Proc. of the 2nd Int. Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2006), Dresden, Germany, July 2006.

Efficient EDF Implementation for Small Embedded Systems

Giorgio Buttazzo
Scuola Superiore Sant'Anna
Pavia, Italy
buttazzo@sssup.it

Paolo Gai
Evidence S.r.l.
Pisa, Italy
pj@evidence.eu.com

Abstract

Modern embedded systems are often required to execute under stringent real-time constraints to satisfy high performance requirements. When the available computational resources are scarce, an efficient implementation of the scheduler is of crucial importance for containing system costs. In this paper, we present an efficient implementation of the Earliest Deadline First (EDF) scheduler, which is typically considered difficult and expensive to implement with respect to fixed priority algorithms, such as the Rate Monotonic (RM) scheduler. The higher resource utilization and the greater flexibility in handling aperiodic requests and overload conditions make EDF highly desirable for real-time embedded systems. We show that by using suitable kernel mechanisms for time representation, EDF can be effectively used even in small microprocessors for achieving precise time representation and increasing system utilization. Another important issue for adopting new schedulers in the market is their compliance with existing standards. For this reason, in this paper we discuss how to achieve a compliance with the OSEK/VDX API, widely used on automotive embedded systems, and we show how commercial kernels can integrate an EDF scheduler while maintaining an OSEK compliant API. Finally, we provide some performance measurements to show that implementing and EDF scheduler on a commercial OSEK-like kernel, such as ERIKA Enterprise, only introduces negligible overhead.

1 Introduction

Many modern embedded systems have to satisfy high performance requirements and have to execute in dynamic environments where the characteristics of the computational load cannot be precisely predicted at design time. In this context, to cope with the uncertainty, the operating system must provide efficient support for real-time scheduling and must be adaptive to modify the internal policies as a function of the current load conditions. When system resources are scarce, as usual in small embedded devices, an efficient implementation of the scheduler is of crucial importance for achieving the required performance while containing system costs.

Scheduling methods are typically distinguished into off-line and on-line algorithms. Off-line algorithms build the scheduling sequence before task execution, assuming that all tasks have an expected behavior that can be predicted by analyzing the task code. The scheduling table is built to satisfy a set of constraints specified on the task set and is used on line to activate the tasks according to the precomputed sequence. The strength of this approach is that complex constraints (such as deadlines, jitter, precedence relations, resource conflicts, non preemption, and fault tolerance) can be taken into account to produce a feasible schedule. Another advantage of this approach is that, once the schedule is computed and stored in a table, the runtime overhead is negligible, because all scheduling decisions are taken off line, and only a dispatcher is needed to read the next task from the table and put in execution. On the other hand, there are two main disadvantages in the off-line approach: one is that storing the entire scheduling table may require a lot of memory, depending on the task parameters, which may be prohibitive in embedded microprocessor with small memory requirements. Another disadvantage of this method is that it works only if all the tasks are known in advance and behave as predicted.

Unfortunately, in many practical situations, tasks are dynamically activated upon the occurrence of specific events and their execution is quite difficult to predict in advance, due to a complex dependency on input data. As a consequence, when using an off-line approach, an a-priory guarantee can only be achieved by making very pessimistic assumptions on the environment, which cause a waste of system resources. This problem is even more significant for small embedded systems, where the resource are scarce and need to be carefully allocated.

To overcome these limitations, mainly due to rigid a-priori assumptions, on-line algorithms build the schedule as tasks enter the systems. When tasks consist of an indefinite sequence of jobs (which may be activated periodically or aperiodically), on-line algorithms are in turn distinguished in two classes: fixed priority and dynamic priority algorithms. In fixed priority algorithms, all jobs of a task are assigned the same priority, so the relative priority between any two tasks does not change. On the contrary, in dynamic priority algorithms, jobs belonging to the same task may have different priority, so the relative priority of two tasks is not fixed, but can change with time.

A typical example of a fixed priority assignment is given by the Rate Monotonic (RM) algorithm [12], according to which a periodic task is assigned a fixed priority proportional to its rate: the higher the activation rate, the higher the priority. Another example of fixed priority assignment is given by the Deadline Monotonic (DM) algorithm [11], according to which the priority of a task is inversely proportional to its relative deadline: the shorter the relative deadline, the higher the priority. Since the relative deadline is fixed for all the jobs of a task, DM is a fixed priority algorithm. The Earliest Deadline First (EDF) algorithm [12] is an example of dynamic priority policy, because a job is assigned a priority that is inversely proportional to its absolute deadline: the shorter the absolute deadline, the higher the priority. Since the absolute deadline changes from job to job, the relative priority between tasks is not fixed, but depends on the absolute deadlines of the current active jobs.

Fixed priority algorithms are simpler to implement on top of commercial kernels, but dynamic algorithms are superior in many aspects, such as processor utilization, schedulability analysis, flexibility in overload management, and efficiency in aperiodic task handling. A detailed comparison between Rate Monotonic and EDF has been discussed by Buttazzo [4] under several perspectives. In spite of their advantages, however, dynamic scheduling methods are not widely used in embedded real-time systems, mainly because fixed priority algorithms are much simpler to implement, especially on top of commercial kernels.

In this paper we show that, by using suitable kernel mechanisms for time representation and scheduling, EDF can be effectively used, even in small microprocessors, for increasing system utilization and achieving a timely execution of periodic and aperiodic tasks.

Moreover, we show how EDF can easily be integrated in commercial kernels to achieve a compliance with the OSEK/VDX standard [15]. This is an important issue to consider when supporting legacy applications. In fact, many industries are reluctant to use novel scheduling techniques only because of the potential change they could introduce in the software design and implementation. Hence, a wide acceptance of a new scheduling paradigm can only be obtained if the enhancements can be provided without changing the current industrial practice. To address this issue, we show how EDF can be integrated with the existing OSEK/VDX standard without modifying application source code. Specific performance results are presented for the ERIKA Enterprise kernel [8], showing that the extra overhead introduced by our EDF implementation is negligible, compared with fixed priority algorithms.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 presents the problem of time representation in deadline-based schedulers and introduces an efficient method suitable for small real-time kernels. Section 4 shows how to implement resource reservation and budget management mechanisms. Section 5 shows the modifications done on an OSEK-like kernel to support EDF scheduling. Section 6 shows some performance results, and Section 7 states our conclusions and future work.

2 Related Work

A lot of work has been done in the context of small scale embedded real-time systems. For example, the MCX11 kernel [14] is a kernel totally written in assembly code for the Motorola 68HC11 microcontroller. It can manage up to 126 tasks and has a memory footprint of 3 Kbytes. Task scheduling is performed based on a fixed priority scheme and task communication may occur through shared memory buffers or message queues. The kernel does not provide a notion of global time, but only allows posting events with a time granularity of 50 milliseconds. TinyOS [20] is another kernel for embedded devices typically used for sensor network devices. It is mainly written in NesC (a dialect of the C language), and it implements a set of features useful for small-scale embedded devices. It lacks basic real-time support, as the schedule is a non-preemptive FIFO scheduler. SSX5 is a commercial real-time system produced by ETAS Gmbh [19] for M68HC12 and other microcontrollers. This kernel provides fixed priority preemptive scheduling and it is based on the single-shot execution model, in which tasks run to completion when activated. They are preempted as appropriate, but always complete before returning control to the lower priority task they have preempted. Single shot execution allows the use of a single stack, which leads to significant reduction in RAM requirements. MaRTE OS [17] is another real-time kernel for embedded applications, which allows greater flexibility. Most of its code is written in Ada with some C and assembler parts and it follows the Minimal Real-Time POSIX.13 subset. MaRTE implements an application defined scheduler that can support EDF scheduling on top of it.

MICOS [13] is a microkernel developed for the Motorola 68HC11 micocontroller, which is also available for Intel X86 architectures. To facilitate portability on different platforms, the kernel consists of two main layers: the kernel layer and the hardware layer. MICOS is one of the few kernels which implements dynamic EDF-based scheduling for better resource utilization. It handles hard real-time tasks, soft real-time tasks, and non critical activities. The time management method adopted in the kernel is the one described in this paper. It was used to implement a resource reservation mechanism for providing temporal isolation among tasks [1, 2], as well as good aperiodic responsiveness. The reservation mechanism allows the user to reserve a fraction of processor to each soft task to guarantee a minimum level of performance. The efficiency of the time management also allows monitoring task execution times for detecting execution overruns and collecting statistical information on timing parameters. The kernel with all real-time features requires 8 Kbytes of memory. ERIKA Enterprise [8] is a modular kernel for real-time systems produced by Evidence Srl. It implements the OSEK/VDX API, and it also proposes extensions to handle embedded multiprocessor-on-a-chip, and dynamic priority scheduling, like EDF and soft reservations. For architectures with very small RAM requirements, ERIKA Enterprise provides a mono-stack task management policy, where all tasks can share the same stack. The kernel also supports multi-stack models to provide blocking synchronization and communication primitives.

On the side of the operating system standards, we would like to cite the OSEK/VDX standard, which was developed by major automotive industries for the specification of the kernel API of embedded control units (ECU). The OSEK standard [15] is specifically designed for small embedded systems, having in mind the need to save as much RAM/ROM space as possible. For these reason, OSEK-like systems only support static configuration of kernel objects (such as tasks, resources, alarms, etc.) through an additional language named OIL. The OSEK/VDX standard supports preemptive and non-preemptive fixed priority scheduling using Immediate Priority Ceiling. In Section 5 we will describe how an efficient implementation of the EDF scheduler can be integrated in the OSEK/VDX implementation of ERIKA Enterprise to provide innovative algorithms without changing the current industrial practice.

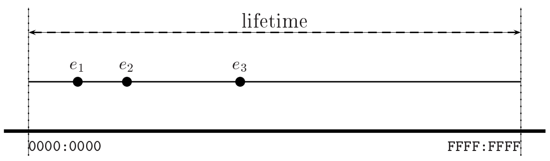

Figure 1. Linear time model with 32 bits.

3 Time representation

One of the problems that need to be solved when implementing an EDF scheduler is to provide a time representation for expressing absolute deadlines. Notice that fixed priority schedulers do not have this problem, since periods and relative deadlines can be mapped into a set of priority levels and do not have to be explicitly represented. In a deadline-based operating system, time can be represented in two ways: as a single integer or as a pair of integers. In the first case, an integer variable is used to track the elapsed time since reset; the value is in fact set to zero at system initialization and periodically incremented at every timer interrupt. The interval between two consecutive timer interrupts is called the system *tick* and defines the time resolution. In the second case, typical of POSIX systems, time is represented by a more complex structure (`struct timespec`), that is used to represent seconds and nanoseconds. Operations on timespec are typically avoided in small microcontrollers because of their complexity. For this reason, only the first option will be considered in the following paragraphs.

The system's *lifetime* is the maximum time the system can operate without causing a real-time clock overrun. It depends on the number of bits used for time representation and on the time resolution. For example, with a 10 ms resolution, a linear time clock represented on 16 bits has a lifetime of about 11 minutes, which is not suitable for most real-time applications.

Typical operating systems for medium size machines use a linear time model, where time is represented using a 32 bit variable with 1 millisecond resolution. In this case, the system lifetime has a value of about a few months. An example of linear time model is illustrated in Figure 1.

The main advantage of such a solution is that an event e_i precedes another event e_j if and only if $e_i < e_j$. The disadvantage of a linear time model, however, is that it imposes a finite lifetime. Increasing the lifetime requires either using a larger number of bits or setting a lower time resolution. Unfortunately, both solutions can be inappropriate for an embedded system with stringent memory requirements and real-time constraints. In fact, handling 32-bit time variables on a 8-bit microcontroller would cause a large overhead, since each 32-bit integer operation must be split into several 8-bit instructions, also considering the reduced number of registers available (3 for the M68HC11). A large lifetime with a small number of bits can be achieved by decreasing the time resolution. For example, a lifetime of 1 month with 16-bit time representation can be achieved using a resolution of 1.6 seconds, which however is unsuitable for most practical cases.

In general, a time management mechanism should have the following characteristics:

1. Time resolution should be as high as possible in order for the kernel to provide high responsiveness to asynchronous events. High timer resolution also allows to increase processor utilization in the case task periods or deadlines are not multiple of the system tick. In this situation, in fact, to avoid missing deadlines, a period which is not multiple of the tick, should be reduced to the closest multiple. Hence, the higher the resolution, the smaller the utilization increase due to the period reduction.
2. The maximum time interval P handled by the system should be as long as possible in order to manage tasks with large periods or long relative deadlines (with respect to the system tick).
3. In embedded systems with stringent memory requirements, the system time should be represented using the minimum number of bits. Such a requirement is in contrast with the previous ones and imposes a trade off in the kernel.
4. The time management mechanism should not introduce a large runtime overhead.

In general, possible compromises depend on several factors, including the processor speed, the available memory, the efficiency of the kernel, the time horizon required by the application, the maximum extension of the relative timing constraints, the task criticality, and the number of tasks in the system.

A reasonable compromise among the four requirements stated above is to use a circular time model. It differs from the linear one in that it handles the overflow condition occurring when the n-bit variable used to represent the system time passes from $2^n - 1$ to 0.

Figure 2 shows a circular time model, implemented using a 16-bit variable. In this model, each cycle has a length P = 10000H (hexadecimal), hence two events with a time difference greater than or equal to P cannot be handled by the system without additional information. For example, Figure 2 shows two cycles in which four events are represented. Since events e_1 and e_2 have the same value ($e_1 = e_2$), they are considered simultaneous by

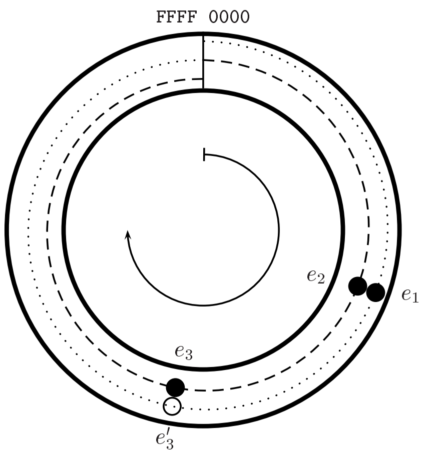

Figure 2. Circular time model with 16 bits.

the system, although they occur in two different cycles. Similarly, the time distance between e_1 and e_3 ($e_3 - e_1$) is considered to be the same as the one between e_1 and e'_3 ($e'_3 - e_1 = e_3 - e_1$).

In summary, when working with 8-bit or 16-bit microcontrollers, a long lifetime can be achieved either using a 32-bit linear timer (with a large overhead) or with a 16-bit circular timer, by handling the overflow of the system clock variable (in this case the lifetime becomes infinite). It is worth observing that, when using a circular timer, the overflow has to be managed at every time comparison, and hence it must be efficiently handled.

In this work, we implemented the Implicit Circular Timer's Overflow Handler (ICTOH), first proposed by Carlini and Buttazzo [5], which is able to handle such a timer overflow with a very small overhead compared with other existing techniques [16, 9, 7].

3.1 The ICTOH algorithm

This section describes the Implicit Circular Timer's Overflow Handler (ICTOH) [5], which allows an efficient representation of absolute deadlines in a circular time model. We first introduce the following definitions.

[Def.] An event and its temporal reference on the circular timer is denoted as e_i. Thus, for example, we can say that e_i is a task activation and that $e_i = 04F3H$.

[Def.] The set of temporal references stored in the system at time t is denoted as $E(t)$.

[Def.] The absolute time at which an event $e_i \in E(t)$ occurs is denoted as $t(e_i)$.

[Def.] The circular timer period is denoted as P. In other words, P is the minimum interval of time between two non simultaneous events characterized by the same representation in the system.

[Def.] We say that two events $e_i,\ e_j \in E(t)$ belong to the same cycle if the interval $[t(e_i),\ t(e_j)]$ represented on the circular timer does not include the values FFFFH and 0000H. For example, in Figure 2 e_1 and e_2 do not belong to the same cycle, whereas e_2 and e_3 do belong.

Then the ICTOH method can be defined as follows.

ICTOH: If events are represented by n-bit unsigned integers, such that

$$\forall t\ \forall e_i, e_j \in E(t)\ |t(e_i) - t(e_j)| < \frac{P}{2} \qquad (1)$$

then $\forall t\ \forall e_i, e_j \in E(t)$ we have:

1. $t(e_i) > t(e_j) \iff (e_i \ominus e_j) < \frac{P}{2}\ ,\ (e_i \ominus e_j) \neq 0$
2. $t(e_i) < t(e_j) \iff (e_i \ominus e_j) > \frac{P}{2}$
3. $t(e_i) = t(e_j) \iff (e_i \ominus e_j) = 0$

where $\ominus$ denotes a subtraction (modulo 2^n) between n-bit integers, evaluated as an unsigned n-bit integer.

It is worth observing that for 8/16/32-bit integers such a subtraction operation does not require a special support since it is implemented in all CPUs. Figure 3 shows a set of events which satisfies condition (1).

Notice that condition (1) represents the price we have to pay for implementing a high resolution timer with an infinite lifetime. It means that the system can only handle tasks with timing constraints that cannot exceed the value of $P/2$ ticks.

So, for example, if the tick is set to 1 ms, and we use a 16 bit variable for storing the system time, then $P = 2^{16} = 65536$, meaning that the longest timing variable cannot be greater than 32.768 seconds. If the application includes a task with a greater period, we can increase the system tick, until a value equal to the least timing value in the system.

So, actually, $P/2$ represents the maximun ratio between the longest and the smallest timing parameter. If there is a task set where such a ratio is greater than $P/2$, then the ICTOH method cannot be used, and we are forced to pay a greater overhead for handling time variables with a higher number of bits.

The main property of the $\ominus$ operator is that

$$\forall a, b \in [0, 2^n - 1]\ \ unsigned(b \ominus a) = dist(a, b)$$

where

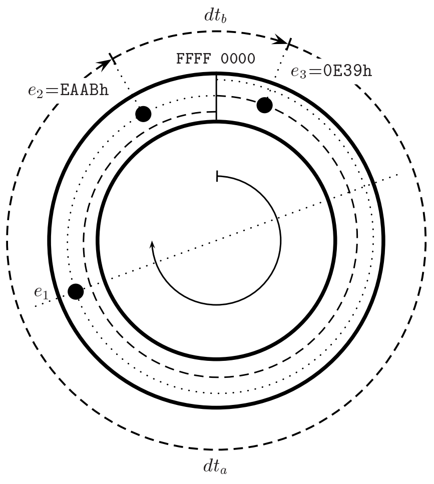

Figure 3. Example of events evaluated by ICTOH.

- $dist(x, y)$ is the distance from x to y evaluated on the time circle in the direction of increasing time values. Notice that $dist(x, y) = d$ means that if $t = x$ then, after a delay of d, we have $t = y$, independently of the fact that x and y belong to two different cycles.
- $unsigned(x)$ is the value of x, interpreted as an n-bit unsigned value. We recall that according to the 2's complement representation,

$$unsigned(x) = \begin{cases} x & \text{if } x \geq 0 \\ 2^n + x & \text{otherwise.} \end{cases}$$

For example, when evaluating events e_2 and e_3 in Figure 3, we have that $dt_a = (e_2 - e_3) = DC72H > 8000H = P/2$. Hence, we conclude that e_2 must precede e_3 and that the actual time difference between the events is $dt_b = (e_3 - e_2) = 238EH < P/2$.

3.2 Extension

The constraint expressed by equation (1) can be relaxed if we consider disjoined sets of events.

[Def.] Two sets of events $E_i(t)$ and $E_j(t)$ are said to be disjoined if every element of the first set is never compared with an event of the second set.

[Def.] Let F be the set of all the disjoined sets in the system.

Then, we can formulate constraint (1) as follows:

$$\forall t\ \forall E_k(t) \in F(t)\ \forall e_i, e_j \in E_k(t)$$
$$|t(e_i) - t(e_j)| < \frac{P}{2}. \qquad (2)$$

In this way it is possible to manage temporal events which are spread in intervals greater than

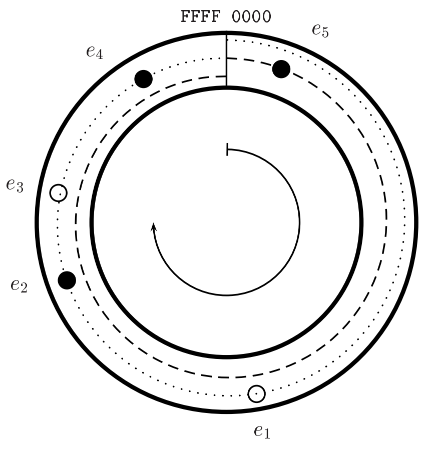

Figure 4. Two groups of disjoint events $\{e_1, e_3; e_2, e_4, e_5\}$ that can be handled by the ICTOH method.

$P/2$, provided that events belonging to the same group are not separated by a time difference greater than $P/2$. Figure 4 illustrates two groups of events that satisfy constraint (2).

Task activation times and deadlines represent a typical example of two disjoint groups of events. In fact, although an absolute deadline is computed from task's activation time (by summing the corresponding relative deadline), the deadline event enters the system only after task activation, hence there is no need to compare the two events.

3.3 Implementation notes

Given a pair of events e_i end e_j represented through variables with 8, 16, or 32 bits, then by computing the difference $(e_i \ominus e_j)$ as a signed integer we can say that:

$$t(e_i) > t(e_j) \iff (e_i - e_j) > 0$$
$$t(e_i) < t(e_j) \iff (e_i - e_j) < 0$$
$$t(e_i) = t(e_j) \iff (e_i - e_j) = 0$$

It is worth noting that such a result is valid only for variables represented on 8/16/32 bits, since only in this case all unsigned numbers with a value greater than or equal to $P/2$ (evaluated using the two's complement representation) are considered to be negative. For variables with a different number of bits, the test has to be performed as shown in the previous section, which in several CPUs does not cause a larger overhead in terms of time and memory.

4 Implementing resource reservations

The ICTOH method can in general be used to efficiently implement various EDF-based algorithms. For example, on the ERIKA Enterprise kernel [8], we easily realized a soft resource reservation server, which is basically an EDF scheduler with an additional run-time monitoring mechanism for checking the execution budget of each task. Whenever a task finishes its budget, its deadline is postponed, thus decreasing its priority (see [2] for more details).

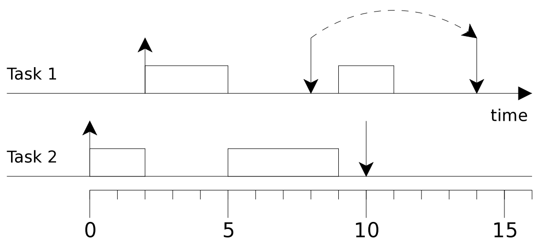

Figure 5. τ_1 initially preempts τ_2, but due to a deadline postponement it is later on preempted by τ_2 at time 5.

The main problem when implementing an efficient resource reservation mechanism is that deadline postponement cannot be done forever, because of the circular timer implementation. To address this issue in ERIKA Enterprise, a number of design choices have been taken. They are described below.

4.1 Stack sharing

Stack sharing implies that different tasks can somehow share the same stack. To achieve a correct result, the system has to prevent interleaved executions among tasks. In fact, suppose that two tasks, τ_1 and τ_2, share the same stack and suppose that τ_1 preempts τ_2; in this case, τ_2 must not be scheduled before the end of τ_1, otherwise it can overwrite τ_1's stack data. This situation typically occurs when a task blocks on a locked semaphore, held by the task that has been preempted.

Unfortunately, a similar problem can also occur when adopting a deadline postponement mechanism to enforce resource reservations, as done by the Constant Bandwidth Server (CBS) [2]. In fact, if τ_1 preempts τ_2 and consumes all its budget, its deadline is postponed, giving the possibility of being preempted by τ_2. This situation is illustrated in Figure 5. To prevent such a problem, stack sharing must be avoided when using resource reservation mechanisms, or at least restricted to those tasks having predictable execution times.

4.2 Timer handling

To implement the resource reservation mechanism, the kernel has to export a timing service that supports at least a fast timer read and a timer interrupt. A *fast* timer read is needed because the timer is frequently used to perform time accounting and check for deadlines. The timer interrupt

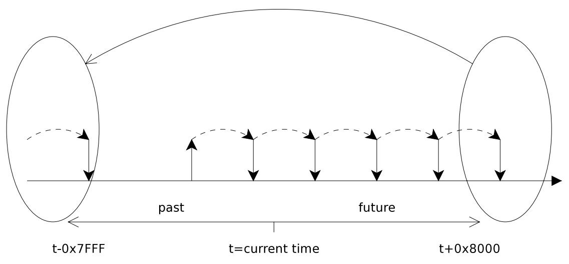

Figure 6. Deadline postponement causing a wrapup of the time reference.

is also needed to implement the preemption mechanism upon budget exhaustion. Fortunately, these features are common in most microcontrollers, which usually have at least one timer that can be read as a CPU register and programmed in a few clock cycles.

In addition to that, every primitive that may cause a preemption (e.g., a primitive that activates a task with priority higher than the running task) has to be modified to automatically reprogram the timer upon a system reschedule. The timer must be set to raise an interrupt in a future instant equal to the current time plus the budget of the task just being scheduled. At that time, two things may happen:

- The timer is reprogrammed again before the interrupt is raised, which means that another rescheduling operation has taken place before the exhaustion of the budget of the running task. In this case, the running task will be accounted for the time it has executed.
- The timer interrupt is raised by the hardware timer, which means that the budget of the task has finished before the end of the current instance of the running task. In this case, the timer interrupt handler performs the following operations:
  - the deadline of the running task is postponed;
  - the budget of the running task is recharged;
  - the system is rescheduled to check whether the running task has to be preempted because of the deadline postponement;
  - the hardware timer is reprogrammed again considering the new budget.

  De facto, the timer interrupt is the responsible for enforcing temporal isolation.

4.3 Timer wrapup

A high number of deadline postponements due to continuous budget exhaustion may cause a wrapup of the time reference, as shown in Figure 6, where a deadline may pass from the far future to the far past.

This can occur when the allocated budget is too small, or when a task enters an unpredicted long branch or a loop. To prevent this to happen, a check is performed when the deadline is going to be postponed. If a timer wrapup is caught, the deadline is set to the longest time instant in the future, which is equivalent of executing the task in background. Notice that, from a theoretical point of view, the timer wrapup phenomenon prevents implementing a *perfect* resource reservation. However, in practice, such a reservation error does not create significant problems.

4.4 Task periodicity

In ERIKA Enterprise, and in many OSEK kernels, there is no concept of task periodicity, which is handled independently of the task objects using alarm objects. An alarm object is simply a notification mechanism based on external events that can be programmed to periodically activate a task. The kernel does not store any information about the periodicity of a task and, in this sense, the action of activating a task is separated from its activation pattern. When adding resource reservations to an OSEK-like kernel, we must consider that task budgets have to be handled independently of task activation patterns. The kernel has no knowledge of the status of a task with respect to its activation pattern. Hence, the kernel has to monitor tasks that have not been activated for a while, to avoid their deadline to pass from the far past directly into the far future.

In our implementation, a periodic check is performed every fourth of the system lifetime. The check simply sets a *disable* flag for every task having its deadline in the past. Then, the disable flag must be checked every time a task has to be activated:

- if the task is not disabled, deadline calculation is done as usual;
- if the task is disabled, its deadline is set to the current time plus its replenishment period, regardless of the value of its absolute deadline field.

4.5 Locking and temporal isolation

A final check is needed to handle the case in which a budget is exhausted when a task is inside a critical section. In fact, if the task locking a semaphore is preempted due to budget exhaustion, then the main properties of SRP and Immediate Priority Ceiling protocols do not hold anymore (see [3] for details). As a consequence, a task could even block after passing the preemption test!

This problem has been solved by temporarily disabling temporal isolation (i.e., the timer that fires upon budget exhaustion) inside critical sections. This is equivalent of giving the task some extra budget to complete the critical section, if needed.

As a result, the implementation of resource reservation in ERIKA Enterprise turned out to be fairly efficient (low runtime overhead) and robust against the possible traps deriving by other interacting kernel features. Performance figures of the implementation are included in Section 6.

5 Integrating EDF into OSEK

To enable a wide acceptance of a new scheduling method in the industry, besides technical reasons, it is important to show how the new algorithm can be integrated in the current state-of-the-art industrial practice. For this reason, in this section we show how EDF can be easily integrated in the OSEK/VDX standard, which proposes an API for small-scale operating systems for automotive embedded control units. The OSEK/VDX standard proposes an API that supports fixed priority scheduling featuring preemptive, non preemptive and preemption threshold options, together with shared resource handling using Immediate Priority Ceiling, and periodic activations using alarms. It also proposes a static approach for the system configuration, where all the objects are specified at design time using a configuration language called OIL. In practice, an OIL specification is composed by an *implementation definition*, usually provided by the RTOS maker, and by an *application definition*, which defines the objects (Tasks, Resources, ...) that are really present in the system, together with their parameters. Although the OIL language is specified by a standard, the particular parameters that can be specified heavily depends on the particular kernel, which makes OILs from different vendors almost incompatible [6]. Static configuration through OIL, together with one-shot task execution supported by OSEK Basic tasks allow stack sharing among different tasks in the system. This feature is important because it enables the implementation of OSEK-compliant systems that fit into a few kilobytes of ROM and RAM. The OSEK-VDX standard is a perfect candidate for supporting an alternative scheduling algorithm, such as EDF, because all the configuration parameters are static, hence they do not affect the actual parameters in the API (Task-related primitives only receive the task ID as a parameter). The only modification required to implement EDF on an OSEK system is the integration of the EDF parameters together with the standard OIL parameters. In the case of ERIKA Enterprise, we added a RELATIVEDEADLINE parameter in the TASK implementation section, allowing the specification of a relative deadline that is added to the current time to get the absolute deadline needed for EDF task scheduling. Then, resource usage and mutual exclusion are obtained using the SRPT algorithm [10], which combines EDF with the SRP [3] for handling shared resources, and with Preemption Thresholds [18, 21] for reducing preemptions and the overall stack space in real-time applications.

Version	Bytes
OSEK BCC1	2136
Simplified BCC1 (FP)	1716
EDF	2004
EDF without stack sharing	2772
EDF with Resource Reservation	4008

Table 3. ERIKA Enterprise ROM (code) footprint on ARM7TDMI.

6 Performance measurements

This section describes some performance measurements that have been taken on ERIKA Enterprise running on a 50 MHz ARM7TDMI, with cache disabled and external SDRAM. Table 3 gives an idea of the typical footprint of ERIKA Enterprise: The first row considers a full-fledged OSEK configuration, whereas the remaining rows show the footprint of typical Fixed Priority / EDF implementations (the OSEK BCC1 row includes boot, StartOS, ActivateTask, Schedule, TerminateTask, ChainTask, GetResource, and ReleaseResource; the others include the corresponding reduced versions). As it can be seen, Fixed Priority and EDF implementation on monostack configurations differ only by around 300 bytes, which is an affordable (if not negligible) price on the final footprint. Introducing multi-stack (that allows the application to use blocking primitives), the footprint increases by around 750 bytes. Additional support for resource reservations accounts for around 1.7 Kb, and it is mainly due to timer handling reprogramming needed in the implementation of the resource reservation mechanism.

Table 1 shows the impact of the implementation of EDF on the run-time performance of a kernel. The queue implementation chosen for the tests is a $O(n)$ queue both for Fixed Priority and EDF. ERIKA Enterprise also supports $O(1)$ queuing for Fixed Priority, which is not showed in the Table because it seems that most commercial OSEK kernels are currently using linear queuing [6]. As it can be seen, the EDF implementation is only slightly more complex than a linear queuing algorithm on Fixed Priority.

In particular, Tables 2, 4 and 5 show the C source code and the corresponding gcc generated assembly code for a possible implementation of the preemption test under fixed priorities and EDF, using wraparound timers. As it can be seen, the EDF implementation only adds 9 assembler instructions to the test, which is a reasonable overhead considering

Version	FP w. ss	FP no ss	EDF w. ss	EDF no ss	Res. Res
Mutex Lock	1.84	1.84	1.84	1.84	2.5
Mutex Unlock	5.0	9.7	5.52	10.2	22.3
ActivateTask	8.4	13.4	9.5	14.5	20.3
Task End	9.0	14.5	9.0	14.5	16.1

Table 1. ERIKA Enterprise timings, 50 MHz-ARM7TDMI (ss stands for stack sharing). All the numbers are in μsec.

```
// C implementation using fixed priority
if (ready != NIL &&
    system_ceiling < ready_prio[ready])

// C implementation using EDF with wraparound timers
if (running == NIL ||
    (ready != NIL &&
     (signed)(absdline[running] - absdline[ready]) > 0 &&
     system_ceiling < EE_th_ready_prio[ready]
    )
   )
```

Table 2. C source code needed to implement a preemption test.

the length of a typical RTOS primitive.

A major hit on the timing performance is done by the resource reservation mechanism; that time reported in Table 1 basically accounts for the reprogramming of the timer that needs to be done every time the system is rescheduled. Notice, however, that the overhead for timing reprogramming has to be taken into account also when implementing resource reservations under fixed priorities.

It is worth noting that, in general, the choice of the best implementation method for enforcing temporal protection is a tradeoff that depends on the frequency of the system calls invoked by the application, the precision required for task suspension, and the overhead of reprogramming a timer. If a high precision is not mandatory, or the overhead for reprogramming the hardware timer is high, then a periodic timer can be used, without the need of reprogramming the timer at every rescheduling. In our case, we chose the most efficient method for the microcontroller used. In particular, the overhead of Table 1 is mainly related to the computation of the new timer value (that has to be done in any case), whereas the hardware timer reprogramming simply consists of a write operation on a memory mapped register.

7 Conclusions

In this paper we showed that dynamic scheduling algorithms, such as EDF, can be efficiently implemented in small operating systems for embedded applications, enabling a more efficient use of the computational resources and a more flexible management of aperiodic events and overload conditions. We also illustrated how EDF can be easily integrated with existing operating systems standards, like OSEK-VDX, which is widely used in the automotive industry.

Moreover, we used the efficient EDF implementation to support the paradigm of soft resource reservations, and we showed the impact of our implementation on a real kernel, identifying the major features that affect the memory footprint and the runtime overhead. Finally, we evaluated the implementation and showed that the overhead introduced is affordable for most microcontroller applications.

As a future work, we plan to study the effectiveness of the EDF implementations on reconfigurable hardware, such as Altera Nios II, to see whether the techniques presented in this paper can be efficiently implemented in hardware, thus limiting the additional system overhead.

References

[1] L. Abeni and G. Buttazzo, “Integrating Multimedia Applications in Hard Real-Time Systems,” *Proc. of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] L. Abeni and G. Buttazzo, “Resource Reservation in Dynamic Real-Time Systems,” Real-Time Systems, Vol. 27, No. 2, July 2004.

[3] T.P. Baker, Stack-Based Scheduling of Real-Time Processes, The Journal of Real-Time Systems 3(1), pp. 76-100, (1991).

[4] Giorgio Buttazzo, ”Rate Monotonic vs. EDF: Judgment Day”, Real-Time Systems, Vol. 28, pp. 1-22, 2005.

```
// Assembly of the fixed priority implementation
// read ready
ldr r3, .L15+12
ldr r0, [r3, #0]
// ready != NIL
cmn r0, #1
beq .L5
// read system_ceiling and ready_prio[ready]
ldr lr, .L15+16
ldr r3, .L15+20
ldr r3, [r3, r0, asl #2]
// read system_ceiling value
ldr ip, [lr, #0]
// system_ceiling ¡ ready_prio[ready]
cmp r3, ip
bls .L5
```

Table 4. Compiler generated ARM7 assembler needed to implement a fixed priority preemption test.

```
// Assembly of the EDF implementation
// read ready
ldr r3, .L18+24
ldr ip, [r3, #0]
// read running
ldr r3, .L18+28
ldr r2, [r3, #0]
// running == NIL
cmn r2, #1
beq .L12
// ready != NIL
cmn ip, #1
beq .L7
// (signed)(absdline[running] - absdline[ready]) > 0
ldr r3, .L18+12
ldr r2, [r3, r2, asl #2]
ldr r3, [r3, ip, asl #2]
rsb r2, r3, r2
cmp r2, #0
ble .L7
//system_ceiling ¡ EE_th_ready_prio[ready]
ldr r3, .L18+32
ldr r2, [r3, ip, asl #2]
ldr r3, .L18+36
ldr r3, [r3, #0]
cmp r2, r3 bls .L7
```

Table 5. Compiler generated ARM7 assembler needed to implement an EDF preemption test using wraparound timers.

[5] A. Carlini and G. Buttazzo, ”An Efficient Time Representation for Real-Time Embedded Systems”, Proceedings of the ACM Symposium on Applied Computing (SAC 2003), Melbourne, Florida, USA, March 9-12, pp. 705-712, 2003.

[6] R. Dreier, K. D. Mller-Glaser, “Requirements for Real Time Operating Systems and Features of Operating Systems Implementing the OSEK/VDX Standard API” Proceeding MASCOTS 2004, Volendam, The Netherlands, October 2004.

[7] R. Elz and R. Bush, “Serial Number Arithmetic”, August 1996, Network Working Group, request for comments, 1982. URL: ftp://ftp.isi.edu/in-notes/rfc1982.txt.

[8] Evidence Srl, “ERIKA Enterprise RTOS” URL: http://www.evidence.eu.com.

[9] Pedro Fonseca, “Approximating linear time with finite count clocks”, Tech. Rep., Dep. de Electrónica, Universidade de Aveiro, Revista do DETUA vol.3,n.4,pp:359-361, ISSN-1645-0493, Sept.2001.

[10] P. Gai, G. Lipari, M. Di Natale, “Design Methodologies and Tools for Real-Time Embedded Systems”, Special Issue of Design Automation for Embedded Systems, 2002.

[11] J. Leung and J. W. Whitehead, ”On the Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks”, Performance Evaluation, 2(4), 1982.

[12] C.L. Liu and J.W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment,” *Journal of the ACM* 20(1), pp. 40–61, 1973.

[13] A. Carlini, “A real-time kernel for embedded applications on a Motorola 68HC11 microcontroller”, Technical Report, Robotics Lab, University of Pavia, TR-2001-01, December 2001.

[14] The MCX11 kernel, http://www.introl.com/ introl-demo/demo/MCX11/contents.html.

[15] OSEK/VDX standard, http://www.osek-vdx.org.

[16] Moonju Park, Lui Sha, and Yookun Cho, “A Pratical Approach to Earliest Deadline Scheduling”, Technical Report, School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea, December 2001.

[17] M. Aldea Rivas and M. Gonzlez Harbour, “POSIX-Compatible Application-Defined Scheduling in MaRTE OS,” Euromicro Conference on Real-Time Systems (WiP), Delft, The Netherlands, June 2001.

[18] M. Saksena and Y. Wang, “Scalable Real-Time System Design using Preemption Thresholds”, Proc. of the Real Time Systems Symposium, Dec. 2000

[19] The SSX5 kernel, URL: http://www.realogy.com/.

[20] The TinyOS kernel, URL: http://www.tinyos.net/.

[21] Y. Wang and M. Saksena, “Fixed Priority Scheduling with Preemption Threshold”, Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications, December 1999.