4

An Efficient Hierarchical Scheduling Framework for the Automotive Domain

Mike Holenderski, Reinder J. Bril and Johan J. Lukkien
Eindhoven University of Technology
The Netherlands

1. Introduction

Modern real-time systems have become exceedingly complex. A typical car is controlled by over 100 million lines of code executing on close to 100 Electronic Control Units (ECU). With more and more functions being implemented in software, the traditional approach of implementing each function (such as engine control, ABS, windows control) on a dedicated ECU is no longer viable, due to increased manufacturing costs, weight, power consumption, and decreased reliability and serviceability (Nolte et al., 2009). With the ECUs having increasingly more processing power, it has become feasible to integrate several functions on a single ECU. However, this introduces the challenge of supporting independent and concurrent development and analysis of individual functions which are later to be integrated on a shared platform. A popular approach in the industry and literature is component based engineering, where the complete system is divided into smaller software components which can be developed independently. The Automotive Open System Architecture (AUTOSAR) (AUTOSAR, 2011) standard is an example of such an approach in the automotive domain. It relies on a formal specification of component interfaces to verify the functional properties of their composition. Many functions in automotive systems, however, also have real-time constraints, meaning that their correct behavior is not only dependent on their functional correctness but also their temporal correctness. AUTOSAR does not provide temporal isolation between components. Verifying the temporal properties of an integrated system requires complete knowledge of all functions comprising the components mapped to the same ECU, and therefore violates the requirement for independent development and analysis.

In this chapter we address the problem of providing temporal isolation to components in an integrated system. Ideally, temporal isolation allows to develop and verify the components independently (and concurrently), and then to seamlessly integrate them into a system which is functioning correctly from both a functional and timing perspective (Nolte, 2011; Shin & Lee, 2008). The question is how to provide true temporal isolation when components execute on a shared processor. We address this problem by means of an hierarchical scheduling framework (HSF).

An HSF provides the means for the integration of independently developed and analyzed components into a predictable real-time system. A component is defined by a set of tasks, a local scheduler and a *server*, which defines the component's time budget (i.e. its share of the processing time) and its replenishment policy.

An HSF-enabled platform should provide the following general functionalities:

1. Interface for the creation of servers and assigning tasks to servers.
2. *Virtual timers*, which are relative to a components's budget consumption, as well as *global timers*, which are relative to a fixed point in time.
3. Local scheduling of tasks within a component, and global scheduling of components on the system level.

In this chapter we focus on providing temporal isolation and preventing interference between components. We aim at satisfying the following additional requirement:

4. Expiration of events local to a component, such as the arrival of periodic tasks, should not interfere with other components. In particular, the handling of the events local to inactive components should be deferred until the corresponding component is activated. The time required to handle them should be accounted to the corresponding component, rather than the currently active one.

These requirements should be met by a modular and extensible design, with low performance overhead and minimal modifications to the underlying RTOS. It should exhibit predictable overhead, while remaining efficient to support resource-constrained embedded systems in the automotive domain.

Real-time applications will often require support for periodic task arrival. Periodic tasks rely on timers to represent their arrival time. For servers, we also need timers representing the replenishment and depletion of a budget. Vital, and a starting point for our design, is therefore the support for simple timers (or timed events), i.e. the assumption that an event can be set to arrive at a certain time. This simple timer support is typically available in an off-the-shelf Real-Time Operating System (RTOS) (Labrosse, 2002). Some RTOSes provide much more functionality (for which our work then provides an efficient realization) but other systems provide just that. As a result, the emphasis lies with the management of timers. The timer management should support long event interarrival times and long lifetime of the system at a low overhead.

Contributions

We first present the design of a general timer management system, which is based on Relative Timed Event Queues (RELTEQ) (Holenderski et al., 2009), an efficient timer management system targeted at embedded systems. Pending timers are stored in a queue sorted on the expiration time, where the expiration time of each timer is stored relative to the previous timer in the queue. This representation makes it possible to reduce the memory requirements for storing the expiration times, making it ideal for resource constrained embedded systems. We have implemented RELTEQ within μC/OS-II, and showed that it also reduces the processor overhead compared to the existing timer implementation.

We then leverage RELTEQ to implement periodic tasks and design an efficient HSF. The proposed HSF extension of RELTEQ supports various servers (including the polling, idling-periodic, deferrable and constant-bandwidth servers), and provides access to both virtual and global timers. It supports independent development of components by separating the global and local scheduling, and allowing each server to define a dedicated scheduler. The HSF design provides a mechanism for tasks to monitor their server's remaining budget,

and addresses the system overheads inherent to an HSF implementation. It provides temporal isolation and limits the interference of inactive servers on the system level. Moreover, it avoids recalculating the expiration of virtual events upon every server switch and thus reduces the worst-case scheduler overhead.

The proposed design is evaluated based on an implementation within μC/OS-II, a commercial operating system used in the automotive domain. The results demonstrate low overheads of the design and minimal interference between the components.

In this chapter we focus on the means for implementing a HSF. The corresponding analysis falls outside of the scope.

Outline

Section 2 discusses related work, followed by the system model description in Section 3. Section 4 introduces RELTEQ, describing its provided interface, underlying data structures, and algorithms for fast insertion and deletion of timed events. Subsequently, the RELTEQ interface is used to implement periodic tasks in Section 5 and fixed-priority servers in Section 6. The servers form an integral part of the HSF presented in Section 7. In Section 8 the HSF design is evaluated based on an implementation on top of a commercial operating system. Section 9 concludes this chapter.

2. Related work

In this section we discuss the work related to timer management, HSFs in general, and HSFs in automotive systems.

2.1 Timer management

The two most common ways to represent the timestamps of pending timers are: *absolute* timestamps are relative to a fixed point in time (e.g. January 1st, 1900), while *relative* timestamps are relative to a variable point in time (e.g. the last tick of a periodic timer).

In (Oikawa & Rajkumar, 1999; Palopoli et al., 2009) each timer consists of a 64-bit absolute timestamp and a 32-bit overflow counter. The timers are stored in a sorted linked list. A timer Interrupt Service Routine (ISR) checks for any expiring timers, and performs the actual enforcement, replenishment, and priority adjustments. In (Oikawa & Rajkumar, 1999) the timer ISR is driven by a one-shot high resolution timer which is programmed directly. Palopoli et al. (2009) use the Linux timer interface, and therefore their temporal granularity and latency depend on the underlying Linux kernel.

The Eswaran et al. (2005) implementation is based on the POSIX time structure timeval, with two 32-bit numbers to represent seconds/nanoseconds. The authors assume the absolute timestamp value is large enough such that it will practically not overflow.

Carlini & Buttazzo (2003) present the Implicit Circular Timers Overflow Handler (ICTOH), which is an efficient time representation of absolute deadlines in a circular time model. It assumes a periodic timer and absolute time representation. It's main contribution is handling the overflow of the time due to a fixed-size bit representation of time. It requires managing the overflow at every time comparison and is limited to timing constraints which do not exceed 2^{n-1}, where n is the number of bits of the time representation. Buttazzo & Gai (2006) present

an implementation of an EDF scheduler based on ICTOH for the ERIKA Enterprise kernel (Evidence, n.d.) and focus on minimizing the tick handler overhead.

The μC/OS-II (Labrosse, 2002) real-time operating system stores timestamps relative to the current time. The timers are stored in an unordered queue. It assumes a periodic timer, and at every tick it decrements the timestamp of all pending timers. A timer expires when its timestamp reaches 0. Timestamps are represented as 16-bit integers. The lifetime of their queue is therefore 2^{16} ticks.

In (Holenderski et al., 2009) we introduced Relative Timed Event Queues (RELTEQ), which is a timed event management component targeted at embedded operating systems. It supports long event interarrival time (compared to the size of the bit representation for a single timestamp), long lifetime of the event queue, and low memory and processor overheads. By using extra "dummy" events it avoids the need to handle overflows at every comparison due to a fixed bit-length time representation, and allows to vary the size of the time representation to trade the processor overhead for handling dummy events for the memory overhead due to time representation. Similar to (Engler et al., 1995; Kim et al., 2000), our RELTEQ implementation is tick based, driven by a periodic hardware timer.

2.2 Hierarchical scheduling frameworks

HSFs are closely related to resource reservations. Mercer et al. (1994) introduce the notion of processor reservations, aiming at providing temporal isolation for individual components comprising a real-time system. Rajkumar et al. (1998) identify four mechanisms which are required to implement such reservations: *admission control*, *scheduling*, *monitoring* and *enforcement*. Run-time monitoring of the consumed resources is intrinsic to realizing correct implementation of the scheduling and enforcement rules. Monitoring of real-time systems can be classified as synchronous or asynchronous (Chodrow et al., 1991). In the synchronous case, a constraint (e.g worst-case execution time) is examined by the task itself. In the asynchronous case, a constraint is monitored by a separate task. The approaches in (Chodrow et al., 1991) are based on program annotations and, hence, are synchronous. In reservation-based systems, however, monitoring should be asynchronous to guarantee enforcement without relying on cooperation from tasks. Moreover, monitoring should not interfere with task execution, but should be part of the operating system or middleware that hosts the real-time application. Our HSF takes the asynchronous monitoring approach.

Shin & Lee (2003) introduce the periodic resource model, allowing the integration of independently analyzed components in compositional hard real-time systems. Their resource is specified by a pair (Π_i, Θ_i), where Π_i is its replenishment period and Θ_i is its capacity. They also describe the schedulability analysis for a HSF based on the periodic resource model under the Earliest Deadline First and Rate Monotonic scheduling algorithms. While the periodic-idling server Davis & Burns (2005) conforms to the periodic resource model, the deferrable (Strosnider et al., 1995) and polling (Lehoczky et al., 1987) servers do not. The HSF presented in this chapter supports various two-level hierarchical processor scheduling mechanisms, including the polling, periodic idling, deferrable servers, and constant-bandwidth (Abeni & Buttazzo, 1998) servers. We have reported on the benefits of our constant-bandwidth server implementation in (van den Heuvel et al., 2011). In this chapter we focus on the underlying timer management and illustrate it with fixed-priority servers.

2.2.1 HSF implementations

Saewong et al. (2002) present the implementation and analysis of an HSF based on deferrable and sporadic servers using an hierarchical rate-monotonic and deadline-monotonic scheduler, as used in systems such as the Resource Kernel (Rajkumar et al., 1998).

Inam et al. (2011) present a FreeRTOS implementation of an HSF, which is based on our earlier work in (Holenderski et al., 2010). It supports temporal isolation for fixed-priority global and local scheduling of independent tasks, including the support for the idling-periodic and deferrable servers. Their goal is to minimize the changes to the underlying OS. Consequently they rely on absolute timers provided by FreeRTOS. They do not address virtual timers. The HSF presented in this chapter relies on relative times, which allow for an efficient implementation of virtual timers. Also, our HSF implementation is modular and supports both fixed-priority as well as EDF scheduling on both global and local levels, as well as constant-bandwidth servers.

Kim et al. (2000) propose a two-level HSF called the SPIRIT uKernel, which provides a separation between components by using partitions. Each partition executes a component, and uses the Fixed-Priority Scheduling (FPS) policy as a local scheduler to schedule the component's tasks. An offline schedule is used to schedule the partitions on a global level.

Behnam et al. (2008) present an implementation of a HSF based on the periodic resource model in the VxWorks operating system. They keep track of budget depletion by using separate event queues for each server in the HSF by means of absolute times. Whenever a server is activated (or switched in), an event indicating the depletion of the budget, i.e. the current time plus the remaining budget, is added to the server event queue. On preemption of a server, the remaining budget is updated according to the time passed since the last server release and the budget depletion event is removed from the server event queue. When the server's budget depletion event expires, the server is removed from the server ready queue, i.e. it will not be rescheduled until the replenishment of its budget.

Oikawa & Rajkumar (1999), describe the design and implementation of Linux/RK, an implementation of a resource kernel (Portable RK) within the Linux kernel. They minimize the modifications to the Linux kernel by introducing a small number of call back hooks for identifying context switches, with the remainder of the implementation residing in an independent kernel module. Linux/RK introduces the notion of a resource set, which is a set of processor reservations. Once a resource set is created, one or more processes can be attached to it to share its reservations. Although reservations are periodic, periodic tasks inside reservations are not supported. The system employs a replenishment timer for each processor reservation, and a global enforcement timer which expires when the currently running reservation runs out of budget. Whenever a reservation is switched in the enforcement timer is set to its remaining budget. Whenever a reservation is switched out, the enforcement timer is cancelled, and the remaining budget is recalculated.

AQuoSA (Palopoli et al., 2009) also provides the Linux kernel with EDF scheduling and various well-known resource reservation mechanisms, including the constant bandwidth server. Processor reservations are provided as servers, where a server can contain one or more tasks. Periodic tasks are supported by providing an API to sleep until the next period. Similar to Oikawa & Rajkumar (1999) it requires a kernel patch to provide for scheduling hooks and updates the remaining budget and the enforcement timers upon every server switch.

Faggioli et al. (2009) present an implementation of the Earliest Deadline First (EDF) and constant bandwidth servers for the Linux kernel, with support for multicore platforms. It is implemented directly into the Linux kernel. Each task is assigned a period (equal to its relative deadline) and a budget. When a task exceeds its budget, it is stopped until its next period expires and its budget is replenished. This provides temporal protection, as the task behaves like a hard reservation. Each task is assigned a timer, which is activated whenever a task is switched in, by recalculating the deadline event for the task.

Eswaran et al. (2005) describe Nano-RK, a reservation-based RTOS targeted for use in resource-constrained wireless sensor networks. It supports fixed-priority preemptive multitasking, as well as resource reservations for processor, network, sensor and energy. Only one task can be assigned to each processor reservation. Nano-RK also provides explicit support for periodic tasks, where a task can wait for its next period. Each task contains a timestamp for its next period, next replenishment and remaining budget. A one-shot timer drives the timer ISR, which (i) loops through all tasks, to update their timestamps and handle the expired events, and (ii) sets the one-shot timer to the next wakeup time.

Unlike the work presented in (Behnam et al., 2008), which implements a HSF on top of a commercial operating system, and in (Faggioli et al., 2009; Oikawa & Rajkumar, 1999; Palopoli et al., 2009), which implement reservations within Linux, our design for HSF is integrated within a RTOS targeted at embedded systems. Kim et al. (2000) describe a micro-kernel with a two-level HSF and time-triggered scheduling on the global level.

Our design aims at efficiency, in terms of memory and processor overheads, while minimizing the modifications of the underlying RTOS. Unlike Behnam et al. (2008); Oikawa & Rajkumar (1999); Palopoli et al. (2009) it avoids recalculating the expiration of local server events, such as budget depletion, upon every server switch. It also limits the interference of inactive servers on system level by deferring the handling of their local events until they are switched in. While Behnam et al. (2008) present an approach for limiting interference of periodic idling servers, to the best of our knowledge, our work is the first to also cover deferrable servers.

2.3 Hierarchical scheduling in automotive systems

Asberg et al. (2009) make first steps towards using hierarchical scheduling in the AUTOSAR standard. They sketch what it would take to enable the integration of software components by providing temporal isolation between the AUTOSAR components. In (Nolte et al., 2009) they extend their work to systems where components share logical resources, and describe how to apply the SIRAP protocol (Behnam et al., 2007) for synchronizing access to resources shared between tasks belonging to different components. In this work we consider independent components and focus on minimizing the interference between components due to them sharing the timer management system.

3. System model

In this paper we assume a system is composed of independently developed and analyzed components. A components consists of a set of tasks which implement the desired application, a local scheduler, and a server. There is a one-to-one mapping between components and servers.

3.1 Tasks

We consider a set Γ of periodic tasks, where each task $\tau_i \in \Gamma$ is specified by a tuple (i, ϕ_i, T_i, C_i), where i is a fixed priority (smaller i means higher priority), ϕ_i is the task's phasing, T_i is the interarrival time between two consecutive jobs, and C_i is its worst-case execution time. Tasks are preemptive and independent.

3.2 Servers

We consider a set of servers Σ, where each server $\sigma_i \in \Sigma$ is specified by a tuple (i, Π_i, Θ_i), where i is the priority (smaller i means higher priority), Π_i is its replenishment period and Θ_i is its capacity. During runtime, its available budget β_i may vary. Every Π_i time units β_i is replenished to Θ_i. When a server is running, every time unit its available budget β_i is decremented by one.

The mapping of tasks to servers is given by $\gamma(\sigma_i) \subseteq \Gamma$ which defines the set of tasks mapped to server σ_i. We assume that each task is mapped to exactly one server. A task $\tau_j \in \gamma(\sigma_i)$ which is mapped to server σ_i can execute only when $\beta_i > 0$.

3.2.1 Deferrable server

The deferrable server Strosnider et al. (1995) is bandwidth preserving. This means that when a server is switched out because none of its tasks are ready, it will preserve its budget to handle tasks which may become ready later. A deferrable server can be in one of the states shown in Figure 1. A server in the *running* state is said to be *active*, and in either *ready*, *waiting* or *depleted* state is said to be *inactive*. A change from inactive to active or vice-versa is accompanied by the server being *switched in* or *switched out*, respectively.

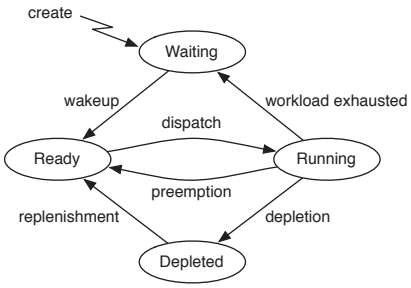

Fig. 1. State transition diagram for the deferrable server. The replenishment transitions from the Ready, Running and Waiting states pointing to the same state are not shown.

A server σ_i is created in the waiting state, with $\beta_i = \Theta_i$. When it is dispatched by the scheduler it moves to running state. A running server may become inactive for one of three reasons:

- It may be preempted by a higher priority server, upon which it preserves its budget and moves to the ready state.
- It may have available budget $\beta_i > 0$, but none of its tasks in $\gamma(\sigma_i)$ may be ready to run, upon which it preserves its budget and moves to the waiting state.
- Its budget may become depleted, upon which it moves to the depleted state.

When a depleted server is replenished it moves to the ready state and becomes eligible to run. A waiting server may be woken up by a newly arrived periodic task or a delay event.

3.2.2 Idling periodic server

When the idling periodic server Davis & Burns (2005) is replenished and none of its tasks are ready, then it idles its budget away until either a proper task arrives or the budget depletes. An idling periodic server follows the state transition diagram in Figure 1, however, due to its idling nature it will never reach the waiting state (and can therefore be regarded as created in ready state).

3.3 Hierarchical scheduling

In two-level hierarchical scheduling one can identify a global scheduler which is responsible for selecting a component. The component is then free to use any local scheduler to select a task to run.

In order to facilitate the reuse of existing components when integrating them to form larger systems, the platform should support (at least) fixed-priority preemptive scheduling at the local level within components (since it is a de-facto standard in the industry). To give the system designer the most freedom it should support arbitrary schedulers at the global level. In this paper we will focus on a fixed-priority scheduler on both local and global level.

3.4 Timed events

The platform needs to support at least the following timed events: task delay, arrival of a periodic task, server replenishment and server depletion.

Events local to server σ_i, such as the arrival of periodic tasks $\tau_j \in \gamma(\sigma_i)$, should not interfere with other servers, unless they wake a server, i.e. the time required to handle them should be accounted to σ_i, rather than the currently running server. In particular, handling the events local to inactive servers should not interfere with the currently active server and should be deferred until the corresponding server is switched in.

4. RELTEQ

To implement the desired extensions in μC/OS-II we needed a general mechanism for different kinds of timed events, exhibiting low runtime overheads. This mechanism should be expressive enough to easily implement higher level primitives, such as periodic tasks, fixed-priority servers and two-level fixed-priority scheduling.

4.1 RELTEQ time model

RELTEQ stores the arrival times of future events relative to each other, by expressing their time *relative to their previous event*. The arrival time of the head event is relative to the current time[1], as shown in Figure 2.

[1] Later in this chapter we will use RELTEQ queues as an underlying data structure for different purposes. We will relax the queue definition: all event times will be expressed relative to their previous event, but the head event will not necessarily be relative to "now".

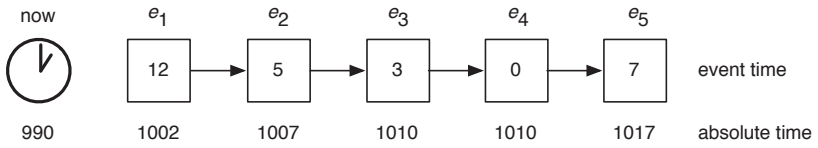

Fig. 2. Example of the RELTEQ event queue.

Unbounded interarrival time between events

One of our requirements is for long event interarrival times with respect to time representation. In other words, given d as the largest value that can be represented for a fixed bit-length time representation, we want to be able to express events which are kd time units apart, for some parameter $k > 1$.

For an n-bit time representation, the maximum interval between two consecutive events in the queue is $2^n - 1$ time units[2]. Using k events, we can therefore represent event interarrival time of at most $k(2^n - 1)$. RELTEQ improves this interval even further and allows for an *arbitrarily long interval* between *any* two events by inserting "dummy" events, as shown in Figure 3.

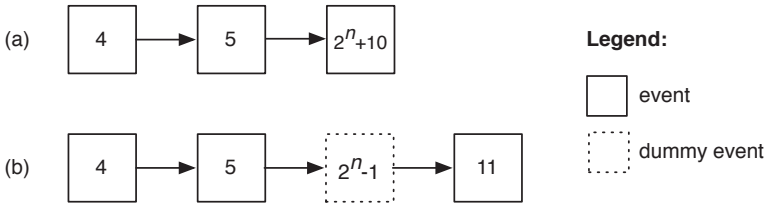

Fig. 3. Example of (a) an overflowing relative event time (b) RELTEQ inserting a dummy event with time $2^n - 1$ to handle the overflow.

If t represents the event time of the last event in the queue, then an event e_i with a time larger than $2^n - 1$ relative to t can be inserted by first inserting dummy events with time $2^n - 1$ at the end of the queue until the remaining relative time of e_i is smaller or equal to $2^n - 1$.

In general, dummy events act as placeholders in queues and can be assigned any time in the interval $[0, 2^n - 1]$.

4.2 RELTEQ data structures

A RELTEQ *event* is specified by the tuple $(kind, time, data)$. The *kind* field identifies the event kind, e.g. a delay or the arrival of a periodic task. *time* is the event time. *data* points to additional data that may be required to handle the event and depends on the event kind. For example, a delay event will point to the task which is to be resumed after the delay event expires. Decrementing an event means decrementing its event time and incrementing an event means incrementing its event time. We will use a dot notation to represent individual fields in the data structures, e.g. $e_i.time$ is the event time of event e_i.

A RELTEQ *queue* is a list of RELTEQ events. *Head(q_i)* represents the head event in queue q_i.

[2] With n bits we can represent 2^n distinct numbers. Since we start at 0, the largest one is $2^n - 1$.

4.3 RELTEQ tick handler

While RELTEQ is not restricted to any specific hardware timer, in this chapter we assume a periodic timer which invokes the *tick handler*, outlined in Figure 4.

The tick handler is responsible for managing the *system queue*, which is a RELTEQ queue keeping track of all the timed events in the system. At every tick of the periodic timer the time of the head event in the queue is decremented. When the time of the head event is 0, then the events with time equal to 0 are popped from the queue and handled.

The scheduler is called at the end of the tick handler, but only in case an event was handled. If no event was handled the currently running task is resumed straightway.

The behavior of a RELTEQ tick handler is summarized in Figure 4.

```
Head(system).time := Head(system).time - 1;
if Head(system).time = 0 then
    while Head(system).time = 0 do
        HandleEvent(Head(system));
        PopEvent(system);
    end while
    Schedule();
end if
```

Fig. 4. Pseudocode for the RELTEQ tick handler.

How an event is handled by *HandleEvent()* depends on its kind. E.g. a *delay event* will resume the delayed task. In general, the event handler will often use the basic RELTEQ primitives, as described in the following sections.

Note that the tick granularity dictates the granularity of any timed events driven by the tick handler: e.g. a server's budget can be depleted only upon a tick. High resolution one-shot timers (e.g. High Precision Event Timer) provide a fine grained alternative to periodic ticks. In case these are present, RELTEQ can easily take advantage of the fine time granularity by setting the timer to the expiration of the earliest event among the active queues. The tick based approach was chosen due to lack of hardware support for high resolution one-shot timers on our example platform. In case such a one-shot timer is available, our RELTEQ based approach can be easily modified to take advantage of it.

4.4 Basic RELTEQ primitives

Three operations can be performed on an event queue: a new event can be inserted, the head event can be popped, and an arbitrary event in the queue can be deleted.

NewEvent(k, t, p) Creates and returns a new event e_i with $e_i.kind = k$, $e_i.time = t$, and $e_i.data = p$.

InsertEvent(q_i, e_j) When a new event e_j with absolute time t_j is inserted into the event queue q_i, the queue is traversed accumulating the relative times of the events until a later event e_k is found, with absolute time $t_k \geq t_j$. When such an event is found, then (i) e_j is inserted before e_k, (ii) its time $e_j.time$ is set relative to the previous event, and (iii) the arrival time of e_k is set relative to e_j (i.e. $t_k - t_j$). If no later event was found, then e_j is appended at the end of the queue, and its time is set relative to the previous event.

PopEvent(q_i) When an event is popped from a queue it is simply removed from the head of the queue q_i. It is not handled at this point.

DeleteEvent(q_i, e_j) Since all events in a queue are stored relative to each other, the time $e_j.time$ of any event $e_j \in q_i$ is critical for the integrity of the events later in the queue. Therefore, before an event e_j is removed from q_i, its event time $e_j.time$ is added to the following event in q_i.

Note that the addition could overflow. In such case, instead of adding $e_j.time$ to the following event in q_i, the kind of e_j is set to a dummy event and the event is not removed. If e_j is the last event in q_i then it is simply removed, together with any dummy events preceding it.

4.5 Event queue implementation

The most straightforward queue implementation is probably a doubly linked list. The time complexity of the *InsertEvent()* operation is then linear in the number of events in the queue, while the complexity of the *DeleteEvent()* and *PopEvent()* operations is constant.

The linear time complexity of the insert operation may be inconvenient for large event queues. An alternative implementation based on a heap or a balanced binary tree may seem more appropriate, as it promises logarithmic time operations. However, as the following theorem states, the relative time representation coupled with the requirement for long event interarrival times (compared to the time representation) make such an implementation impossible.

Theorem 4.1. *Assume that the maximum value we can represent in the time representation is d and also assume that we store times in a tree using relative values no greater than d. Finally, assume that any two events in the tree are at most kd apart in real time, for some parameter k. Then a logarithmic time retrieval of an event from a tree is not possible.*

Proof. If there are k events, the largest time span these k events can represent is kd time units, i.e., the time difference between the first and last event can be at most kd units. If we are to obtain this value by summing over a path this path has to be of length k which leads to a linear representation. This argument pertains to any representation that sums contributions over a path. □

We can illustrate Theorem 4.1 using dummy events: assuming that we start at time 0, the real time of a newly inserted event is at most kd. We would need to insert dummy events until a root path can contain this value. This means we would need to add dummy events until there is a root path of length k.

Conversely, if we assume a tree representation, then we would like to obtain kd as a sum of $\log(k)$ events. If we assume an even distribution over all events, which is the best case with respect to the number of bits required for the time representation, then each event time will be equal to $\frac{k}{\log(k)}d$. This means that $\left\lceil \log\left(\frac{k}{\log(k)}\right)\right\rceil$ extra bits are needed. Therefore, in a tree implementation one cannot limit the time representation to a given fixed value, independent of kd (i.e. the tree span).

In order to satisfy our initial requirement for long event interarrival time, we chose for a linked-list implementation of RELTEQ queues. In future work we look into relaxing this requirement.

5. Periodic tasks

The task concept is an abstraction of a program text. There are roughly three approaches to periodic tasks, depending on the primitives the operating system provides. Figure 5 illustrates the possible implementations of periodic tasks, where function $f_i()$ represents the body of task τ_i (i.e. the actual work done during each job of task τ_i).

```
Task τ_i :
k := 0;
while true do
   now := GetTime();
   DelayFor(φ_i + k * T_i − now);
   k := k + 1;
   f_i();
end while
```

(a)

```
Registration:
TaskMakePeriodic(τ_i, φ_i, T_i);

Task τ_i :
while true do
   TaskWaitPeriod();
   f_i();
end while
```

(b)

```
Registration:
RegisterPeriodic(f_i(), φ_i, T_i);
```

(c)

Fig. 5. Possible implementations of a periodic task.

In Figure 5.a, the periodic behavior is programmed explicitly while in Figure 5.b this periodicity is implicit. The first syntax is typical for a system without support for periodicity, like μC/OS-II. It provides two methods for managing time: $GetTime()$ which returns the current time, and $DelayFor(t)$ which delays the execution of the current task for t time units relative to the time when the method was called. As an important downside, the approach in Figure 5.a may give rise to jitter, when the task is preempted between $now := GetTime()$ and $DelayFor()$.

In order to go from Figure 5.a to 5.c we extract the periodic timer management from the task in two functions: a registration of the task as periodic and a synchronization with the timer system. A straightforward implementation of *TaskWaitPeriod()* is a suspension on a semaphore. Note that we wait at the beginning of the while loop body (rather than at the end) in case $\phi_i > 0$. Going from interface in Figure 5.b to 5.c is now a simple implementation issue.

Note that the task structure described in Figure 5.b guarantees that a job will not start before the previous job has completed, and therefore makes sure that two jobs of the same task will not overlap if the first job's response time exceeds the task's period.

RELTEQ primitives for periodic tasks

In order to provide the periodic task interface in 5.b, we need to implement a timer which expires periodically and triggers the task waiting inside the *TaskWaitPeriod()* call.

To support periodic tasks we introduce a new kind of RELTEQ events: a *period event*. Each period event e_i points to a task τ_i. The expiration of a period event e_i indicates the arrival of

a periodic task τ_i upon which (i) the event time of the e_i is set to T_i and reinserted into the system queue using *InsertEvent()*, and (ii) the semaphore blocking τ_i is raised.

To support periodic tasks we have equipped each task with three additional variables: *TaskPeriod*, expressed in the number of ticks, *TaskPeriodSemaphore*, pointing to the semaphore guarding the release of the task, and *TaskPeriodEvent*, pointing to a RELTEQ period event. For efficiency reasons we have added these directly to the Task Control Block (TCB), which is the μC/OS-II structure storing the state information about a task. Our extensions could, however, reside in a separate structure pointing back to the original TCB.

A task τ_i is made periodic by calling *TaskMakePeriodic*(τ_i, ϕ_i, T_i), which

1. sets the *TaskPeriod* to T_i,
2. removes the *TaskPeriodEvent* from the system queue using *DeleteEvent()*, in case it was already inserted by a previous call to *TaskMakePeriodic()*, otherwise creates a new period event using *NewEvent(period, T_i, τ_i)* and assigns it to *TaskPeriodEvent*.
3. sets the event time of the *TaskPeriodEvent* to ϕ_i if $\phi_i > 0$ or T_i if $\phi_i = 0$, and inserts it into the system queue.

6. Servers

A server σ_i is created using *ServerCreate*$(\Pi_i, \Theta_i, kind)$, where *kind* specifies whether the server is *idling periodic* or *deferrable*. A task τ_i is mapped to server σ_i using *ServerAddTask*(σ_i, τ_i).

In Section 4.3 we have introduced a system queue, which keeps track of pending timed events. For handling periodic tasks assigned to servers we could reuse the system queue. However, this would mean that the tick handler would process the expiration of events local to inactive servers within the budget of the running server.

In order to limit the interference from inactive servers we would like to separate the events belonging to different servers. For this purpose we introduce additional RELTEQ queues for each server. We start this section by introducing additional primitives for manipulating queues, followed by describing how to use these in order to implement fixed-priority servers.

6.1 RELTEQ primitives for servers

We introduce the notion of a pool of queues, and define two pools: *active queues* and *inactive queues*. They are implemented as lists of RELTEQ queues. Conceptually, at every tick of the periodic timer the heads of all active queues are decremented. The inactive queues are left untouched.

To support servers we extend RELTEQ with the following methods:

ActivateQueue(q_i) Moves queue q_i from the inactive pool to the active pool.

DeactivateQueue(q_i) Moves queue q_i from the active pool to the inactive pool.

IncrementQueue(q_i) Increments the head event in queue q_i by 1. Time overflows are handled by setting the overflowing event to $2^n - 1$ and inserting a new dummy event at the head of the queue with time equal to the overflow (i.e. 1).

SyncQueueUntilEvent(q_i, q_j, e_k) Synchronizes queue q_i with queue q_j until event $e_k \in q_j$, by conceptually computing the absolute time of e_k, and then popping and handling all the events in q_i which have occurred during that time interval.

6.2 Limiting interference of inactive servers

To support servers, we add an additional *server queue* for *each* server σ_i, denoted by $\sigma_i.sq$, to keep track of the events local to the server, i.e. delays and periodic arrival of tasks $\tau_j \in \gamma(\sigma_i)$. At any time at most one server can be active; all other servers are inactive. The additional server queues make sure that the events local to inactive servers do not interfere with the currently active server.

When a server σ_i is switched in its server queue is activated by calling *ActivateQueue(σ_i.sq)*. In this new configuration the hardware timer drives two event queues:

1. the *system queue*, keeping track of system events, i.e. the replenishment of periodic servers,
2. the *server queue* of the *active* server, keeping track of the events local to a particular server, i.e. the delays and the arrival of periodic tasks belonging to the server.

When the active server is switched out (e.g. a higher priority server is resumed, or the active server gets depleted) then the active server queue is deactivated by calling *DeactivateQueue(σ_i.sq)*. As a result, the queue of the switched out server will be "paused", and the queue of the switched in server will be "resumed". The system queue is never deactivated.

To keep track of the time which has passed since the last server switch, we introduce a *stopwatch*. The stopwatch is basically a counter, which is incremented with every tick. In order to handle time overflows discussed in Section 4.1, we represent the stopwatch as a RELTEQ queue and use *IncrementQueue(stopwatch)* to increment it.

During the time when a server is inactive, several other servers may be switched in and out. Therefore, next to keeping track of time since the last server switch, for each server we also need to keep track of how long it was inactive, i.e the time since that particular server was switched out. Rather than storing a separate counter for each server, we multiplex the stopwatches for all servers onto the single stopwatch which we have already introduced, exploiting the RELTEQ approach. We do this by inserting a *stopwatch event*, denoted by $\sigma_i.se$, at the head of the stopwatch queue using *InsertEvent(stopwatch, σ_i.se)* whenever server σ_i is switched out. The event points to the server and its time is initially set to 0. The behavior of the tick handler with respect to the stopwatch remains unchanged: upon every tick the head event in the stopwatch queue is incremented using *IncrementQueue(stopwatch)*.

During runtime the stopwatch queue will contain one stopwatch event for every inactive server (the stopwatch event for the currently active server is removed when the server is switched in). The semantics of the stopwatch queue is defined as follows: the accumulated time from the head of the queue until (and including) a stopwatch event $\sigma_i.se$ represents the time the server σ_i was switched out.

When a server σ_i is switched in, its server queue is synchronized with the stopwatch using *SyncQueuesUntilEvent(σ_i.sq, stopwatch, σ_i.se)*, which handles all the events in $\sigma_i.sq$ which might have occurred during the time the server was switched out. It accumulates the time in the stopwatch queue until the stopwatch event $\sigma_i.se$ and handles all the events in $\sigma_i.sq$ which have expired during that time. Then $\sigma_i.se$ is removed from the stopwatch queue. When σ_i is switched out, $\sigma_i.se$ with time 0 is inserted at the head of the stopwatch queue.

6.2.1 Example of the stopwatch behavior

The stopwatch queue is a great example of RELTEQ's strength. It provides an efficient and concise mechanism for keeping track of the inactive time for *all* servers. Figure 6 demonstrates the behavior of the stopwatch queue for an example system consisting of three servers A, B and C. It illustrates the state of the stopwatch queue at different moments during execution, before the currently running server is switched out and after the next server is switched in.

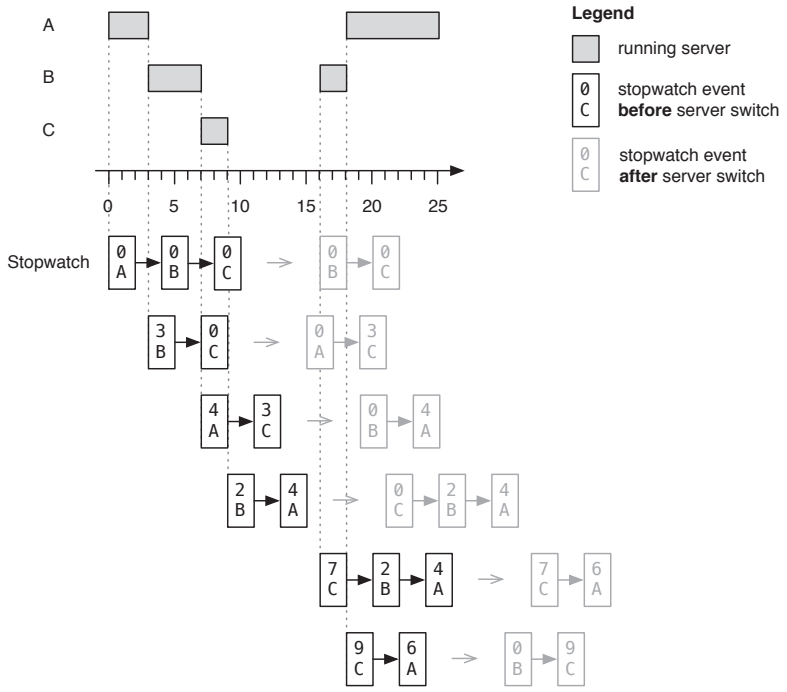

Fig. 6. Example of the stopwatch queue.

Initially, when server σ_i is created, a stopwatch event $\sigma_i.se$ with time 0 is inserted into the stopwatch queue. At time 0 server A is switched in and its stopwatch event is removed. While server A is running, the tick handler increments the head of the stopwatch queue, which happens to be the stopwatch event of server B. At time 3, when server A is switched out and server B is switched in, server B synchronizes its absolute queue with the stopwatch queue until and including $B.se$, $B.se$ is deleted, and $A.se$ with time 0 is inserted. Note that when $B.se$ is deleted, its time is added to $C.se$.

At time 7 server C is switched in, its absolute queue is synchronized with time $4 + 3 = 7$, after which $C.se$ is deleted, and $B.se$ with time 0 is inserted.

At time 9, since no server is switched in, no synchronization is taking place and no stopwatch event is deleted. Only stopwatch event $C.se$ with time 0 is inserted, since server C is switched out.

At time 16, when server B is switched in and its stopwatch event $B.se$ is deleted, the time of $B.se$ is added to $C.se$.

6.2.2 Deferrable server

When the workload of a deferrable server σ_i is exhausted, i.e. there are no ready tasks in $\gamma(\sigma_i)$, then the server is switched out and its server queue $\sigma_i.sq$ is deactivated. Consequently, any periodic task which could wake up the server to consume any remaining budget cannot be noticed. One could alleviate this problem by keeping its server queue active when σ_i is switched out. This, however, would make the tick handler overhead linear in the number of deferrable servers, since a tick handler decrements the head events in all active queues (see Section 6.6).

Instead, in order to limit the interference of inactive deferrable servers, when a deferrable server σ_i is switched out and it has no workload pending (i.e. no tasks in $\gamma(\sigma_i)$ are ready), we deactivate the σ_i's server queue, change its state to waiting, and insert a *wakeup event*, denoted as $\sigma_i.we$, into the system queue. The wakeup event has its *data* pointing to σ_i and time equal to the arrival of the first event in $\sigma_i.sq$. When the wakeup event expires, the σ_i's state is set to the ready state. This way handling the events inside $\sigma_i.sq$ is deferred until σ_i is switched in.

When a deferrable server is switched in while it is in the waiting state, its wakeup event $\sigma_i.we$ is removed from the system queue.

6.2.3 Idling periodic server

An idling periodic server is a special kind of a deferrable server containing an idle task (with lowest priority). The idle task is switched in if no higher priority task is ready, effectively idling away the remaining capacity. In order to save memory needed for storing the task control block and the stack of the idle task, one idle task is shared between all idling periodic servers in the system.

6.3 Virtual timers

When the server budget is depleted an event must be triggered, to guarantee that a server does not exceed its budget. We present a general approach for handling budget depletion and introduce the notion of *virtual timers*, which are events relative to server's budget consumption.

We can implement virtual timers by adding a *virtual server queue* for *each* server, denoted by $\sigma_i.vq$. Similarly to the server queues introduced earlier, when a server is switched in, its virtual server queue is activated. The difference is that the virtual server queue is not synchronized with the stopwatch queue, since during the inactive period a server does not consume any of its budget. When a server is switched out, its virtual server queue is deactivated.

The relative time representation by RELTEQ allows for a more efficient virtual queue activation than an absolute time representation does. An absolute time representation (e.g. in (Behnam et al., 2008; Inam et al., 2011)) requires to recompute the expiration time for *all* the events in a virtual server queue upon switching in the corresponding server, which is linear in the number of events. In our RELTEQ-based virtual queues the events are stored relative to each other and their expiration times do not need to be recomputed upon queue activation. Note that it will never be necessary to handle an expired virtual event upon queue activation, since such an event would have been already handled before the corresponding server was switched out. Therefore, our HSF design exhibits a constant time activation of a virtual server queue.

6.4 Switching servers

The methods for switching servers in and out are summarized in Figures 7 and 8.

```
SyncQueuesUntilEvent(σi.sq, stopwatch, σi.se);
ActivateQueue(σi.sq);
ActivateQueue(σi.vq);
if σi.we ≠ ∅ then
    DeleteEvent(system, σi.we);
    σi.we = ∅;
end if
```

Fig. 7. Pseudocode for *ServerSwitchIn*(σ_i).

```
DeleteEvent(stopwatch, σi.se);
σi.se = NewEvent(stopwatch, 0, σi);
InsertEvent(stopwatch, σi.se);
DeactivateQueue(σi.sq);
DeactivateQueue(σi.vq);
if σi.readyTasks = ∅ then
    σi.we = NewEvent(wakeup, Head(σi.sq).time, σi);
    InsertEvent(system, σi.we);
end if
```

Fig. 8. Pseudocode for *ServerSwitchOut*(σ_i).

6.5 Server replenishment and depletion

We introduce two additional RELTEQ event kinds to support servers: *budget replenishment* and *budget depletion*. When a server σ_i is created, a replenishment event e_j is inserted into the *system queue*, with $e_j.data$ pointing to σ_i and $e_j.time$ equal to the server's replenishment period Π_i. When e_j expires, $e_j.time$ is updated to Π_i and it is inserted into the system queue.

Upon replenishment, the server's depletion event e_j is inserted into its *virtual server queue*, with $e_j.data$ pointing to σ_i and $e_j.time$ equal to the server's capacity Θ_i. If the server was not depleted yet, then the old depletion event is removed from the virtual server queue using *DeleteEvent*($\sigma_i.vq$, e_j).

6.6 RELTEQ tick handler with support for servers

An example of the RELTEQ queues managed by the tick handler in the proposed RELTEQ extension with servers is summarized in Figure 9. Conceptually, every tick the stopwatch queue is incremented and the heads of the system queue, the active server queue and the active virtual server queue are decremented. If the head of any queue becomes 0, then their head event is popped and handled until the queue is exhausted or the head event has time larger than 0.

Actually, rather than decrementing the head of each active queue and checking whether it is 0, a *CurrentTime* counter is incremented and compared to the *Earliesttime*, which is set whenever the head of an active queue changes. If they are equal, then (i) the *CurrentTime* is subtracted from the heads of all the active queues, (ii) any head event with time 0 is popped and handled,

(iii) *CurrentTime* is set to 0, and (iv) *Earliesttime* is set to the earliest time among the heads of all active queues[3].

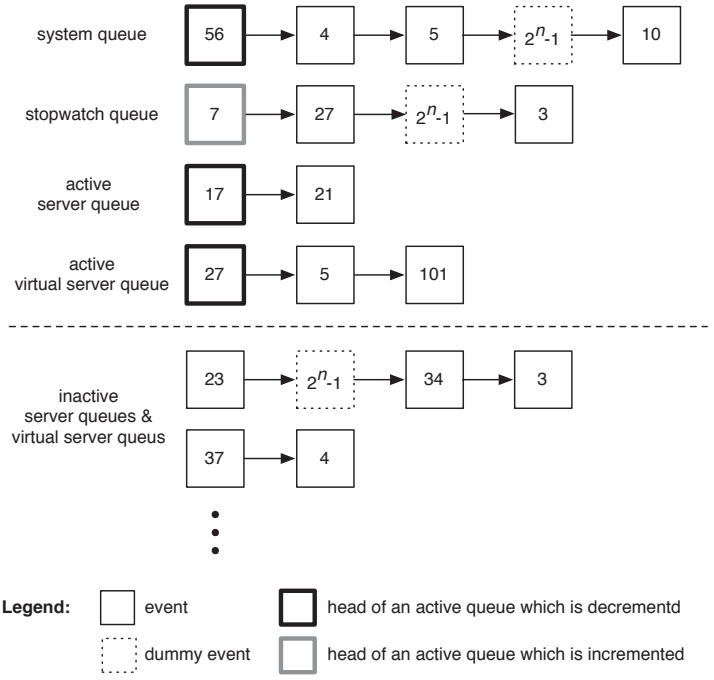

Fig. 9. Example of the RELTEQ queues managed by the tick handler.

The behavior of a RELTEQ tick handler supporting servers is summarized in Figure 10.

```
IncrementQueue(stopwatch);
CurrentTime := CurrentTime + 1;
if Earliesttime = CurrentTime then
   for all q ∈ activequeues do
      Head(q).time := Head(q).time − CurrentTime;
      while Head(q).time = 0 do
         HandleEvent(Head(q));
         PopEvent(q);
      end while
   end for
   CurrentTime := 0;
   Earliesttime := Earliest(activequeues);
   Schedule();
end if
```

Fig. 10. Pseudocode for the RELTEQ tick handler supporting hierarchical scheduling.

Note that at any moment in time there are at most four active queues, as shown in Figure 9.

[3] Note that the *time* of any event will never become negative.

6.7 Summary

We have described a generic framework for supporting servers, which is tick based (Section 4.3) and limits the interference of inactive servers on system level (Section 6.2). The interference of inactive servers which are either ready or depleted was limited by means of a combination of inactive server queues and a stopwatch queue. Deactivating server queues of waiting servers was made possible by inserting a wakeup event into the system queue, in order to wake up the server upon the arrival of a periodic task while the server is switched out.

7. Hierarchical scheduling

Rajkumar et al. (1998) identified four ingredients necessary for guaranteeing resource provisions: admission, monitoring, scheduling and enforcement. In this section we describe how our implementation of HSF addresses each of them.

7.1 Admission

We allow admission of new components only during the integration, not during runtime. The admission testing requires analysis for hierarchical systems, which is outside the scope of this chapter.

7.2 Monitoring

There are two reasons for monitoring the budget consumption of servers: (i) handle the budget depletion and (ii) allow the assigned tasks to track and adapt to the available budget.

In order to notice the moment when a server becomes depleted we have introduced a virtual *depletion event* for every server, which is inserted into its virtual server queue. When the depletion event expires, then (i) the server's capacity is set to 0, (ii) its state is set to depleted, and (iii) the scheduler is called.

In order to allow tasks $\tau_j \in \gamma(\sigma_i)$ to track the server budget β_i we equipped each server with a *budget counter*. Upon every tick the budget counter of the currently active server is decremented by one. The depletion event will make sure that a depleted server is switched out before the counter becomes negative. We also added the *ServerBudget()* method, which can be called by any task.

ServerBudget(σ_i) Returns the current value of β_i, which represents the lower bound on the processor time that server σ_i will receive within the next interval of Π_i time units.

7.3 Scheduling

The μC/OS-II the scheduler does two things: (i) select the highest priority ready task, and (ii) in case it is different from the currently running one, do a context switch. Our hierarchical scheduler replaces the original *OS_SchedNew()* method, which is used by the μC/OS-II scheduler to select the highest priority ready task.

It first uses the *global scheduler HighestReadyServer()* to select the highest priority ready server, and then the server's *local scheduler HighestReadyTask()*, which selects the highest priority ready

task belonging to that server. This approach allows to implement different global and local schedulers, and also different schedulers in each server. Our fixed-priority global scheduler is shown in Figure 11.

```
highestServer := HighestReadyServer();
if highestServer ≠ currentServer then
   if currentServer ≠ ∅ then
      ServerSwitchOut(currentServer);
   end if
   if highestServer ≠ ∅ then
      ServerSwitchIn(highestServer);
   end if
   currentServer := highestServer;
end if
if currentServer ≠ ∅ then
   return currentServer.HighestReadyTask();
else
   return idleTask;
end if
```

Fig. 11. Pseudocode for the hierarchical scheduler.

The *currentServer* is a global variable referring to the currently active server. Initially $currentServer = \emptyset$, where $\emptyset$ refers to a null pointer.

The scheduler first determines the highest priority ready server. Then, if the server is different from the currently active server, a server switch is performed, composed of 3 steps:

1. If there is a currently active server, then it is switched out, using *ServerSwitchOut()* described in Section 6.4.
2. If there is a ready server, then it is switched in, using *ServerSwitchIn()* described in Section 6.4.
3. The *currentServer* is updated.

Finally the highest priority task in the currently active server is selected, using the current server's local scheduler *HighestReadyTask()*. If no server is active, then the idle task is returned.

7.4 Enforcement

When a server becomes depleted during the execution of one of its tasks (i.e. if a depletion event expires), the task will be preempted and the server will be switched out. This is possible, since we assume preemptive and independent tasks.

8. Evalulation

In this section we evaluate the modularity, memory footprint and performance of the HSF extension for RELTEQ. We chose a linked-list as the data structure underlying our RELTEQ queues and implemented the proposed design within μC/OS-II.

8.1 Modularity and memory footprint

The design of RELTEQ and the HSF extension is modular, allowing to enable or disable the support for HSF and different server types during compilation with a single compiler directive for each extension.

The complete RELTEQ implementation including the HSF extension is 1610 lines of code (excluding comments and blank lines), compared to 8330 lines of the original μC/OS-II. 105 lines of code were inserted into the original μC/OS-II code, out of which 60 were conditional compilation directives allowing to easily enable and disable our extensions. No original code was deleted or modified. Note that the RELTEQ+HSF code can replace the existing timing mechanisms in μC/OS-II, and that it provides a framework for easy implementation of other scheduler and servers types.

The 105 lines of code represent the effort required to port RELTEQ+HSF to another operating system. Such porting requires (i) redirecting the tick handler to the RELTEQ handler, (ii) redirecting the method responsible for selecting the the highest priority task to the HSF scheduler, and (iii) identifying when tasks become ready or blocked.

The code memory footprint of RELTEQ+HSF is 8KB, compared to 32KB of the original μC/OS-II. The additional data memory foot print for an application consisting of 6 servers with 6 tasks each is 5KB, compared to 47KB for an application consisting of 36 tasks (with a stack of 128B each) in the original μC/OS-II.

8.2 Performance analysis

In this section we evaluate the system overheads of our extensions, in particular the overheads of the scheduler and the tick handler. We express the overhead in terms of the maximum number of events inside the queues which need to be handled in a single invocation of the scheduler or the tick handler, times the maximum overhead for handling a single event.

8.2.1 Handling a single event

Handling different events will result in different overheads.

- When a dummy event expires, it is simply removed from the head of the queue. Hence, handling it requires $O(1)$ time.
- When a task period event expires, an event representing the next periodic arrival is inserted into the corresponding server queue. In this section we assume a linked-list implementation, and consequently insertion is linear in the number of events in a queue. Note that we could delay inserting the next period event until the task completes, as at most one job of a task may be running at a time. This would reduce the handling of a periodic arrival to constant time, albeit at the additional cost of keeping track for each task of the time since its arrival, which would be taken into account when inserting the next period event. However, if we would like to monitor whether tasks complete before their deadline, then we will need to insert a deadline event into $\sigma_i.sq$ anyway. Hence the time for handling an event inside of a server queue is linear in the number of events in a server queue. Since there are at most two events per task in a server queue (period and deadline events), handling a period event is linear in the maximum number of tasks assigned to a server, i.e. $O(m(\sigma_i))$.

- When a task deadline event expires, it is simply removed from the head of the queue an the system is notified that a deadline was missed. Hence, handling it requires $O(1)$ time.
- When a server replenishment event expires, an event representing the next replenishment is inserted into the system queue[4]. Since there are at most two events in the system queue per server (replenishment and wakeup event), handling a replenishment event is linear in the number of servers, i.e $O(|\Sigma|)$.
- When a server depletion event expires, it is simply removed from the queue. Hence, handling it requires $O(1)$ time.

8.2.2 Scheduler

Our HSF supports different global and local schedulers. For the sake of a fair comparison with the μC/OS-II which implements a fixed-priority scheduler, we also assume fixed-priority global and local schedulers in this section. For both global and local scheduling we can reuse the bitmap-based approach implemented by μC/OS-II, which has a constant time overhead for selecting the highest priority ready task as well as indicating that a task is ready or not (Labrosse, 2002). Consequently, in our HSF we can select the highest priority server and task within a server in constant time.

Once a highest priority server σ_i is selected, the overhead of switching in the server depends on the number of events inside the stopwatch queue and σ_i's server queue (which needs to be synchronized with the stopwatch), and the overhead of selecting the highest priority task.

The stopwatch queue contains one stopwatch event for each inactive server. The length of the stopwatch queue is therefore bounded by $|\Sigma| + d_s$, where $|\Sigma|$ is the number of servers, and $d_s = \max_{\sigma_i \in \Sigma} \left\lfloor \frac{t_s(\sigma_i)}{2^n - 1} \right\rfloor$ is the maximum number of dummy events inside the stopwatch queue. $t_s(\sigma_i)$ is the longest time interval that a server can be switched out, and $2^n - 1$ is the largest relative time which can be represented with n bits.

The only local events are a task delay and the arrival of a periodic task. Also, each task can wait for at most one timed event at a time. The number of events inside the server queue is therefore bounded by $m(\sigma_i) + d_l(\sigma_i)$, where $m(\sigma_i)$ is the maximum number of tasks assigned to server σ_i, and $d_l(\sigma_i) = \left\lfloor \frac{t_l(\sigma_i)}{2^n - 1} \right\rfloor$ is the maximum number of dummy events local to the server queue $\sigma_i.sq$. $t_l(\sigma_i)$ is the longest time interval between any two events inside of $\sigma_i.sq$ (e.g. the longest task period or the longest task delay).

The complexity of the scheduler is therefore $O(|\Sigma| + d_s + m(\sigma_i) + d_l(\sigma_i))$. Note that the maximum numbers of dummy events d_s and $d_l(\sigma_i))$ can be determined at design time.

8.2.3 Tick handler

The tick handler synchronizes all active queues with the current time, and (in case an event was handled) calls the scheduler. The active queues are comprised of the system queue and two queues for the server σ_i which is active at the time the tick handler is invoked (its server queue $\sigma_i.sq$ and virtual server queue $\sigma_i.vq$).

[4] Inserting the next replenishment event could be deferred until the server is depleted, at a similar cost and benefit to deferring the insertion of the task period event.

The system queue contains only replenishment and wakeup events. Its size is therefore proportional to $|\Sigma| + d_g$, where $d_g = \left\lfloor \frac{t_g}{2^n - 1} \right\rfloor$ is the maximum number of dummy events inside the global system queue. t_g is the longest time interval between any two events inside the global system queue (i.e. the longest server period).

The size of $\sigma_i.sq$ is linear in the number of tasks assigned to the server. Similarly, since $\sigma_i.vq$ contains one depletion event and at most one virtual timer for each task, its size is linear in the number of tasks assigned to the server.

Therefore, when server σ_i is active at the time the tick handler is invoked, the tick handler will need to handle $t_t(\sigma_i) = |\Sigma| + d_g + m(\sigma_i) + d_l(\sigma_i)$ events. The complexity of the tick handler is therefore $O(\max_{\sigma_i \in \Sigma} m(\sigma_i) t_t(\sigma_i))$. Note that the tick handler overhead depends only on tasks belonging to the server σ_i which is active at the time of the tick. It does not depend on tasks belonging to other servers.

8.2.4 Experimental results

In Section 6.2 we introduced wakeup events in order to limit the interference due to inactive servers. In order to validate this approach we have also implemented a variant of the HSF scheduler which avoids using wakeup events and instead, whenever a deferrable server σ_i is switched out, it keeps the server queue $\sigma_i.sq$ active. Consequently, the scheduler does not need to synchronize the server queue when switching in a server. However, this overhead is shifted to the tick handler, which needs to handle the expired events in all the server queues from inactive deferrable servers. In the following discussion we refer to this approach as *without limited interference*, as opposed to *with limited interference* based on wakeup events.

Figures 12.a to 12.e compare the two variants. We have varied the number of deferrable servers and the number of tasks assigned to each server (all servers had the same number of tasks). The server replenishment periods and the task periods were all set to the same value (100ms), to exhibit the maximum overhead by having all tasks arrive at the same time. Each task had a computation time of 1ms and each server had a capacity of 7ms. We have run the setup within the cycle-accurate hardware simulator for the OpenRISC 1000 architecture OpenCores (2010). We have set the processor clock to 345MHz and the tick to 1KHz, which is inline with the platform used by our industrial partner. Each experiment was run for an integral number of task periods.

Figures 12.a and 12.b show the maximum measured overheads of the scheduler and the tick handler, while Figures 12.c and 12.d show the total overhead of the scheduler and the tick handler in terms of processor utilization. The figures demonstrate that wakeup events reduce the tick overhead, at the cost of increasing the scheduler overhead, by effectively shifting the overhead of handling server's local events from the tick handler to the scheduler. Since the scheduler overhead is accounted to the server which is switched in, as the number of servers and tasks per server increase, so does the advantage of the limited interference approach. Figure 12.e combines Figures 12.c and 12.d and verifies that the additional overhead due to the wakeup events in the limited interference approach is negligible.

Figure 12.f compares the system overheads of our HSF extension to the standard μC/OS-II implementation. As the standard μC/OS-II does not implement hierarchical scheduling, we have implemented a flat system containing the same number of tasks with the same parameters as in Figures 12.a and 12.b. The results demonstrate the temporal isolation and

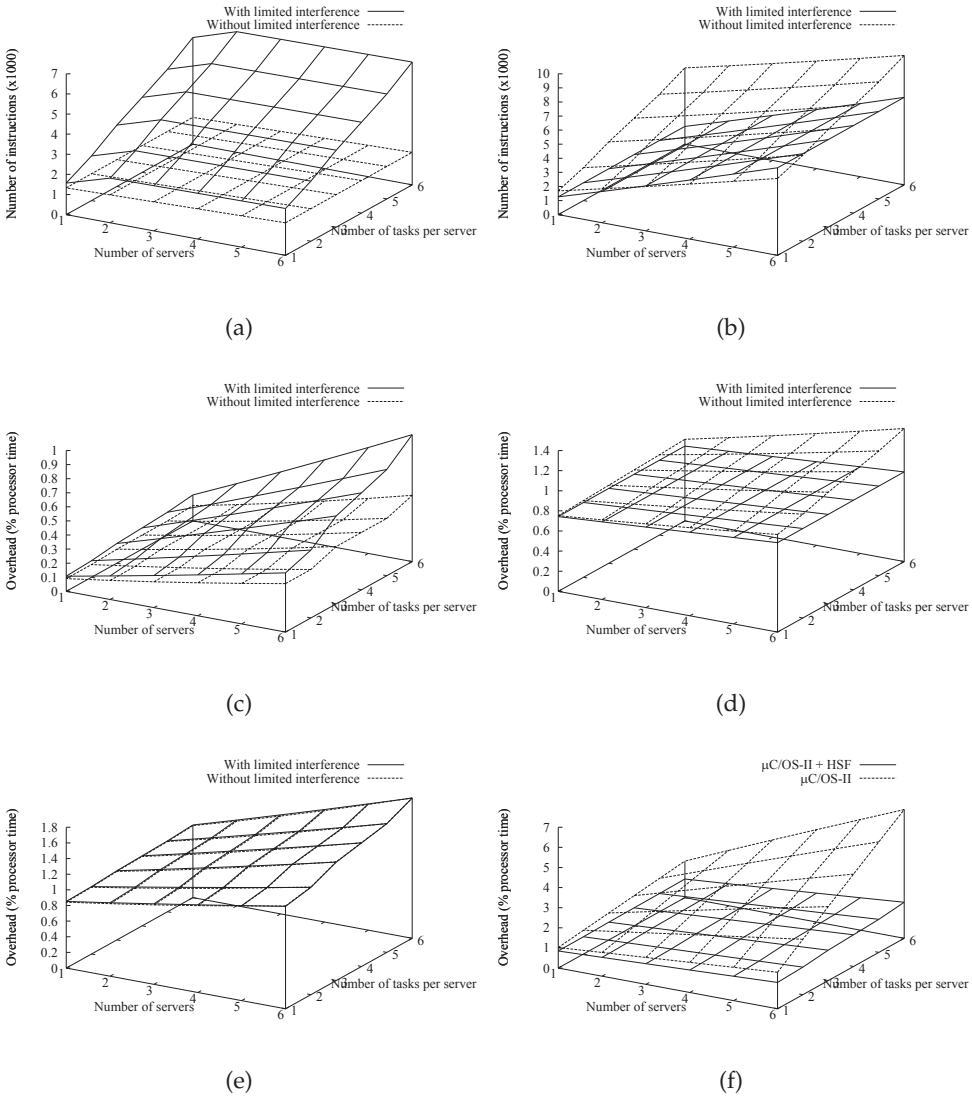

Fig. 12. (a) maximum overhead of the (local + global) scheduler, (b) maximum overhead of the tick handler, (c) total overhead of the scheduler, (d) total overhead of the tick handler, (e) and (f) total overhead of the tick handler and the scheduler.

efficiency of our HSF implementation. While the standard μC/OS-II scans through all tasks on each tick to see if any task delay has expired, in the HSF extension the tick handler needs to consider only head event in the server queue of the currently running server.

Figure 13 compares the best-case and worst-case measured overheads of the scheduler and tick handler between μC/OS-II with our extensions, compared to the standard μC/OS-II, for which we have implemented a flat system containing the same number of tasks with the same parameters as for the μC/OS-II+HSF case.

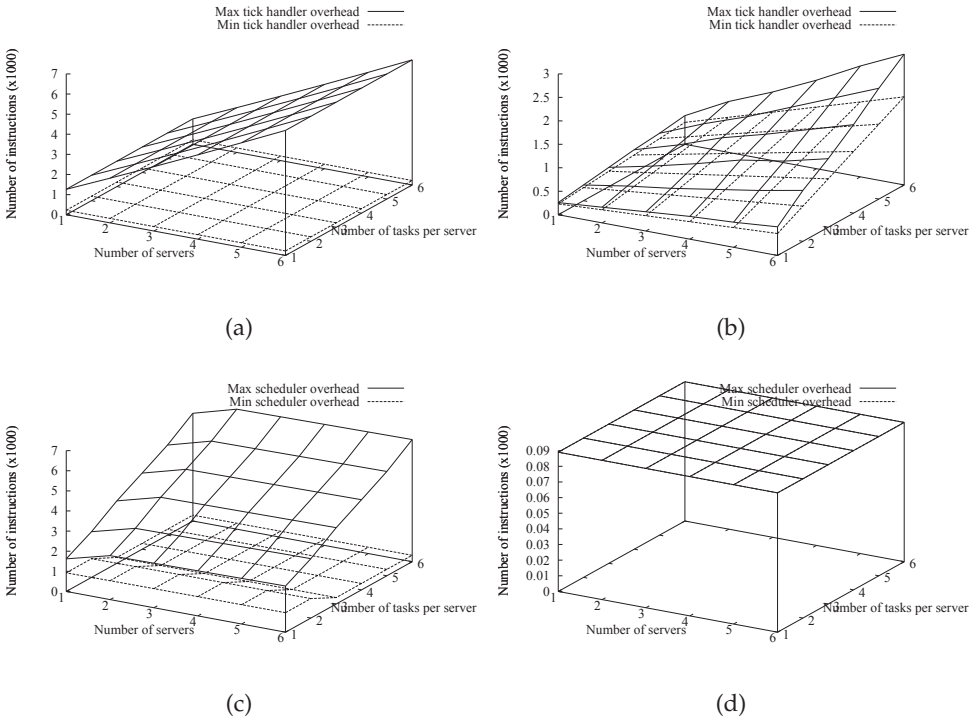

Fig. 13. (a) tick handler overhead in μC/OS-II+HSF, (b) tick handler overhead in μC/OS-II, (c) scheduler overhead in μC/OS-II+HSF, (d) scheduler overhead in μC/OS-II.

The figure shows that both scheduler and tick handler suffer larger execution time jitter under μC/OS-II+HSF, than the standard μC/OS-II. In the best case the μC/OS-II+HSF tick handler needs to decrement only the head of the system queue, while in μC/OS-II the tick hander traverses all the tasks in the system and for each one it checks whether its timer has expired.

In a system with small utilization of individual tasks and servers (as was the case in our experiments), most local events will arrive while the server is switched out. Since handling local events is deferred until the server is switched in and its server queue synchronized with the stopwatch queue, it explains why the worst-case tick handler overhead is increasing with the number of servers and the worst-case scheduler overhead is increasing with the number of tasks per server.

9. Conclusions

We have presented an efficient, modular and extendible design for enhancing a real-time operating system with a two-level HSF. It relies on Relative Timed Event Queues (RELTEQ), a general timer management system targeting embedded systems. Our design supports various server types (including polling, idling periodic, deferrable and constant-bandwidth servers), and global and virtual timers. It supports fixed-priority and EDF schedulers on both local and global level. It provides temporal isolation between components and limits the interference of inactive servers on the active server, by means of wakeup events and a combination of inactive server queues with a stopwatch. We have evaluated a fixed-priority based implementation of our RELTEQ and HSF within the μC/OS-II real-time operating system used in the automotive domain. The results demonstrate that our approach exhibits low performance overhead and limits the necessary modifications of the underlying operating system.

We have assumed a linked-list implementation of our RELTEQ queues, and indicated the challenges of a tree-based implementation due to the relative time representation. In the future we want to investigate in more detail other advanced data structures for implementing RELTEQ queues.

10. References

Abeni, L. & Buttazzo, G. (1998). Integrating multimedia applications in hard real-time systems, *Real-Time Systems Symposium (RTSS)*, pp. 4–14.

Asberg, M., Behnam, M., Nemati, F. & Nolte, T. (2009). Towards hierarchical scheduling in AUTOSAR, *Emerging Technologies Factory Automation (ETFA)*, pp. 1 –8.

AUTOSAR (2011). Automotive open system architecture (AUTOSAR). URL: *http://www.autosar.org*

Behnam, M., Nolte, T., Shin, I., Åsberg, M. & Bril, R. J. (2008). Towards hierarchical scheduling on top of VxWorks, *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pp. 63–72.

Behnam, M., Shin, I., Nolte, T. & Nolin, M. (2007). Sirap: a synchronization protocol for hierarchical resource sharingin real-time open systems, *International Conference on Embedded Software (EMSOFT)*, pp. 279–288.

Buttazzo, G. & Gai, P. (2006). Efficient EDF implementation for small embedded systems, *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*.

Carlini, A. & Buttazzo, G. C. (2003). An efficient time representation for real-time embedded systems, *ACM Symposium on Applied Computing*, pp. 705–712.

Chodrow, S., Jahanian, F. & Donner, M. (1991). Run-time monitoring of real-time systems, *Real-Time Systems Symposium (RTSS)*, pp. 74–83.

Davis, R. I. & Burns, A. (2005). Hierarchical fixed priority pre-emptive scheduling, *Real-Time Systems Symposium (RTSS)*, pp. 389–398.

Engler, D. R., Kaashoek, M. F. & O'Toole, Jr., J. (1995). Exokernel: an operating system architecture for application-level resource management, *Symposium on Operating Systems Principles (SOSP)*, pp. 251–266.

Eswaran, A., Rowe, A. & Rajkumar, R. (2005). Nano-RK: An energy-aware resource-centric RTOS for sensor networks, *Real-Time Systems Symposium (RTSS)*, pp. 256–265.

Evidence (n.d.). ERIKA Enterprise: Open Source RTOS for single- and multi-core applications. URL: *http://www.evidence.eu.com*

Faggioli, D., Trimarchi, M., Checconi, F. & Scordino, C. (2009). An EDF scheduling class for the linux kernel, *Real-Time Linux Workshop*.

Holenderski, M., Cools, W., Bril, R. J. & Lukkien, J. J. (2009). Multiplexing real-time timed events, *Emerging Technologies and Factory Automation (ETFA)*, pp. 1718–1721.

Holenderski, M., Cools, W., Bril, R. J. & Lukkien, J. J. (2010). Extending an open-source real-time operating system with hierarchical scheduling, *Technical Report CS-report 10-10*, Eindhoven University of Technology.

Inam, R., Maki-Turja, J., Sjodin, M., Ashjaei, S. & Afshar, S. (2011). Support for hierarchical scheduling in freertos, *Emerging Technologies Factory Automation (ETFA)*, pp. 1 –10.

Kim, D., Lee, Y.-H. & Younis, M. (2000). SPIRIT-μKernel for strongly partitioned real-time systems, *Real-Time Systems and Applications (RTCSA)*, pp. 73–80.

Labrosse, J. J. (2002). *MicroC/OS-II: The Real Time Kernel*, 2nd edition edn, CMP Books.

Lehoczky, J. P., Sha, L. & Strosnider, J. K. (1987). Enhanced aperiodic responsiveness in hard real-time environments, *Real-Time Systems Symposium (RTSS)*, pp. 261–270.

Mercer, C., Rajkumar, R. & Zelenka, J. (1994). Temporal protection in real-time operating systems, *IEEE Workshop on Real-Time Operating Systems and Software*, pp. 79–83.

Nolte, T. (2011). Compositionality and CPS from a platform perspective, *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Vol. 2, pp. 57 –60.

Nolte, T., Shin, I., Behnam, M. & Sjödin, M. (2009). A synchronization protocol for temporal isolation of software components in vehicular systems, *IEEE Transactions on Industrial Informatics* 5(4): 375–387.

Oikawa, S. & Rajkumar, R. (1999). Portable RK: a portable resource kernel for guaranteed and enforced timing behavior, *Real-Time Technology and Applications Symposium (RTAS)*, pp. 111–120.

OpenCores (2010). Openrisc 1000: Architectural simulator. URL: *http://opencores.org/openrisc,or1ksim*

Palopoli, L., Cucinotta, T., Marzario, L. & Lipari, G. (2009). Aquosa—adaptive quality of service architecture, *Software – Practice and Experience* 39(1): 1–31.

Rajkumar, R., Juvva, K., Molano, A. & Oikawa, S. (1998). Resource kernels: A resource-centric approach to real-time and multimedia systems, *Conference on Multimedia Computing and Networking (CMCN)*, pp. 150–164.

Saewong, S., Rajkumar, R. R., Lehoczky, J. P. & Klein, M. H. (2002). Analysis of hierarchical fixed-priority scheduling, *Euromicro Conference on Real-Time Systems (ECRTS)*, p. 173.

Shin, I. & Lee, I. (2003). Periodic resource model for compositional real-time guarantees, *Real-Time Systems Symposium (RTSS)*, pp. 2–13.

Shin, I. & Lee, I. (2008). Compositional real-time scheduling framework with periodic model, *ACM Transactions on Embedded Computing Systems* 7: 30:1–30:39.

Strosnider, J. K., Lehoczky, J. P. & Sha, L. (1995). The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments, *IEEE Transactions on Computers* 44(1): 73–91.

van den Heuvel, M. M. H. P., Holenderski, M., Bril, R. J. & Lukkien, J. J. (2011). Constant-bandwidth supply for priority processing, *IEEE Transactions on Consumer Electronics* 57(2): 873–881.

van den Heuvel, M. M. H. P., Holenderski, M., Cools, W., Bril, R. J. & Lukkien, J. J. (2009). Virtual timers in hierarchical real-time systems, *Work in Progress session of the Real-time Systems Symposium (RTSS)*.

Real-Time Systems, Architecture, Scheduling, and Application
Edited by Dr. Seyed Morteza Babamir

ISBN 978-953-51-0510-7
Hard cover, 334 pages
Publisher InTech
Published online 11, April, 2012
Published in print edition April, 2012

This book is a rich text for introducing diverse aspects of real-time systems including architecture, specification and verification, scheduling and real world applications. It is useful for advanced graduate students and researchers in a wide range of disciplines impacted by embedded computing and software. Since the book covers the most recent advances in real-time systems and communications networks, it serves as a vehicle for technology transition within the real-time systems community of systems architects, designers, technologists, and system analysts. Real-time applications are used in daily operations, such as engine and break mechanisms in cars, traffic light and air-traffic control and heart beat and blood pressure monitoring. This book includes 15 chapters arranged in 4 sections, Architecture (chapters 1-4), Specification and Verification (chapters 5-6), Scheduling (chapters 7-9) and Real word applications (chapters 10-15).

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Mike Holenderski, Reinder J. Bril and Johan J. Lukkien (2012). An Efficient Hierarchical Scheduling Framework for the Automotive Domain, Real-Time Systems, Architecture, Scheduling, and Application, Dr. Seyed Morteza Babamir (Ed.), ISBN: 978-953-51-0510-7, InTech, Available from: http://www.intechopen.com/books/real-time-systems-architecture-scheduling-and-application/an-efficient-hierarchical-scheduling-framework-for-the-automotive-domain

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821