Timing-Driven HW/SW Codesign Based on Task Structuring and Process Timing Simulation*

Dinesh Ramanathan†
Dept. of Info. & Computer Sci.
University of California
Irvine, CA 92697
dinesh@ics.uci.edu

Ali Dasdan
Dept. of Computer Sci.
University of Illinois
Urbana, IL 61801
dasdan@cs.uiuc.edu

Rajesh Gupta
Dept. of Info. & Computer Sci.
University of California
Irvine, CA 92697
rgupta@ics.uci.edu

Abstract

Task structuring is the process of determining the individual tasks of a system, leading to the system's description as a task graph. This paper shows that RADHA-RATAN, our rate derivation algorithms, can be used to validate various tradeoffs made during task structuring, making this step timing aware. We show how RADHA-RATAN enables construction of a high-level timing model of the system leading to a *process timing simulation* of the entire system. An interesting aspect of process timing simulation is that it provides the ability to observe system level timing behavior based on timing requirements and analysis *before* an implementation of the tasks has been carried out. Based on task structuring and process timing simulation we propose a codesign methodology by which a system designer can gain insight into the system's timing performance. This approach enables the designer to reduce expensive timing driven design iterations. We have implemented this methodology in the RADHA-RATAN framework. We illustrate its application by an example.

1 Introduction

Timing plays an important role in the design of embedded systems. Unfortunately, the problem of designing a temporally correct system is a difficult one, and the current practice for this problem is ad hoc; it is based on trial and error, guided by engineering experience [7]. Moreover, the emphasis is usually on designing a functionally correct system [8]. The temporal correctness of the system will usually be checked after the system's components are integrated and the resulting system's functional correctness is ensured. This approach usually results in expensive re-design iterations in order to satisfy temporal constraints.

This paper proposes a methodology by which a designer can gain insight into the system's timing performance by simulating and validating timing tradeoffs at very high levels of abstraction. This enables the designer to reduce expensive timing driven design iterations. This methodology is based on the notion of process timing simulation that involves simulation of the system's timing behavior. This simulation is based on rate derivation and task structuring. Rate derivation and related problems have been discussed in detail in [4, 5, 6]. In this paper, we focus on task structuring and process timing simulation.

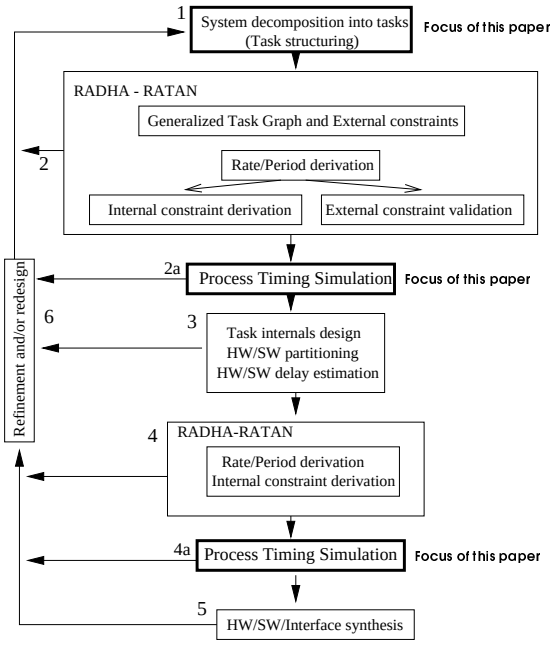

Figure 1: *Timing-driven HW/SW codesign methodology.*

Figure 1 shows the timing-driven HW/SW codesign methodology which builds upon earlier work in the area, e.g., POLIS [1]. We begin with task structuring (Step 1).

Task structuring is the phase where we define the system's tasks and communication between them. The resulting group of interrelated functionalities (modeled as tasks) are then subject to timing constraint analysis in Step 2 using RADHA-RATAN [4, 5]. The objective of RADHA-RATAN analysis is to validate external user imposed rates and user imposed internal rate constraints, thereby deriving timing budgets for the tasks. While the scope of this analysis is large, there are cases when this analysis needs to be *augmented* by a simulation of the timing behavior. To enable the simulation, in step 2a, our tool automatically emits the individual tasks in a hardware description language (HDL) or a programming language, both at the behavioral level. Each task has been transformed into its equivalent form in the HDL. The HDL form does not have the internal functionality of the task described, since this is not known to our tool at this stage. Instead, the HDL form captures the tasks' interactions with its environment based on the

*The authors would like to acknowledge support from NSF award numbers MIP 95-01615 (CAREER) and CCR-9806898, from DARPA DABT63-98-C-0045, and the UC's MICRO program.

†This work was done while this author was at Synopsys Inc. He is presently with C2 Design Automation Inc.

task graph model, and the internal functionality is modeled by an empty function. We then perform a process timing simulation of the system and observe its timing behavior, using the HDL as a vehicle to perform simulation.

In Step 3, the system's tasks are refined to include the functionality of the task in the HDL form obtained from Step 2a. This step actually corresponds to the first step in many existing codesign methodologies. In Step 3, the task's implementation platforms are also determined by partitioning them into hardware and software tasks. In Step 4, RADHA-RATAN is performed again on the generalized task graph, since we now know more about the delays within the system. In Step 4a, we again emit the individual tasks in HDL and validate the timing behavior of the system by performing another process timing simulation. This time the simulation provides a more timing accurate behavior of the system, since the tasks' functionality and delays have been refined by Step 4. In Step 5, all the tasks are synthesized in their respective implementation platforms together with the interfaces between these platforms. This step is explained well in [1, 2, 11]. Finally, Step 6 is needed to perform modifications on the system in case there are functional or temporal constraint violations.

This paper is organized as follows. We present a brief review of our task graph model and rate derivation in section 2. In section 3, we present task structuring and illustrate it with the dashboard controller example taken from [1]. In section 4, we introduce process timing simulation and illustrate it with two simulation scenarios applied to the example. Section 5 concludes with a summary of our contributions.

2 Generalized Task Graph Model and RADHA-RATAN

We model an embedded system using a directed graph called a *generalized task graph* [3, 4, 5] (the task graph for short). It corresponds to the system's data/control flow diagram. Each node represents a task, and each edge represents a unidirectional communication channel between its *producer* and *consumer*. The sensors and actuators of the system are also included in its task graph, in which they are referred to as *input* and *output tasks*, respectively. We assume that each task is periodic or sporadic, i.e., each task has a bounded rate interval. The *rate* of a task is equal to the number of its executions per unit time such that it executes only once during its *period*. There have been other approaches to model embedded systems, e.g. the SPI model [13].

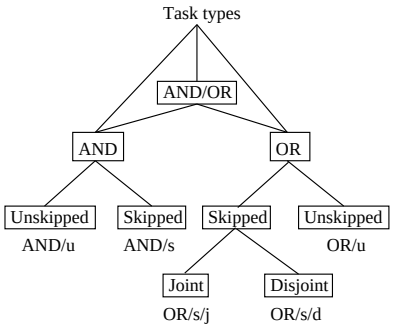

Figure 2: *Task classification.*

Both data and control flow through a channel are modeled using token flow. The granularity of every token is *channel-specific*, and once fixed, the tokens are indistinguishable within and across channels for timing analysis purposes. A task is *enabled* when it is ready to run. We classify the tasks in the task graph as depicted in Figure 2. We use the following criteria: (i) An AND task waits for all its predecessors to get enabled, whereas an OR task needs only one predecessor. An AND/OR task has both AND and OR behavior. (ii) A *skipped* task may sometimes allow intentional loss of input tokens whereas an *unskipped* task does not. Both AND and OR tasks can have these behavior. (iii) An OR/unskipped task reads from all its predecessors before completion whereas an OR/skipped task may skip some or all of the predecessors other than the enabling one. In particular, an OR/skipped/joint (OR/s/d) task skips all of the predecessors other than the enabling one whereas an OR/skipped/disjoint task uses all of the tokens to start a new execution after every reading.

The user inputs a generalized task graph along with the token consumption and production rates for the actuators of the system being modeled using our tool's GUI. RADHA-RATAN first derives the internal rate constraints of a system using its external rate constraints. It then uses these rate constraints to derive and validate the remaining timing constraints (provided by the user) of the system. Thus, RADHA-RATAN contains both derivation and validation algorithms for both rate and separation constraints. Our earlier works in [4, 3, 12, 5, 6] explain these algorithms in detail. Note that the rate of a task is its frequency of its execution, and its period is the reciprocal of its rate. Due to this relationship between the rate and the period, we will use them interchangeably in the sequel. We will use T for a given period, RT for a required period, and DT for a derived period.

3 Timing-driven Task Structuring

Task structuring is the phase where we define the system's tasks and communication between them. It changes the granularity of the data flow/control flow diagram by successive refinement into smaller tasks.

Task structuring reduces the complexity of the system's description by grouping "related" functions into the same task using structuring criteria which are very well explained in [8, 9]. They also address whether and how transformations should be grouped into concurrent tasks, e.g., grouping functionally related transformations into the same task or those that execute sequentially into the same task. Task structuring criteria typically address how transformations for physical device I/O, e.g., defining a task to poll a passive I/O device, and those for internal transformations, e.g., defining a task for a periodic transformation or a task for user interaction, are mapped to tasks.

Unfortunately, the current practice for task structuring is *not* timing-driven: timing decisions during task structuring are made in an ad hoc manner. Moreover, the timing decisions are not validated as task structuring decisions are being made. While task structuring itself seems difficult to automate due to large number of design decisions, timing analysis during task structuring can be made more systematic and rigorous. One related approach is given by Hou and Wolf in [10]. Their approach presents a heuristic algorithm to partition the real-time software system's data flow graph into a task graph under deadline and size constraints. This approach assumes that it is possible to determine the cost of a process in terms of its execution time and monetary value before partitioning. It is not useful at the level where task structuring is performed since task structuring is performed before the internals of tasks are designed.

In our methodology, task structuring can be made timing driven by performing analysis and validation of timing tradeoffs made by system designers. Consider task a that needs refinement by the system designer. The designer is presented with two choices: (a) to break a into two sequential tasks b and c or (b) into two parallel tasks b and c. In the first case, the designer increases the latency and decreases the throughput of the system. In the second case, the designer decreases the latency and increases the throughput of the system. However, the designer does not have a quantitative measure of change in latency and throughput of the *entire* system since task a could have been interacting with several other tasks of the system.

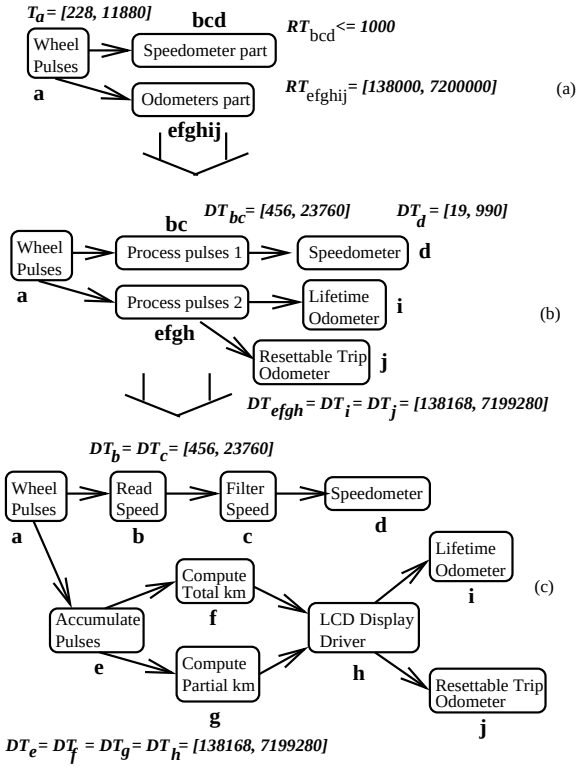

Figure 3: *Task structuring for the dashboard controller in three phases (labeled a-c). The labels for the period intervals are T for given, RT for required, and DT for derived period intervals.*

This is crucial to the designer since it effects design decisions further down the design process. Using RADHA-RATAN on the transformed task graph, the designer can make quantitative assessments about the timing behavior of the system. If task a was not along the critical path the designer may choose to split task a as in case (a) as this may ease the implementation of tasks b and c. On the other hand, if task a was timing critical, the designer may choose the parallel route for tasks b and c. In either case, RADHA-RATAN allows the designer to make high level timing driven decisions during task structuring by providing timing tradeoff information for various system design choices.

3.1 Example of Task Structuring

Consider the design of a dashboard controller also used in [1]. The controller contains two parts: the speedometer part and the odometers' part. There is one speedometer and two odometers: the lifetime odometer and the trip odometer. The speedometer registers vehicle speed in the range of 0-260 km/h where any speed value less than 5 km/h is regarded as zero. The odometers register distance traveled at increments of 0.1 km starting from 0 km. The trip odometer can sometimes be reset by the driver. The dashboard controller gets four pulses from the sensors placed on one of the wheel shafts so that every rotation of the corresponding tire produces four pulses, each of which corresponds to 1/4 of a rotation. The tire travels 0.66 meter per rotation.

The timing constraints on the dashboard controller are all in the form of rate constraints. The speedometer consists of two coils and a magnetized needle. It must get its inputs at a rate of at least 100 Hz to drive these coils. The odometers must be so fast that the time it takes to display an distance increment of 0.1 km must be less than the time it takes for the car to travel 0.1 km.

We now illustrate timing-driven task structuring as shown in Figure 3. These graphs represent a *functional* structuring of the system that a designer would create.

3.2 Initial task structuring phase

Consider Figure 3(a). RADHA-RATAN requires that the rate of task a be provided since this rate is an input to the system. Task a generates a pulse for every 1/4 rotation of the tire, which means a pulse for every 0.66/4 m. As the speed of the car ranges between 5 km/h and 260 km/h, the separation between the times of two consecutive pulses ranges from 2.28 ms to 118.80 ms. Hence, the integral period interval for task a is $T_a = [228, 11880]$, the unit of which is 100 ms. This unit will be used for all the period intervals in the sequel.

The period intervals for the other tasks in Figure 3(a) are as follows. The speedometer part must read at least two pulses from task a to compute the speed of the car; so we derive the period interval of $DT_{bcd} = 2 * T_a = [456, 23760]$. The odometer part needs at least $\lfloor 0.1\,\text{km}/(0.66/4\,\text{m}) \rfloor = 606$ pulses to register a distance increment of 0.1 km that the car travels. Hence, for the odometer part, we derive a period interval of $DT_{efghij} = 606 * T_a = [138168, 7199280]$.

When we validate the resulting period intervals, we see that the odometer part satisfies its rate requirement because it takes at least (0.1 km)/(260 km/h)=1.38 s and at most (0.1 km)/(5 km/h)= 72 s for the car to travel 0.1 km, i.e., the rate requirement is $RT_{efghij} = [138000, 7200000]$. We see, however, that the speedometer part violates its rate requirement of 100 Hz, which means that $RT_{bcd} \leq 1000$.

3.3 Refinement phase

We go from (a) to (b) in Figure 3 for two reasons: (1) there is a rate violation in the speedometer part, and (2) each part of the system has a rather large granularity. The refinement consists of exposing the speedometer and the odometers. They are connected to task a, the input task of the system, by two generic tasks called "process pulses".

We know the rate of task a and the rate requirements on the output tasks, the speedometer and the odometers. We use them to derive a period interval for all the tasks in the task graph as follows. The period interval for the "process pulses 2" task is the same as the one we derived for task $efghij$, i.e., $DT_{efgh} = [138168, 7199280]$, because the same reasoning applies. Similarly, the period interval for the "process pulses 1" task is $DT_{bc} = [456, 23760]$. However, the rate requirement on the speedometer leads to a period interval of $DT_c = [19, 990]$ for the speedometer so that it can satisfy its rate requirement. This period interval means that task bc must produce 24 speed values for the speedometer for every change in the car's speed. Later, we will see that this in fact enables a smoother movement of the speedometer's needle. Finally, the period interval for each odometer is the same as that of task $efgh$ because each odometer displays every distance value sent by task $efgh$ as soon as they arrive. We now see that the output tasks, thereby the system, satisfy their rate requirements.

3.4 Final structuring phase

The task graph in (b) is not the final one since we should further refine the rather generic "process pulses" tasks. This leads to the task graph (c) in Figure 3. This task graph is actually the one given for the dashboard controller in [1]. The structuring of the "process pulses" tasks is the choice of the original designers of the dashboard controller, so we will not question it; we will however derive the rates of all the tasks in the task graph and check to see if the rate requirements are still satisfied.

In the speedometer part, the "process pulses 1" task is structured into tasks b and c. Task b reads the pulses and computes the speed of the car whereas task c reads one speed value and

produces 24 speed values between the current one and the previous one. For example, if the speed changes from 30 to 54, the speedometer will get 24 values, 31, 32, ..., 54. This is done to smooth out the movement of the speedometer's needle. We now see that the derived period intervals for tasks b and c are the same as that of the "process pulses 1" task.

In the odometer part, the "process pulses 2" task is structured into tasks e, f, g, and h. This is done in such a way that the distance traveled to display on the odometers is computed by different tasks, one task for each odometer. Task h combines the distances computed and sends them to the respective odometers. Since each of these tasks gets one value and produces one value, the derived period intervals for each task is the same and is equal to that of the "process pulses 2" task.

In [1], the period interval of task b is fixed at 250 ms. Fixing the period interval means that task b will read different number of pulses from task a to compute the speed. This does not violate the functionality; it may, however, lead to average rather than instantaneous speed values in short intervals. Using this fixed value, we can see that the new derived period (interval) for task c is also 250 ms. The period intervals for the other tasks do not change.

We can now validate the rate requirements of the system. Since the rates of the output tasks have not been changed by the final structuring phase, the rate requirements are still satisfied. The system does not have any response time requirements, but if it had, we could also validate them. The importance of validations in this particular example stems from the fact that each structuring phase increases the response time of the system.

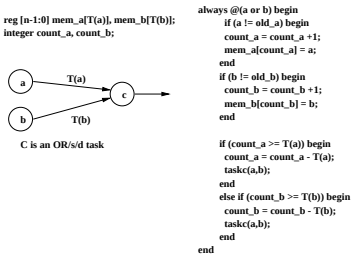

Figure 4: *Verilog HDL for task c that is an OR/s/d task. Verilog task, taskc, captures the functionality of the task. Initially, $old\_a = a$, $old\_b = b$ and $count\_a = count\_b = 0$.*

In the above task structuring phases, RADHA-RATAN only expects to know the rate of the input task, task a, and some important details related to the functionality of the tasks, such as there must be at least two pulses to compute a distance value and thereby a speed value. We should mention that RADHA-RATAN handles all the other derivations and validations automatically. The derivations and validations in the above discussion, which may seem to be carried out manually by the designers, are to illustrate how RADHA-RATAN works internally.

4 Timing Driven Design Exploration

RADHA-RATAN derives a time budget for each task in the task graph. Hence, we know the timing constraints that each task should obey. Also, from the task graph, we know the input/output characteristics of each task in terms of token usage. These two can be combined to define a high-level process timing model for the system in order to perform a high-level simulation of the system to evaluate its performance characteristics, which we call *process timing simulation*. For timing simulation, we represent the high-level process timing model of the tasks of the system using Verilog HDL. In contrast to functional simulation, process timing simulation does not assume knowledge of task internals. Functional simulation is performed only to present valid internal values while for process timing simulation, the contents of tokens used to communicate between tasks do not matter.

4.1 Transformation to Verilog HDL

The generalized task graph classifies its tasks according to their behavior upon receiving tokens from their predecessors as in Figure 4. Using this classification, we capture the *functionality* of the task into a Verilog task. Note that this functionality may not be known in the very early stages of the design cycle, and may result in an empty Verilog task. Our tool provides a "token protocol layer" around the functionality of the task that triggers the functionality (which may be an empty Verilog task, when the implementation of the functionality is not known) based on the tokens received from its predecessors. Figure 4 shows an example of such a protocol layer for an OR/s/d task. Our tool automatically generates the Verilog code from the input that the user has provided: the generalized task graph and the interaction of the tasks using tokens. Using this generated code as a template, the user can then refine the functionality of the task by providing the implementation details of the task.

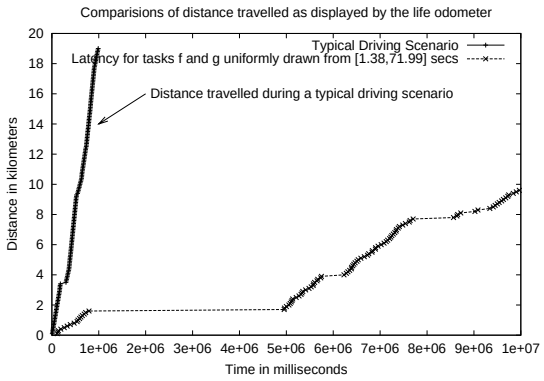

Figure 5: *Comparisons of distance traveled by the car as displayed by the life odometer under a typical driving scenario and when we pick the latency of tasks f and g uniformly from the interval* $[1.38, 71.99]$. *The fewer number of data points indicates that we have token loss, and the distance traveled is incorrect when we reduce the latencies of tasks f and g.*

We now describe how we can transform a typical task to Verilog HDL for process timing simulation. Recall that a task needs enabling tokens to get enabled. These tokens are generated by the task's predecessors. For example, only one token sent from the "read speed" task to the "filter speed" task in the dashboard controller is enough to enable the latter task. However, we do not exactly know how many tokens the "read speed" task needs to get enabled but we know that it should get enabled every 250 ms. This has been obtained by running RADHA-RATAN during Step 2 or Step 4 of our codesign methodology. We can achieve this behavior using a timer (a clock in the context of Verilog HDL). The timer can also be represented as a task in the task graph so it does not disrupt the task graph semantics.

Now, consider the "filter speed" task. We know its functionality in that we know it gets enabled by one token from the "read speed" task and generates 25 tokens for the speedometer, one every 10 ms. The only detail that we do *not* know is how the "filter speed" task actually filters the speed which means we do *not* know the internal details of the task filter speed. Therefore, the token sent from the "filter speed" task to the speedometer does not have a valid and filtered speed value as would be the case in functional simulation. The key issue in process timing simulation is the fact that the token is communicated between these tasks under the required timing constraints.

4.2 Timing simulation scenarios

A time budget for a task is a period interval and it gives the smallest and largest period possible for the task. For example, the "wheel pulses" task has a period interval of $T_a = [2.28, 118.80]$ ms, which is derived by knowing that the vehicle travels at a speed in the range of 5-260 km/h. The rate derivation algorithm derives the period interval for the other tasks using T_a.

To understand the timing behavior under typical operating conditions, simulation is necessary. Assume that under a typical driving scenario the vehicle's driver decides between acceleration, deceleration, and changes speed in a uniformly distributed manner. We assume the acceleration and deceleration periods are normally distributed with a mean of 20 s and a standard deviation of 1 s. We assume the acceleration or deceleration of the vehicle is such that it takes 10 s for the vehicle to have a speed increment of 100 km/h. These are normally distributed such that the time to get this speed increment has a mean of 10 s and a standard deviation of 4 s. We assume the vehicle will not accelerate and decelerate constantly. We also assume that it will not change its speed for twice as long as the acceleration/deceleration period. We prevent the car going above 80 km/h. These restrictions are placed in a test bench file that drives the generation of the wheel pulses. We now describe two scenarios with these simulation parameters.

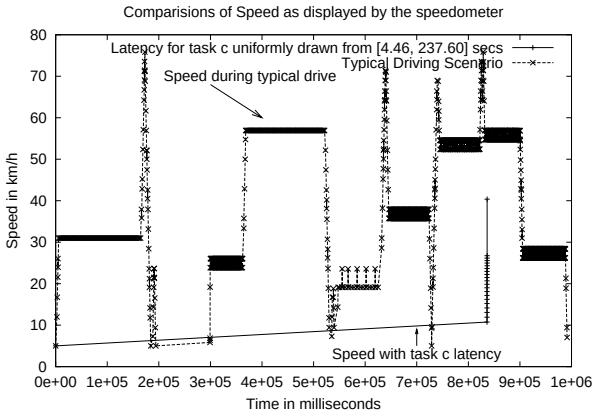

Figure 6: *Plot of the speed of the car when we pick the latency of task c from a uniform distribution in the interval* [4.46, 237.60] *s versus the speed during a typical driving scenario. The token loss is indicated by the fact the we have lesser data points and the displayed speed is incorrect when we vary the latency of task c. If there were no token loss, we would expect identical speed values in both cases.*

Case 1: In this scenario, we wish to determine the rate at which tasks f and g should run so that we can ascertain their latencies. From our analysis results, we know that tasks f and g will have a period interval of $[1.38, 71.990]$ s, so their latency should be at most 1.38 s. Once the latencies have been determined, we can impose a timing requirement on the specific implementation of tasks f and g. To validate the latencies with timing simulation, we chose the latency of tasks f and g such that they are uniformly distributed within the interval $[1.38, 71.990]$.

A timing simulation of the system, with these parameters for tasks f and g, reveals that these tasks are slower than required. The system drops almost 1/4 of the tokens that should reach the odometers. Figure 5 illustrates this scenario: it displays the number of tokens that get registered at the life and trip odometers under a typical driving scenario (presented above) and when the latency of tasks f and g are drawn uniformly from the interval $[1.38, 71.99]$. This shows that the latency of both tasks has to be 1.38 s, otherwise we will drop tokens and report the distance traveled incorrectly (Note that the distances traveled as reported by the life odometer is different).

Case 2: From our analysis results, we know that task c will have a period of 250 ms. Since it may take at most 237.6 s for task b to produce a token for task c, any latency of less than 237.6 s for task c will result in loss of tokens. In order to validate this, we draw the latency of task c uniformly from $[4.46, 237.60]$ s, and see that the system drops many tokens. Figure 6 illustrates this scenario, and compares it with the timing simulation of the dashboard controller under a typical driving scenario.

From these examples it is clear that multiple system design scenarios can explored via simulation using rate derivation and task structuring. In this way, a designer can get a preliminary idea of the system's performance and validate the system's timing prior to detailed implementation. Further, detailed implementation can now be timing driven as well as shown in the first scenario.

5 Summary

We proposed a timing driven codesign methodology based on high-level timing constraint derivation and validation (RADHA-RATAN). This is applied to task structuring and enables process timing simulation during the design of embedded real-time systems. We show that this tool can be used for systematic treatment of timing issues during task structuring. Using the timing bounds obtained from RADHA-RATAN, we can perform process timing simulation of the system and explore various timing dependent system design choices.

References

[1] Balarin, F., et al. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach.* Kluwer Academic Publ., Boston, MA, USA, 1997.

[2] Chou, P., Walkup, E. A., and Borriello, G. Scheduling for reactive real-time systems. *IEEE Micro* (Aug. 1994), 37–47.

[3] Dasdan, A., Ramanathan, D., and Gupta, R. K. A timing-driven design and validation methodology for embedded real-time systems. *ACM Trans. on Design Automation of Electronic Systems. 3*, 4 (Oct. 1998).

[4] Dasdan, A., Ramanathan, D., and Gupta, R. K. Rate derivation and its applications to reactive, real-time embedded systems. In *Proc. the 35th Design Automation Conf.* (1998), pp. 263–268.

[5] Ramanathan, D., Dasdan, A., and Gupta, R. K. High-Level Modeling of Communication in Real-Time Embedded Systems. *IEEE High-Level Design Validation and Test Workshop*, Nov 1998, pp. 172–180.

[6] Dasdan, A. Timing Analysis of Embedded Real-Time Systems. Ph.D Thesis, University of Illinois at Urbana-Champaign, 1998.

[7] Gerber, R., Hong, S., and Saksena, M. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Trans. Software Eng. 21*, 7 (July 1995), 579–92.

[8] Gomaa, H. *Software Design Methods for Concurrent and Real-Time Systems.* Addison-Wesley, Reading, MA, USA, 1993.

[9] Hatley, D. J., and Pirbhai, I. A. *Strategies for Real-Time System Specification.* Dorset House, New York, NY, USA, 1987.

[10] Hou, J., and Wolf, W. Process partitioning for distributed embedded systems. In *Proc. Int. Wrkshp on HW/SW Codesign.* (1996), IEEE, pp. 70–76.

[11] Ku, D., and Micheli, G. D. *High Level Synthesis of ASICs Under Timing and Synchronization Constraints.* Kluwer Academic Publ., Boston, MA, USA, 1992.

[12] Mathur, A., Dasdan, A., and Gupta, R. K. Rate analysis of embedded systems. *ACM Trans. on Design Automation of Electronic Systems 3*, 3 (July 1998).

[13] Dirk Ziegenbein, and Rolf Ernst A Framework for High-Level Performance Validation of Embedded HW/SW Systems *IEEE International Workshop on High-Level Design Validation and Test*, Nov 1998.