Prototyping and Code Synthesis of Hierarchically Scheduled Systems using 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 as a tree of nodes, where each node schedules its child nodes. A node contains tasks and/or subsystems, where a subsystem is typically developed by a development team. Given a system where each part is subcontracted to different developers, they can benefit from hierarchical scheduling by parallel development and simplified integration of subsystems. Each team should have the possibility to test their system before integration. Hence, we show how a node, in a hierarchical scheduling tree, can be analyzed in the Times tool by replacing all interference from nodes with a small set of higher priority tasks. We show an algorithm that can generate these tasks, including their parameters. Further, we use the Times code-generator, in combination with operating system extensions, to generate source code that emulates the scheduling environment for a subsystem, in an arbitrary level in the tree. Our experiments include two example systems. In the first case we generate source code for an industrial oriented platform (VxWorks) and conduct a performance evaluation. In the second example we generate source code that emulates the scheduling environment for a video application, running in Linux, and we perform a frame-rate evaluation.**

I. Introduction

The increase in global competitiveness and requirement of shorter time-to-market has increased the need for rapid development of embedded software systems. A crucial characteristic, in being fast and reliable in the development of embedded software systems, is to do analysis and prototyping early in the development process, in order to decrease the load, complexity and cost in the integration phase.

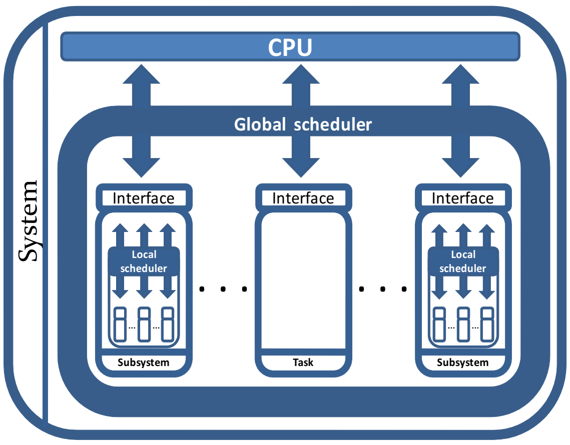


Fig. 1. Hierarchical scheduling

Recently, the technique of hierarchical scheduling (HS) has been introduced in order to simplify parallel development of embedded systems. HS facilitates integration of such systems, by providing mechanisms for temporal isolation of system parts, called subsystems. Essentially, a system consists of a number of subsystems that typically represents a particular function/feature of the whole system. For example, a car could have one subsystem implementing a engine control system, and another being the anti-lock braking system. These two subsystems should ideally be developed in parallel, and at the integration phase, no integration related problems should occur [1]. One such integration related problem is software that turn out to require more time to execute than originally intended, and therefore causing unforseen interference with the rest of the system. Another integration problem is the introduction of new subsystems, not apparent at early design. Integration of unforseen subsystems should not cause too much interference, i.e., the entire system should not be required to be verified/validated again. HS insures that no unpredictable interference will occur, related to timing, hence by allowing for timing analysis of subsystems in isolation before the integration. Figure 1 illustrates HS. The top node is defined as the *Global scheduler*. It is responsible for multiplexing the entire *CPU* resource to the second layer of the scheduling tree. A node can be either a *Subsystem*, or a *Task* (except for the top node which is a scheduler). In this way, a node schedules its child nodes with its *Local scheduler*. All nodes have an *Interface* (set of scheduling parameters) which specifies the amount of CPU that the node may access. The schedulers uses these interfaces to schedule its nodes.

It is desirable to be able to conduct analysis of a subsystem's functional and non-functional properties in isolation, i.e., without requiring details of the rest of the system. It is hard to get access to all details of other subsystems, especially at an early stage in the construction of a system. Our proposed technique makes it possible to perform schedulability analysis of tasks, with respect to its subsystem interface. Also, the subsystem can be realized by generating source code (for our target platforms VxWorks and Linux) that will emulate the subsystem (under development) executing together with other subsystems/tasks. The subsystem's schedule will look like it is executing together with the other subsystems in the tree (early prototyping). What is required are the interfaces of the other subsystems/tasks, i.e., no subsystem internal data such as task source code, execution time, period etc. are needed. Also, there is no need to implement any scheduler. The internal scheduler of the Times tool is responsible for the schedulability analysis, and the generated source code will emulate the scheduler(s) in the system.

Recently, automata based techniques have been proposed as a generic way to describe and analyze a broad variety of real-time scheduling algorithms. One of the strengths of these techniques is the possibility to encode general release patterns of tasks. In the task automata model [2], release patterns are modeled using timed automata [3]. The schedulability analysis problem has shown to be decidable for both fixed and dynamic priority scheduling algorithms. Further, this approach has the possibility to perform simulation and formal verification of timing and functional safety properties, as well as code-synthesis [4]. For the model of task automata, the Times tool provides this support [5].

In this paper our overall goal is to provide a technique for analysis and synthesis of hierarchically scheduled real-time systems, at an early stage in the development process. Our main contributions are[1]:

1) We have enabled timing analysis of hierarchically scheduled, fixed-priority preemptive systems, in the Times tool.
2) We have transformed and made extensions to the generated source code (from Times) for VxWorks and Linux, allowing for early prototyping/testing of hierarchically scheduled, fixed-priority, preemptive systems.
3) Related to the above contribution (2), we have conducted experiments on the generated code (for both VxWorks and Linux). We have included response time measurements, overhead measurements of both the generated scheduler, and a manually coded scheduler, and we have compared these. Also, we have been running a video processing application (VLC) in Linux, and conducted frame-rate performance comparisons using a 2-level hierarchical scheduler, as well as task tracing.

The outline of the 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 including its limitations, and in Section IV we show our solution. Section V shows two case-studies, including an example system, code generation and a performance evaluation. Section VI presents related work, and finally, Section VII concludes.

II. Preliminaries

A. Hierarchical scheduling

Hierarchical scheduling has been introduced to facilitate resource sharing among applications under different scheduling policies. Hierarchical scheduling can be represented as a tree of nodes (Figure 1), where each node corresponds to an application, equipped with a scheduler that schedules internal workloads. Looking at the tree-structure representation of HS, CPU resources are reserved from a parent node to its children nodes (Shin and Lee [7]). One of the advantages of HS is that it provides a way to decompose a complex system into well-defined parts (subsystems). HS provides the mechanism for predictable composition (in the time domain) of coarse-grained subsystems. This makes it possible for subsystems to be developed independently and later integrated, without introducing timing errors. Also, HS makes it easy to reuse subsystems, since their computational demands are characterized by well defined interfaces.

Subsystems and tasks are scheduled according to the scheduling scheme of the above scheduler and the parameters in the interface of the subsystem. In this paper, we assume that the schedulers follow the fixed-priority preemptive scheduling policy. Subsystems can be viewed as "virtual tasks", where the interface parameters corresponds to those in the periodic task model [8]. At runtime, subsystems reserve a defined time (*budget*) at every *period* and the execution order is based on their *priority*. This is similar to a traditional periodic task, scheduled preemptively with a fixed-priority scheduler. When a subsystem is selected for execution by the overlaying scheduler, the subsystem's tasks are executed and scheduled according to the scheduling policy of the subsystem local scheduler. In the general case, the schedulers in HS may all have different scheduling schemes.

B. Task automata and Times

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

More recently, the model of timed automata has been extended with a notion of real-time tasks. *Task automata* (of *timed automata with tasks*), associates asynchronous tasks with the states of a timed automaton. It assumes that tasks are executed with static or dynamic priorities by a preemptive or non-preemptive scheduling algorithm. Task automata is supported by the Times tool [5][2], it facilitates schedulability analysis, formal verification by model-checking and code synthesis.

An input system to the Times tool can consist of 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 and period (if applicable), interface, semaphore usage, and its C-code. Alternatively, a task can be of type *controlled* which means that its release pattern is defined by a user defined timed automata.

III. Problem statement

The aim of this paper is to consider a subsystem (potentially with tasks and a fixed-priority scheduler), residing in a scheduling tree, and to perform schedulability analysis of

[1]This work is an extension of our previous work [6]

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

it. The analysis is done by the Times tool, although it does not support schedulability analysis of hierarchically scheduled systems. The solution to this is to map the rest of the tasks and subsystems in the tree to a small amount of interference tasks. Also, for the sake of prototyping, we 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: (1) the inequalities are recommended in order to have a resource efficient system, (2) analysis of the system is simplified given the fulfillment of the above 2 inequalities.

B. Approach

The objective is to perform schedulability analysis of the contents (tasks/subsystems resident in $\mathcal{T}_i$) of a subsystem S_i, with respect to its interface and the interference from the rest of the tree. 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. Hence, the interference from $\mathcal{I}_i$ represents the interference from the whole tree (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 in S_i. Moreover, the Times tool is used for code synthesis, allowing for early prototyping of hierarchically scheduled subsystems.

In order to perform analysis of a complete system, i.e., for each subsystems in 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, then we can 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 analyze a whole system, and focus on the analysis of one subsystem.

IV. Analysis of hierarchical systems

In order to analyze 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 because it supports analysis of several properties, as well as code synthesis (see Section V).

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

b) Step 1:: First 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, 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 (including S_i itself), on the same level of the scheduling tree as S_i (with the same parent as S_i), that have higher priority than subsystem S_i. The recursiveness is defined in that HEP must also be calculated for the parent of S_i (Eq. 1). However, the HEP set of the root node is empty (Eq. 2).

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

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

For the set of tasks $HEP(S_i)$, we compute the schedule s_i for the time interval $[0, l_i]$, where

$$l_i = \text{LCM}(\{\forall\ k \in HEP(S_i) : P_k\})$$

,i.e., upto the least common multiple of the periods in the set $HEP(S_i)$.

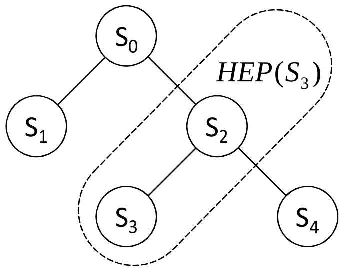


Fig. 2. Example hierarchical system.

c) Example:: To show how the procedure works, we use a simple example of a hierarchical scheduled 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 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.

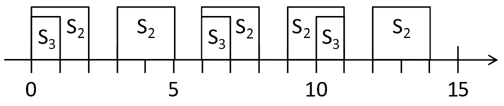


Fig. 3. Schedule s_3 given S_3 and $l_3 = 15$.

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($\{\forall\ k \in HEP(S_i) : P_k\}$), 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 scheduled initially 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($\{\forall\ k \in HEP(S_i) : P_k\}$) 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

$$\partial_0 = \langle 15, 0\ \ , 0, pr_0\rangle$$
$$\partial_1 = \langle 15, 1\ \ , 5, pr_1\rangle$$
$$\partial_2 = \langle 15, 7\ \ , 3, pr_2\rangle$$
$$\partial_3 = \langle 15, 11, 4, pr_3\rangle$$

Once the above three steps are finished, all interference tasks stored in $\mathcal{I}_i$, together with the tasks and subsystems ($\mathcal{T}_i$) in the subsystem under analysis, 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, 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 schedulability analysis of these tasks, in order to verify whether 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). This will be shown in section V-B2 and V-C2.

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 does 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 **A** and **C**'s content in this example), with respect to the interface(s) of subsystem **A**, respectively **C**, and the rest of the subsystems, and
2) generate executable code, a scheduler to be precise, that execute subsystem **A** and **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 **A** and **C**).

An assumption is 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 your subsystem (i.e., subsystem **A** or **C** in this case), which is the period and capacity of these subsystems. The responsibility of the development team is to develop an application consisting of a set of tasks that are schedulable given the timing parameters of their subsystem. The issue for the development team to solve, is to assure 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 their subsystem, **C** for example, 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 subsystem **C**). The timing analysis of a subsystem (and its tasks) must consider all subsystem (of the same or higher level and with higher priority) parameters, including its own.

The first step is to analyze whether 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, like the one in Figure 4, under their subsystem, and check if they are schedulable with relation to the interfaces of the subsystems in the tree. This can be done with a schedulability test such as Response Time Analysis (RTA) [9] for hierarchical systems [10]. However, we want to show how this can be done in Times, by generating interference tasks (called dummy tasks in this section). These tasks 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. 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 subsystem under analysis (as described in Section IV). Once the dummy tasks are generated (which 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. by setting offset, priority, period etc.) or by constructing an automata. The latter has an advantage when generating code (this will be covered in more detail in Section V-B2). However, for schedulability analysis of tasks in Times, the easier approach is to specify the dummy tasks in the task-parameter table. After entering the dummy task parameters together with the subsystem tasks in Times, it can simulate the system and do response-time analysis as shown in Figure 6 and 14. Times will output whether or not the system is schedulable, and if schedulable, it will also give the Worst Case Response Time (WCRT) of all tasks.

In conclusion, the schedulability analysis performed in Times, is a simulation which will produce the WCRT of each task. So we have actually simplified the problem into a response time analysis of a set of periodic tasks (belonging to the subsystem under analysis), together with a set of periodic tasks with offsets (the dummy tasks). The WCRT value will include the interference from subsystems (that can reside at different levels of the scheduling tree), which is actually modeled as interference from higher priority tasks, as well as the execution time of the task itself. Hence, for the sake of timing analysis, timing analysis tools other than Times can be used. However, we are not only interested in timing analysis, but also in generating code for early prototyping of the subsystem under analysis.

A. *Code synthesis*

The Times tool is equipped with an automatic code generator which can generate C-code of the modeled system to the platform brickOS[3], as well as a simulator for Linux. We have used this code generator to generate code of our example system. We show two examples, where we synthesize code for a scheduler for VxWorks (section V-B2) and Linux (section V-C2). The generated code is then transformed (extended) to fit the new software platform, i.e., VxWorks or Linux. 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 (since the code generator generates brickOS code).

For Linux, we generate the Linux simulator code from Times, then we remove the simulator code manually (could be done automatically). What is left is the actual automata code (i.e., the scheduler). The automata code in turn is extended to fit in the Linux kernel, such that it can schedule tasks. This is a manual step (which can be automated).

B. *Subsystem **C***

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 [8], i.e, the shorter the period, the higher the priority. Subsystem **C** resides in the tree represented in Figure 4.

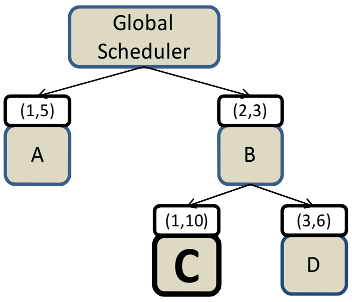


Fig. 4. Subsystem **C**

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.

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

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**

1) Schedulability analysis: The corresponding schedule for s_C, executing in the example system, is illustrated in Figure 5.

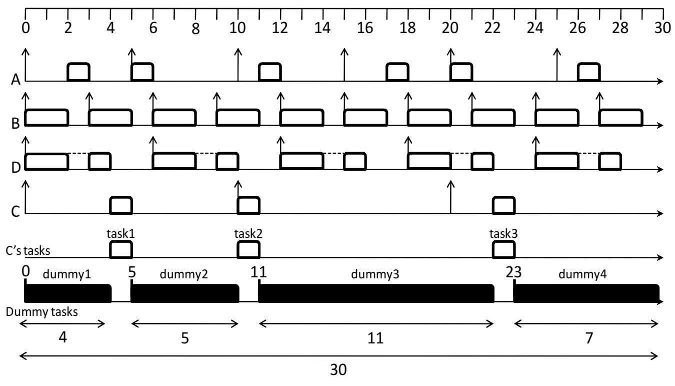


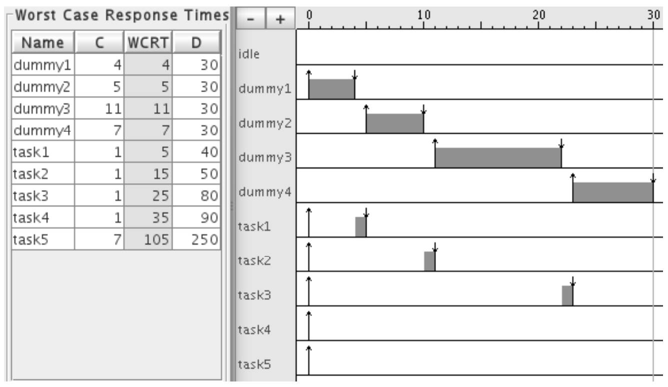
Fig. 5. Schedule for subsystem **C**

From this schedule we can conclude which dummy tasks that we need (∂_1-∂_4), as shown in Table II.

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 FOR SUBSYSTEM **C**

The last step is to input all tasks in the Times tool and let it perform a simulation. Figure 6 shows that subsystem **C**'s tasks are schedulable with the 4 dummy tasks, i.e., the other three subsystems in the system.


Worst Case Response Times

Name	C	WCRT	D
dummy1	4	4	30
dummy2	5	5	30
dummy3	11	11	30
dummy4	7	7	30
task1	1	5	40
task2	1	15	50
task3	1	25	80
task4	1	35	90
task5	7	105	250

Fig. 6. Times schedulability analysis (for subsystem **C**)

2) Code synthesis to VxWorks (kernel version 6.6): In the analysis part (Section V-B1), 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 automata in Figure 7 models the releasing of dummy tasks (a similar automata, but with other release times, is used for the example in section V-C2).

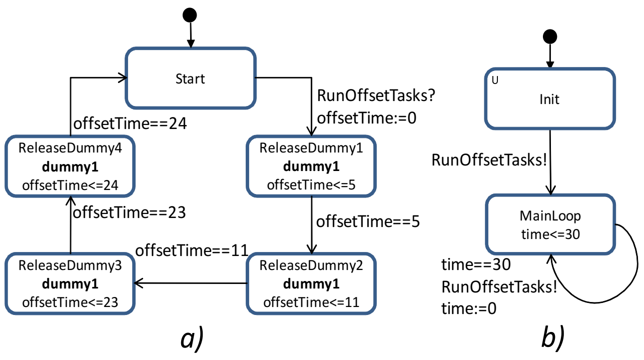


Fig. 7. Task automata

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.

```
task( ) {
    while(TRUE) {
        wait_event(task_release, release_flag)
        // Task code here
    }
}
controller( ) {
    wait_event(check_trans, 0)
}
```

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

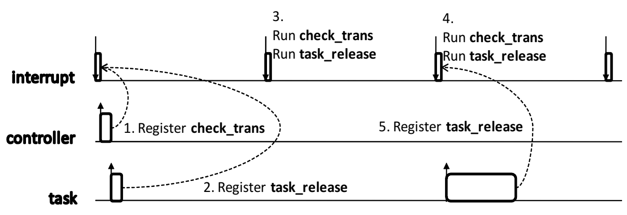


Fig. 9. brickOS scheduling

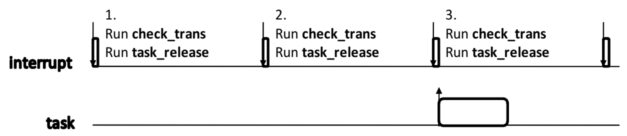


Fig. 10. VxWorks scheduling

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 automata (both user defined automata 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[4].

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

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

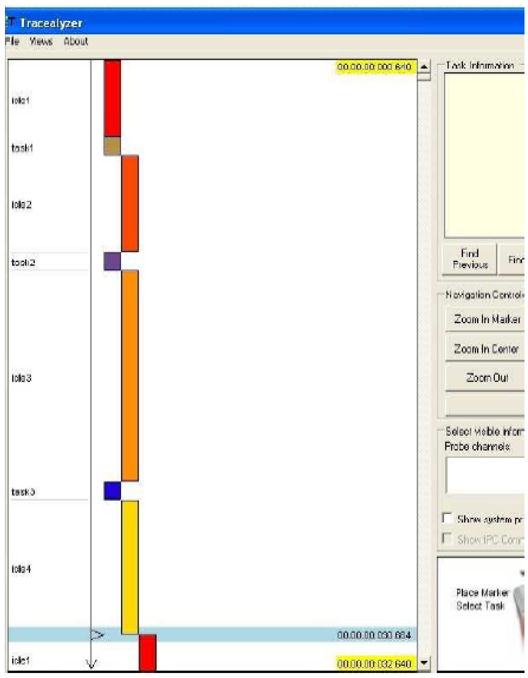


Fig. 11. Tracealyzer screenshot

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

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; the Hierarchical Scheduling Framework (HSF) [11]. 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 [11]).

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

TABLE IV
SCHEDULING OVERHEAD

C. *Subsystem* **A**

This example also assumes fixed-priority preemptive scheduling of periodic tasks/subsystems, as well as rate monotonic priority assignment.

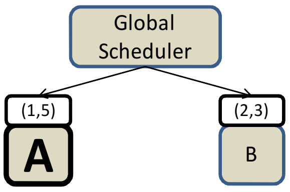


Fig. 12. Subsystem **A**

The content of subsystem **A** is one task (Table V), which correspond to the parameters of its subsystem. Subsystem **A**'s position in the scheduling tree is shown in Figure 12.

Name	T	C	D	pr
taskA	5	1	5	1

TABLE V
TASK SET OF SUBSYSTEM **A**

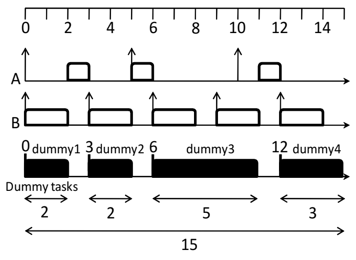


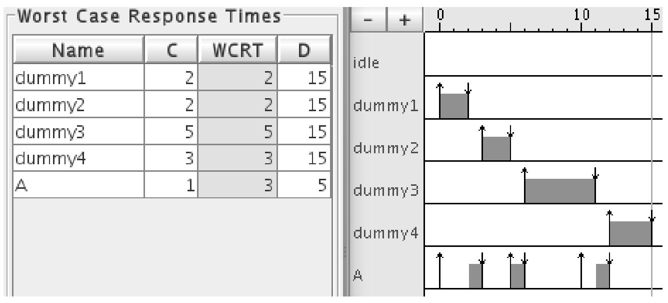
Fig. 13. Schedule for subsystem **A**

1) Schedulability analysis: By laying out the schedule for subsystem **A** (Figure 13), we have generated the necessary dummy tasks (Table VI).

By inserting all tasks (Table V and VI) into Times and running its simulation, we can get the schedulability analysis for subsystem **A**'s task. This is shown in Figure 14, the tool will output the worst case response times of all tasks if the system is schedulable.

Name	T	O	C	pr
dummy1 (∂_1)	15	0	2	2
dummy2 (∂_2)	15	3	2	2
dummy3 (∂_3)	15	6	5	2
dummy4 (∂_4)	15	12	3	2

TABLE VI
GENERATED DUMMY TASKS FOR SUBSYSTEM **A**


Worst Case Response Times

Name	C	WCRT	D
dummy1	2	2	15
dummy2	2	2	15
dummy3	5	5	15
dummy4	3	3	15
A	1	3	5

Fig. 14. Times schedulability analysis (for subsystem **A**)

2) Code synthesis to Linux (kernel version 2.6.31-9): The subsystem (**A**) execution trace is illustrated in Figure 13, as illustrated, the four dummy tasks replace subsystem **B**. We let a video processing application (VLC[5]) replace task $taskA$ in subsystem **A** in our experiments. The release of subsystem **A** and dummy tasks 1-4 is done with two automata similar to the ones in Figure 7. We generate code, using the Times code generator for generating a Linux simulator. The simulator will run the automata, which is also generated by Times. We then replace the simulator with Linux kernel scheduling functions, which are exported by the scheduling framework Resch [12]. Resch is unique in that it does not require the user to make any changes in the Linux kernel, when implementing a scheduler in Resch. It runs as a kernel module, and the user implemented scheduler will act as a plugin kernel module to Resch (hence no kernel patches are required). The automata code generated from Times, is wrapped with Resch scheduling primitives, and it is executed as a kernel module in Linux. In the experiments, all tasks, i.e., the VLC application and the dummy tasks, are running as Linux real-time tasks.

We also ran the VLC application in a 2-level hierarchical scheduling framework, which is able to run a global scheduler, scheduling an arbitrary number of subsystems in one level. The subsystems themselves may have their own local scheduler. All schedulers (local and global) schedule with fixed-priority preemptive scheduling of periodic tasks/subsystems. The framework is implemented by the authors of the paper, and it runs as a plugin scheduler in Resch, i.e., as a kernel module. We executed subsystem **A** and **B** (Figure 12) with corresponding parameters, including rate monotonic priorities, in the hierarchical scheduling framework. Subsystem **B** corresponds to B in Figure 15 and subsystem **A** maps to A ($Idle$ is the idle subsystem). The VLC application (referred to as vlc_A in Figure 15) was running in subsystem A, the dummy

[5]VLC http://www.videolan.org/vlc

task $task\_B$ was running in B and dummy task $idle$ was running in subsystem $Idle$ (which has lowest priority among the subsystems). Task $linux$ is the Linux idle task which will run whenever $task\_B$, vlc_A or $idle$ does not run.

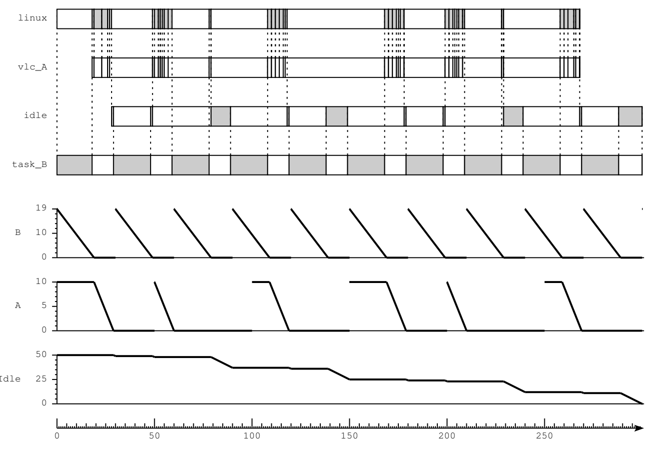


Fig. 15. Execution recording from the HSF scheduler

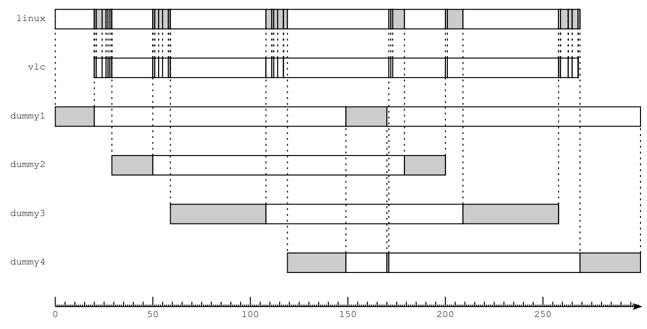


Fig. 16. Execution recording from the Times scheduler

Figure 16 shows the execution trace when running the Times automata in Resch, as a plugin scheduler. The dummy tasks ($dummy1$, $dummy2$, $dummy3$ and $dummy4$) in Figure 16 corresponds to our generated dummy tasks in Figure 13 (these tasks have highest priority). The VLC application was running as task vlc (intermediate priority) and the Linux idle task $linux$ was running with lowest task priority.

We ran all the experiments on an Intel Pentium Dual-Core (E5300 2,6GHz) platform, equipped with a Linux kernel version 2.6.31.9, running with load balancing disabled (no automatic task migration) for simplicity. The task execution recording was done with the tool Ftrace [13], and the recording of subsystem scheduling events were done by our own recorder [14] (which is integrated in HSF). The recordings were visualized (Figure 15 and 16) with the tool Grasp [15].

We measured the execution time of the VLC application, while it processed a 91 frame long video, with corresponding audio. The measurements were done 10 times for each scheduler, and the data presented (Table VII) represents the average values. The resulting data is presented as the number of frames displayed per second (Table VII). The measurements were done while scheduling the VLC application with the HSF scheduler, and the Times scheduler. When running VLC with only the native Linux scheduler, the video processing reached approximately 30 fps. The presented fps values shows that both schedulers (HSF and Times) gives almost the same amount of CPU power (approximately 20%) to the VLC application. However, VLC does not use all of its allocated CPU time (Figure 15 and 16) because its internal clock will decide when to process and display frames, which is dependant on the intended frame-rate of the application (which is 30 fps).

Scheduler	fps (average)
Times scheduler	25.3174938
HSF scheduler	25.3582266
Linux scheduler	30

TABLE VII
FRAMES PER SECOND (FPS) MEASUREMENTS OF VLC

The time points when the Times scheduler allocates CPU time to VLC (Figure 16), matches the points that are generated by the scheduling framework HSF (Figure 15), which implements the scheduler that is intended to be used in the final system. However, HSF "leaks" CPU time, as can be seen in Figure 15. This is due to that we set the budget of subsystem **B** to less than 20, so that the budget does not deplete at the same time as the other subsystem is released (which may cause our scheduler to execute the scheduling events in wrong order).

VI. Related work

Related work in the area of hierarchical scheduling originated in open systems [16] in the late 1990's, and it has been receiving an increasing research attention ever since. Since Deng and Liu [16] introduced a two-level hierarchical scheduling framework, its schedulability has been analyzed under fixed-priority global scheduling [17] and under EDF-based global scheduling [18], [19]. Mok *et al.* [20] proposed the bounded-delay resource model so as to achieve a clean separation in a multi-level hierarchical scheduling framework, and schedulability analysis techniques [21], [22] have been introduced for this resource model. In addition, Shin and Lee [7] 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 [10], [23], [24] and under EDF scheduling [7].

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 is similar to [25], 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. Lime *et al.* [26] model fixed and dynamic priority scheduling using time petri nets, which is similar to the work in [27]. Scheduler modeling is showed in [28] using the controller paradigm.

VII. Conclusion

We have shown how to perform schedulability analysis in the Times tool, where a subsystem within fixed-priority preemptive hierarchical scheduling is the system under analysis. The concept we present simplifies the analysis of the whole system by analysing one subsystem and abstracting the rest of the system (black-boxing). Iterating through all subsystems in this manner results in analysing the whole system. In each step, the black-boxing is done by replacing interfering subsystems with a small set of high priority tasks (which we refer to as dummy tasks). The procedure is described with an algorithm in the paper, and the output of the algorithm is a set of dummy tasks that are periodic with offsets. These tasks, and the tasks of the subsystem to be analyzed, are then modeled in the Times tool (with a task-table or timed automata). The last step is to run a simulation in Times which will generate the worst case response time of each task, thereby deciding if the subsystem is schedulable or not. The Times tool could traverse the scheduling tree and analyze each subsystem, resulting in a complete analysis of the whole tree. The simulation itself is essentially a response time analysis of tasks that are periodic, whereas some of them will also have offsets (the dummy tasks).

We have used the Times code synthesis and shown how to generate C-code of two example systems. The code has been extended to execute on an industrial platform (i.e. VxWorks), and also on a PC desktop platform (Linux). Hence, our proposed method has shown to be practical. After the code generation, a subsystem can be executed as if it would be running within a hierarchically scheduled system. Hence, our proposed approach supports early prototyping of hierarchically scheduled systems, by using our dummy-task algorithm together with our code synthesis for VxWorks and Linux.

Our example in VxWorks shows 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 compared to a manually coded scheduler. Our other example in Linux shows how a video processing application (VLC) is affected when running it in a prototyped subsystem. We have measured the frame-rate and compared the results from the same example system running in a 2-level hierarchical scheduling framework.

As future work, we plan to optimize the code synthesis (in order to minimize scheduler overhead) as well as to model and generate code for hierarchical scheduling frameworks. This is interesting in the context of proving the correctness of scheduling, since model checking could be used to verify the schedulers. As a last step of the contribution of this paper we plan to implement the concept in a tool, which will provide graphical modeling of systems, automatic generation of dummy tasks as well as automatic synthesis for various platforms (such as VxWorks, Linux and FreeRTOS).

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] M. Åsberg, T. Nolte, and P. Pettersson, "Prototyping Hierarchically Scheduled Systems using Task Automata and TIMES," in *Proc. of the 5th International Conference on Embedded and Multimedia Computing*, August 2010.

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

[8] 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.

[9] 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.

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

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

[12] S. Kato, R. Rajkumar, and Y. Ishikawa, "A Loadable Real-Time Scheduler Suite for Multicore Platforms," Technical Report CMU-ECE-TR09-12, 2009. [Online]. Available: http://www.contrib.andrew.cmu.edu/~shinpei/papers/techrep09.pdf

[13] T. Bird, "Measuring Function Duration with Ftrace," in *Proc. of the Japan Linux Symposium*, 2009.

[14] M. Åsberg, T. Nolte, and S. Kato, "A Loadable Task Execution Recorder for Hierarchical Scheduling in Linux," School of Innovation Design and Engineering (Mälardalen University), Tech. Rep., 2010.

[15] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems," in *Proc. of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, July 2010.

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

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

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

[19] 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.

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

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

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

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

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

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

[26] D. Lime and O. H. Roux, "Formal verification of real-time systems with preemptive scheduling," *Real-Time Syst.*, vol. 41, no. 2, pp. 118–151, 2009.

[27] K. Altisen, G. Gosler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine, "A framework for scheduler synthesis," in *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1999, p. 154.

[28] J. Sifakis, "Scheduler modeling based on the controller synthesis paradigm," in *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. London, UK: Springer-Verlag, 2002, pp. 107–110.