An application-based EDF scheduler for OSEK/VDX

Claas Diederichs
INCHRON GmbH
14482 Potsdam, Germany
claas.diederichs@inchron.de

Ulrich Margull
1 mal 1 Software GmbH
90762 Fürth, Germany
margull@1mal1.com

Frank Slomka
University of Ulm
89069 Ulm, Germany
frank.slomka@uni-ulm.de

Gerhard Wirrer
SiemensVDO AG
93055 Regensburg
gerhard.wirrer@siemens.com

Abstract

Earliest deadline first scheduling performs processor utilization up to 100 percent and improved robustness in overload situations. However, most automotive applications are running under static priority policy. Because of this, the standard operating system in the automotive industry, OSEK/VDX, just supports priority scheduling. This paper describes an EDF scheduler plug-in for OSEK/VDX. The plug-in provides EDF scheduling without changes to the operating system by delaying task activations.

The add-on was tested for an engine management system developed by SiemensVDO. Results of this experiment are presented and discussed, showing that the EDF scheduling techniques can improve the system in aspects of robustness and resource utilization.

1. Introduction

Because most commercial real-time operating systems (RTOS) are based on a limited set of fixed priority levels[1], commercial embedded systems using an RTOS mostly use static priority based scheduling. Static scheduling such as rate monotonic scheduling (RMS) or deadline monotonic scheduling (DMS) is known not to be optimal in all cases while dynamic scheduling techniques such as earlies deadline first (EDF) can utilize systems with 100% CPU-load [4],[1]. [1] compares the RMS scheduling with EDF in aspects of implementation complexity, runtime overhead and robustness during overload, as well as jitter and latency. The conclusion states that the advantage of RMS is a simpler implementation for kernels that do not provide support for timing constrains. EDF on the other hand allows full processor utilization and better responsiveness for aperiodic activities.

This paper introduces an EDF plug-in which can extend every priority based scheduled operating system. It will be shown that it is easy to extend a commercial operating system kernel such as the OSEK/VDX implementation RTA-OSEK. It is shown by a case study how the new scheduling algorithm performs in an industrial car environment. An engine management system (EMS) is adapted to the EDF plug-in running in its legacy environment. First, the system is tested by using a new embedded simulation tool. The results of the simulation are then verified by a HIL (Hardware In the Loop) execution.

2 EDF plug-in for RTOS with static priorities

The real-time operating system OSEK/VDX does not support dynamic priority assignment. All priorities are assigned at design time. Therefore, EDF implementations that use dynamic priority assignments such as the CLOCK algorithm ([6]) cannot be used. One possibility to implement EDF scheduling is to change the RTOS core. This is not always feasible as the used RTOS might be closed-source.

The described algorithm allows EDF scheduling on systems with static priorities such as OSEK/VDX without manipulation of the RTOS source code. It uses OS features such as task switching and resource handling of the original priority-based system, implementing the EDF behavior by delaying activations of tasks. The algorithm only works with basic tasks with the states inactive, ready and running (Figure 1). Active waiting cannot be used, therefore OSEK/VDX Extended tasks are not supported by this approach, only conformance classes BCC1 and BCC2 can be used.

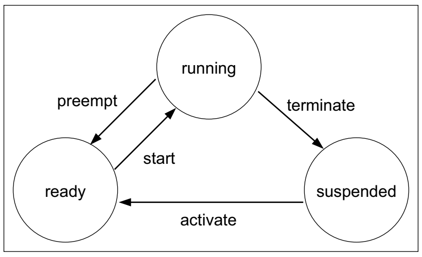

Figure 1. Basic task state model

2.1 Architecture of the plug-in

The EDF plugin requires that the task priorities are assigned Deadline Monotonic (DM). The plug-in is build on top of the RTOS API. All task activations are invoked using the plug-ins API, the plug-in itself uses the RTOS API (Figure 3). The EDF-module organizes tasks in a list sorted by absolute deadlines. Furthermore, a *delayed* state is added to the state model (see Figure 2). The algorithm implements EDF behavior by selectively delaying activations of tasks. A task in the delayed state is in the EDF-list, but its activation is not propagated to the operating system. Therefore, a delayed task cannot interrupt a running one.

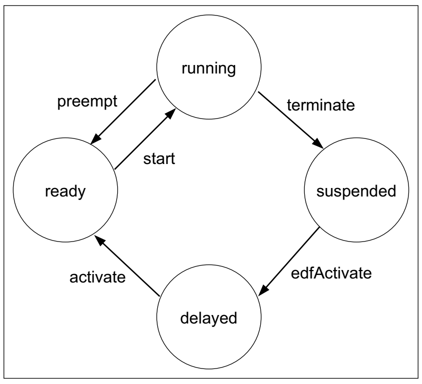

Figure 2. Task state model with extension

2.1.1 Task activation

If a task is activated, its absolute deadline is calculated. It is then inserted into the list and thus in the state delayed. This activation is called *edfActivate*, for a better differentiation from the activation of the RTOS. If the newly activated task is the first in the list, it is activated (Figure 3). Otherwise, it remains delayed to prevent interruption of the current running task.

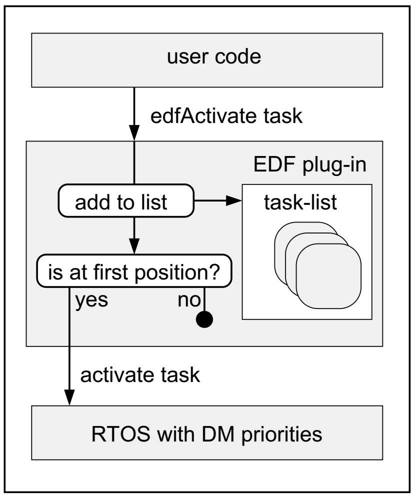

Figure 3. EDF module behavior at task activation

If a newly edfActivated task has a longer absolute deadline than the current running one, it remains in the state delayed and cannot interrupt the running task even if it has a higher priority. If a newly edfActivated task has shorter absolute deadline, its relative deadline is also shorter. Because of the DM priority assignment, its priority is higher and it interrupts the current running task.

2.1.2 Task termination

If a task terminates, it is removed from the EDF-list. The new first task in the list is activated if it is in the delayed state (Figure 4).

2.2 Verification of the plug-ins behavior

A basic task is described through the following parameters:

$$\tau = (r, \Delta e, d)$$

A task consists of a ready time r, an execution time Δe and an absolute deadline d. D is the relative deadline.

$$r_i + D_i = d_i \tag{1}$$

To describe the algorithm, additional parameters are used:

$$\tau = (r, \Delta e, d, pr, st, lp)$$

The parameters are the task priority pr, the task state st and the EDF-list position lp in the EDF task-list L.

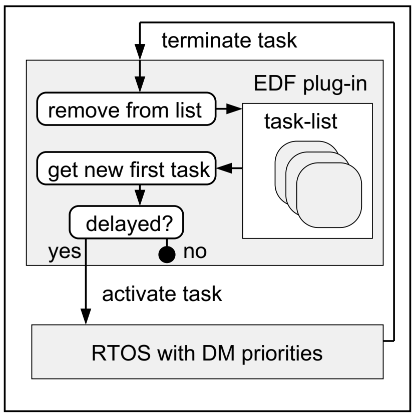

Figure 4. EDF module behavior at task termination

It is assumed:

$$\forall(\tau_i \in L) : r_i, d_i, D_i \in \mathbb{R}^+ \tag{2}$$

The absolute deadline of a task is always greater than the ready time:

$$\forall(\tau_i \in L) : d_i > r_i \tag{3}$$

The priority scheme of the RTOS must be deadline monotonic:

$$\forall(\tau_i, \tau_j \in L) : (D_i < D_j) \Rightarrow (pr_i > pr_j) \tag{4}$$

The EDF-list is sorted by absolute deadlines (d_i). The first position of the list is position zero.

$$\forall(\tau_i, \tau_j \in L) : (d_i < d_j) \Rightarrow (lp_i < lp_j) \tag{5}$$

To proof that the EDF-module behaves like an EDF scheduled system, it must be shown that at any time the task with the shortest absolute deadline is executed. The tasks in the task list are ordered by absolute deadlines. Therefore, it must be proven that the task at the beginning of the list is running. In this Section, an *active* task is in the state ready or running. A task in the task list is either active or delayed.

At system start, the task list is empty. If a task τ_a is edfActivated, it is positioned at the beginning of the list and activated. It is immediately running because it is the only task ready to execute.

If a second task τ_b is edfActivated, there are two cases:

- The new task is inserted behind the currently running task into the list ($d_b > d_a$)
- The new task is inserted at the beginning of the list ($d_b < d_a$)

If the newly arrived task is inserted behind the currently running into the list, it is delayed. Therefore, it cannot interrupt the currently running task.

If τ_b is inserted at the beginning of the list, it is activated. To get executed, τ_b must interrupt the task τ_a and thus the priority of τ_b must be higher than the priority of task τ_a

To guarantee that the task at the beginning of the list is always executed, its priority must be higher than the priority of every other task in the list that is active:

$$(lp_i = 0) \Rightarrow \forall(\tau_j \in L | \tau_i \neq \tau_j \land st_j \neq \text{delayed}) : pr_i > pr_j \tag{6}$$

Delayed tasks in front of active tasks are activated when reaching the start of the list. To fulfill requirement (6), every task in front of an active task must have a higher priority:

$$\begin{aligned}\forall(\tau_i, \tau_j \in L | st_j \neq \text{delayed}) :\\ (lp_i < lp_j) \Rightarrow (pr_i > pr_j)\end{aligned} \tag{7}$$

Equation (7) implies (6).

To proof that the system behaves correctly, it must be shown that equation (7) is satisfied.

A task can only be active, if it has been at the start of the list. Otherwise, it would not have been activated.

$$\forall(\tau_i \in L | st_i \neq \text{delayed}) : lp_i{}' = 0$$

If a task is in front of an active task, it was inserted at a later time:

$$\begin{aligned}\forall(\tau_i, \tau_j \in L | st_j \neq \text{delayed}) :\\ (lp_i < lp_j) \Rightarrow (r_i > r_j)\end{aligned} \tag{8}$$

Using sort criteria (5), the following is true:

$$\begin{aligned}\forall(\tau_i, \tau_j \in L | st_j \neq \text{delayed}) :\\ (lp_i < lp_j) \Rightarrow (d_i \leq d_j)\end{aligned} \tag{9}$$

Combining (9) with (8) results in:

$$\begin{aligned}\forall(\tau_i, \tau_j \in L | st_j \neq \text{delayed}) :\\ (lp_i < lp_j) \Rightarrow (d_i \leq d_j) \land (r_i > r_j)\end{aligned} \tag{10}$$

Ready times and deadlines of tasks are always ≥ 0 (2) and the deadline of a task is always greater than the ready time (3).

Therefore, the following equation is true:

$$(d_i \leq d_j) \wedge (r_i > r_j) \Rightarrow (d_i - r_i < d_j - r_j) \tag{11}$$

The ready times and absolute deadlines can now be replaced by the relative deadlines using (1):

$$(d_i \leq d_j) \wedge (r_i > r_j) \Rightarrow (D_i < D_j) \tag{12}$$

Equation (12) can be inserted into (10):

$$\begin{aligned} &\forall(\tau_i, \tau_j \in L | st_j \neq \text{delayed}) : \\ &(lp_i < lp_j) \Rightarrow (D_i < D_j) \end{aligned} \tag{13}$$

Using the DMS criteria (4), the requirement (7) is satisfied:

$$\begin{aligned} &\forall(\tau_i, \tau_j \in L | st_j \neq \text{delayed}) : \\ &(lp_i < lp_j) \Rightarrow (D_i < D_j) \Rightarrow (pr_i > pr_j) \end{aligned} \tag{14}$$

2.3 Complexity and runtime overhead

2.3.1 List operations

For a list, the time effort for inserting a new task is $O(n)$, where n is the size of the task-list. Using a heap, the time effort is $O(\log n)$. However, tasks that are inserted in the back part of the list have longer deadlines and therefore longer periods. Tasks that are scheduled often have a short deadline – they are inserted in the front part of the list. Therefore, a list might be the best option to implement the algorithm.

2.3.2 Runtime overhead

The runtime overhead of the algorithm at task activation is created by three actions:

- Guard the action to prevent interruptions during the scheduling, that can be done by disabling interrupts that have access to RTOS features (e.g. level 2 interrupts of OSEK/VDX)
- Retrieve timestamp from RTOS and calculate the absolute deadline from the current time and the relative deadline
- Insert the newly activated task into the sorted list (see Section 2.3.1)

The timing overhead when finishing a task is created by two actions:

- Guard the action
- Remove task from the list

Manipulation of the task list must be guarded by a critical Section. In OSEK/VDX, this can be done by using *SuspendOSInterrupts()* and *ResumeOSInterrupts()*, to deactivate level 2 interrupts. The guarding of the list is one part of the runtime overhead of this EDF algorithm.

The relative deadline of a task must be known by the EDF-module. To create the absolute deadline, a timestamp has to be retrieved by the system. The time effort depends on the underlying RTOS and is another part of the runtime overhead of this EDF implementation.

Removing the task from the list is mostly very fast: in full-preemptive systems the task is at the beginning of the list. Only with cooperative tasks or resource sharing it is possible that another task is at the first position of the list, activated but waiting for a resource to be released.

2.3.3 Memory overhead

The memory overhead is linearly dependant on the list size. Using a linked list, every item in the list consists of four parts:

- Pointer to task function
- Absolute deadline
- Task state, if the task is active or delayed
- Pointer to next list entry

For systems with 32 bit pointers, counting the deadlines in μs using 64 bit and using 1 bit for the task state, the space overhead would be 129 bit per list item. The list size can be deduced from the number of tasks and multi-activations. If multi-activation is used in the operation system, it can be removed to save space, because the EDF-module does not activate the same task twice.

In [2], an efficient algorithm for time representation for small embedded systems, using smarter time representations with 32 bit or 16 bit is discussed. Using time representation with smaller sizes does not only save space, but also speeds up deadline calculation and task list management, because adding and comparing 64bit numbers on 32bit or 16bit microcontrollers is time-expensive.

Assuming a list size of 32 maximum tasks, using 32 bit time representation and a linked list, the space overhead of the EDF algorithm is $32 * 97$ bit $= 388$ byte.

When space is a matter, choosing a heap instead of a linked list would save 32 bit per list item: $32 * 65$ bit $=$ 260 byte.

3 Case study engine control

An EMS currently under development by SiemensVDO is adapted to the EDF plug-in running in its legacy environment.

3.1 Engine Management System

An Engine Management System (EMS), also known as Engine Control Unit (ECU) or Engine Control Module (ECM) is an embedded system which controls all major aspects of a combustion engine. For example, an EMS controls ignition timing, fuel quantity and other cylinder related aspects. An EMS has various sensors for observing the engine such as air sensors, pressure sensors, crankshaft/camshaft position sensors or accelerator pedal acquisition sensor. The EMS has communication links to other electronic control units in the car such as the transmission.

The investigated system is a single processor EMS controlling an engine with eight cylinders. The system features about 30 tasks, some time-triggered, some engine-position triggered and some aperiodic triggered tasks. The EMS uses the OSEK/VDX implementation RTA-OSEK. At the time of this experiment, the EMS was under development.

3.2 OSEK/VDX

A proof-of-concept implementation is done for the OSEK/VDX operating system. OSEK/VDX uses the following functions for task control:

ActivateTask(τ_x) can be called from tasks and level 2 ISRs to activate the task τ_x.

TerminateTask() is the last statement of a Task τ_x, τ_x is terminated.

ChainTask(τ_y) is the last statement of the Task τ_x, τ_x is terminated and the task τ_y is activated.

These functions are used by the EDF-Module. The user-code calls to those RTOS functions are replaced with EDFActivateTask(), EDFTerminateTask() and EDFChainTask().

EDFActivateTask() inserts the new task into the task list. It then calls ActivateTask() if needed.

EDFTerminateTask() removes the task from the task list. It then calls ChainTask() if the next tasks needs to be activated, TerminateTask() otherwise.

EDFChainTaks() removes the old task from the list and inserts the new task. It then behaves like EDFTerminateTask().

3.3 Experiment set-up

This Section describes the details of the implementation used for the below described experiment.

System ticks (64 bit) are used for time representation inside the EDF algorithm and an 8 bit type is used for the state. A linked list is used where each list item consumes 17 bytes. The macro for task activation requires the task pointer and the relative task deadline (given in μs). The macro for task termination requires the task pointer for detecting the correct list item. EDFChainTask() is not used in this implementation.

The prototype implementation includes deadline monitoring and support for switching the EDF scheduler on and off. For each task, the amount of missed deadlines and the maximum response time is monitored.

The implementation for the EMS is described in Section 3.1. The two fastest tasks with a period of $1ms$ are not scheduled by the EDF algorithm. Those tasks are activated by the rta-OSEK timetable and could not be transformed to use the EDF activation macros in the short experiment time, this is no limitation of the EDF algorithm. Both tasks have a higher priority than all tasks scheduled by the EDF module. Three tasks are background tasks, those tasks also are not scheduled by the EDF scheduler. All background task have a lower priority than the EDF tasks. The calls to the OSEK/VDX ActivateTask and TerminateTask macros for the EDF scheduled tasks are replaced with the EDF macros. The task priorities are assigned deadline monotonic.

To manually increase the CPU-load, a dummy loop with an arithmetic computation was inserted into one of the tasks with 1ms period. During execution it can be manipulated, how often the loop is iterated.

3.4 Simulation

Because the time of the test devices (hardware In the Loop – HIL) is limited, the algorithm was tested and verified using a real-time simulation tool for embedded systems. The real-time simulation tool chronSim [3] was used to test the implementation. ChronSim simulates the system behavior offline and is capable of visualizing the preemption and displacement of tasks. The simulator works with c-code and features an OSEK/VDX implementation.

3.5 Hardware In The Loop test

The system was investigated at the HIL (Hardware In the Loop) testing equipement, where the environment of the embedded system is close to the real operation environment. The engine speed was simulated with 4160 rpm. The pedal value (PVS) was set to 50%. At this engine speed, the CPU-load is 81.7% with DMS. With higher engine speed, the CPU-load decreases again because some functionality is not executed anymore when the system exceeds a specific engine speed.

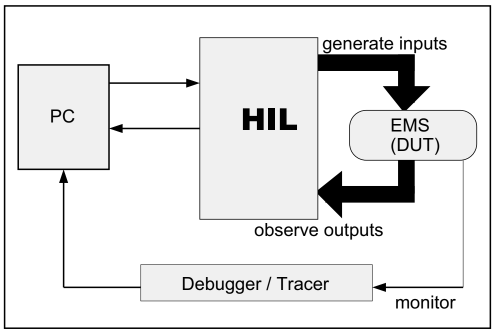

Figure 5. Architecture of the HIL test of a Device Under Test (DUT)

Basically, two things are monitored: the CPU-load that can be reached with DM and EDF, and the CPU-load overhead caused by the EDF module. The deadline monitoring was observed for both EDF and DM.

To determine the overhead of the EDF-Module, deadline monitoring is disabled and the CPU-load is compared to the CPU-load caused by DM (also without any deadline monitoring). Enabling the EDF algorithm at the above described engine speed and PVS value increased the CPU-load from 81.7% (with DM) to 82.3% (EDF). Measurements with other engine speed also showed an EDF-module runtime overhead of 0.5% to 0.8% compared to DMS.

To determine the maximum CPU-load that can be reached with both scheduling techniques the CPU-load is increased by performing more iterations of the dummy loop for the overhead generation.

This technique of increasing CPU-load is not optimal. Inside the task with the highest priority and a period of 1ms runtime is ”stolen” from the system. For smoother overhead generation, an interrupt with smaller runtime and higher frequency could be used to provide a better distribution of the additional runtime.

The DM scheduling could be increased to a CPU-load of 85.2% (using the above described technique). Tasks are lost by the system if the CPU-load is increased further (approx. 1 task every 4 seconds). The OSEK/VDX variable E_OS_LIMIT is increased every time a task is activated and the maximum number of active instances of this task is already reached. The new activation is dropped.

The EDF scheduling could be increased to a CPU load of 99.9% without loosing tasks or raising a system safety event. The maximum size of the EDF task list at this load is 30, while the maximum list size at 82.3% is 16. The maximum list size describes the maximum number of tasks that are active or delayed. The system was observed at this CPU-load for over 15 minutes.

During the high load of 99.9%, some deadlines are missed by a small amount, but no deadline is exceeded by more the ten percent and the system safety monitoring is not showing any critical state.

4 Conclusion

The paper presents a new method to improve the timing behavior of automotive applications. By adding a new plug-in controller to OSEK/VDX based systems it is shown by a case study that it is possible to implement an EDF scheduling policy on top of a legacy operating system to improve the processor utilization of the final system. This leads to easier manageable systems and a better performance of the application, such as the car engine control system presented in this paper.

References

[1] G. Buttazzo. Rate monotonic vs. EDF: Judgment day. *Real-Time Systems*, 29(1), 2005.

[2] G. Buttazzo and P. Gai. Efficient implementation of an EDF scheduler for small embedded systems. In *2nd Workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2006.

[3] T. Komarek, M. Dörfel, and R. Münzenberger. Developing real-time constrained embedded software using task models. *proceedings of the Advanced Automotiv Electronics (AAE 2007)*, 2007.

[4] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[5] OSEK-Group. *OSEK/VDX Operation System Specification*, 2005.

[6] M. Park, J. Hong, and S. Y. Shin. A priority assignment method for earliest deadline scheduling. In *ISCA 18th International Conference - Computers and their Applications*, 2003.

[7] P. Pedreiras and L. Almeida. Edf message scheduling on controller area network. *Computing & Control Engineering Journal*, 13(4):163–170, 2002.