RePP Submission: Flexible Scheduling of Predictable Software with Logical Execution Time Constraints

Stefan Resmerita, Patricia Derler

Technical Report
September 7, 2009

C. Doppler Laboratory
Embedded Software Systems
University of Salzburg
Austria

RePP Submission: Flexible Scheduling of Predictable Software with Logical Execution Time Constraints

Stefan Resmerita and Patricia Derler

University of Salzburg, Austria
firstName.lastName@cs.uni-salzburg.at

Abstract. Various programming models for embedded, time-triggered software employ the logical execution time (LET) abstraction in order to achieve predictable timing behavior. 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 that are valid at the beginning of the LET and writes output values at the end of the LET. While existing approaches constrain scheduling of a task execution to the LET bounds, we present a more efficient alternative where tasks can be executed outside of the LET bounds without changing the observable behavior of the system. Our methodology uses detailed timing information about the embedded software and about the physical environment. This extends the class of systems that are schedulable under the LET constraints. Moreover, in mixed time-/event-triggered systems the relaxed scheduling constraints can decrease response times of event-triggered tasks.

1 Introduction

Developing complex real time applications requires programming disciplines that take into account suitable abstractions of execution and communication times, which are specific to execution platforms. The Giotto programming model [5] introduced the concept of logical execution time (LET) of a software component (a task), representing a fixed logical duration for one execution of the task. The LET specifies the real time instants when task inputs and outputs are updated for every execution of the task. A runtime system performs the I/O actions at the right times, and also dispatches the task for execution. The task should be executed such that it uses input values that are valid at the beginning of the LET and it must issue outputs to the runtime system before the end of the task's LET, when they will be made available to the task's environment.

The LET concept is used in Giotto successors such as TDL [6] and HTL [3]. Advantages of LET-based programming models are that the application development process benefits from platform independence and from a dual separation of concerns: timing versus functionality and reactivity versus scheduling [4]. Furthermore, the I/O behavior of the application is predictable.

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 the scheduling 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 the total physical computation of a task to take place within the LET interval. In other words, the beginning of LET is the release time of the task when the task becomes ready for execution, and the end of the LET forms the latest termination time of the task. Figure 1(a), which can be found in [2] and [6], illustrates the classical LET view with the I/O semantics and the scheduling constraints.

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 conservative scheduling constraints is a sufficient condition for schedulability of the application. Thus, if no schedule with these constraints can be found, the application is declared unschedulable. Note that these conservative constraints are platform independent and reflect a black-box view of a task, since they use no information about the task's internal structure. Yet the schedulability analysis uses these platform independent constraints in conjunction with platform specific information such as the scheduling policy and the WCET information of tasks which is usually obtained from a detailed analysis of the code and the target platform.

In this work, we propose deriving more relaxed scheduling constraints from the LET specifications, by using information about: (1) Internal structure of a task, (2) Execution times of parts of the tasks' code, (3) Scheduling policy of the target platform, and (4) Dynamics of input signals from the physical environment. We describe circumstances under which a task can be released for execution before the beginning of its LET and can be allowed to execute past the end of the LET, thus extending the time frame for scheduling the task's execution, while preserving the LET-based I/O semantics. This is schematically depicted in Figure 1(b). A task T can be released for execution earlier than the beginning of its LET if: (1) Each input port that is read by T before the start of the LET has been updated by the runtime system with the latest value, and (2)

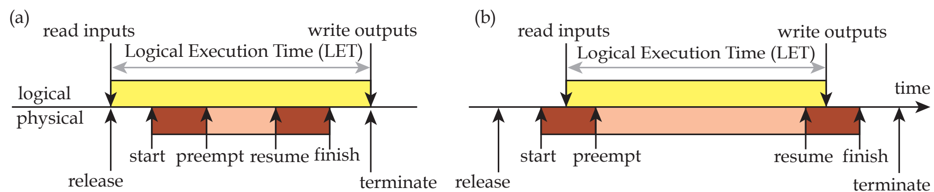

Fig. 1. Logical Execution Time, (a) LET with classical scheduling constraints, (b) LET with flexible scheduling constraints

The value of the corresponding input source does not change from the moment of the update until the beginning of LET. Similarly, a task's execution can be allowed to spill over the end of the LET if it is guaranteed that the task updates no output port during the spillover part.

By relaxing the scheduling constraints, the search space for feasible schedules is increased. In particular, an application that is declared unschedulable with the conservative constraints can become schedulable with the relaxed constraints. Moreover, a better processor utilization is achieved, where a time-triggered task can partially execute outside of its LET when the processor would be otherwise idle. Consequently, response times of low-priority event tasks decrease. Also, this reduces the risk of LET deadline violations in systems with high-priority event tasks that may preempt time-triggered tasks.

2 Using System Information for Enhanced Schedulability

The contents of a software task (at source code level, or assembly level) can reveal *syntactical* and *semantical* information that can be used to loosen the LET-based scheduling constraints. Some of this information is described in the sequel, with the help of a simple example involving one time-triggered task T with two inputs $i1$ and $i2$, and one output $o1$. When associating a LET to T in a programming model such as Giotto or TDL, the task is considered as an opaque component, or black-box, as illustrated in 2(a). The task implementation (source code or binary code) is used only to obtain its WCET. In our approach, we open up the task and extract more information, as follows. Let us consider two alternative implementations of T, schematically depicted in Figure 2(b) and Figure 2(c), where the input and output ports are implemented as global variables.

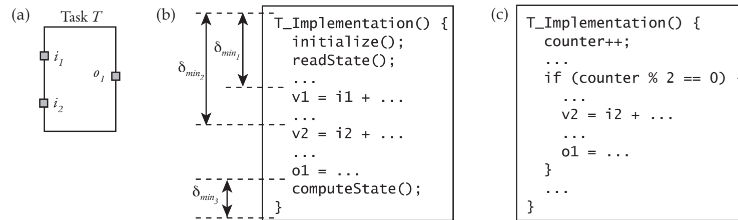

Fig. 2. A Task T with inputs and outputs. (a) shows the black-box view, (b) and (c) show possible implementations of T.

Syntactical information refers to lines of code where input and output ports are accessed. For example, some of the task's code is executed before accessing any input port (e.g. reset or initialization routines). Also, some code can be executed after all output ports are updated (e.g. computation of internal state). Using timing analysis, one can determine the shortest execution time between the beginning of the execution and the moment when an input port is accessed

in the code, as well as the shortest execution time between a last access to an output port and the end of execution. In Figure 2(b), δ_{min1} denotes the minimum execution time between the beginning of the task and any access to input ports (assuming that variable i_1 is the first input port to be used in any execution of the task), δ_{min2} is the minimum execution time between the beginning of the task and the first access to i_2, and δ_{min3} is the minimum execution time between the last access to any output port and the end of execution.

Semantical information reveals possible behaviors of the task's I/O activity. A typical example in this respect is illustrated in Figure 2(c), where the code of the *if* block is executed every second invocation of the task. If the ports i_1 and o_1 are accessed only within that block, then we know that i_1 need not be updated by the runtime system in invocations $1, 3, 5, \ldots$ of the task, and that o_1 has a constant value during two periods of the task.

Timing predictability of inputs means that input values of a task may be known in advance over a time window. In general, we distinguish among two types of predictable input sources for time-triggered tasks in a LET-based system: Time-triggered and event-triggered tasks. Since the I/O behavior of all time-triggered tasks is specified by their LETs, the outputs of a time-triggered tasks are unchanged between two consecutive LET ends. An event usually represents a value change of a physical parameter, detected by an active sensor. Control applications are designed based on a model of the physical process to be controlled. The model contains information about real-time evolution of physical parameters, which could be used to predict events. In particular, we are interested in the minimum period of time during which a sensor value stays unchanged in the system. 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 any two consecutive changes in the sensor's output.

3 Example

To obtain larger scheduling bounds, one should aggregate information from different sources of predictability, as described above. We illustrate this aggregation process in an example of an embedded application with two time-triggered tasks T_1, T_2 and an event-triggered task E. For this example, we will determine release times for the task T_1 that precede the beginning of T_1's LET, such that T_1's LET-based I/O semantics is preserved. Figure 3(b) shows a time window with the placement of LETs for the time-triggered tasks and two executions of the event task. The implementation of T_1 is outlined in Figure 3(a), where T_1 first reads an input from E via the input port p_E, then from T_2 via p_2 and later from a sensor via p_s. For an input port p of task T_1, we denote by $\delta_{min}(T_1, p)$ the minimum execution time of T_1's code segment from the beginning until the place where p is accessed for the first time. We assume that there exists a minimum time period between any two consecutive executions of E, also called the

minimum inter-arrival time of E, denoted by δ_E. We use $t_{Ls}(T_1)$ to represent the beginning of T_1's LET shown in the figure and $t_{Le}(T_2)$ for the end of T_2's LET which precedes $t_{Ls}(T_1)$.

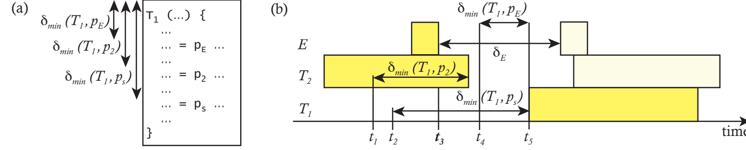

Fig. 3. T_1 is released before the start of the LET

Let us consider the sources of T_1's inputs. We assume that p_s is connected to an unpredictable sensor, so one cannot predict the value on p_s at time $t_{Ls}(T_1)$ ahead of time. Consequently, p_s must be updated by the runtime system at $t_{Ls}(T_1)$ and T_1 must not read the variable p_s earlier. This is satisfied if T_1 begins execution at any time in the interval $(t_2, t_{Ls}(T_1)]$, where $t_2 = t_{Ls}(T_1) - \delta_{min}(T_1, p_s)$.The source of p_2 is time-predictable: it is updated only at the end of T_2's LET, thus a correct execution of T_1 must not read from p_2 before $t_{Le}(T_2)$. The earliest time at which T_1 can be released under this condition is $t_1 = t_{Le}(T_2) - \delta_{min}(T_1, p_s)$. The port p_E is updated by the event-triggered task E at times which cannot be statically predicted. The earliest release time for T_1 which guarantees that T_1 reads from p_E at or after $t_{Ls}(T_1)$ is $t_4 = t_{Ls}(T_1) - \delta_{min}(T_1, p_E)$. It follows that the earliest static release time for T_1 that ensures a correct LET-based I/O operation is $t_r(T_1) = max\{t_1, t_2, t_4\} = t_4$. The value of $t_r(T_1)$ can be decreased by using a dynamic scheduling procedure. Assume that $t_3 + \delta_E > t_{Ls}(T_1)$ meaning that p_E remains unchanged between t_3 and $t_{Ls}(T_1)$. Hence the earliest release time for T_1 can be determined by the dynamic scheduler at time t_3, as $t_r(T_1) = max\{t_1, t_2, t_3\}$.

4 Discussion

The relaxed scheduling constraints have to be evaluated in conjunction with the scheduler used by the operating system on the target platform to make sure that the parts of tasks which can be executed outside of the corresponding LETs are in fact executed when the processor would otherwise be idle. For example, when using a priority-based preemptive scheduler, the priority of a task can be set to be lower than the normal priority of any task in the system for the time intervals the task is executing outside the LET bounds. This will ensure that the code executed outside of the LET will not preempt or delay any other regular execution of a time- or event-triggered task.

Information about task internal structure and inter-task dependencies has been applied to general scheduling problems. In [1], Gerber and Hong present a language where timing semantics is based solely on observable events, which are

I/O send and receive actions. Tasks are split such that the part of code that does not contain observable events (send and receive actions) is put into a separate task which can be scheduled later. We also consider code starting code earlier than stated in the timing specification. This means using shortest execution time of code segments, while the vast majority of scheduling research considers worst case (longest) execution time analysis.

While the relaxed constraints provide an increased scheduling flexibility, they may weaken a compositionality property of the system. Using the classical scheduling constraints, a system is still schedulable if a new task is added to an existing system with a LET that does not overlap with existing tasks LETs. This guarantee cannot be given with the scheduling constraints described in this work, since tasks can be executed outside of their LETs. However, notice that in any situation in which such a new task cannot be added to a given system, this implies that the original system is not schedulable with the classical constraints. Note that the classical requirement $ET \leq WCET \leq LET$ is not necessary with the relaxed scheduling constraints, where a task is schedulable even if it's WCET is bigger than its LET. Another potential disadvantage of the proposed constraints is that the static schedule of actions that must be executed by the runtime system is platform-specific. In the classical case, the time steps of timing instructions depend only on the LET specifications. This can be easily overcome by using a tool to automatically generate the relaxed constraints.

Further work on this topic involves deriving the precise formulas for relaxed execution bounds, and proving various properties of the system as consequences of using these bounds. We plan to evaluate the benefits and costs of using our approach on several relevant examples of embedded applications and provide an implementation of the new scheduling.

References

1. Richard Gerber and Seongsoo Hong. Semantics-based compiler transformations for enhanced schedulability. In *Proceedings IEEE Real-Time Systems Symposium*, pages 232–242. IEEE Computer Society Press, 1993.
2. 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.
3. 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.
4. T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, and W. Pree. From control models to real-time code using giotto. *Control Systems Magazine, IEEE*, 23(1):50–64, 2003.
5. Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic. Composable code generation for distributed giotto. *SIGPLAN Not.*, 40(7):21–30, 2005.
6. Wolfgang Pree and Josef Templ. Modeling with the timing definition language (TDL). *Second Automotive Software Workshop, ASWSD 2006, San Diego, CA, USA*, pages 133–144, 2008.