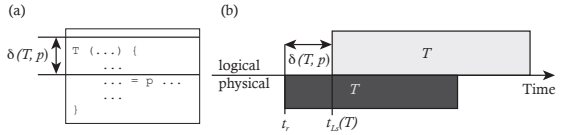


Fig. 4. Task T reads from a sensor

2) Inputs from time-triggered tasks: Assume now that T reads from ports p_1 and p_2, which are connected to visible output ports of two LET-based tasks T_1 and T_2, respectively (see Figure 5(a)). In this case, a release time which satisfies the conditions of (O4) is $t_r = max\{t_{Le}(T_1) - \delta(T,p_1), t_{Le}(T_2) - \delta(T,p_2)\}$. This is illustrated in Figure 5(b) for a specific placement of LETs. Note that operation (O3) ensures that p_1 and p_2 are updated with the values of their corresponding sources at the LET ends of T_1 and T_2, respectively.

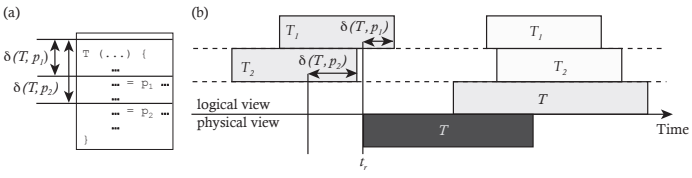


Fig. 5. Task T reads from two time-triggered tasks

B. Computing release times

Let us first formally specify the required task timing information, as follows. The control flow graph (CFG) of task T is a directed acyclic graph representing all the possible execution paths of T. CFG is commonly used in timing analysis based on abstract interpretation and path analysis [1]. Let G_T be the CFG of T with every loop unrolled exactly once. For a path x_T in G_T and an input port p of T we denote by $\delta(x_T,p)$ the minimum execution time of the prefix of x_T up to the place where p is first accessed. If there is no access to p in x_T, then $\delta(x_T,p) = \infty$ by convention. Let:

$$\delta(T,p) = \min\{\delta(x_T,p) | x_T \in G_T\}$$
$$\delta(T,P^S) = \min\{\delta(T,p) | p \in \mathbf{P_{IN}}(T),\ s_p \text{ is a sensor}\}$$

For any task $U \in \mathbf{T}$, we take the latest end of a U's LET that precedes $t^i_{Ls}(T)$ as:

$$t^{\rho,i}_{Le}(U,T) = \begin{cases} 0 & \text{, if } t^0_{Le}(U) > t^i_{Ls}(T) \\ \max_{j=0,1,\ldots}\left\{t^j_{Le}(U) \quad | \quad t^j_{Le}(U) \le t^i_{Ls}(T)\right\}, & \\ & \text{otherwise} \end{cases}$$

Note that $t^{\rho,i}_{Le}(T,T) = t^{i-1}_{Le}(T)$. The set of all inputs of T connected to visible outputs of $U, U \neq T$, is denoted by:

$$P^U(T) = \{p \in \mathbf{P_{IN}}(T) \quad | \quad s_p \text{ is a visible output of } U\}$$

Now we can define the minimum execution time of the initial code segment of T until reading the first input from U:

$$\delta(T,P^U) = \begin{cases} \min\{\delta(T,p) | p \in P^U(T)\} & \text{, if } P^U(T) \neq \emptyset \\ +\infty & \text{, otherwise} \end{cases}$$

By convention, we consider $\delta(T,P^T) = 0$.

For any task $T \in \mathbf{T}$ take the release time of the i-th execution of T ($i = 0,1,\ldots$) as follows:

$$t^i_r(T) = \max\left\{0, t^i_{Ls}(T) - \delta(T,P^S), \max_{U\in\mathbf{T}}\left\{t^{\rho,i}_{Le}(U,T) - \delta(T,P^U)\right\}\right\} \quad (1)$$

Theorem 3.1: Any run of the embedded application in conjunction with a runtime system that implements operations (O1) - (O3), where every execution $i = 0,1,\ldots$ of any task T is released at the time $t^i_r(T)$ given by relation 1 and terminates before $t^i_{Le}(T)$, is LET-compliant.

Proof: We need to show that the conditions of operation $(O4)$ are satisfied. First, note that $t^i_r(T) \ge t^{\rho,i}_{Le}(T,T) - \delta(T,P^T) = t^{i-1}_{Le}(T)$. Also, since $t^{\rho,i}_{Le}(U,T) \le t^i_{Ls}(T)$ for any U, it follows that $t^i_r(T) \le t^i_{Ls}(T)$.

Let p be any input port connected to a sensor. From relation 1, we have that $t^i_r(T) \ge t^i_{Ls}(T) - \delta(T,p)$ which is equivalent to $t^i_r(T) + \delta(T,p) \ge t^i_{Ls}(T)$. Note that $t^i_r(T) + \delta(T,p)$ is the earliest possible time at which p can be accessed during the i^{th} execution of T. Thus, no input port updated from a sensor is accessed before $t^i_{Ls}(T)$. Now take any input port p of T connected to a visible output s_p of some other task U, and assume that T reads from p at a time $t_p < t^i_{Ls}(T)$. We have $t_p \ge t^i_r(T) + \delta(T,p) \ge t^i_r(T) + \delta(T,P^U) \ge t^{\rho,i}_{Le}(U,T)$. According to operation $(O1)$, the last update of the source s_p before $t^i_{Ls}(T)$ happens at time $t_s = t^{\rho,i}_{Le}(U,T)$. Consequently, since $t_p \ge t_s$, the value of the source of p does not change between t_p and $t^i_{Ls}(T)$. We conclude that all conditions of $(O4)$ are satisfied for every task execution, hence by Theorem 2.2 the run is LET-compliant. ■

IV. Schedulability Analysis

Schedulability testing for a system with modified release times may be harder than for classical release times, due to loss of periodicity in the task model. Thus, it is of interest to investigate conditions under which schedulability of the original system is retained when shifting release times to the left. This property is called scheduling *sustainability* in [2], where it is shown that the earliest deadline first (EDF) policy is sustainable with respect to release times. Thus, if the original LET-based system is schedulable under EDF, then the release times given by relation 1 can be used without jeopardizing schedulability.

In this section, we examine consequences of release time changes for two types of preemptive uniprocessor scheduling policies: fixed task-level priority (FP) and fixed job-level priority. FP is widely used in the embedded systems industry due to its predictability. In a fixed job-level priority scheme, distinct jobs of the same task may have different priorities, but the priority of each job is constant.

We consider a scheduling task model where each task $T \in \mathbf{T}$ is associated to a collection of jobs $\mathcal{J}(T) = \{x_i(T) = (r_i(T), c_i(T), d_i(T))\}_{i\geq 1}$, where r_i denotes the release time (ready time), c_i is the execution time, and d_i is the deadline of the i-th job of task T. Note that the term *job* in the scheduling context is related to the term *execution* in the LET context. The collection of all task jobs is $\mathcal{J} = \bigcup_{T\in\mathbf{T}} \mathcal{J}(T)$. We assume that jobs of the same task are non-trivial and do not overlap in time, i.e., $d_{i-1}(T) \leq r_i(T) < d_i(T)$. Moreover, all jobs in $\mathcal{J}$ are independent. In the standard task model for schedulability checking of LET-based systems, jobs are defined as follows: $r_i(T) = t^i_{Ls}(T)$, $c_i(T) = WCET(T)$, and $d_i(T) = t^i_{Le}(T)$.

We need to examine a modified job set $\mathcal{J}' = \bigcup_{T\in\mathbf{T}} \mathcal{J}'(T)$, where $\mathcal{J}'(T) = \{x'_i(T) = (r'_i(T), c_i(T), d_i(T))\}_{i\geq 1}$ such that $d_{i-1}(T) \leq r'_i(T) \leq r_i(T)$. For a job $x_i(T)$, the time interval $[r_i(T), d_i(T)]$ is referred to as the *execution interval* of $x_i(T)$ in $\mathcal{J}(T)$. A sufficient condition of schedulability invariance under preemptive policies is given next.

Theorem 4.1: If a job set $\mathcal{J}$ is schedulable by a preemptive policy, then the set of modified jobs $\mathcal{J}'$ is schedulable by any preemptive policy which introduces no new preemption points in the execution intervals of $\mathcal{J}$.

Proof: Take any job of some task T in the modified system: $x'_i(T) \in \mathcal{J}'(T)$. Under a scheduling policy which does not create new preemption points in the execution intervals of $\mathcal{J}$, any task that preempts the execution of T in the interval $[r_i(T), d_i(T)]$ in the modified system also preempts the execution of T in the same interval in the original system. This, together with the fact that $[r_i(T), d_i(T)] \subseteq [r'_i(T), d_i(T)]$, implies that the amount of time available for T's execution in $[r'_i(T), d_i(T)]$ in the modified case is at least as large as the amount of time available to T in $[r_i(T), d_i(T)]$ in the original case. Hence, the modified system is schedulable. ■

Let us discuss in our context the example ([2], Example 1), which was used to show that the Leung and Whitehead test for fixed priority scheduling [10] is not sustainable with respect to the type of change considered here. Consider a periodic task system with two tasks T_1 and T_2 where T_1 has higher priority, $r_i(T_1) = 0.5 + 2\cdot(i-1)$, $c_i(T_1) = 1$, $d_i(T_1) = 2\cdot i$, and $r_i(T_2) = 3\cdot(i-1)$, $c_i(T_2) = 1.5$, $d_i(T_2) = 3\cdot i$. As described in [2], this system becomes unschedulable if *all* jobs of T_1 are shifted to the left by 0.5 units of time, that is $r'_i(T_1) = r_i(T_1) - 0.5$. In fact, schedulability is lost for any value of the shifting. However, this seems to be a consequence of the constraint that the modified system must be periodic (all jobs must suffer the same change). We allow *independent* job changes, where periodicity may be lost but schedulability can be maintained. In this example, one can readily see that the modified system with $r'_{3k+1}(T_1) = r_{3k+1}(T_1) - 0.5$ and $r'_{3(k+1)} = r_{3(k+1)(T_1)-0.5}$ (for all $k = 0, 1, ...$) is FP schedulable - notice that no new preemption point is created. Moreover, any change of r_{3k+2} (for any $k = 0, 1, \ldots$) leads to loss of schedulability - incidentally, such a change would create a new preemption point in the execution interval $[r_{3k+1}(T_2), d_{3k+1}(T_2)]$.

Theorem 4.1 can be used to determine release time changes which are acceptable from a schedulability viewpoint. Alternatively, schedulability can be retained under given changed release times by suitably setting scheduling parameters such that this condition is satisfied. These cases are elaborated upon in the next two subsections.

A. Task-Level Fixed-Priority

One can determine release times that guarantee preservation of schedulability under FP as follows. Let $lp(T)$ denote the set of tasks with lower priorities than T. For every job $(r_i(T), c_i(T), d_i(T)) \in \mathcal{J}(T)$, define the modified job $(r'_i(T), c_i(T), d_i(T))$ with

$$r'_i(T) = \begin{cases} r_i(T) & \text{, if } \exists \tilde{T} \in lp(T) \text{ and } x_k \in \mathcal{J}(\tilde{T}) \text{ s.t. } r_i(T) \in [r_k(\tilde{T}), d_k(\tilde{T})] \\ \max_{\tilde{T}\in lp(T)}\{d_k(\tilde{T}) | x_k \in \mathcal{J}(\tilde{T}) \text{ and } d_k(\tilde{T}) \leq r_i(T)\}, & \text{otherwise.} \end{cases} \quad (2)$$

According to the relation 2, a release time must remain unchanged if it falls into an execution interval of a lower priority task. Otherwise, the release time can be decreased to the last preceding deadline of all lower priority jobs. It can be readily seen that this introduces no additional preemption points inside the original execution intervals under FP, hence by Theorem 4.1 schedulability is not compromised.

For LET-based systems with FP scheduling, it is sufficient to take as release times the maximum between relation 1 and relation 2 to guarantee both the LET I/O semantics and the schedulability of the system.

B. Job-Level Fixed-Priority

Consider a job set $\mathcal{J}$ that is schedulable by some scheduling policy which assigns a priority to each job. We assume that priorities are drawn from a finite set. Each job is subject to a change in release time, leading to a modified set $\mathcal{J}'$. The problem is to find a scheduling policy that ensures schedulability of $\mathcal{J}'$. We address this problem by proposing a scheduling policy derived from the original one, but which allows a job to execute with two priorities: a nominal priority,

which is the one given by the original policy, and a dual priority. For every job $x \in \mathcal{J}'$, the dual priority of x is an arbitrary priority that is lower than the nominal priorities of all the other jobs. This is an application of Dual Priority (DP) scheduling [3] that ensures conservation of schedulability under *any* given shifting to the left of release times, which is achieved by setting parameters of the scheduling algorithm, instead of restricting release times.

At runtime, the dual priority is assigned to the part of x which is executed outside the execution interval of x in $\mathcal{J}$, while the nominal priority is assigned to the part which is executed inside the original execution interval of x. More precisely, when jobs in $\mathcal{J}'$ are executed, the DP scheduler sets the priority of job x_i to the dual value at time r'_i (the release time) and then upgrades the priority to the nominal value at time r_i. If $r'_i = r_i$, then only the nominal priority is used.

Theorem 4.2: If a job set $\mathcal{J}$ is schedulable by a job-level fixed-priority preemptive policy, then any job set $\mathcal{J}'$ obtained by shifting release times of jobs in $\mathcal{J}$ to the left is schedulable by the DP algorithm.

Proof: It can be readily seen that DP introduces no new preemption points. Hence, by Theorem 4.1, the modified system is DP-schedulable. ■

Note that DP can be used when the original scheduling is FP, where all priorities are known a priori. If the modified release times are statically determined, then the points in time where priorities are changed are statically defined. In this case, the DP scheduling policy proposed here offers the same predictability as FP.

Consider an application with mixed time-/event-triggered tasks where the periodic tasks are LET-based and scheduled by FP, and event-triggered tasks are executed in the background. More precisely, all event-triggered task have one low priority and they are executed in FIFO order of their arrival times. In this case, one can show that by introducing the release margins given by relation 1 for the LET-based tasks and by using a DP scheduler, response times of event-triggered tasks in the system may be decreased (and are never increased). We take the dual priority of each LET-based task as an arbitrary priority that is lower than the priority of the event-triggered tasks. For simplicity, assume a suitable buffering of pending events such that every event is eventually processed.

Theorem 4.3: In a mixed time-/event-triggered systems where event-triggered tasks have a lower priority than LET-based tasks and are executed in FIFO order, the response times of event-triggered tasks under DP with flexible constraints are not larger than in the case of FP scheduling with classical constraints.

Proof: Since the priority of any event task is higher than the dual priority of any LET-task, if a LET-based task is in execution at a time t outside of the task's LET, then there is no pending request for an event-triggered task at t. Also, events are processed in the same order in both cases. Hence, no outside-LET execution can increase response times of event-triggered tasks. ■

The next section describes an application where response times of event tasks become significantly smaller when shifting release times to the left.

V. Application Example

The magnitude of improvement that can be achieved by using our approach is application dependant. While generic examples can be easily constructed to demonstrate the benefits described in Section I, it is important to consider real applications in this respect.

We present here an evaluation of the impact of using flexible execution constraints on the behavior of a common control application: the inverted pendulum control system. The top-level Simulink model, obtained from the official Matlab/Simulink Demo Library, is shown in Figure 6. The control application has one event-triggered and two time-triggered components, from which C code was generated with Matlab's Real Time Workshop. Then the software was optimized for memory and response time. The code was compiled for the NEC V850 processor, and the execution time estimator $a3$ from AbsInt [1] was used to obtain the relevant execution time information. A processor clock of 25MHz was assumed. The original application is scheduled with FP scheduling, with priorities given by the rate monotonic policy. The task properties are summarized in Table I, where the timing of the event task (the sensor task) is given in terms of the interval of inter-arrival times τ and relative deadline D. The table also shows the minimum and maximum execution times of the tasks: BCET and WCET, respectively.

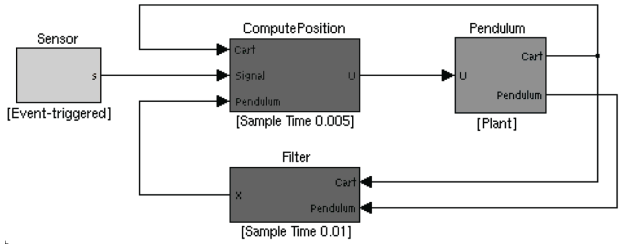
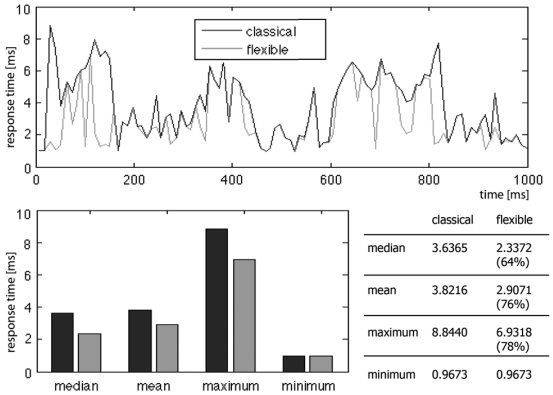


Fig. 6. The Simulink model of the inverted pendulum control system

TABLE I
Task Parameters

Property	Computation	Filter	Sensor
Priority	1	2	3
Timing(in ms)	LET = 4, ϕ = 0, π = 5	LET = 5, ϕ = 2, π = 5	$\tau = [10, 20]$, D = 8
BCET/WCET	1.18/2.1	1.95/2.05	1.53/1.75
δ (input ports access times)	0.5 (filter) 1.24 (sensor)	0.31 (any input)	

The timing of an execution was simulated over an interval of 100 seconds, with execution times of tasks and event arrival times chosen randomly with normal distribution between the given bounds, under an FP scheduling with standard execution bounds and under DP scheduling with flexible execution bounds given by relation 1. Figure 7 shows the response times of the event-triggered task in the simulation interval [0, 1000ms] for the two scheduling variants, as well as statistics for the entire simulation time.



	classical	flexible
median	3.6365	2.3372 (64%)
mean	3.8216	2.9071 (76%)
maximum	8.8440	6.9318 (78%)
minimum	0.9673	0.9673

Fig. 7. Event response times with classical and flexible constraints

The graph shows that response times with classical execution constraints are always higher than the response times with flexible execution constraints. For this application, the mean response times are decreased by 25%.

VI. Conclusions and Related Work

In this work we propose and analyze a more efficient way of scheduling LET-based tasks by taking into consideration execution times of tasks and dependencies between tasks. This approach enables a flexible execution of tasks without changing the observable system behavior as specified by the LET requirements. The main price that has to be paid is losing portability of the timed program. However, this is compensated by the possibility to fully automatize the compilation process. A compiler can take into account platform specific information to generate the release margins. If such information is not available, then the compiler yields the classical constraints.

Clearly, one can make an implementation more efficient by using more information about the platform and the application. The main concern of this paper is to point out the main sources of information and to explore the consequences of modifying the existing operational semantics of the LET model on the schedulability of the system. In this respect, we provide a sufficient condition for sustainability of preemptive scheduling with respect to release time changes. For FP scheduling, we give bounds on these changes for which the condition is satisfied. Moreover, we present a dual priority scheduling algorithm that guarantees schedulability of any variation of a system that is schedulable by a job-level fixed priority scheme.

Further work along the same lines needs to be done in relation to multi-mode LET systems, other scheduling algorithms, and other cases of mixed time-event triggered systems.

In related work, the authors of [8] briefly mention that a task can be released earlier than the LET if all its inputs are available, a concrete example being given in [9]. We add to that the observation that a task can be released even if its inputs are not available, if it is guaranteed that no input is accessed in the execution before the beginning of the LET. This is a property of the task in isolation and can be employed even if the task's environment is unpredictable. One can propose other, more intrusive approaches to increase processor utilization. For example, a task can be released earlier, execute initial code and then voluntarily suspend itself until the beginning of its LET. This is a form of cooperative scheduling [11].

Taking into account internal task structure has been shown to improve scheduling margins. A language where timing semantics is based solely on observable events was proposed in [4]. Observable events are send and receive actions. Timing requirements are placed only on the send and receive actions and the unobservable code can be transformed to support low-level tuning of the scheduling algorithm. A task is split such that code, which does not contain observable events is put into a separate task and can be scheduled later. More closely related to this paper is the study of sustainable scheduling presented in [2], which focuses on sustainability of scheduling tests. In our work, we address the weaker requirement that a task system deemed schedulable remains so with "better" parameters, in the context of preemptive policies and smaller release times. The Dual Priority approach was proposed in [3] as a way to increase system utility for fixed task-level priorities. Here, we specify the promotion times (relative times when priorities are upgraded) such that the LET I/O semantics and the schedulability of the system are preserved. The type of mixed time/event-triggered applications analyzed in Section IV-B and illustrated in Section VI are direct applications of DP, making use of three bands of priority levels.

References

[1] aiT worst-case execution time analyzers. http://www.absint.com/ait/, 2009.

[2] A. Burns and S. Baruah. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, 2(1):74–97, 2008.

[3] A. Burns and A. J. Wellings. Dual priority assignment: A practical method for increasing processor utilization, 1993.

[4] Richard Gerber and Seongsoo Hong. Semantics-based compiler transformations for enhanced schedulability. In *Proceedings IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1993.

[5] Arkadeb Ghosal, Tom Henzinger, Christoph Kirsch, and Marco Sanvido. Event-driven programming with logical execution times. In George Alur, Rajeev; Pappas, editor, *Proceedings of the 7th International Workshop, Hybrid Systems Computation and Control*, March 2004.

[6] Arkadeb Ghosal, Alberto Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger, and Daniel Iercan. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 132–141, New York, NY, USA, 2006. ACM.

[7] Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic. Composable code generation for distributed giotto. *SIGPLAN Not.*, 40(7):21–30, 2005.

[8] Tom Henzinger, Christoph Kirsch, Rupak Majumdar, and Slobodan Matic. Time-safety checking for embedded programs. In *Proceedings of the Second International Workshop on Embedded Software*. Lecture Notes in Computer Science, Springer-Verlag,, January 2002.

[9] Benjamin Horowitz. *Giotto : a time-triggered language for embedded programming*. PhD thesis, University of California, Berkeley, 2003.

[10] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[11] Stefan Poledna, Th. Mocken, J. Schiemann, and Th. Beck. Ercos: An operating system for automotive applications. Research Report 21/1996, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1996.

[12] Wolfgang Pree and Josef Templ. Modeling with the timing definition language (TDL). *Model-Driven Development of Reliable Automotive Services: Second Automotive Software Workshop, ASWSD 2006, San Diego, CA, USA, March 15-17, 2006, Revised Selected Papers*, pages 133–144, 2008.