SLOTH ON TIME: Efficient Hardware-Based Scheduling for Time-Triggered RTOS*

Wanja Hofer, Daniel Danner, Rainer Müller,
Fabian Scheler, Wolfgang Schröder-Preikschat, Daniel Lohmann
Friedrich–Alexander University Erlangen–Nuremberg, Germany
E-Mail: {hofer,danner,raimue,scheler,wosch,lohmann}@cs.fau.de

***Abstract*—Traditional time-triggered operating systems are implemented by multiplexing a single hardware timer—the system timer—in software, having the kernel maintain dispatcher tables at run time. Our SLOTH ON TIME approach proposes to make use of multiple timer cells as available on modern microcontroller platforms to encapsulate dispatcher tables in the timer configuration, yielding low scheduling and dispatching latencies at run time. SLOTH ON TIME instruments available timer cells in different roles to implement time-triggered task activation, deadline monitoring, and time synchronization, amongst others.**

By comparing the SLOTH ON TIME kernel implementation to two commercial kernels, we show that our concept significantly reduces the overhead of time-triggered operating systems. The speed-ups in task dispatching that it achieves range up to a factor of 171 x, and its dispatch latencies go as low as 14 clock cycles. Additionally, we demonstrate that SLOTH ON TIME minimizes jitter and increases schedulability for its real-time applications, and that it avoids situations of priority inversion where traditional kernels fail by design.

I. INTRODUCTION AND MOTIVATION

In operating system engineering, the overhead induced by the kernel is a crucial property since operating system kernels do not provide a business value of their own. This is especially true in *embedded real-time systems*, where superfluous bytes in RAM and ROM as well as unnecessary event latencies can decide whether a kernel is used for the implementation of an embedded device or not. In previous work on the SLOTH approach, we have shown that by using commodity microcontroller hardware in a more sophisticated manner in the kernel, we can achieve lower footprints in RAM and ROM as well as very low system call overheads [7]. To achieve this, the SLOTH kernel maps run-to-completion tasks to interrupt handlers and lets the interrupt hardware schedule them, eliminating the need for a software task scheduler completely. Additionally, we have been able to show that implementing a full thread abstraction with blocking functionality in the SLEEPY SLOTH kernel still yields a significant performance boost over traditional, software-based embedded kernels [8].

However, both the SLOTH and the SLEEPY SLOTH kernels target *event-triggered* real-time systems with event-driven task dispatching. In this paper, we discuss how the SLOTH principle of making better use of hardware facilities in the implementation of embedded kernels can be applied to *time-triggered operating systems*. The resulting SLOTH ON TIME kernel uses the fundamental task dispatching mechanisms as introduced in the SLOTH kernel and makes better use of the central hardware part of any time-triggered kernel: the hardware timer facility.

This paper provides the following contributions:

- We develop a comprehensive design for mapping time-triggered tasks and kernel timing services to hardware timer cells and present SLOTH ON TIME, the first real-time kernel that utilizes *arrays* of hardware timers (see Section IV).
- By evaluating our SLOTH ON TIME kernel and comparing it to traditional time-triggered kernel implementations, we show that our design minimizes the number of interrupts and the kernel-induced overhead at run time, yielding lower latencies, reduced jitter, increased schedulability, and better energy efficiency for real-time applications (see Section V).
- We discuss the implications of the SLOTH ON TIME approach on the design of time-triggered kernels and time-triggered applications (see Section VI).

Before explaining the SLOTH ON TIME design and system, we first describe the time-triggered system model that we use in this paper in Section II and provide necessary background information on the original SLOTH and SLEEPY SLOTH kernels and the microcontroller hardware model that we use as a basis for our description in Section III.

II. SYSTEM MODEL FOR TIME-TRIGGERED TASK ACTIVATION

In this section, we describe the system model that we use for time-triggered task scheduling throughout this paper. Since our model is motivated by the kernel point of view on time-triggered tasks, we take the terminology and semantics from publicly available embedded operating system specifications. Those include the specification for the time-triggered OSEKtime kernel [13] and the AUTOSAR OS specification, which targets an event-triggered kernel that features a schedule table abstraction and timing protection facilities [2]. Both of those standards were developed by leading automotive manufacturers and suppliers based on their experiences with requirements on real-time kernels; this is why they are widely used in the automotive industry. Additionally, since the standards are also implemented by commercially available kernels, we are able to directly compare kernel properties between our SLOTH ON TIME operating system and those kernels by writing benchmarking applications against the respective OS interfaces.

*This work was partly supported by the German Research Foundation (DFG) under grants no. SCHR 603/8-1, SCHR 603/9-1, and SFB/TR 89 (project C1).

Appeared in: Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS '12), pages 237–247, IEEE Computer Society, 2012, ISBN 978-0-7695-4869-2

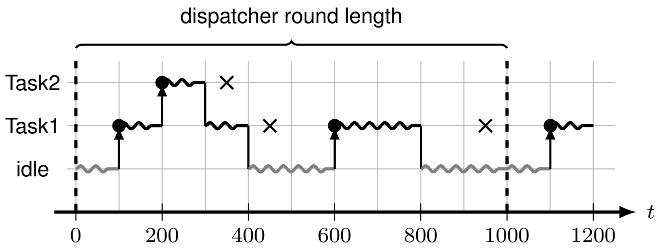


Fig. 1: The model for time-triggered activation and deadlines used in this paper, based on the OSEKtime specification [13]. In this example of a dispatcher table, task activations are depicted by circles, their deadlines by crosses. Later task activations preempt currently running tasks, yielding a stack-based execution pattern.

A. *Time-Triggered OSEKtime*

The central component of OSEKtime is a preplanned dispatcher table of fixed round length that cyclically activates tasks at given offsets (see example in Figure 1). An activated task always preempts the currently running task until its termination, resulting in a stack-based scheduling policy. Additional OSEKtime features include synchronization with a global time source and task deadline monitoring for detecting misbehaving tasks. The latter is done by checking whether a task is neither running nor preempted at given points in time within a dispatcher round (see crosses in Figure 1).

B. *AUTOSAR OS Schedule Tables and Execution Budgets*

In contrast to OSEKtime, AUTOSAR OS approaches time-triggered activations in a more complex way and offers seamless integration with the event-triggered model specified by the same system. Although the basic structure of statically defined dispatcher tables (called *schedule tables*) is the same as in OSEKtime, a time-triggered activation in AUTOSAR OS will not inevitably result in preempting any running task, but instead the activated task will be scheduled according to its own static priority and the priorities of other active tasks. In an AUTOSAR system, a distinction between time-triggered and event-triggered tasks does not exist, and both types of activations share the same priority space. Further extensions consist in the possibility to have multiple schedule tables run simultaneously and to have non-cyclic tables, which are executed only once.

AUTOSAR OS also specifies the ability to restrict the execution budget available to individual tasks. Although this is not related to time-triggered scheduling in particular, we will show that the mechanisms developed in SLOTH ON TIME are suitable for an efficient implementation of this feature as well.

III. BACKGROUND

In order to understand the concept and design principles behind SLOTH ON TIME, we first provide background information about the original event-triggered SLOTH kernel that SLOTH ON TIME is based on, and we describe the requirements on the underlying microcontroller hardware, introducing the abstract terminology used throughout the rest of the paper.

A. SLOTH *Revisited*

The main idea in the original event-triggered SLOTH kernel [7] is to have application tasks run as interrupt handlers internally in the system. Each task is statically mapped to a dedicated interrupt source of the underlying hardware platform at compile time; the IRQ source's priority is configured to the task priority and the corresponding interrupt handler is set to the user-provided task function. SLOTH task system calls are implemented by modifying the state of the hardware IRQ controller: The activation of a task, for instance, is implemented by setting the pending bit of the corresponding IRQ source. This way, the interrupt controller automatically schedules and dispatches SLOTH tasks depending on the current system priority. By these means, SLOTH is able to achieve low overheads in its system calls, both in terms of execution latency and in terms of binary code size and lines of source code.

The original SLOTH kernel can only schedule and dispatch basic run-to-completion tasks as mandated by the OSEK BCC1 conformance class, in which the execution of control flows is strictly nested—which means that a task preempted by higher-priority tasks can only resume after those have run to completion. Thus, all tasks are executed on the same stack—the interrupt stack—which is used both for the execution of the current task and for storing the contexts of preempted tasks. The enhanced SLEEPY SLOTH kernel [8] additionally handles extended tasks that can block in their execution and resume at a later point in time (specified by OSEK's ECC1 conformance class). SLEEPY SLOTH implements these extended tasks by providing a full task context including a stack of its own for each of them; additionally, a short task prologue is executed at the beginning of each task's interrupt handler every time that a task is being dispatched. The prologue is responsible for switching to the corresponding task stack and then either initializes or restores the task context depending on whether the task was unblocked or whether it is being run for the first time.

B. *Microcontroller Hardware Model and Requirements*

With SLOTH ON TIME, we describe a hardware-centric and efficient way to design time-triggered services in a real-time kernel. Our concept makes use of timer arrays with multiple timer cells, which are available on many modern microcontroller platforms such as the Freescale MPC55xx and MPC56xx embedded PowerPC families (64 timer cells) or the Infineon TriCore TC1796 (256 timer cells)—the reference platform for SLOTH ON TIME. As shown in the following section on its design, SLOTH ON TIME requires one timer cell per task activation and deadline in a dispatcher round in its base implementation; the SLOTH approach is to leverage *available* hardware features (in this case, an abundance of timer cells) to improve non-functional properties of the kernel. For platforms with fewer timer cells than activation and deadline points, however, we describe a slightly less efficient alternative design that uses partial multiplexing in Section IV-F.

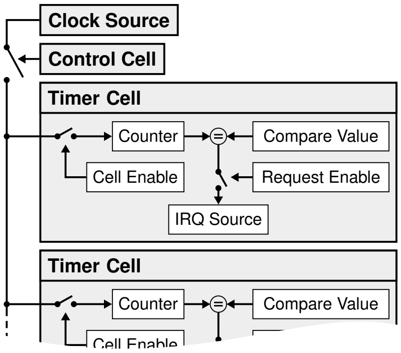


Fig. 2: The abstract model for available timer components on modern microcontroller platforms, introducing the terminology used in this paper.

If the hardware offers *hierarchical* timer cells for controlling lower-order cells, SLOTH ON TIME can optionally make use of that feature, too. Additionally, as in the original SLOTH kernel, we require the hardware platform to offer as many IRQ sources and IRQ priorities as there are tasks in the system; the platforms mentioned before offer plenty of those connected to their timer arrays.

In the rest of the paper, we use the following terminology for the timer array in use (see Figure 2). We call an individual timer of the array a *timer cell*, which features a *counter* register; it is driven by a connected clock signal, which increments or decrements the counter depending on the cell configuration. If the counter value matches the value of the *compare* register or has run down in decrement mode, it triggers the pending bit in the attached *IRQ source*, but only if the *request enable* bit is set. The whole cell can be deactivated by clearing the *cell enable* bit, which stops the internal counter.

IV. SLOTH ON TIME

The main design idea in SLOTH ON TIME is to map the application timing requirements to hardware timer arrays with multiple timer cells—pursuing efficient execution at run time. SLOTH ON TIME tailors those timer cells for different purposes within a time-triggered operating system, introducing different *roles* of timer cells (see overview in Figure 3)—including task activation cells, table control cells, deadline monitoring cells, execution budget cells, and time synchronization cells, as described in this section. Additionally, we show how time-triggered and event-triggered systems can be integrated in SLOTH ON TIME, and we highlight the design of multiplexed timer cells for hardware platforms with fewer available timer cells.

A. *Time-Triggered* SLOTH

Traditional time-triggered kernels implement task activations by instrumenting a single hardware timer, which then serves as the system timer. The system timer is programmed and reprogrammed by the kernel at run time whenever a scheduling decision has to be performed; the next system timer expiry point is usually looked up in a static table representing the application schedule.

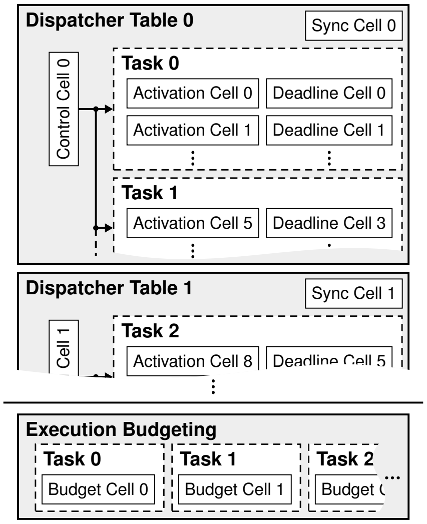


Fig. 3: The different roles that SLOTH ON TIME uses available hardware timer cells for. Some roles are only used in OSEKtime-like systems, others are only used in AUTOSAR-OS-like systems.

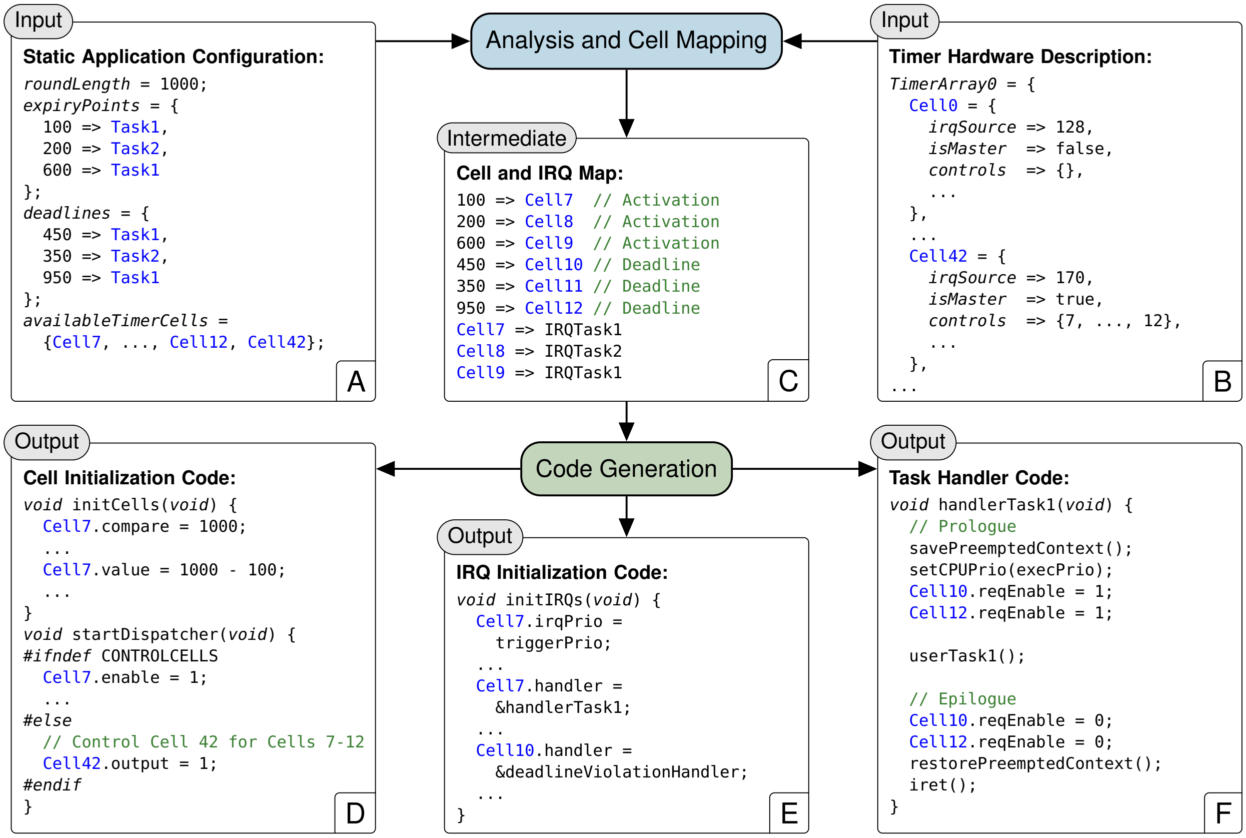
SLOTH ON TIME tries to avoid dynamic decisions—and, therefore, run time overhead—as far as possible by instrumenting multiple timer cells. This way, programming the timers can mostly be limited to the initialization phase; during the system's productive execution phase, the overhead for time-triggered task execution is kept to a minimum.

1) Static Design: SLOTH ON TIME not only comprises the actual time-triggered kernel, but also consists of a static analysis and generation component (see Figure 4). As its input, the analysis tool takes the task activation schedule as provided by the application programmer (see Artifact A in Figure 4) and the platform description of the timer hardware (Artifact B), and, in an intermediate step, outputs a mapping of the included expiry points to individual *activation cells* (Artifact C), which are subject to platform-specific mapping constraints due to the way the individual timer cells are interconnected[1]. The timer cells for SLOTH ON TIME to use are taken from a pool of cells marked as available by the application; this information is also provided by the application configuration. If the number of available timer cells is not sufficient, SLOTH ON TIME uses partial multiplexing, which we describe in Section IV-F.

In the next step, the calculated mapping is used to generate initialization code for the involved timer cells (Artifact D). The compare values for all cells are set to the length of the dispatcher round so that the cell generates an interrupt and resets the counter to zero after one full round has been completed. The initial counter value of a cell is set to the round length minus the expiry point offset in the dispatcher round. This way, once the cell is enabled, it generates its first interrupt after its offset is reached, and then each time a full dispatcher round has elapsed.

Additionally, code for starting the dispatcher is generated

[1] The mapping algorithm is work in progress; currently, the timer cells still have to be assigned manually to respect platform restrictions.



Static Application Configuration:

```
roundLength = 1000;
expiryPoints = {
  100 => Task1,
  200 => Task2,
  600 => Task1
};
deadlines = {
  450 => Task1,
  350 => Task2,
  950 => Task1
};
availableTimerCells =
  {Cell7, ..., Cell12, Cell42};
```

Cell and IRQ Map:

```
100 => Cell7  // Activation
200 => Cell8  // Activation
600 => Cell9  // Activation
450 => Cell10 // Deadline
350 => Cell11 // Deadline
950 => Cell12 // Deadline
Cell7 => IRQTask1
Cell8 => IRQTask2
Cell9 => IRQTask1
```

Timer Hardware Description:

```
TimerArray0 = {
  Cell0 = {
    irqSource => 128,
    isMaster  => false,
    controls  => {},
    ...
  },
  ...
  Cell42 = {
    irqSource => 170,
    isMaster  => true,
    controls  => {7, ..., 12},
    ...
  },
...
```

Cell Initialization Code:

```
void initCells(void) {
  Cell7.compare = 1000;
  ...
  Cell7.value = 1000 - 100;
  ...
}
void startDispatcher(void) {
#ifndef CONTROLCELLS
  Cell7.enable = 1;
  ...
#else
  // Control Cell 42 for Cells 7-12
  Cell42.output = 1;
#endif
}
```

IRQ Initialization Code:

```
void initIRQs(void) {
  Cell7.irqPrio =
    triggerPrio;
  ...
  Cell7.handler =
    &handlerTask1;
  ...
  Cell10.handler =
    &deadlineViolationHandler;
  ...
}
```

Task Handler Code:

```
void handlerTask1(void) {
  // Prologue
  savePreemptedContext();
  setCPUPrio(execPrio);
  Cell10.reqEnable = 1;
  Cell12.reqEnable = 1;

  userTask1();

  // Epilogue
  Cell10.reqEnable = 0;
  Cell12.reqEnable = 0;
  restorePreemptedContext();
  iret();
}
```

Fig. 4: Static analysis and generation in SLOTH ON TIME, producing the mapping of expiry points and deadlines to timer cells and IRQ sources, the corresponding timer and interrupt initialization code, and task handler code with prologues and epilogues. The example values correspond to the sample application schedule depicted in Figure 1.

that enables all involved activation cells consecutively (Artifact D). If the underlying hardware platform features hierarchically connected cascading timer cells, then a higher-order cell is used as a so-called *control cell* to switch all connected lower-order timer cells on or off simultaneously. If such a control cell is available, enabling it will enable all connected activation cells of a dispatcher round (see also the timer model in Figure 2). This mechanism enables completely accurate and atomic starting and stopping of dispatcher rounds by setting a single bit in the respective control cell (see also evaluation in Section V-B3).

Furthermore, since tasks are bound to interrupt handlers for automatic execution by the hardware, the interrupt system needs to be configured appropriately (Artifact E). This entails setting the IRQ priorities of the involved cells to the system trigger priority (see explanation in Section IV-A2) and registering the corresponding interrupt handler for the cell's task.

2) Run Time Behavior: At run time, no expiry points and dispatcher tables need to be managed by SLOTH ON TIME, since all required information is encapsulated in the timer cells that are preconfigured during the system initialization. Once the dispatcher is started by enabling the control cell or the individual timer cells, the interrupts corresponding to task dispatches will automatically be triggered at the specified expiry points by the timer system. The hardware interrupt system will then interrupt the current execution, which will either be the system's idle loop or a task about to be preempted, and dispatch the associated interrupt handler, which in SLOTH ON TIME basically corresponds to the task function as specified by the user, surrounded by a small wrapper.

The only functions that are not performed automatically by the hardware in SLOTH ON TIME are saving the full preempted task context when a new time-triggered task is dispatched and lowering the CPU priority from the interrupt *trigger priority* to the *execution priority* (see generated code in Artifact F in Figure 4). This lowering is needed to achieve the stack-based preemption behavior of tasks in the system, such as mandated by the OSEKtime specification [13], for instance (see also Figure 1). By configuring all interrupts to be triggered at a high trigger priority and lowering interrupt handler execution to a lower execution priority, every task can be preempted by any task that is triggered at a later point in time, yielding the desired stack-based behavior. Thus, a task activation with a later expiry point implicitly has a higher priority than any preceding task activation.

B. Deadline Monitoring

Deadlines to be monitored for violation are implemented in SLOTH ON TIME much in the same way that task activation expiry points are (see Figure 4). Every deadline specified in the application configuration is assigned to a *deadline* *cell* (see also Figure 3), which is a timer cell configured to be triggered after the deadline offset, and then after one dispatcher round has elapsed (Artifacts C and D in Figure 4). The interrupt handler that is registered for such deadline cells is an exception handler for deadline violations that calls an application-provided handler to take action (Artifact E).

In contrast to traditional implementations, SLOTH ON TIME disables the interrupt requests for a deadline cell once the corresponding task has run to completion, and re-enables them once the task has started to run. This way, deadlines that are *not* violated do not lead to unnecessary IRQs, which would disturb the execution of other real-time tasks in the system (see also evaluation in Section V-B). In the system, this behavior is implemented by enhancing the generated prologues and epilogues of the time-triggered tasks; here, the request enable bits of the associated deadline cells are enabled and disabled, respectively (see generated Artifact F in Figure 4).

C. Combination of Time-Triggered and Event-Triggered Systems

In the OSEK and AUTOSAR automotive standards that we use as a basis for our investigations, there are two approaches to combine event-triggered elements with a time-triggered system as implemented by SLOTH ON TIME. The OSEK specifications describe the possibility of a mixed-mode system running an event-triggered OSEK system during the idle time of the time-triggered OSEKtime system running on top, whereas the AUTOSAR OS standard specifies a system with event-triggered tasks that can optionally be activated at certain points in time using the schedule table abstraction. In this section, we show how we implement both approaches in the SLOTH ON TIME system.

1) Mixed-Mode System: Since the original SLOTH kernel implements the event-triggered OSEK OS interface [14], whereas SLOTH ON TIME implements the time-triggered OSEKtime interface, we combine both systems by separating their priority spaces by means of configuration. By assigning all event-triggered tasks priorities that are lower than the time-triggered execution and trigger priorities, the event-triggered system is only executed when there are no time-triggered tasks running; it can be preempted by a time-triggered interrupt at any time. Additionally, we make sure that the event-triggered SLOTH kernel synchronization priority, which is used to synchronize access to kernel state against asynchronous task activations, is set to the highest priority of all event-triggered tasks but lower than the time-triggered priorities. Thus, the integration of both kinds of systems can easily be achieved without jeopardizing the timely execution of the time-triggered tasks.

2) Event-Triggered System with Time-Triggered Elements: In contrast to the mixed-mode approach, AUTOSAR OS defines an event-triggered operating system with static task priorities; its schedule table abstraction only provides means to *activate* the event-triggered tasks at certain points in time, which does not necessarily lead to *dispatching* them as in purely time-triggered systems (in case a higher-priority task is currently running). AUTOSAR tasks have application-configured and potentially distinct priorities; at run time, they can raise their execution priority by acquiring resources for synchronization or even block while waiting for an event.

The schedule table abstraction therefore does not strictly follow the time-triggered paradigm, but it is implemented in SLOTH ON TIME in a way that is very similar to the time-triggered dispatcher table. Instead of configuring the priorities of the IRQ sources attached to the timer system to the system trigger priority, however (see Artifact E in Figure 4), they are set to the priority of the task they activate. This way, the time-dependent activation of tasks is seamlessly integrated into the execution of the rest of the SLOTH system, since after the IRQ pending bit has been set, it does not matter whether this was due to a timer expiry or by synchronously issuing a task activation system call.

To fully implement AUTOSAR OS schedule tables, the SLOTH ON TIME timer facility is enhanced in three ways. First, AUTOSAR allows multiple schedule tables to be executed simultaneously and starting and stopping them at any time. Thus, SLOTH ON TIME introduces the corresponding system calls to enable and disable the control cell for the corresponding schedule table (see also Section IV-A1). Second, AUTOSAR defines non-repeating schedule tables, which encapsulate expiry points for a single dispatcher round, executed only once when that schedule table is started. SLOTH ON TIME implements this kind of schedule table by preconfiguring the corresponding timer cells to one-shot mode; this way, they do not need to be manually deactivated at the end of the schedule table. Third, schedule tables can be started with a global time offset specified dynamically at run time. In that case, SLOTH ON TIME reconfigures the statically configured timer cells for that schedule table to include the run time parameter in its offset calculation before starting it by enabling its control cell.

D. Execution Time Protection

In contrast to deadline monitoring, which is used in time-triggered systems like OSEKtime (see Section IV-B), AUTOSAR prescribes timing protection facilities using execution time budgeting for fault isolation. Each task is assigned a maximum execution budget per activation, which is decremented while that task is running, yielding an exception when the budget is exhausted. In its design, SLOTH ON TIME employs the same mechanisms used for expiry points and deadlines to implement those task budgets. It assigns one *budget cell* to each task to be monitored (see also Figure 3), initializes its counter with the execution time budget provided by the application configuration, and configures it to run down once started. The associated IRQ source is configured to execute a user-defined protection hook as an exception handler in case the budget cell timer should expire.

Furthermore, the dispatch points in the system are instrumented to pause, resume, and reset the budget timers appropriately. First, this entails enhancing the task prologues, which pause the budget timer of the preempted task and start the budget timer of the task that is about to run. Second, the task epilogues executed after task termination are added instructions to reset the budget to the initial value configured for this task, as suggested by the AUTOSAR specification, and to resume the budget timer of the previously preempted task about to be restored.

This design allows for light-weight monitoring of task execution budgets at run time without the need to maintain and calculate using software counters; this information is implicitly encapsulated and automatically updated by the timer hardware in the counter registers.

E. Synchronization with a Global Time Base

Real-time systems are rarely deployed stand-alone but often act together as a distributed system. Therefore, in time-triggered systems, synchronization of the system nodes with a global time base needs to be maintained. This feature has to be supported by the time-triggered kernel by adjusting the execution of dispatcher rounds depending on the detected clock drift.

If support for synchronization is enabled, SLOTH ON TIME allocates a dedicated *sync cell* (see also Figure 3) and configures its offset to the point after the last deadline in a dispatcher round (e.g., 950 in the example schedule in Figure 1). This point has to be specified in the configuration by the application programmer and can be used to apply a limited amount of negative drift depending on the remaining length of the dispatcher round (50 in the example). Positive drifts are, of course, not restricted in this way.

The interrupt handler attached to the sync cell then checks at run time whether a drift occurred. If so, it simply modifies the counter values of all activation, deadline, and sync cells that belong to the dispatcher table, corrected by the drift value (see Figure 5). Since the cell counters are modified *sequentially*, the last counter is changed later than the first counter of a table. However, since the read–modify–write cycle of the counter registers always takes the same amount of time and the modification value is the same for all counters, in effect it does not matter when exactly the counter registers are modified. In case the synchronization handler is not able to reprogram all affected cell counters by the next task activation point in the schedule, it resumes execution at the next cell to be reprogrammed once it becomes active again in the next round.

F. Timer Cell Multiplexing

If the hardware platform does not have enough timer cells to allocate one cell per time-triggered event, SLOTH ON TIME also allows to partially fall back to multiplexing—allocating only one timer cell for each role (activation and deadline) *per task* and partly reconfiguring it at run time.

For multiplexed deadline monitoring, if the current deadline has not been violated, the task epilogue reconfigures the expiration point of the deadline cell to the *next* deadline instead of disabling it. The deltas between the deadline points are retrieved from a small offset array held in ROM.

For multiplexed activations of the same task, another offset array contains the deltas between the activation points of that task. The enhanced task prologue then reconfigures the compare value of the activation cell to the next activation point every time that task is dispatched.

V. EVALUATION

In order to assess the effects of our proposed timer-centric architecture on the non-functional properties of a time-triggered kernel, we have implemented SLOTH ON TIME with all the kernel features described in Section IV on the Infineon TriCore TC1796 microcontroller, which is widely used for control units in the automotive domain. Since from a *functional* point of view, SLOTH ON TIME implements the OSEKtime and AUTOSAR OS standards, we can directly compare our kernel to a commercial OSEKtime system and a commercial AUTOSAR OS system, both of which are available for the TC1796. This way, we can take benchmarking applications written against the respective OS interface and run them unaltered on both SLOTH ON TIME and the commercial kernels.

A. The Infineon TriCore TC1796 Microcontroller

The TC1796, which serves as a reference platform for SLOTH ON TIME, is a 32-bit microcontroller that features a RISC load/store architecture, a Harvard memory model, and 256 interrupt priority levels. The TC1796's timer system is called its general-purpose timer array and includes 256 timer cells, which can be configured to form a cascading hierarchy if needed. The cells can be routed to 92 different interrupt sources, whose requests can be configured in their priorities and the interrupt handlers that they trigger. The timer and interrupt configuration is performed by manipulating memory-mapped registers.

For the evaluation, we clocked the chip at 50 MHz (corresponding to a cycle time of 20 ns); however, we state the results of our latency and performance measurements in numbers of clock cycles to be frequency-independent. We performed all measurements from internal no-wait-state RAM (both for code and data), so caching effects did not apply. The actual measurements were carried out using a hardware trace unit by Lauterbach. All of the quantitative evaluation results were obtained by measuring the number of cycles spent between two given instructions (e.g., from the first instruction of the interrupt handler to the first user code instruction of the associated task) repeatedly for at least 1,000 times and then averaging these samples. In some situations, the distribution of the samples exhibits two distinct peaks of similar height, which are located exactly 4 cycles apart and presumably related to unstableness in measuring conditional jumps. Aside from this effect, the deviations from the average have shown to be negligible in all measurements of SLOTH ON TIME.

B. Qualitative Evaluation

While running our test applications on SLOTH ON TIME and both commercial kernels, we could observe several effects of our design on the *qualitative* execution properties by examining the execution traces from the hardware tracing unit.

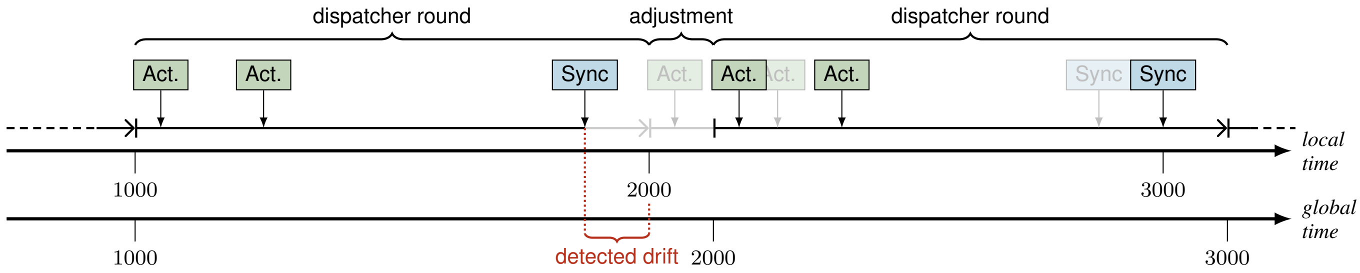
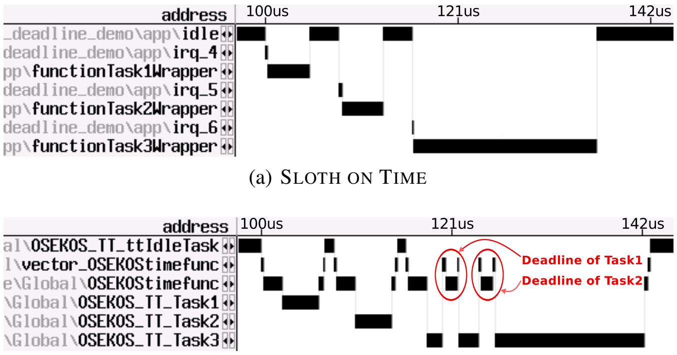


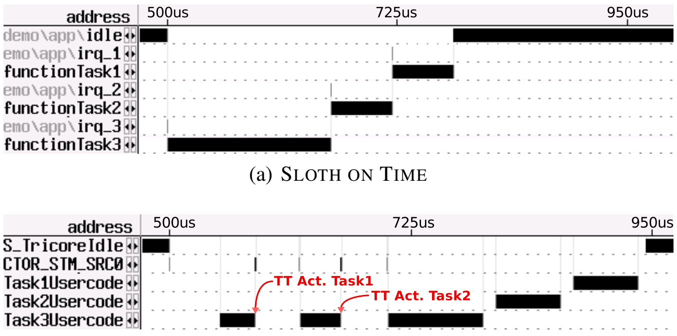
Fig. 5: Synchronization to a global time is implemented in SLOTH ON TIME by adjusting the current counter values of the cells involved in a dispatcher round one after the other, requiring only one read–modify–write cycle per cell.



(a) SLOTH ON TIME

(b) Commercial OSEKtime system

Fig. 6: Comparison of execution traces of an OSEKtime application with two deadlines in **(a)** SLOTH ON TIME and **(b)** a commercial OSEKtime system. Non-violated deadlines of Task1 and Task2 interrupt the execution of Task3 in the commercial system, but not in SLOTH ON TIME.



(a) SLOTH ON TIME

(b) Commercial AUTOSAR OS system

Fig. 7: Execution trace revealing rate-monotonic priority inversion in a commercial AUTOSAR OS system occurring on time-triggered activation of lower-priority tasks Task1 and Task2. The trace of the same dispatcher table in SLOTH ON TIME shows no interruption of Task3.

1) Avoiding Unnecessary IRQs: For one, both commercial kernels exhibit unnecessary interrupts with unnecessary interrupt handlers executing, possibly interrupting and disturbing application control flows. All of those interrupts are not needed by the application semantics, and the SLOTH ON TIME design can avoid all of them in its running implementation.

Figure 6 shows the execution traces for an application with three tasks and three deadlines per dispatcher round, running on SLOTH ON TIME and the commercial OSEKtime implementation. The commercial OSEKtime issues an interrupt for every deadline to be monitored, since it then checks if the corresponding task is still running (see interruptions of Task3 in Figure 6b). These interrupts take 95 clock cycles each, possibly interrupting application tasks for the violation check; this number multiplies by the number of deadlines stated in the application configuration to yield the overhead per dispatcher round. Those unnecessary IRQs in the commercial OSEKtime kernel are especially problematic since they also occur for *non-violated* deadlines—which are, after all, the normal case and not the exception.

SLOTH ON TIME can avoid those unnecessary IRQs completely (see continuous execution of Task3 in Figure 6a). It uses dedicated deadline timer cells, which are turned off using a single memory-mapped store instruction when the task has run to completion in its task epilogue (see Section IV-B); this takes 10 clock cycles per deadline (see also Section V-C1). SLOTH ON TIME effectively trades the overhead introduced by interrupt handlers in the schedule as in traditional systems for overhead introduced when a task terminates; this trade-off decision has the advantage that it does not interrupt the execution of other tasks, facilitating real-time analyses on the schedule. Note that the commercial kernel *could* be implemented in a way similar to SLOTH ON TIME to avoid additional interrupts; however, the overhead for its software logic would probably exceed the overhead that SLOTH ON TIME introduces (compare 10 cycles per deadline in SLOTH ON TIME to 95 cycles per deadline check interrupt in the commercial kernel).

2) Avoiding Priority Inversion: Second, in the commercial AUTOSAR OS system, we could observe a certain kind of priority inversion, which occurs when a low-priority task is activated by the timer while a high-priority task is running (see gaps in the execution of Task3 in Figure 7b). The high-priority task is interrupted by the timer interrupt, whose handler then checks which task is to be activated, and inserts this low-priority task into the ready queue. Thus, code is executed on behalf of a low-priority task while a high-priority task is running or ready to run; Leyva del Foyo et al. coined the term *rate-monotonic priority inversion* for that phenomenon [5]. The priority inversion interruptions exhibited by the commercial AUTOSAR OS kernel are really heavy-weight: The corresponding handlers execute for 2,075 clock cycles each. This can lead to serious deviations between the statically calculated WCET for a task and its actual run time, which is potentially prolonged by several of those low-priority interrupts.

In Sloth on Time, those interrupts do not occur at all since the corresponding timer cell activates the task not by executing code on the main CPU but by setting the pending bit of the associated IRQ source, configured with the task priority. Since the high-priority task that is currently running runs at high *interrupt* priority, the activation does not lead to an interruption until the high-priority task blocks or terminates (see continuous execution of `Task3` in Figure 7a).

Thus, the Sloth approach of running tasks as interrupt service routines in combination with the Sloth on Time concept of using dedicated timer cells for time-dependent activations minimizes the number of interrupts in the system, facilitating real-time analyses. Interrupts only occur if an action needs to be taken by the kernel on behalf of the application. Traditional kernels with a single system timer cannot avoid the described kind of priority inversion since they *have to* put the activated task into the ready queue (even if it has a low priority) and reconfigure the timer to the next expiry point; this is an inherent design issue that Sloth on Time overcomes. In Sloth on Time, those problems are avoided by design; the timer hardware runs concurrently to the main CPU and activates a task by setting the corresponding IRQ pending bit without having to interrupt the CPU.

Furthermore, in traditional systems, the timer interrupt handler needs to be synchronized with the kernel since it accesses the kernel ready queue; this leads to jitter in the task dispatch times since the timer interrupt handler might additionally be delayed by a task executing a system call. The Sloth on Time approach eliminates that jitter source since explicit kernel synchronization is not necessary—the ready queue is implemented implicitly in the hardware IRQ state and does not need to be synchronized in software.

3) Preciseness: We also investigated the preciseness of task dispatch times as specified by the static schedule and the drift between several consecutive dispatcher rounds. Both in our Sloth on Time kernel and the two commercial kernels, we could not observe any drift since all of them rely on hardware timers and static execution overhead in their timer interrupt handlers.

Additionally, we could show that by using control cells as proposed in Section IV-A1, all activation cells of the dispatcher round or schedule table can be started simultaneously. This way, the additional overhead introduced by starting all timer cells in sequential machine instructions does not need to be respected in the offset calculation for the individual activation cells.

C. Quantitative Evaluation

Since non-functional properties such as kernel execution times, latencies, and memory footprint are crucial to real-time kernels, we also took comprehensive measurements to be able to state the *quantitative* effects of our Sloth on Time design on these important properties.

```
<handlerTask2>:
    mov %d0,2944
    mtcr $psw,%d0  // enable global address registers
    isync  // synchronize previous instruction
    st.a [+%a9]-4,%a8  // save preempted task ID on stack
    mov.a %a8,2  // set new task ID
    st.t <GPTA0_LTCCTR11>,3,1  // enable deadline cell
    bisr 2  // set exec prio 2, save context, enable IRQs
    call userTask2  // enter user code
    disable  // suspend IRQs for synchronization
    st.t <GPTA0_LTCCTR11>,3,0  // disable deadline cell
    rslcx  // restore context
    ld.a %a8,[%a9+]  // restore preempted task ID from stack
    rfe  // return from interrupt handler (re-enables IRQs)
```

Fig. 8: Compiled time-triggered task interrupt handler in Sloth on Time on the TC1796 for `Task2` with one deadline.

1) OSEKtime Evaluation: Since Sloth on Time encapsulates the expiry points of a dispatcher round in its timer cell configuration and traditional implementations need a look-up table to multiplex the system timer at run time, we expect differences in the overhead of time-triggered task dispatch and termination in Sloth on Time and the commercial OSEKtime kernel. The top of Table I shows our measurements, which confirm that our approach yields very low latencies at run time compared to traditional implementations like the commercial OSEKtime, yielding speed-up numbers of 8.6 and 2.7 for task dispatch and termination, respectively. This reduced overhead in Sloth on Time leads to additional slack time in a dispatcher round, which can be used to include additional application functionality (compare the idle times in Figure 6).

Note that the number of 14 clock cycles for the time-triggered task dispatch includes *all* costs between the preemption of the running task or the idle loop to the first user instruction in the dispatched task (see assembly instructions in Figure 8). Thus, the number reflects the complete Sloth on Time prologue wrapper, which itself entails saving the context of the preempted task on the stack; since the system is strictly stack-based, all tasks run on the same stack, so the stack pointer does not have to be switched. The overhead numbers of 60–74 cycles for a task activation in an event-triggered Sloth system (presented in [7]) exactly correspond to the 14 cycles for the context save prologue plus the activation system call issued by the calling task. Since Sloth on Time activations are time-triggered, the overhead for the system call is not applicable, yielding the very low total number of 14 cycles. Enabling activation cell multiplexing (see Section IV-F) adds 18 cycles to the activation overhead for any task that benefits from this feature due to multiple activations in a single dispatcher round. Tasks activated only once per dispatcher round are not affected by this and retain the usual overhead.

If deadline monitoring is used in the application, both overhead numbers in Sloth on Time increase by about 10 cycles for every deadline associated with a given task (see bottom of Table I). This stems from the fact that Sloth on Time activates and deactivates the deadline cells for that task, which compiles to a single memory-mapped store instruction per cell (see also Figure 8); due to memory bus synchronization,

	Sloth on Time	OSEKtime	Speed-Up
Time-triggered (TT) dispatch	14	120	8.6
Terminate	14	38	2.7
TT dispatch w/ 1 deadline	26	120	4.6
TT dispatch w/ 2 deadlines	34	120	3.5
Terminate w/ 1 deadline	24	38	1.6
Terminate w/ 2 deadlines	34	38	1.1

TABLE I: Run time overhead of time-triggered task dispatch and termination with deadline monitoring enabled, comparing Sloth on Time with a commercial OSEKtime implementation (in number of clock cycles).

this instruction needs 10 clock cycles. If multiplexing of deadline cells as described in Section IV-F is used instead, no additional overhead is incurred during dispatch, but an increase of 18 cycles is measured for the task termination, representing the cost of maintaining the state of the offset array and reconfiguring the deadline cell. However, this increased overhead does not increase further with additional deadlines to be monitored for the same task; starting with two deadlines per task, multiplexing yields a performance advantage. This advantage is traded for a slight increase in memory footprint for the offset array and the associated index variable.

The dispatch overhead in the commercial OSEKtime system remains the same independent of the number of deadlines; however, as discussed in Section V-B1, it introduces additional IRQs to a dispatcher round, executing for 95 cycles per deadline.

In the mixed-system case, when an event-triggered OSEK system runs in the idle time of the time-triggered OSEKtime system, the commercial implementation exhibits the same latency as in the time-triggered-only case (120 cycles). Sloth's latency depends on the conformance class of the underlying event-triggered system: If the application only includes basic run-to-completion tasks (class BCC1), then its latencies remain the same as in the time-triggered-only case (14 cycles for task activation and 14 cycles for task termination). If it also includes extended tasks that can block at run time (class ECC1), then the latencies rise the same way as described in Sleepy Sloth [8]. Since extended tasks have to run on stacks of their own because they potentially block, an additional stack switch is needed when a time-triggered task preempts an extended task of the underlying event-triggered system. With the raised latency being 28 cycles, even in that case Sloth is still considerably faster than the commercial kernel (speed-up of 4.3).

2) AUTOSAR OS Evaluation: We also evaluated the latencies of event-triggered systems featuring time-triggered task activation; the measurements use the AUTOSAR interface of both Sloth on Time and the commercial AUTOSAR OS kernel.

First, we show the measurement numbers for the two relevant system calls: `StartScheduleTableRel()`, which starts a schedule table dispatcher round relative to the current time, and `StopScheduleTable()`, which stops the execution of further expiry points of a given table. The results are shown in Table II. Due to the preconfiguration of corresponding timer cells during the initialization, the system call latencies in Sloth on Time compared to the commercial implementation reach speed-up numbers of 10.2 and 37.6 for starting and stopping a schedule table at run time, respectively.

	Sloth on Time	AUTOSAR OS	Speed-Up
StartScheduleTableRel()	108	1,104	10.2
StopScheduleTable()	20	752	37.6

TABLE II: Overhead of time-triggered system services in event-triggered AUTOSAR OS systems, comparing Sloth on Time with a commercial implementation of the AUTOSAR standard (in number of clock cycles).

Second, we measured the latencies for activating a task at certain points in time as specified by an AUTOSAR schedule table; the results are shown in Table III. In Sloth on Time, activating a task using preconfigured timer cells and user task functions connected to an interrupt source totals to between 14 and 77 clock cycles, depending on the types of the involved tasks. If full stack switches are needed (since extended tasks are able to block), the latency is higher compared to when only basic tasks are involved, which run to completion and can therefore run on the same stack. Again, those numbers reflect the execution of the whole wrapper prologue, measuring the time from the timer interrupt to the first user task instruction; thus, they include the whole context switch, including the stack switch if an extended task is involved. The commercial AUTOSAR system needs 2,344 to 2,400 clock cycles to achieve this, resulting in speed-up numbers for Sloth on Time between 31.2 and 171.4. Dispatches resulting from task termination amount to 14 to 88 cycles, again depending on the involved task types. The commercial AUTOSAR implementation takes 382 to 532 clock cycles for those test cases, yielding speed-up numbers of 6.0 to 38.0 for Sloth on Time.

3) Memory Footprint: Since Sloth on Time is highly configurable, its memory footprint depends on factors such as the selected set of features and the configured number of tasks, activation points, deadlines, et cetera. As a ballpark figure for the memory usage of Sloth on Time, we created a minimal time-triggered application with one task and one deadline but no actual user code and measured a total of 8 bytes of RAM usage and 1,480 bytes of ROM (of which 624 bytes are claimed by the platform start-up code). In comparison, equivalent setups in the commercial implementations allocate 52 bytes of RAM and 2,600 bytes of ROM (OSEKtime) and 900 bytes of RAM and 34,514 bytes of ROM (AUTOSAR OS).

D. Overall Benefit for the Application

By using Sloth on Time instead of one of the commercial kernels, a time-triggered application will be executed more predictably, since unnecessary interrupts and priority inversion can be avoided in Sloth on Time. The gained slack time per dispatcher round depends on the degree that the application makes use of the operating system—on its number of activation points, deadline monitoring points, and executed system calls.

		SLOTH ON TIME	AUTOSAR OS	Speed-Up
Time-triggered task activation with dispatch	idle loop to basic task	14	2,344	167.4
Time-triggered task activation with dispatch	basic task to basic task	14	2,400	171.4
Time-triggered task activation with dispatch	extended task to extended task	77	2,400	31.2
Task termination with dispatch	basic task to idle loop	14	382	27.3
Task termination with dispatch	basic task to basic task	14	532	38.0
Task termination with dispatch	extended task to extended task	88	532	6.0

TABLE III: Latencies of time-triggered task activation and dispatching in event-triggered AUTOSAR OS systems, comparing SLOTH ON TIME with a commercial AUTOSAR OS implementation (in number of clock cycles).

However, since SLOTH ON TIME has a lower overhead in *all* microbenchmarks, the application will *always* experience a benefit.

For OSEKtime systems, the number of additionally available clock cycles per dispatcher round is 130 per task activation point without a deadline, plus 114 per task activation with one or more associated deadlines, plus 95 per monitored deadline. For AUTOSAR OS systems, the total benefit in clock cycles is at least 2,767 per task activation, plus 996 per schedule table start system call, plus 732 per schedule table stop system call, plus the SLOTH and SLEEPY SLOTH benefit for the regular event-triggered operation (see [7] and [8]). The gained slack time allows the application to include additional functionality.

VI. DISCUSSION

In this section, we discuss the general applicability of our SLOTH ON TIME approach and the impact it has on applications running on top of the kernel.

A. Applicability

The applicability of the proposed SLOTH ON TIME design depends on the timer architecture of the underlying hardware platform. Due to the instrumentation of timer cells in different role types, an appropriate number of timer cells that are not otherwise used by the application needs to be available for the kernel to use, specified in the configuration (see Artifact A in Figure 4). Many modern microcontrollers, especially those that are used in control systems, offer plenty of configurable timers—like the Freescale MPC55xx and MPC56xx embedded PowerPC families and the Infineon TriCore TC1796, the reference platform for SLOTH ON TIME.

Since timer cells and connected interrupt sources are usually not freely configurable, the mapping of scheduling points to timer cells can be challenging for the developer of the hardware model (see Artifact B in Figure 4). On the TC1796, for instance, restrictions apply that make it necessary to use two adjacent cells per activation; additionally, four cells are connected to a single interrupt source. Thus, on that platform, a second activation of the same task in a dispatcher round can be accommodated with minimal additional hardware resources. More than two activations will be subject to a trade-off decision, probably favoring a multiplexing implementation if cells become scarce (see Section IV-F).

In theory, SLOTH ON TIME competes with the application for the timer cells, which may limit their availability for the kernel. In practice, however, timer arrays are only used for control algorithms that bear latency and activation rate requirements so tight that traditional RTOS cannot fulfill them; by using the timer hardware directly, the application also becomes less portable. SLOTH ON TIME, on the other hand, offers very low latencies, but hides its implementation beneath a platform-independent OSEKtime API and configuration, shielding the developer from porting the application from one hardware timer API to another. We are convinced that, given an RTOS that offers hardware-comparable latencies for task activations such as SLOTH ON TIME, application developers would happily migrate from using timer arrays directly to using time-triggered task abstractions.

By using platform-specific timer hardware extensively, the SLOTH ON TIME kernel itself is less portable than a traditional time-triggered kernel with software multiplexing. Our reference implementation runs on the Infineon TriCore TC1796; from our experiences in porting the event-triggered SLOTH and SLEEPY SLOTH kernels, however, we can state that the additional porting effort can be contained by using a clear internal abstraction boundary.

Since multi-core processors are used mainly for consolidation purposes in the automotive market, the AUTOSAR standard recently introduced hard task partitioning for multi-core applications. Schedule tables, which encapsulate task activations, are therefore also bound to specific cores; thus, the SLOTH ON TIME approach can be applied to each schedule table separately by statically initializing the task interrupt sources to route interrupt requests to the configured core.

B. Impact on Schedulability, Predictability, and Efficiency

The benefits of improved latency and system call performance introduced by the SLOTH ON TIME concept have a positive impact on the schedulability of tasks in the application. As directly perceivable by comparing the idle times in the execution traces in SLOTH ON TIME and the commercial kernels (see Figures 6 and 7), the increased slack time can be used to include additional application functionality by either extending existing tasks or by introducing additional time-triggered tasks. In application scenarios with highly loaded schedules, an implementation using traditional kernels might not even be possible, whereas the reduced overhead in SLOTH ON TIME might make it feasible.

Schedules with activation points that are very close together in time will cause problems in traditional kernels, since the software scheduler will delay the second activation through its overhead for the first task activation. By activating and dispatching in hardware, the minimal overhead caused by SLOTH ON TIME can accommodate close activation points—as they occur when scheduling tasks with high activation frequencies, for instance. Taking into account the dispatching overheads caused by the kernels, SLOTH ON TIME supports a maximum dispatch frequency of 1.7 MHz of a single minimal task, whereas the commercial AUTOSAR kernel only supports 17 kHz, for example.

The fact that SLOTH ON TIME has very few data structures in the software part of the kernel not only reduces its footprint in RAM, but also in the platform's data cache. This reduced kernel-induced cache load increases application performance by letting it execute out of the cache more often, and, more importantly, reduces caching effects *caused* by the kernel—thereby increasing the predictability of the application. This facilitates the development of the real-time schedule with tightened WCETs.

Additionally, the reduced kernel-induced load in SLOTH ON TIME systems positively influences the energy consumption of embedded devices. Since most of those systems spend the majority of their time in sleep mode, the lower overhead introduced by the operating system has a significant impact on energy efficiency—and, therefore, battery life, which is crucial in *mobile* embedded systems.

VII. RELATED WORK

The idea of hardware-based and hardware-assisted real-time scheduling is not new. Existing approaches, like Atalanta [17], cs2 [11], HW-RTOS [3], FASTCHART [9], and Silicon TRON [12], but also the work presented in [16], [4], however, focus on *event-triggered* real-time systems and employ *customized* hardware synthesized on an FPGA or a similar component. SLOTH ON TIME, in contrast, employs *commodity* hardware to implement scheduling with focus on *time-triggered* systems; the seamless integration of mixed-mode systems is possible by employing the techniques presented in our previous papers [7], [8].

The fact that only little work so far has focused on hardware assistance for time-triggered schedulers might be rooted in the generally simple and straight-forward software implementations of such schedulers [10]. On the other hand, aiming for efficient software-based timer abstractions has a long tradition in the operating systems community, including concepts such as timer wheels [42], soft timers [1], and adaptive timers [15]. These concepts, however, are all based on the assumptions that 1) hardware timers are sparse and that 2) they are costly to reprogram [6]. The first assumption is no longer valid with current 32-bit microcontroller platforms; the second still is, but SLOTH ON TIME can avoid the reprogramming costs by using dedicated timer cells for each task and deadline.

VIII. CONCLUSION

We have presented our SLOTH ON TIME RTOS design, which exploits standard timer array hardware available on modern microcontrollers to efficiently offload schedules in a time-triggered or mixed-mode real-time system to the hardware. With our design, tasks are scheduled and dispatched with low latency by the platform's timer and interrupt controller. SLOTH ON TIME instruments the available timer cells not only for task activation, but also for schedule table control, deadline monitoring, and time synchronization; thereby it entirely prevents issues of rate-monotonic priority inversion.

The resulting performance boost is convincing: We have evaluated our approach by implementing the OSEKtime OS standard and the AUTOSAR OS schedule table facility, both of which are omnipresent in the automotive industry. With a dispatch latency of 14 cycles, SLOTH ON TIME outperforms leading commercial implementations of these standards by a factor of up to 171 x. Our results show that it is time (sic!) to exploit the capabilities of modern microcontrollers in time-triggered real-time kernels.

REFERENCES

[1] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM TOCS*, 18(3):197–228, 2000.

[2] AUTOSAR. Specification of operating system (version 4.0.0). Technical report, Automotive Open System Architecture GbR, 2009. http://autosar.org/download/R4.0/AUTOSAR_SWS_OS.pdf.

[3] Sathish Chandra, Francesco Regazzoni, and Marcello Lajolo. Hardware/software partitioning of operating systems: A behavioral synthesis approach. In *GLSVLSI '06*, pages 324–329, 2006.

[4] Uwe Dannowski and Hermann Härtig. Policing offloaded. In *RTAS '00*, pages 218–228, 2000.

[5] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *RTAS '06*, pages 14–23, 2006.

[6] Antônio Augusto Fröhlich, Giovani Gracioli, and João Felipe Santos. Periodic timers revisited: The real-time embedded system perspective. *Computers & Electrical Engineering*, 37(3):365–375, 2011.

[7] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *RTSS '09*, pages 204–213, 2009.

[8] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Sleepy Sloth: Threads as interrupts as threads. In *RTSS '11*, pages 67–77, 2011.

[9] Lennart Lindh and Frank Stanischewski. FASTCHART – A fast time deterministic CPU and hardware based real-time-kernel. In *Euromicro Workshop on Real-Time Systems*, pages 36–40, 1991.

[10] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[11] Andrew Morton and Wayne M. Loucks. A hardware/software kernel for system on chip designs. In *SAC '04*, pages 869–875, 2004.

[12] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. Hardware implementation of a real-time operating system. In *12th TRON Project International Symposium (TRON '95)*, pages 34–42, 1995.

[13] OSEK/VDX Group. Time triggered operating system specification 1.0. Technical report, OSEK/VDX Group, 2001. http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf.

[14] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

[15] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 30 seconds is not enough! A study of operating system timer usage. In *EuroSys '08*, pages 205–218, 2008.

[16] John Regehr and Usit Duongsaa. Preventing interrupt overload. In *LCTES '05*, pages 50–58, 2005.

[17] Di-Shi Sun, Douglas M. Blough, and Vincent John Mooney III. Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications. Technical report, Georgia Institute of Technology, 2002.

[42] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *SOSP '87*, pages 25–38, 1987.