Prototyping Hierarchically Scheduled Systems using Task Automata and TIMES

Mikael Åsberg, Thomas Nolte and Paul Pettersson
MRTC/Mälardalen University
P.O. Box 883, SE-721 23 Västerås, Sweden
{mikael.asberg, thomas.nolte, paul.pettersson}@mdh.se

***Abstract*—In hierarchical scheduling, a system is organized into multiple levels of individually scheduled subsystems (hierarchical scheduling tree), which provides several benefits for developers including possibilities for parallel development of subsystems. In this paper, we study how the model of task automata and the Times tool can be applied to provide support for rapid and early prototyping of hierarchically scheduled embedded systems.**

As a main result, we show how a single node, in an arbitrary level in a hierarchical scheduling tree (scheduled with fixed-priority preemptive scheduling), can easily be analyzed in Times by replacing all interfering nodes with a small set of higher priority (dummy) tasks. We show with an algorithm how these dummy tasks are generated (including task-parameters such as period, offset etc.). Further, we generate executable source code, with the Times code-generator, that emulates the scheduling environment (with our dummy tasks), i.e., the hierarchical scheduling tree and all of its preemptions, of a small example system. Yet another contribution is that we transform the generated (brickOS) source code to run on an industrial oriented platform (VxWorks), and conduct an performance evaluation.

I. Introduction

Ever increasing global competitiveness and demands of shortened time-to-market has increased pressure on rapid development of embedded software systems. A key component in being a fast player in developing embedded software systems, is to be able to do analysis and prototyping early in the development of these systems.

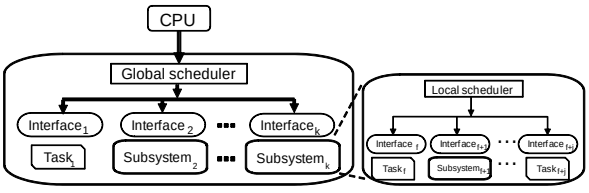

Fig. 1. Hierarchical scheduling

In recent years, hierarchical scheduling techniques have been introduced as a means to simplify parallel development of embedded software systems. Hierarchical scheduling simplifies integration of such systems, by providing mechanisms for temporal isolation between system parts, called subsystems. Essentially, a system comprises a number of subsystems that typically implement a particular function or feature of the whole system. Imagine a car where one subsystem can be the engine control system, and another could be the anti-lock braking system. These subsystems should ideally be developed in parallel, and, at the time of integration, no integration related problems should occur [1]. One such integration related problem is a piece of software requiring more time to execute than originally intended, and therefore causes unforseen interference with the rest of the system. Another example is the introduction of new software pieces needed to realize the function of a subsystem, not apparent at early design. Hierarchical scheduling makes sure that no unpredictable interference will happen in the time domain, hence, allowing for timing analysis of subsystems in isolation, before integration. Figure 1 illustrates a scheduling tree. We define the top node as the *Global scheduler*, it is responsible for distributing the the whole *CPU* resource to the second layer in the scheduling tree. Any node can be either a *Subsystem* or a *Task* (except for the top node which is a scheduler). In this manner, any node schedules its nodes with its *Local scheduler*. All nodes have an *Interface* (scheduling parameters) which specifies the amount of CPU resource that the node has access to. The overlaying scheduler uses these interfaces to schedule its nodes.

In the ideal case, it is desirable to be able to perform detailed analysis of a subsystem's functional and non-functional properties by looking at the subsystem in isolation, i.e., without requiring details of all subsystems in the system. In particular at an early stage in the construction of a system, it may be hard to get access to such details. Our proposed approach makes it easy to analyze task schedulability with respect to its subsystem interface and generate source code (for our target platform VxWorks) that will emulate a subsystem running together with other subsystems/tasks, i.e., the subsystem (and its tasks) schedule will behave as if it would be executed together with the rest of the subsystem tree (early prototyping/testing). All that is required are the interfaces of the other subsystems/tasks, i.e., no subsystem internals such as task source code, task execution time, task period etc. are required. Also, it is not required to implement any scheduler. Times internal scheduler is responsible for the schedulability analysis and the generated code will emulate the scheduler(s) in the system.

In recent years, automata based approaches have been proposed as generic ways to describe and analyze a wide variety of real-time scheduling policies. One of the potential strengths of these approaches is the possibility to encode

generic release patterns of tasks. In the model of task automata [2], release patterns are modeled using timed automata [3] and it has been shown that the schedulability analysis problem is decidable for both fixed-priority and dynamic scheduling policies (including EDF). Another strength of this approach is the possibility to perform simulation, formal verification of timing and functional safety properties, as well as code-synthesis [4]. For the task automata model, the tool Times provides this support [5].

In this paper our overall goal is to allow for detailed analysis of hierarchically scheduled real-time systems at an early stage in the development process. The main contributions of this paper are:

1) We have enabled analysis of hierarchically scheduled fixed priority preemptive systems in Times.
2) We have transformed automatically generated source code (from Times) for VxWorks (could easily be extended to other RTOS), allowing for early prototyping, testing and verification of hierarchically scheduled fixed priority preemptive systems.
3) Related to the above contribution (2), we have conducted experiments on the generated code. We have included response time measurements, overhead measurements of both the generated scheduler and a manually coded scheduler and compared these.

The outline of this paper is as follows: in Section II we outline preliminaries on hierarchical scheduling, task automata and Times. In Section III we outline the problem statement together with its limitations, and in Section IV we show our solution. Section V shows a case-study, including an example system, code generation and an performance evaluation. Section VI presents related work, and finally, Section VII concludes.

II. Preliminaries

A. Hierarchical scheduling

Hierarchical scheduling has been introduced to support hierarchical resource sharing among applications under different scheduling services. Hierarchical scheduling can generally be represented as a tree of nodes (Figure 1), where each node represents an application with its own scheduler for scheduling internal workloads (e.g., tasks). In this paper, we call these nodes subsystems. Further, looking at the tree-structure representation, resources are allocated from a parent node to its children nodes (Shin and Lee [6]). One of the main advantages of hierarchical scheduling is that it provides means for decomposing a complex system into well-defined parts (subsystems). In essence, hierarchical scheduling provides a mechanism for time-predictable *composition* of coarse-grained subsystems. This means that subsystems can be independently developed and tested, and later assembled without introducing unwanted temporal behavior. Also, hierarchical scheduling facilitates *reusability* of subsystems in time-critical and resource constrained environments, since their computational requirements are characterized by well defined *interfaces*.

Subsystems are scheduled according to the scheduling policy of the overlaying scheduler (for example Fixed-Priority Preemptive Scheduling (FPPS) or Earliest Deadline First (EDF)) and the parameters in the subsystem interface. In this paper, we assume that both local and global schedulers follow the FPPS. Subsystems can be represented as ”virtual tasks”, where the parameters in the interface corresponds to those in the periodic task model [7]. At runtime, subsystems are allocated a defined time (*budget*) every predefined *period* and they are executed based on their *priority* (these parameters are part of the subsystem *interface*). This is similar to a traditional real-time task executing a defined time periodically at every period, preempted when required, according to its priority. When a subsystem is selected for execution by its overlaying scheduler, the subsystem's internal tasks are executed and scheduled according to the scheduling strategy of the subsystem local scheduler, i.e., FPPS in this paper. Note that, in the general case, the global and local schedulers may all have different scheduling strategies.

To summarize the above, as subsystems are periodically scheduled in a hierarchical manner, the subsystem interface contains information on the fraction of the CPU required by a subsystem in each subsystem period. As long as this fraction of CPU is always provided to the subsystem, it is guaranteed that the subsystem will function according to its specifications, i.e., that the extra-functional temporal requirements (schedulability) of the subsystem are met.

B. Task automata and Times

The modeling language *timed automata* [3] is widely used for formal modeling and analysis of real-time systems. A timed automaton is essentially a finite state automaton to which real-valued clocks that can be tested and reset are added. The formalism has shown to be suitable for a wide range of real-time systems.

More recently, the timed automata model has been extended with an explicit notion of tasks with parameters such as priorities, computation times, deadlines etc. The model, designated *task automata* (of *timed automata with tasks*), associates asynchronous tasks with the locations of a timed automaton, and assumes that the tasks are executed using static or dynamic priorities by a preemptive or non-preemptive scheduling policy. The model is supported by the Times tool [5][1], a tool supporting schedulability analysis, formal verification by model-checking of safety properties, and code synthesis. In particular, the tool can check if a model is schedulable in the sense that all tasks, released by the timed automaton, are guaranteed to always meet their deadlines using a given scheduling policy.

In case all tasks are released periodically (possibly with offsets), or aperiodically, an input system to the Times tool merely constitutes a task table in which the following parameters are defined for each task: name, computation time, (relative) deadline, priority (in case of static priority scheduling), offset

[1]For more information about Times, see `http://www.times-tool.com/`.

and period (if applicable), interface, semaphore usage, and its C-code. Alternatively, a task can be of type *controlled*, meaning that its release pattern is defined by a given timed automata.

III. Problem statement

The objective of this paper is to perform schedulability analysis and generate executable code (that emulates the scheduling of a scheduling tree) of hierarchically scheduled systems. In this section, we first outline the system model used, followed by some limitations and a description of our approach.

A. System model

A system $\mathcal{S}$ consists of a root S_0 and n subsystems $S_1, ..., S_n$. We assume independent tasks, i.e., there is no synchronization between tasks in the scheduling tree. Each subsystem S_i is defined as a tuple $\langle P_i, Q_i, \mathcal{T}_i, p_i, pr_i \rangle$, where P_i is the subsystem period, Q_i is the amount of CPU (or computation time) provided to the subsystem in each P_i, $\mathcal{T}_i$ is the set of subsystems (S) and tasks (τ) residing in subsystem S_i, $p_i \in [0..n]$ is the index of the parent of S_i, and pr_i is the fixed priority of S_i (higher value means higher priority). Each task τ_j is defined as a tuple $\langle T_j, C_j, D_j, pr_j \rangle$, where T_j is the task period, C_j is the task worst case execution time, D_j is the relative deadline and pr_j is the task priority (higher value means higher priority). The root S_0 is defined by the tuple $\langle \mathcal{T}_0 \rangle$, i.e., just a set of subsystems and tasks.

An example system with root S_0, subsystems S_1 and S_2 (of S_0), and subsubsystems S_3 and S_4 (subsystems of S_2), is illustrated in Figure 2.

a) Limitations:: We assume that the whole system and all subsystems are scheduled by fully preemptive fixed-priority schedulers. Generalizing the considered scheduling policy is deferred to future work. Given the system model defined above, we also impose the following two limitations on the relationship between task and subsystem periods:

- $\{\forall\ S_{i,i\in[1,n]} : P_i \geq P_{p_i}\}$, i.e., all subsystem periods are greater or equal to their respective parent's subsystem period and
- $\{\forall\ S_{i,i\in[1,n]}, \forall\ \tau_k \in \mathcal{T}_i : T_k \geq P_{p_i}\}$, i.e., all task periods are greater or equal to its corresponding subsystem's period.

The main reasons for these assumptions are twofold: first, the inequalities are recommended in order to have a resource efficient system, and secondly, analysis of the system is simplified given the fulfillment of the above 2 inequalities.

B. Approach

The objective is to perform detailed analysis of the contents of a subsystem S_i, i.e., detailed analysis of tasks resident in $\mathcal{T}_i$. This analysis is intended to assist engineers in the development of a subsystem. In doing the analysis, we create a set of interference tasks $\mathcal{I}_i$, representing (and consuming the computation time of) the rest of the system, i.e., the whole system excluding the subsystem under analysis. Each interference task is described by period a T, an offset O, and a computation time C. Given the interference tasks and the contents of the subsystem under analysis (i.e. the subsystem tasks), the Times tool is used to calculate timing properties (worst case response time) of the task set. Moreover, the Times tool is used for code-generation allowing for early prototyping of hierarchically scheduled subsystems.

In order to perform analysis of a complete system, i.e., all subsystems of a system, the approach outlined above can be repeated for each subsystem in the system. If the analysis shows that the scheduling of each subsystem is successful, we conclude that the whole system is schedulable. Traversing the system tree and analysing each subproblem can be performed automatically, either encoded as an automata in Times, or using an external script program. In this paper however, we leave the details of how to analyse a whole system, and focus on the analysis of one subsystem.

IV. Analysis of hierarchical systems

In order to analyse the tasks and subsystems, residing inside a subsystem (i.e., the subsystem under analysis), we create a set of interference tasks $\mathcal{I}_i$. Tasks and subsystems residing in the subsystem under analysis are then, together with the interference tasks $\mathcal{I}_i$, used as input to a tool for timing analysis. In this paper, we use the Times tool as it supports analysis of several properties, as well as code synthesis (see Section V).

In the following, we outline how to obtain $\mathcal{I}_i$ — a procedure with the following three main steps:

b) Step 1:: In this step, we create a partial schedule s_i, i.e. execution sequence (an example can be found in Figure 3). This schedule includes all subsystems and tasks interfering with the subsystem under analysis, and including the subsystem itself (S_i). The set of subsystems and tasks influencing the execution of a given subsystem is computed by the function HEP.

We define the recursive function $HEP(S_i)$ for a given system $\mathcal{S}$ in the following way. $HEP(S_i)$ is the set of subsystems, on the same level of the scheduling tree (with the same parent as S_i), that has higher priority than subsystem S_i, HEP of the parent of S_i and S_i itself (Eq. 1). The HEP set of the root node is empty (Eq. 2).

For the set of tasks $HEP(S_i)$ we compute the schedule s_i for the time interval $[0, l_i]$, where l_i = LCM($HEP(S_i)$), i.e. upto the least common multiple of the set $HEP(S_i)$.

$$HEP(S_i) = HEP(S_{p_i}) \cup \{\forall\ S_k \in \mathcal{T}_{p_i} : pr_k \geq pr_i\} \cup S_i \quad (1)$$

$$HEP(S_0) = \{\} \quad (2)$$

c) Example:: To show how the procedure works, we use a simple example hierarchical system consisting of 4 subsystems with the following parameters:

$$S_1 = \langle 4, 1, \mathcal{T}_1, 0, 3 \rangle$$
$$S_2 = \langle 3, 2, \mathcal{T}_2, 0, 4 \rangle$$
$$S_3 = \langle 5, 1, \mathcal{T}_3, 2, 2 \rangle$$
$$S_4 = \langle 6, 2, \mathcal{T}_4, 2, 1 \rangle$$

The example system is outlined in Figure 2. Suppose that subsystem S_3 is the subsystem that we are analyzing. Looking

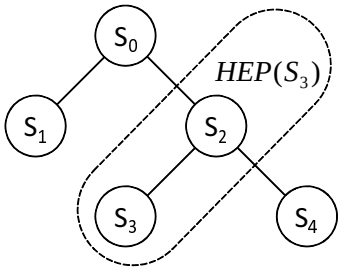

Fig. 2. Example hierarchical system.

at S_3, HEP(S_3) = $\{S_2, S_3\}$ (highlighted in Figure 2) and l_3 = LCM(HEP(S_3)) = 15.

Scheduling the example system, for the interval 0 to $l_3 = 15$, gives the schedule s_3, depicted in Figure 3.

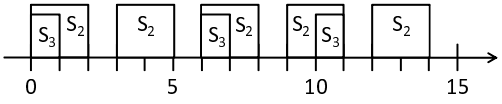

Fig. 3. Schedule s_3 given S_3 and $l_3 = 15$.

d) Step 2:: In this step, we take schedule s_i as input and create an ordered set of time points ϕ_i. The first element is 0, the last is l_i = LCM(HEP(S_i)), and the intermediate are the time-points when subsystem S_i is scheduled for execution, and is started, preempted or finished, in the time interval [0, l_i].

e) Example (continued):: Given the example system above, ϕ_3 is as follows:

$$\phi_3 = \{0, 0, 1, 6, 7, 10, 11, 15\}$$

representing a schedule starting at time 0, where the subsystem under analysis is first scheduled at time 0, finished at time 1, scheduled again at time 6, finished at time 7, scheduled again at time 10, finished at time 11, and LCM is 15.

f) Step 3:: In this step, given ϕ_i as input, we create a set of interference tasks $\mathcal{I}_i$. Let $|\phi_i|$ denote the number of elements in ϕ_i. We have to create $m = \frac{|\phi_i|}{2}$ interference tasks, $\partial_0, ..., \partial_{m-1}$. The task parameters are $\partial_j = \langle T_j, O_j, C_j, pr_j \rangle$, where T_j is the period of the task (set to T_j = LCM(HEP(S_i)) for all interference tasks), O_j is the offset of the interference tasks given by $O_j = \phi_i[j*2]$, given that $\phi_i[x]$ returns the value stored in ϕ_i at position x (given that positions are indexed starting with 0 and finishing with $|\phi_i| - 1$), $C_j = \phi_i[1 + j * 2] - \phi_i[j * 2]$, and for pr_j the following holds: $pr_j > pr_k$, where index k is defined by the set $\forall\ (\tau_k \wedge S_k) \in \mathcal{T}_i$.

g) Example (continued):: Looking at the example system again, $m = \frac{|\phi_3|}{2} = 4$, hence $\mathcal{I}_3$ hosting the set of 4 interference tasks is $\mathcal{I}_3 = \{\partial_0, \partial_1, \partial_2, \partial_3\}$ with

$$\begin{aligned}
\partial_0 &= \langle 15, 0 \ \ , 0, pr_0 \rangle \\
\partial_1 &= \langle 15, 1 \ \ , 5, pr_1 \rangle \\
\partial_2 &= \langle 15, 7 \ \ , 3, pr_2 \rangle \\
\partial_3 &= \langle 15, 11, 4, pr_3 \rangle
\end{aligned}$$

Once the above three steps are finished, all interference tasks stored in $\mathcal{I}_i$, together with the tasks and subsystems stored in the subsystem under analysis $\mathcal{T}_i$, are taken as input to Times, giving detailed analysis of all tasks in $\mathcal{T}_i$.

V. Modeling example

In order to illustrate our solution, we have modeled an example system consisting of 4 subsystems, arranged in a hierarchical tree (scheduling tree) as depicted in Figure 4. The engineering challenge, highlighted in this example, is how a development team (given a scheduling tree and a dedicated subsystem within it) can develop an application, consisting of real-time tasks, and be able to perform timing analysis of these tasks in order to verify weather or not they meet their respective deadlines. Such a verification should be possible when specifying and allocating task parameters, preferably early during the development and testing phase, allowing for early prototyping. The latter requires a way to execute the tasks, on a given platform, within their corresponding time slots, determined by the actual scheduling of the whole system (of subsystems).

Recall, in this paper it is assumed that tasks within one subsystem do not need to synchronize/communicate with tasks residing in other subsystems. Given this assumption, we do not need to consider detailed scheduling of tasks in other subsystems, since their exact scheduling do not affect the scheduling of the subsystem under analysis.

To summarize the above, in this example, we want to:

1) conduct schedulability analysis of a subsystems content (subsystem **C**'s content in this example), with respect to the interface(s) of subsystem **C** and the rest of the systems subsystems, and
2) generate executable code that execute subsystem **C**'s content, within its precise time slots, as if the whole system of subsystems was executing (even though we only have source code and task parameters of subsystem **C**).

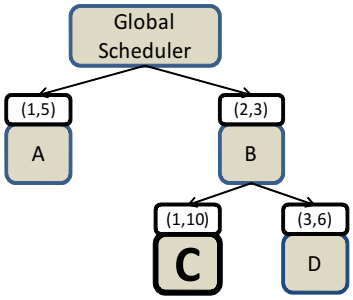

Fig. 4. Example system

In this example, the global scheduler and all local schedulers (i.e. the internal scheduler of each subsystem) schedule their tasks/subsystems according to fixed priority preemptive scheduling. The priority assignment is done according to Rate Monotonic [7], i.e, the shorter the period, the higher the priority. In doing schedulability and response-time calculations, we

need a detailed description of the task set resident in subsystem **C**; these details are represented in Table I.

Name	T	C	D	pr
task1 (τ_1)	40	1	40	5
task2 (τ_2)	50	1	50	4
task3 (τ_3)	80	1	80	3
task4 (τ_4)	90	1	90	2
task5 (τ_5)	250	7	250	1

TABLE I
TASK SET OF SUBSYSTEM **C**

A. Schedulability analysis

We assume that the subsystems in the tree are schedulable (for which they are in this example) and that the scheduling tree is pre-determined by the system description or similar. As a development team, you are given the timing parameters of subsystem **C**, i.e., its period and capacity. The responsibility of the development team is to develop an application consisting of a set of tasks (Table I) that are schedulable given the timing parameters of subsystem **C** (i.e., given subsystem period and capacity). The issue for the development team to solve, is assuring that their application is schedulable considering that their application will (in the future and final system) be scheduled together with other subsystems in the hierarchical scheduling tree. Hence, the development team cannot assume that subsystem **C** will get 1 time slot exactly every 10 time units because subsystems, at the same or higher level in the scheduling tree, might interfere (as they may have higher priority than that of the subsystem allocated to the application). The timing analysis of a subsystem (and its tasks) must consider all subsystem (of the same or higher level and with higher priority) parameters.

The first step is to analyze weather the chosen task parameters are sufficient in order for the tasks to meet their deadlines. What should be done is to add these tasks to the scheduling tree (Figure 4), under subsystem **C**, and check if they are schedulable. This can be done with a schedulability test such as Response Time Analysis (RTA) [8] for hierarchical systems [9]. However, we want to show how this can be done in Times, by generating interference tasks, belonging to the set $\mathcal{I}_C$, (called dummy tasks in this section) that emulate correct execution of the subsystem under analysis by blocking out time representing higher priority subsystem execution time, as well as time when the system should be idle.

Executing the example system, the corresponding schedule (s_C) of subsystem and task execution is illustrated in Figure 5.

By laying out the schedule of all subsystems, one can identify the time-slots when the subsystem under analysis should be executed, and thereby also the inverse of this time. This inverse time represents the time that should be "blocked out" in order to simulate interference from higher priority subsystems, as well as idle time. We achieve this "blocking out" (interference) by creating dummy tasks with higher priority than that of the tasks in the analyzed subsystem (as described in Section IV). Once the dummy tasks are generated (which

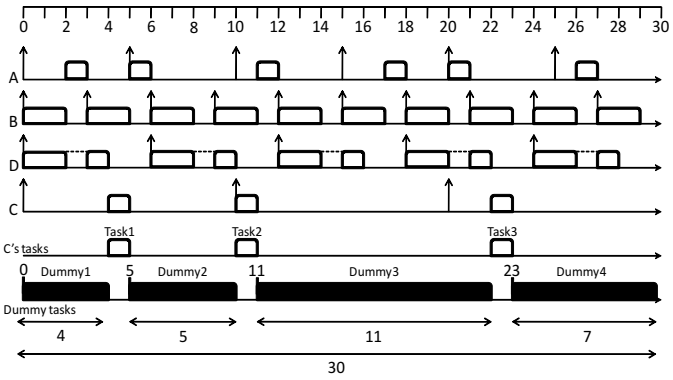

Fig. 5. System schedule

can be done following the steps in Section IV), they can be inserted into the Times tool. The dummy tasks' release pattern can either be described (in Times) in a task-parameter table (e.g. set offset, priority etc.) or by constructing an automata. The latter has an advantage when generating code (this will be covered in more detail in Section V-B). However, for analyzing timing and schedulability of tasks in Times, the easier approach is to specify the dummy tasks in the task-parameter table. After entering the dummy task parameters (Table II) together with the subsystem tasks (Table I) in Times, it can simulate the system and do (schedulability and) response-time analysis as shown in Figure 6. Times will output weather or not the system is schedulable, and if schedulable, it will also give the Worst Case Response Time (WCRT) of all tasks.

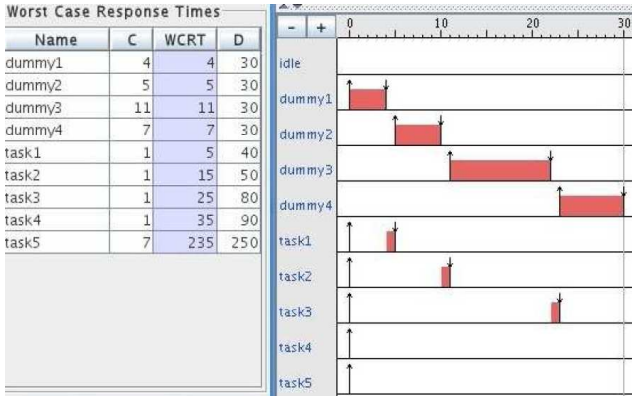

Fig. 6. Times schedulability analysis

Summarizing this section on analysis, the schedulability analysis performed in Times, is simulation which will produce the WCRT. This will include interference from subsystems (that can reside at different levels of the scheduling tree), which is actually modeled as interference from higher priority tasks.

B. Code synthesis

Times is equipped with an automatic code generator which can generate C-code of the modeled system in the platform

Name	T	O	C	pr
dummy1 (∂_1)	30	0	4	6
dummy2 (∂_2)	30	5	5	6
dummy3 (∂_3)	30	11	11	6
dummy4 (∂_4)	30	23	7	6

TABLE II
GENERATED DUMMY TASKS

brickOS[2]. We have used this code generator to generate code of our example system in Section V-A. The generated code is then transformed to fit a new software platform, namely VxWorks. This transformation was done manually but could also be done automatically. The reason for choosing VxWorks is that we are well familiar with task scheduling, execution tracing etc. in this platform, it provides an industry standard task scheduler, and it is a preferred platform of several of our industrial partners. Having knowledge of scheduling is specially important since we need to map brickOS scheduling to VxWorks.

In the analysis part (Section V-A), we analyzed the system based on dummy tasks (with offsets). We created periodic tasks and assigned the offsets through the task parameter table (all other tasks were also created in this manner). Creating tasks with offsets can also be done by creating an automata. This has the advantage that we can specify that only one dummy task is released at all offset instances and thereby replacing all dummy tasks with only one. This is good when generating code, since most RTOSs have an upper limit on the amount of tasks. At code level, the execution time of this dummy task must be set to be dynamic, since it is replacing tasks which most probably have different execution times. The two automatas in Figure 7 models the releasing of dummy tasks.

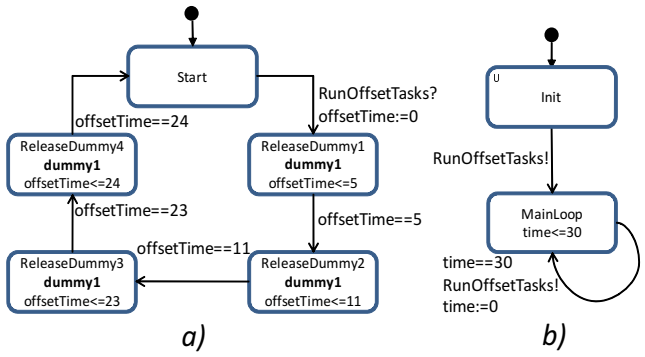

Fig. 7. Task automatas

The automata in Figure 7, **b)**, releases the second automata (Figure 7, **a)**) every 30 time units by calling a synchronization function **RunOffsetTasks!** which starts a transition in the edge where **RunOffsetTasks?** is located. The second automata releases the dummy tasks according to the calculated offsets (with relation to the period). **time** and **offsetTime** are two clocks that progresses in discrete time. An invariant such as **offsetTime<=5** (located inside a state) means that the automata may only be in that state until this condition does not hold. A condition at an edge such as **offsetTime==5** means that the transition can be made only when this condition holds. A statement such as **time:=0** means that the variable (in this case a clock) is assigned a value. Whenever there is a transition to a state with a task name, such as **dummy1**, this task is released for execution.

[2]http://brickos.sourceforge.net/

```
1: task( ) {
2.    while(TRUE) {
3.          wait_event(task_release, release_flag)
4.          // Task code here
5.    }
6. }
7: controller( ) {
8:     wait_event(check_trans, 0)
9: }
```

Fig. 8. Function task() and controller()

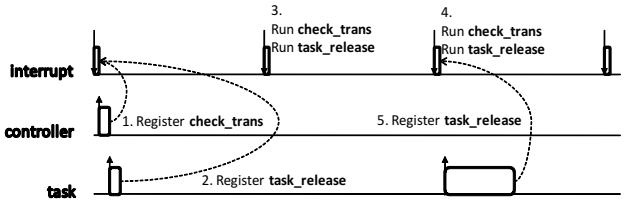

Fig. 9. brickOS scheduling

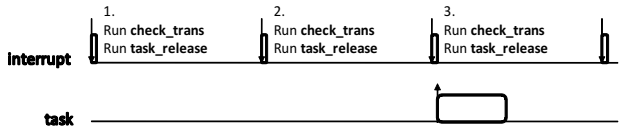

Fig. 10. VxWorks scheduling

The mapping from the C-code (generated by Times) to VxWorks consists mostly of changing the way the task is suspended and released. In the brickOS generated code, an initializer task called **controller** (Figure 8, lines 7-9) calls **wait_event** in order to register a function **check_trans** that will be executed at every system tick by an interrupt routine. This will stop when the function returns a non-zero value (which is not the case for **check_trans**). This function traverses the automatas (both user defined automatas and Times default generated automata) and sets a flag whenever there should be a task release. Each task (Figure 8, lines 1-6) registers a function **task_release** at the beginning of its execution, before it suspends. This function checks whether the flag is set, if so, it will return a non-zero value that in turn will release the corresponding task. Figure 9 illustrates how the scheduling is done in the generated code for brickOS. The mapping of

this scheduling to VxWorks is illustrated in Figure 10. We create an interrupt routine that is executed at every system tick. This routine executes both the **check_trans** function and each tasks **task_release** function. Whenever **check_trans** sets the task flag, i.e. that is when **task_release** returns a non-zero value, the corresponding task is inserted into the VxWorks ready queue.

We have successfully generated C-code for the example system in Figure 4, that is comprised of the tasks in Table I and Table II. We transformed the generated code and ran the system in VxWorks 6.6 on a Intel Pentium4 platform. Further, we recorded and visualized the execution trace with the Tracealyzer tool[3].

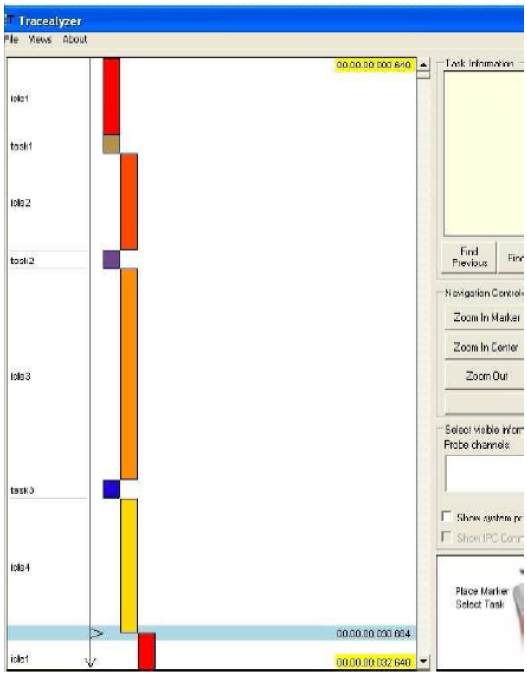

Fig. 11. Tracealyzer screenshot

Task	Execution time (µs)		Response time (µs)	
	Avg.	Max.	Avg.	Max.
task1	996	999	11999	14003
task2	996	999	16998	24000
task3	995	997	27994	33996
task4	995	997	32042	63982
task5	6267	6973	228643	291888
idle1	3995	4004	3995	4004
idle2	5000	5001	5000	5001
idle3	11000	11001	11000	11001
idle4	6999	7007	10994	11004

TABLE III
TRACEALYZER RESULT

Figure 11 shows the graphical representation of the running tasks (note that tasks 'dummy1' etc. from Figure 6 are named 'idle1' etc. in Figure 11) at critical instant and the recorded data is shown in Table III. Figure 6 shows the WCRT of the simulation, corresponding to **Max. Response time** in Table III, note that the time-base is 1000 times bigger in Table III. The maximum response times in Table III are significantly higher than the simulation values because of overhead (scheduling, context switches etc.). This prolonged response time is illustrated in Figure 11. **task2** does not finish its entire execution before **idle3** starts, leading to that **task2** has to wait for it to finish (which will take 11 time units), and then execute the final part (it is a very small amount so it does not show in this resolution). This kind of execution scenario is valuable for a development team and can only be discovered in time, in the development process, through early prototyping/testing.

Table IV shows the scheduling overhead (from running the tasks in Table I and II) from the generated scheduler (Times) and a manually coded scheduler, Hierarchical Scheduling Framework (HSF) [10]. We measured the schedulers execution times with micro-second resolution, 10 times each (Table IV shows the average values), between time zero (when the system started) and LCM of all tasks (18000000 µs). The HSF scheduler only executes at task release and task deadline (in the latter case it checks if the task has finished), while the Times scheduler executes at every system tick (i.e. every milli-second), and releases tasks if necessary. VxWorks itself handles task switching due to that a task has finished. The conclusion is that even though Times runs more frequently (and the fact that it is automatically generated code) than HSF, HSF still produces more overhead (the majority of it comes from queue-management [10]).

Scheduler	Avg. overhead/Duration (µs)	Avg. overhead (%)
Times	1952/18000000	0.01084
HSF	3283/18000000	0.01824

TABLE IV
SCHEDULING OVERHEAD

VI. RELATED WORK

Related work in the area of hierarchical scheduling originated in open systems [11] in the late 1990's, and it has been receiving an increasing research attention ever since. Since Deng and Liu [11] introduced a two-level hierarchical scheduling framework, its schedulability has been analyzed under fixed-priority global scheduling [12] and under EDF-based global scheduling [13], [14]. Mok *et al.* [15] proposed the bounded-delay resource model so as to achieve a clean separation in a multi-level hierarchical scheduling framework, and schedulability analysis techniques [16], [17] have been introduced for this resource model. In addition, Shin and Lee [6] introduced the periodic resource model (to characterize the periodic resource allocation behavior), and many studies have been proposed on schedulability analysis with this resource model under fixed-priority scheduling [9], [18], [19] and under EDF scheduling [6]. Looking at the kind of analysis possible with these hierarchical scheduling approaches, typically only timing is considered. In this paper, we are also interested in code synthesis, as well as analysis (using task automata). This

[3] http://www.tracealyzer.se/.

is similar to [20], where the authors show how modeling and schedulability analysis of two-level hierarchical scheduling, with timed automata, can be accomplished in the simulation tool Cheddar.

VII. Conclusion

We have shown how schedulability analysis, of a subsystem, within fixed-priority preemptive hierarchical scheduling, can be done in the Times tool. The key idea is to replace interfering subsystems (in a hierarchical scheduling tree) with a small set of high priority tasks (the proposed technique is described with an algorithm). These tasks, and the tasks of the subsystem to be analyzed, are modeled in Times (with a task-table or timed automata), and later analyzed (schedulability analysis).

Using the Times code generator, we have shown how the generated code (of an example system) can be adjusted to execute on an industrial platform (i.e. VxWorks), hence, our proposed method becomes more practical. After code synthesis, a subsystem can be executed as if it would be scheduled within a hierarchical scheduling tree. In this sense, the approach proposed in this paper supports early prototyping of hierarchically scheduled systems.

Our example shows clearly that response times can vary significantly when moved from simulation to a real platform, even though a very small amount of overhead is introduced. The overhead measurements show that the scheduler, generated from Times, produces less overhead than a manually coded scheduler.

As future work, we plan to optimize the code synthesis (in order to minimize scheduler overhead) as well as model and generate code for hierarchical schedulers.

References

[1] M. Åsberg, M. Behnam, F. Nemati, and T. Nolte, "Towards Hierarchical Scheduling in AUTOSAR," in *ETFA'09*.

[2] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *International Journal of Information and Computation*, vol. 205, no. 8, pp. 1149–1172, August 2007.

[3] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: citeseer.nj.nec.com/alur94theory.html

[4] T. Amnell, E. Fersman, P. Pettersson, W. Yi, and H. Sun, "Code synthesis for timed automata," *Nordic J. of Computing*, vol. 9, no. 4, pp. 269–300, 2002.

[5] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times: A tool for modelling and implementation of embedded systems," in *TACAS'02*, 2002.

[6] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *RTSS'03*, Dec. 2003.

[7] C. Liu and J. Layland, "Scheduling algorithms for multi-programming in a hard-real-time environment," *ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[8] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, 1993.

[9] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *RTSS'05*, December 2005.

[10] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in *OSPERT'08*, July 2008.

[11] Z. Deng and J. W.-S. Liu, "Scheduling real-time applications in an open environment," in *RTSS'97*, Dec. 1997.

[12] T.-W. Kuo and C. Li, "A fixed-priority-driven open environment for real-time applications," in *RTSS'99*, Dec. 1999.

[13] G. Lipari and S. Baruah, "Efficient scheduling of real-time multi-task applications in dynamic systems," in *RTAS'00*, May 2000.

[14] G. Lipari, J. Carpenter, and S. Baruah, "A framework for achieving inter-application isolation in multiprogrammed hard-real-time environments," in *RTSS'00*, Dec. 2000.

[15] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *RTAS'01*, May 2001.

[16] X. Feng and A. Mok, "A model of hierarchical real-time virtual resources," in *RTSS'02*, Dec. 2002.

[17] I. Shin and I. Lee, "Compositional real-time scheduling framework," in *RTSS'04*, Dec. 2004.

[18] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in *ECRTS'03*, July 2003.

[19] S. Saewong, R. Rajkumar, J. P. Lehoczky, and M. H. Klein, "Analysis of hierarchical fixed-priority scheduling," in *ECRTS'02*, June 2002.

[20] F. Singhoff and A. Plantec, "AADL modeling and analysis of hierarchical schedulers," in *SIGAda'07*, 2007.