Framework for Validation, Test and Analysis of Real-time Scheduling Algorithms and Scheduler Implementations

Frank Golatowski, Jens Hildebrandt, Jan Blumenthal, Dirk Timmermann
Institute of Applied Microelectronics and Computer Science
University of Rostock
Richard-Wagner-Str. 31, 18119 Rostock, GERMANY
{Frank.Golatowski, Jens.Hildebrandt, Jan.Blumenthal, Dirk.Timmermann}@etechnik.uni-rostock.de

Abstract

This paper describes our work towards a rapid prototyping system for hard real-time systems focusing on scheduling algorithms and scheduler implementations. The framework aims at speeding up the decision making process during selection of a suitable scheduling algorithm for a real-time application. The framework supports various kinds of real-time scheduling algorithms, which can be simulated for evaluation purposes. Furthermore, implementations of these algorithms can be tested in a real-time operating system (RTOS) with real or synthetic workloads. The algorithms are implemented as software routines, which are part of the operating system (OS), or realized within a co-processor to free the operating system kernel from time consuming scheduling operations. Both kinds of implementations have to provide an application programmers interface that hides the algorithm from implementation issues. The target operating system of our framework is mainly RT-Linux, eventhough development for those systems is possible under Linux as well as Windows-NT. The framework proposed in this paper is new in that it supports the selection of the right algorithm and the right implementation for the target platform based on real-time scheduling analysis.

1 Introduction

When designing an application that requires real time capabilities of the underlying computer system a real time operating system is quite often used to relieve the developer from the burden of managing parallel execution of several tasks. Furthermore, usage of the operating systems API makes a software product portable to different hardware environments and even to different operating systems using the same API. The downside is that very often the developer has to achieve his goal not only due to the possibilities but also despite the limitations implicated by the OS. Especially in real time applications where execution time is a strict constraint, these limitations can not always be overcome by any possible workaround leading to the desired computational result.

One of these limitations is the scheduling algorithm implemented in a particular RTOS. Most commercial systems use static priority systems and only few of them offer alternative algorithms. The availability of open source RTOS like RTEMS or RT-Linux gives an opportunity to employ scheduling methods more suitable for a given set of tasks than the ones offered by off-the-shelf real-time operating systems.

In recent years, some frameworks and tools have been developed based on schedulability analysis. Such frameworks are Perts [5], Real-time Graphic Analyzer RTGA [7], Stress [8] and Timewiz [9].

Furthermore, a reconfigurable scheduling framework for the RED-Linux Real-Time Kernel has been offered [6].

EVASCAN [1] [2] is a general framework that uses YASA [4] as a tool for schedulability analysis. Operating system support is included in form of timing information [3]. The new redesigned framework called YASA adds a more generic framework in that it runs under both Windows and Linux with a real operating system as a target.

The extended framework YASA presented in this paper aims at evaluating a task set with given properties in terms of schedulability and compliance with given execution time constraints when using different scheduling algorithms and scheduler implementations. Special attention was given to dynamic scheduling methods and hardware supported scheduling as these technologies can provide very good performance but need thorough scheduling analysis to guarantee timeliness of the scheduled task set. In Section 2 this paper gives an overview over the main scheduling algorithms used and the technology of scheduling coprocessors

supported by the framework. Section 3 explains the structure and working principle of the framework and section 4 deals with special implementation issues and their consequences for the usage of YASA. Finally, section 5 gives an outlook onto further development of the framework.

2 Basic technologies

This section deals with the dynamic scheduling algorithms used inside the YASA framework and the supported scheduling coprocessors.

2.1 Scheduling algorithms

EDF [10] and LLF [11] have been proven to be optimal dynamic scheduling algorithms. A main advantage of these dynamic scheduling algorithms is a theoretically possible 100% processor utilization without deadline misses. From the implementation point of view, it is easier to realize the EDF than the LLF algorithm (see section 2.2.1). But both algorithms have also some drawbacks. In an overload situation all tasks of the given task sets miss their deadline in the EDF algorithms (known as the domino effect). Another drawback is that with the EDF algorithm it is possible to detect the violation of a deadline only after it has happened. With LLF it is possible for the scheduler to detect an impending deadline miss during the execution of tasks. On the other hand, if two or more tasks have the smallest laxity the algorithm suffers from thrashing, i.e. an excessive number of context switches. Another drawback of the LLF algorithm is its relatively high computational effort.

To overcome this drawback we have developed the Enhanced Least Laxity First algorithm (ELLF). This algorithm combines benefits from both algorithms. These benefits are optimality, possible processor utilizations of up to 100% and early detection of future missed deadlines. To minimize the long runtimes for the determination of laxity values we use a coprocessor as a hardware based accelerator. This coprocessor will also be supported by an application programmers interface in RT-Linux and in the framework YASA presented here.

2.2 Task model for dynamic schedulers

A set of independent tasks τ ($\tau_i \ldots \tau_m$) is to be executed on a real-time system. Each task $\tau_i \in \tau$ is characterized by the following parameters:

$S(\tau)$ start time of task set τ, booting time

$A(\tau_i)$ arrival time of task τ_I

$C(\tau_i)$ computation time of task τ_I

$D(\tau_i)$ deadline of task τ_I

$P(\tau_i)$ period of task τ_I

Based on this parameters, scheduling decisions have to be made.

2.2.1 Least Laxity First

For the Least Laxity First algorithm the following computations and times are important for the selection of the next task to run.

$L(\tau_i)$ laxity of task τ_i

$L(\tau_i,t)$ laxity of task τ_i at time t

t time since bootup, $0 < t < \text{infinity}$

$d(\tau_i,t)$ deadline in current period since bootup

$D(\tau_i,t)$ deadline of task τ_i at time t

$d_i(\tau_i,t) - t$ (1)

$cd(\tau_i,t)$ computed time

$C(\tau_i,t)$ remaining computation time in period

$C(\tau_i) - cd(\tau_I,t)$

To determine the laxity:

$$L(\tau_i) = D(\tau_i)-C(\tau_i) \quad (2)$$

The laxity at time t:

$$L(\tau_i,t) = D(\tau_i,t)-C(\tau_i,t) \quad (3)$$

Now we substitute D and C:

$$L(\tau_i,t) = d(\tau_i,t) - t - (C(\tau_i) - cd(\tau_i,t)) \quad (4)$$

Based on equation (4) the scheduling decision in the LLF algorithm is made. The task with the smallest laxity will be selected for execution.

2.2.2 Earliest Deadline First

The EDF algorithm makes its scheduling decision based on equation (1). The task with the nearest deadline will be selected for execution. The computation and comparison of deadlines is very simple and hence the EDF algorithm is the dynamic scheduling method that currently is implemented most often. The drawback is, that according to this algorithm even tasks are executed which in no way will finish until their deadline. This consumes computation time that may otherwise be needed to execute other tasks in time and can eventually lead to a whole set of tasks missing all their deadlines.

2.2.3 Enhanced Least Laxity First

The ELLF-algorithm [2] is an improvement of the LLF algorithm. ELLF is more complex and makes its scheduling decisions in two steps. Since the detailed description of this algorithm is beyond the scope of this paper, only a short description of the working principle is given here.

In the first step, analogue to the LLF algorithm, the tasks that have the smallest laxity are determined. In the second step, out of these tasks the one with the earliest deadline is chosen and brought to execution while the other tasks enter a newly introduced task state. This state (called *excluded state*) prevents them from preempting the currently running task. This decreases the amount of unnecessary context switches which are typical for the LLF algorithm and make that algorithm practically non-usable.

2.3 Scheduling Coprocessors

One method to speed up scheduling is to move the scheduler function as a whole or in parts to a dedicated hardware device. Of course, such a coprocessor is effective only if the gain in execution speed of the implemented functions is not consumed by the time needed to transfer input data and computation results between the hardware device and the operating system routines. Beside that, only functions that do not require access to CPU internal resources (i.e. registers) can be transferred to the coprocessor. Hence, for schedulers, the parts that are most efficiently implement in hardware are online computation of priorities, priority comparison and – based on that operation - selection of one task to be executed next. Although the latter two operations do not profit from hardware realization in such a high degree as does priority computation that can rely on hardware parallelism, the combination of all three parts of the scheduling operation in one hardware device avoids the necessity to transfer computed task priorities back to the operating system. Provided task states and task parameters used for priority computation are stored inside the coprocessor, communication overhead at scheduling time merely consists of an optional start signal for the coprocessor and an identifier for the next task to run that is transmitted back to the operating system as the scheduling result.

Obviously, such hardware effort is justified only if the performance gain is significant. Hence, the discussed here type of scheduling coprocessor is used primarily for dynamic priority algorithms like EDF, LLF or ELLF.

The coprocessors used in the rapid prototyping framework described in the following sections have a structure as given in Figure 2.1. Each task is represented by a functional block – the so called task module – that determines priority values and for that purpose holds task parameters and task states. These values are accessible via registers that have to be initialised during system setup and when new tasks are added to the system. Task states have to be updated during run time as they determine whether a task is active, i.e. takes part in scheduling, or not. The number and meaning of other parameters stored inside a task module depend upon the implemented scheduling algorithm. For example, EDF algorithm uses task deadlines as scheduling criterion. Hence task modules for EDF scheduling store the deadline in relative form, i.e. the number of remaining time units until deadline is reached. This value is counted down automatically while the task is active. A missed deadline is equivalent with the deadline value being zero while the task is still active. More complex algorithms like LLF or ELLF apart from the deadline use the remaining run time of a task, a value that is updated automatically as long as a task is executed. Task laxity, the difference of remaining time until deadline and remaining run time, is used to determine the order of task execution. These computations are done simultaneously for all tasks at scheduling time and thus make up the largest part in execution speed gain compared to a software solution.

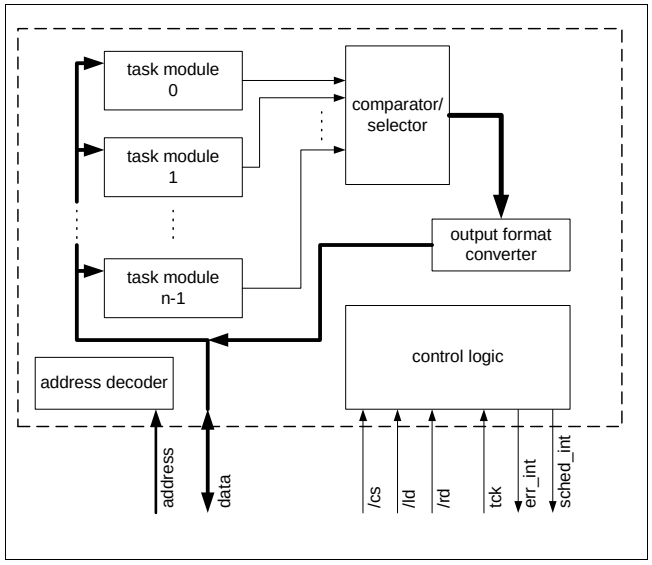

Figure 2.1 Scheduling coprocessor, internal structure

Priority values computed inside the task modules are compared in the comparator module to determine the next task to run. The strategy used therein depends on the employed scheduling algorithm. The scheduling methods mentioned above all search for the task with the smallest priority value. But there may be other algorithms that determine the biggest value or combine the comparison with some other conditions. In any case, the coprocessor returns an identifier that names the task belonging to that value determined by the comparator.

Although the described here coprocessors implement different algorithms, their interface to the host computer system and thus to the operating system is the same. That allows an easy integration of different hardware supported scheduling algorithms into the operating system. For the prototyping framework described in this paper that means that different coprocessors can be compared with low overhead regarding their suitability for a given task set since the surrounding operating system routines do not have to be altered significantly.

3 The YASA Framework

The YASA framework allows selection of the right scheduler during the design phase of a project while relieving the programmer from the necessity to know details of the scheduler implementation in different environments. Therefore, the different schedulers are hidden behind an application programmers interface (API). Usage of YASA in different operating systems is another reason to do so. We call this layer the *executive*. Derived from our goal to use different schedulers and executives there must be defined a small and unambiguous but expandable API. The whole project is designed to run on different hardware platforms and is only restricted due to executives depending on special operating system functions.

Figure 3.1 is shows the internal structure of the framework. There are two main components, the graphical user interface (GUI) using the QT library and the command line based simulation environment (dotted rectangle). The whole developer project can be configured from within the GUI. It is possible to define new tasks with properties like computation time, deadline, period or behaviour in case of a deadline miss. For each task a start function can be declared. Furthermore it is necessary to make some adjustments like selecting the desired executive and scheduler on each CPU.

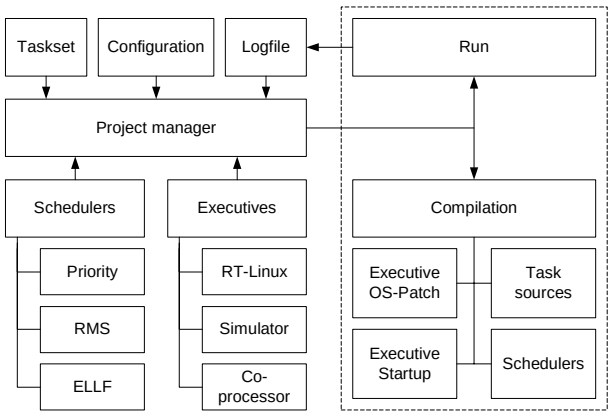

Figure 3.1 Framework, internal structure

After entering all needed information the project can be compiled. The compilation process includes all selected components like executives, schedulers, project functions, task information and startup codes. These parts are linked in different forms depending on characteristics of the executive. That can be a monolithic executable or a system of several shared libraries. In some cases, the kernel of the operating system has to be recompiled. After making the executive the project can be started. An internal log file mechanism is registering all internal information like calling times of the scheduler, deadlines, task switches or locking of resources. After finishing the execution the log file is evaluated by the graphical front end and is illustrated in clearly structured diagrams. By this it is possible to log every action in the RTOS.

In our framework the different restrictions of the particular operating system result in tight limits. It will not be possible to run or compile certain task sets on every executive, if these tasks use specific support functions of the OS. For instance the function *malloc()* to allocate memory is a blocking function. For this reason it is not available in kernel mode within most desktop operating systems in contrast to embedded systems. Task sets using this function can not compile successfully on executives running in kernel mode. The employment of *kmalloc()* restricts the task sets to the maximum page size and to executives running under Linux and its derivates.

Another restriction is given due to the implementation of the scheduling system. If the desired operating system supports only priority based scheduling selection of a scheduler will be restricted.

4 Implementation Issues

4.1 Software Scheduler Modules

The software scheduler modules are defined in small ANSI-C modules. These modules, compiled as shared libraries or statically linked, are loaded and initialised at runtime during the starting phase of the executive. Usually, the modules can contain several member functions like *init()*, *schedule()* or *cleanup()*. In the initialisation phase a pre-defined attribute structure is used to configure the scheduler. Due to compatibility issues, the schedulers are using only functions of the executive to communicate with the environment. Information about the operating system and used hardware are hidden from the scheduler modules.

During the runtime of the executive the member function *schedule()* of the current scheduler module will be called every time the system scheduler is called to determine the next task to be executed. One result of the encapsulation

process is that the actual scheduler function becomes very small. For instance, a priority based scheduler consists of just 25 lines of source code and is able to run on different executives.

4.2 Hardware scheduler modules

Another feature of the framework, although still in the development phase, is support of hardware scheduler modules which can be combined with several executives. These scheduling coprocessors are realized in programmable hardware (FPGAs) that can be loaded with different configuration bit streams to implement different scheduling algorithms. All these different hardware schedulers have an identical user interface so that integration into the executive becomes independent from the implemented scheduling algorithm.

The projects of the framework are independent from the scheduler type. So every task set can be scheduled with either software schedulers or hardware schedulers to compare the performance and trade performance gains against the effort of additional hardware.

4.3 The Executives

The executive describes the environment in which the task sets and the desired schedulers are used. It provides an API and hides the operating system from the task set. Access to different operating system structures and functions like threads and scheduling parameters is possible only via that interface. Furthermore, additional internal functions like making log files or synchronisation of task sets at start time are included in the API.

In real operating systems the executive is integrated mostly by patching the original kernel. The final state of executives depends mostly on features of the operating system as well as implementation details.

There are currently two completely different executives, the *Simulator* and the *RT-Linux executive*.

4.3.1 Algorithm simulation

The algorithm simulator is a virtual multi processor machine. This executive can be combined with all available schedulers defined in the framework. It supports different methods to handle deadline misses and is able to use different protocols to assign resources such as semaphores and mutexes. The simulator is independent from the operating system.

After the definition of the task set the GUI will generate source code containing a task array with attribute information about these tasks. Then, the compilation process will be started with translating the core of the executive as shared library or Windows DLL. Next, the schedulers and the task array will be compiled to several objects and linked to one executable. The advantages in this is that the executive has to be recompiled only if developer is changing its settings. After compilation the project will be started using information from the task array created earlier by the GUI. All actions within the simulator will be logged and parsed later in the GUI.

One of the important restrictions of the simulator is that it is not capable of simulating real programs. This executive is ignoring the start functions of tasks mentioned above. The reason is that different executives are running in different system environments. To run real programs within the simulator, it would be necessary to emulate the real world, in particular the API of the operating system used in each task set. Furthermore the interaction with the environment and the dynamically behaviour would have to be emulated. This is nearly impossible with reasonable simulation effort.

To alleviate these effects the framework provides methods to define start times of asynchronous tasks and allocation times of resources. These times are used to simulate blockings and standby times. It is necessary to have know-how about progress of the tasks to get results close to reality. That is why the simulator is only useful to get information about general schedulability.

4.3.2 RT-Linux (Integration in RT-Linux / API)

Real time Linux (RT-Linux) is a hard real time extension to conventional Linux. The current version 3.0 is available on different hardware platforms like x86, PowerPC and Alpha and supports Symmetric Multiprocessing (SMP). We chose RT-Linux due to the POSIX style API, the possibility to change the original scheduler with small effort and because of the availability of the operating system source code.

Usually, RT-Linux is started by loading several kernel modules like *rtl_sched.o* or *rtl_fifo.o*. The developer can change or improve these modules if necessary.

RT-Linux uses a static priority based scheduler by default. That means that only non-suspended tasks with the highest priority will be chosen. This scheduler is very fast and simple but static. In its spare time, that means if no real time task is running it will execute Linux as lowest priority thread.

RT-Linux does not know anything about computation time, computed time and deadlines so there were some enhancements necessary. We simply added some variable declarations to the specific schedulers, threads and scheduling structures to store the required information. This is done in the “Executive OS-Patch” of the RT-Linux executive (Fig-

ure 3.1). We did not change the current API of RT-Linux due to compatibility issues to conventional projects but we patched some function bodies in the original scheduler module to guarantee the functionality of projects using default values. There are added some API calls like *yasa_setscheduler()* usage of which is optional.

Furthermore, based on *rtl_sched.o* we implemented an additional modular scheduler technology so that the scheduler can be changed during runtime for every CPU. In this way the developer can create his own task set on different CPUs in conjunction with different schedulers.

All schedulers use the same data structure describing tasks. These structures contain more data members than the current scheduler may actually use. This is necessary since these structures have to bridge the difference between the task models of all schedulers and that of the operating system.

When the developer creates a new project YASA will generate the source code of the task set, basically an array of thread attributes. Next, YASA starts the compilation process of the whole RT-Linux kernel and some new modules named "yp_project.o", "ys_scheduler.o" and "logfilereader.o". The first two modules are used to start the project and set the schedulers. The startup code of the RT-Linux executive initialises the threads using the parameters contained in the task arrays. All threads will be started in suspended mode. After successful creation of the task set a synchronized start of the project will follow and will reschedule all real time threads having start time 0.

To integrate or to consider non real-time tasks within RT-Linux systems two possibilities exist. It is simple to integrate those tasks as Linux threads or processes. But in this way it is hard to communicate between RT-Linux threads and the non-real-time Linux threads. The other possibility is the integration as RT-Linux threads. Therefore it is necessary to initialise the task structures with default parameters guaranteeing that these tasks are executed only when there is no real time task pending.

During run time the Linux thread will be in the idle time of the real time kernel. Hence, in fully utilized environments that will seem like a hanging system. After reaching the defined end time, the project will be stopped and the third module "logfilereader.o" will be started to transfer the scheduling information into user space.

Most of the schedulers used in YASA need detailed information about tasks like computation time. The developer should be aware that it can be a lengthy and intricate task to determine this necessary values and this must be done before entering the periodic state of the threads during runtime. The computation time is depending on everything in the computer: CPU speed, memory performance, I/O activities, caches and of course the system environment. If the computation time given to the scheduler is smaller than actual execution time many of the dynamic schedulers will not work correctly.

4.3.3 RT- Linux Executive with HW-Scheduler

The RT-Linux executive with hardware scheduler is an improvement of the RT-Linux executive. It is similar to the executive mentioned in section 4.3.2 but it is using a hardware scheduler to speed up the scheduling process. Task sets running successfully with the RT-Linux executive run with this executive, too

At start time of the executive the coprocessor device is loaded with a configuration according to the selected scheduling algorithm and initialised with the parameters from the task structure. Changes of the parameters in the task structure have to be announced to the coprocessor. Every time the system scheduler is called this executive will start the coprocessor to determine the next task to be executed.

5 Conclusions

The framework presented in this paper gives the developer the possibility to evaluate the schedulability of his application from an early simulation with synthetic load to execution of real code using hardware or software schedulers. Development is possible under and for different operating systems.

The GUI of the framework allows an easy comparison of scheduling methods and technologies by giving an overview over scheduling events, resource requirements or execution times. This makes YASA a good tool for education purposes as well.

Future developments of YASA see the extension of the range of supported scheduling algorithms and executives. Furthermore, a deeper integration of scheduling coprocessors is planned.

References

[1] Golatowski, F.; Hildebrandt, J.; Timmermann, D.: *Rapid Prototyping with Reconfigurable Hardware for Embedded Hard Real-Time Systems*. 19th IEEE Real-Time Systems Symposium 98, WIP -Session, Madrid, Spain, 1998

[2] Golatowski, F.; Hildebrandt, J.; Timmermann, D.: *Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems*, 11th Euromicro Conference on Real-Time Systems, York, England, 1999

[3] Golatowski, F.; Timmermann, D.: *Using Hartstone Uniprocessor Benchmark in a Real-Time Systems Course*. Third

IEEE Real-Time Systems Education Workshop, Poznan, Poland, 11/98

[4] *Yasa- Yet Another Schedulability Anayzer*, University of Rostock, 1998, http://www-md.e-technik.uni-rostock.de/ma/gol/yasa/

[5] Liu, J. W. S.; Liu, C. L.; Deng, Z.; Tia, T. S.; Sun, J.; Storch, M.; Hull, D.; Redondo, J. L.; Bettati, R.; Silberman, A.: *PERTS: A prototyping environment for real-time systems.* International Journal of Software Engineering and Knowledge Engineering, 6(2):161-177, 1996.

[6] Wang, Y-C.; Lin, K-J.: *Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel.* 20th IEEE Real-Time Systems Symposium, 1999, Phoenix, Arizona

[7] Ripoll, I.: *RTGA Real-time Graphic Analyzer*, 2002, http://bernia.disca.upv.es/rtportal/apps/rtga/index.html

[8] Audsley, N.C.; Burns, A.; Richardson, M.F.; Wellings, A.J.: *Stress: A Simulator for Hard Real-Time Systems*. Software-Practice and Experience, Vol. 24(6), p. 543, 564 (June 1994).

[9] *Timewiz- An Architectural modelling, analysis, and simulation environment for real-time systems*. White paper, http://www.timesys.com/pdf/timewiz_ds.pdf

[10] Dertouzos, M.: *Control Robotics: the procedural control of physical processors.* IFIP Congress, pp 807-813, 1974

[11] Mok, A.K.: *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Dissertation, MIT, 1983