SLEEPY SLOTH: Threads as Interrupts as Threads*

Wanja Hofer, Daniel Lohmann, Wolfgang Schröder-Preikschat
Friedrich–Alexander University Erlangen–Nuremberg, Germany
E-Mail: {hofer,lohmann,wosch}@cs.fau.de

***Abstract*—Event latency is considered to be one of the most important properties when selecting an event-driven real-time operating system. This is why in previous work on the SLOTH kernel, we suggested treating threads as ISRs—executing all application code in an interrupt context—and thereby reducing event latencies by scheduling and dispatching solely in hardware. However, to achieve these benefits, SLOTH does not support blocking threads or ISRs, but requires all control flows to have run-to-completion semantics.**

In this paper, we present SLEEPY SLOTH, an extension of SLOTH that provides a new generalized thread abstraction that overcomes this limitation, while still letting the hardware do all scheduling and dispatching. SLEEPY SLOTH abolishes the (artificial) distinction between threads and ISRs: Threads can be dispatched as efficiently as interrupt handlers and interrupt handlers can be scheduled as flexibly as threads.

Our SLEEPY SLOTH implementation of the automotive OSEK OS standard provides much more flexibility to application developers while maintaining efficient execution of application control flows. SLEEPY SLOTH runs on commodity off-the-shelf hardware and outperforms a leading commercial OSEK implementation by a factor of 1.3 to 19.

I. INTRODUCTION AND MOTIVATION

A core task that an operating system has to fulfill in an event-driven real-time system is to manage the control flows present in the computing system, which encompass *threads* and *interrupt service routines* (ISRs).

Threads are managed by *software*: They are activated on behalf of *software events* only (such as signaling a semaphore), and they are scheduled and dispatched by *software mechanisms*, usually provided by the operating system (OS). ISRs, on the other hand, are managed by *hardware*: They are activated by *hardware events* (such as a periphery device requiring service), and they are scheduled and dispatched by *hardware mechanisms*, usually provided by the interrupt controller. Table I lists those two types of control flows in Lines 1 and 2, together with a comparison of their semantics: Threads can block and resume execution at a later point in time, while ISRs have to run to completion. Traditional threads and ISRs form dual priority spaces, with ISRs having a higher priority than all threads in the system.

A. The Issue

From the conceptual point of view, the forced distinction between threads and ISRs is problematic: The control flow semantics of an event handler (blocking or run-to-completion, as well as its priority relative to other event handlers) should not be defined by the source of the event (hardware or software), but by the requirements of the real-time application. Furthermore, the dual priority spaces make the system susceptible to the real-time issue of *rate-monotonic priority inversion*[1] [3].

Several solutions have been proposed to overcome these issues by employing threads only: In the Solaris OS kernel, ISRs can be promoted to threads upon request, enabling blocking semantics in hardware event handlers [8]. At that point, the management and scheduling of the ISR control flow is switched from the hardware to the OS (see Line 4 in Table I). However, to implement this flexibility, Solaris still has to keep the dual priority spaces, with "ISR threads" having a higher priority than all other, "regular" threads. To tackle that problem, solutions have been proposed in which short ISRs always activate a corresponding thread to run and then terminate immediately [3], [4]. This way, all event handlers (hardware and software) are scheduled and dispatched by the OS as threads in a single priority space. However, since OS-managed threads incur significant software overhead compared to ISRs, this advantage comes at a major performance and latency cost: The latency of ISRs becomes 3–10 times higher—even if parts of the ISR processing are outsourced to an external co-processor [21].

That is why in previous work on the SLOTH embedded kernel, we have proposed the opposite approach: to have *threads run as ISRs* [5]. By triggering the corresponding interrupt from within kernel software when activating a thread, SLOTH relies on a single priority space—the interrupt priority space managed by the hardware—and lets the hardware interrupt arbitration system perform the priority-based scheduling and dispatching (see Line 5 in Table I). The SLOTH kernel, which implements this idea, is simple, small, and fast in scheduling and switching control flows (2–7 times faster than in a traditional kernel). Nevertheless, the SLOTH kernel has a significant drawback: It does not support blocking threads, which need an execution stack of their own. Since it only supports run-to-completion control flows, its execution and preemption pattern is strictly last-in, first out (i.e., stacked)—which is why it is a perfect match for execution using an interrupt controller with multiple interrupt levels, since interrupt activations are also strictly stacked and preempted based on their priorities.

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

[1]This term describes the phenomenon that a high-priority thread can be interrupted and delayed by a low-priority ISR because the hardware-managed ISR priorities are inherently higher than the OS-managed thread priorities.

Appeared in: Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS '11), pages 67–77, IEEE Computer Society, 2011, ISBN 978-0-7695-4591-2

		Activation	Scheduling/Dispatching	Execution Semantics
1	Traditional Threads / OSEK Extended Tasks	by OS	by OS	Blocking
2	Traditional ISRs	by HW	by HW	Run-to-Completion
3	OSEK Basic Tasks	by OS	by OS	Run-to-Completion
4	Solaris IRQ Threads [8]	by HW	by HW $\rightarrow$ by OS	Blocking
5	SLOTH Tasks [5]	by OS or HW	by HW	Run-to-Completion
6	SLEEPY SLOTH Threads	by OS or HW	by HW	Run-to-Completion or Blocking

TABLE I
TYPES OF CONTROL FLOWS AND THEIR PROPERTIES (ACTIVATION AND SCHEDULING/DISPATCHING BY HARDWARE (HW) OR THE OPERATING SYSTEM (OS), AS WELL AS EXECUTION SEMANTICS) IN TRADITIONAL OPERATING SYSTEMS, IN OSEK, IN SOLARIS, AND IN SLOTH, COMPARED TO THE THREAD ABSTRACTION PROVIDED BY SLEEPY SLOTH.

B. About This Paper

In this paper, we aim to remedy this situation by designing a new thread abstraction that combines the advantages of the SLOTH control flow with blocking functionality. These new kinds of threads can be activated by software events and hardware events, they can have run-to-completion or blocking semantics, and they run in a single priority space (see Line 6 in Table I). We argue that this new thread abstraction combines the best properties of threads—blocking flexibility—and traditional ISRs—low execution latency. Our proposed thread model is *flexible* since it allows those control flows to block that need to block, and it is *fast* since it relies on commodity interrupt hardware to perform scheduling and dispatching in hardware. Thus, in the SLEEPY SLOTH kernel[2] that we have implemented as an extension of the original SLOTH kernel, we have removed the artificial distinction between threads and ISRs: *Threads can be interrupt handlers and interrupt handlers can be threads*. SLEEPY SLOTH implements the widely used OSEK operating system standard [18], and it outperforms a leading commercial implementation of that standard by a factor of 1.3 to 19.0.

This paper provides the following contributions:

- We present a new generalized thread abstraction and discuss the challenges in implementing it to provide blocking flexibility while being scheduled and dispatched efficiently using interrupt hardware (see Section III).
- We present our SLEEPY SLOTH kernel design that tackles these challenges by providing a tailored task prologue and a static analysis engine (see Sections IV and V).
- We evaluate our SLEEPY SLOTH prototype on the Infineon TriCore microcontroller, showing that it provides the extra blocking flexibility without harming performance and latency for tasks that do not need it (see Section VI).
- We discuss the necessity for different types of control flows in operating systems and the general applicability of the hardware-centric SLEEPY SLOTH approach (see Section VII).

II. SLOTH REVISITED

The original SLOTH kernel described in [5] implements the BCC1 conformance class of the OSEK operating system standard [18], which is omnipresent in the automotive industry.

[2]The name honors the deadly sin and the lazy animal breed named *sloth*, which likes its “control flows” to *sleep* (i.e., to block).

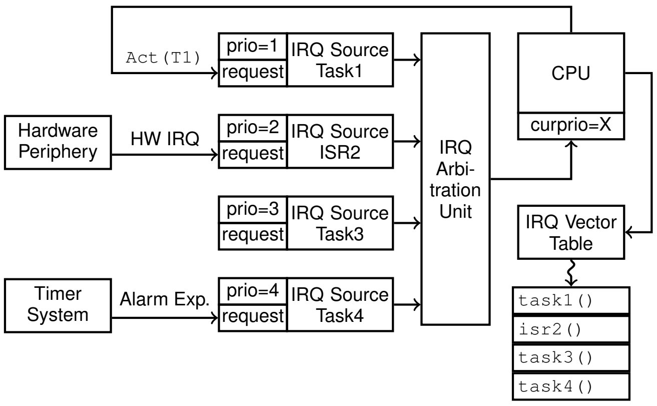

Fig. 1. Design of a SLOTH system, using interrupt handlers for the implementation of threads. The interrupt sources have a statically configured priority and are either triggered synchronously by the CPU through a system-service call (e.g., Task1), through hardware-periphery IRQs (e.g., ISR2), or through the timer system after setting a task alarm (e.g., Task4).

In the following section, we briefly describe OSEK’s features and the corresponding IRQ-based SLOTH design for them; Figure 1 illustrates this by showing an example SLOTH system. OSEK systems (and therefore also SLOTH systems) are configured statically, and, thus, all control flows and their priorities are known at compile time. Note that kernels of that class target microcontrollers without a memory management unit and, together with the application, run in a single execution mode, the supervisor mode; both SLOTH and SLEEPY SLOTH follow that execution model.

a) Task Management: The main idea behind SLOTH is to design every task as an interrupt handler. Every task is assigned an IRQ source with the corresponding priority at compile time, and the corresponding interrupt handler is set to be the user task function. Activation of a task by another control flow is performed by setting the corresponding IRQ request bit by the kernel software (see Task1 in Figure 1), letting the IRQ controller’s arbitration unit decide about preemption depending on the current CPU priority and pending IRQ priorities. A task terminates by returning from the interrupt routine, again relying on the hardware to schedule the task with the next-highest priority.

b) Resource Management: Resources are OSEK’s way of designating critical sections in applications, which are synchronized against preemptions by competing tasks using a stack-based priority ceiling protocol. By acquiring a resource, a task's priority is lifted to the ceiling priority of all tasks that could acquire that resource; SLOTH simply raises the CPU priority to accomplish this. This way, the dispatching of a task that competes for the same resource is delayed until after the critical section is left. Upon release of a resource, the priority is lowered to the previous level, potentially dispatching delayed activated tasks.

c) Alarm Management: Alarms are OSEK's timer abstraction and allow activating a task after a specified amount of time has elapsed. To every task that is configured at compile time to be activated by an alarm at run time, SLOTH assigns an IRQ source that is connected to the timer system (see Task4 in Figure 1). This way, when the timer expires, the task is automatically scheduled by the hardware by triggering the corresponding interrupt, dispatching it depending on the system's current priority situation.

d) ISR Management: OSEK distinguishes between two types of interrupt service routines (ISRs): Category-2 ISRs are allowed to perform system calls and therefore need to be synchronized with tasks in order not to corrupt kernel state, whereas the kernel is oblivious of category-1 ISRs, which are *not* allowed to invoke the kernel. In SLOTH, there is no difference in the handling of category-2 ISRs and tasks; the kernel is oblivious to whether the interrupt request was triggered by a hardware periphery device (see ISR2 in Figure 1) or by software (see Task1 in Figure 1).

III. SLEEPY SLOTH REQUIREMENTS

The only class of OSEK system calls that the original SLOTH kernel does not implement is the one that manages OSEK events. Events are OSEK's means for task notification and, therefore, its means for a task to block (system call `WaitEvent()`) and to be unblocked (`SetEvent()`). OSEK calls tasks that are allowed to potentially block *extended tasks* (which need a *full context* of their own, including a task stack, to be continued in their execution), whereas run-to-completion tasks are called *basic tasks* (which can share parts of their contexts, including their stack, since they preempt each other in a strictly last-in, first-out—that is, stacked—manner).

The overall goal in extending SLOTH to SLEEPY SLOTH is therefore to provide applications the ability to include extended blocking tasks while preserving SLOTH's performance and latency benefits by having threads run as interrupt handlers.

A. SLEEPY SLOTH *Challenges*

In the original SLOTH kernel, where only basic run-to-completion tasks are present, the control flow hierarchy is strictly stacked and strictly nested, which means that control flows are only preempted by higher-priority control flows and returned to after those have run to completion. The SLOTH execution model corresponds exactly to the one that traditional ISRs support using multi-level interrupt controllers, plus the ability to be activated by the operating system (compare Lines 2 and 5 in Table I).

The first and main challenge in SLEEPY SLOTH, however, is to be able to suspend task execution and resume its execution later, which interrupt controllers do not support for interrupt handler execution. This is due to the fact that interrupt handlers are supposed to run to completion *transparently* to the interrupted control flow. Thus, SLEEPY SLOTH needs to find a way to implement both the *suspension* of a blocked ISR and the *re-activation* of an unblocked ISR, saving and restoring its full context including its stack appropriately.

Second, by nature, interrupts are *asynchronous* in their occurrence; that is, no prediction can be made as to where exactly a control flow yields the CPU when interrupted. Thus, performing the necessary context and stack switch in the *preempted* control flow *before* dispatching is impossible in a SLOTH-like system with hardware-triggered preemptions, since the interrupt scheduler and dispatcher are provided by the hardware.

The third challenge regards the execution efficiency of the resulting SLEEPY SLOTH system: The added flexibility for the application developer should not come at the price of lowered system performance. Especially the latencies for scheduling and executing basic run-to-completion tasks should remain comparable to the original SLOTH kernel.

B. Hardware Environment and Requirements

SLEEPY SLOTH's requirements on the hardware platform shall remain the same as stated in [5] for SLOTH; thus, the approach is applicable to any platform with a modern multi-level interrupt controller:

1) The platform needs to be able to trigger interrupts from within software—for instance, by setting a bit in a dedicated register or by offering a special instruction for that matter. This is needed to implement synchronous task activation.
2) The number of available interrupt priorities needs to be at least as high as the number of threads and ISRs in the SLEEPY SLOTH system, since every thread and ISR is assigned a dedicated interrupt source and priority (plus one dedicated priority for each resource; see Section V-E). Our two prototypes run on the Infineon TriCore and the ARM Cortex-M3 platforms, both of which feature 256 interrupt priority levels, enabling SLEEPY SLOTH systems with about 256 real-time control flows.[3] The assignment of a dedicated priority slot per task does not allow for multiple tasks per (semantically equal) priority if the order of activations needs to be preserved.

IV. SLEEPY SLOTH OVERVIEW

In this section, we present the central design ideas in SLEEPY SLOTH to meet the requirements and challenges discussed before. A typical control flow in the system illustrates these ideas, followed by a description of the SLEEPY SLOTH analysis and generation architecture to provide application-tailored context switching.

[3]For the rest of this paper, we assume higher priority levels to be assigned to higher priority numbers (as is the case on the Infineon TriCore platform).

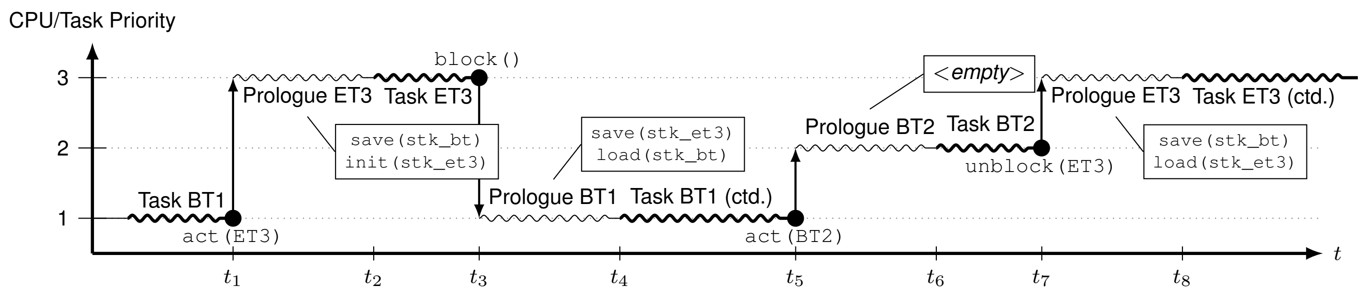

Fig. 2. Example control flow in a SLEEPY SLOTH system with two basic tasks, BT1 and BT2 (with priorities 1 and 2, respectively), which run to completion and share a common stack, `stk_bt`, and one extended task, ET3 (with priority 3), which can block during its execution and therefore has a stack of its own, `stk_et3`. The figure does not reflect timing proportions, as the task prologues are usually very short compared to the task functions themselves.

A. Central Design Ideas

In order to meet the requirements and tackle the challenges stated before, SLEEPY SLOTH is based on three central design ideas.

1) The Task Prologue: SLEEPY SLOTH provides support for blocking tasks by prepending a task prologue to every task function. This prologue is executed whenever a task is dispatched by the interrupt hardware, both when the task is about to run for the first time and when its execution is resumed after being blocked or preempted. The task prologue is the single point to decide whether to save the stack of the interrupted task and whether to restore or initialize the stack of the dispatched task. Note that parts of the task context are saved by the hardware automatically upon dispatching an interrupt handler. The prologue concept is the key enabler for interrupt *re*-activation/resumption, and it addresses the challenge that IRQs occur asynchronously by performing the stack switch (if necessary) in the newly dispatched successor control flow.

2) Threads as ISRs: To provide a performance comparable to SLOTH, SLEEPY SLOTH also relies on the hardware to perform as much of the work as possible for it. This mainly entails the kernel relying on the interrupt system to do the scheduling and dispatching work for it by having all threads run as ISRs with appropriate priorities. Thus, SLEEPY SLOTH has a task run as an ISR whenever possible, and only turns it into a full thread (with a dedicated stack) if the application semantics needs it (in order to be able to block).

3) Static Analysis and System Generation: To achieve its goal to enable efficient execution for basic non-blocking tasks, SLEEPY SLOTH makes use of static application knowledge available at compile time. By statically analyzing the control flow configuration of the application, the SLEEPY SLOTH toolchain can infer information about which task or ISR can preempt which other tasks or ISRs at run time. This information is then used to generate tailored task prologues that omit unnecessary run time checks for preemption conditions before performing stack switches or not. Thus, SLEEPY SLOTH evaluates those conditions statically where possible and dynamically otherwise.

B. SLEEPY SLOTH *Example Control Flow*

Figure 2 shows an example trace of a SLEEPY SLOTH application with two basic tasks, BT1 and BT2 (which share a common stack, `stk_bt`), and a high-priority extended task, ET3 (with a stack of its own, `stk_et3`). Suppose that at some point, only BT1 is running, which then activates ET3 (t_1). ET3 is immediately dispatched because of its high priority—prepended by its prologue, which saves BT1's stack and initializes ET3's stack before executing the actual user function for ET3 (t_2).

ET3 then blocks and releases the CPU, giving control back to BT1 (t_3). Its prologue notes that it follows an extended task (namely ET3, which has previously been running and blocked) and therefore performs a full context switch by saving ET3's stack and loading the common BT stack before resuming execution (t_4). The following activation of BT2 (t_5) dispatches the BT2 prologue, which observes that it has interrupted a basic task and therefore starts executing the task function at t_6 without having to switch stacks.

BT2 then unblocks ET3, triggering its prologue once again (t_7). At this point, the ET3 prologue saves the basic task stack and restores its own stack, resuming execution after the point it had blocked at (t_8).

C. SLEEPY SLOTH *Architecture*

Like OSEK OS, SLEEPY SLOTH is configured completely statically; that is, all threads and ISRs and their priorities are known and configured before compile time. Many crucial parts of the SLEEPY SLOTH system are therefore generated specifically for an application after being analyzed as depicted in Figure 3.

As its input, the SLEEPY SLOTH system takes the configuration of the application as specified by the application programmer. This comprises application objects such as tasks, interrupt service routines, resources, and alarms, together with their names, priorities, and other properties such as whether a task is basic or extended (i.e., it is allowed to block) or which tasks share a given resource.

1) Analysis: The SLEEPY SLOTH analyzer then performs several kinds of analyses on that configuration to provide the subsequent generation step with additional input. First, the configured application control flows (i.e., tasks, category-1

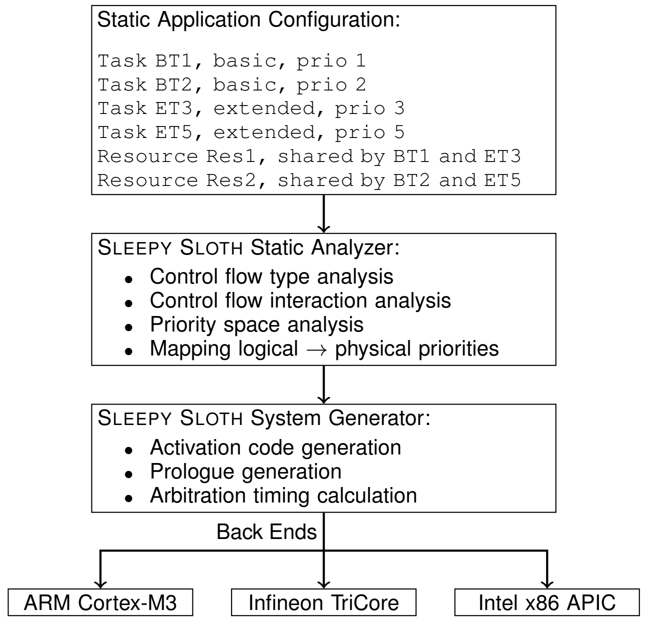
Static Application Configuration:
Task BT1, basic, prio 1
Task BT2, basic, prio 2
Task ET3, extended, prio 3
Task ET5, extended, prio 5
Resource Res1, shared by BT1 and ET3
Resource Res2, shared by BT2 and ET5
SLEEPY SLOTH Static Analyzer:
Control flow type analysis
Control flow interaction analysis
Priority space analysis
Mapping logical → physical priorities
SLEEPY SLOTH System Generator:
Activation code generation
Prologue generation
Arbitration timing calculation
Back Ends
ARM Cortex-M3
Infineon TriCore
Intel x86 APIC

Fig. 3. SLEEPY SLOTH configuration analysis and generation architecture.

ISRs, and category-2 ISRs) are analyzed for their interactions based on their properties and priorities. Internally, this analysis step calculates a preemption graph, which encompasses information about which control flow can be preempted by which other control flows and, therefore, which of the preemptions actually need a stack switch (e.g., a basic task preempting another basic task does not need one). Second, the priority space as specified by the application programmer is analyzed, and the given logical priorities are mapped to physical interrupt priorities. This step comprises both the compacting of the logical priority space if the priority configuration as provided by the application is sparse, and it assigns additional, dedicated priority slots in between control flow priorities for resources. This way, a task holding a resource can be unambiguously identified by its execution priority (see Section V-E).

2) Generation: The actual SLEEPY SLOTH generator then generates application-specific code for the system. This mainly entails code for the activation of a task by setting the interrupt request bit in the appropriately configured IRQ source, and it entails the prologue code for every application task. This prologue code includes only those run time checks that can actually occur in the configured system depending on the calculated preemption graph. Furthermore, as interrupt arbitration systems bear latencies that the kernel needs to consider and as those latencies depend on the involved interrupts, the corresponding timing properties are also calculated in that module (see Section V-C).

3) Back Ends: The back end parts of the SLEEPY SLOTH generator finally generate the architecture-specific parts of the code, which are then compiled with static, non-generated system code (both architecture-dependent and architecture-independent) to produce the combined binary of the SLEEPY SLOTH application and kernel. By producing a single compilation unit using the C preprocessor, the combined application and kernel code is subject to comprehensive compiler optimization, which inlines many of SLEEPY SLOTH's system calls due to their brevity.

V. SLEEPY SLOTH IMPLEMENTATION

The following section first details the SLEEPY SLOTH task prologue and how it interacts with explicit scheduling points such as task termination and task blocking and unblocking. Additionally, resources need to be re-designed in SLEEPY SLOTH, and basic tasks are handled specially to preserve SLOTH's performance characteristics for them as far as possible.

Note that all additional system services and the task prologues in SLEEPY SLOTH as well as SLOTH's original system calls have a statically bounded worst-case execution time for real-time operation.

A. Task Prologue

Whenever the hardware dispatches an interrupt handler that is assigned to an extended task, the task prologue, which is at the core of the SLEEPY SLOTH design (see also Section IV-A) takes action as outlined in Figure 4, with IRQs disabled by the hardware at that point. Depending on the actual task, some of the condition checks and steps are omitted by SLEEPY SLOTH's static analyzer for situations that can never occur at run time (see also Section IV-C).

First, the prologue saves the extended context of the interrupted task (i.e., those registers that have not been saved by the hardware when dispatching the interrupt handler[4]) to the corresponding task stack, whose stack pointer in turn is saved to a kernel context array (Step 1 in Figure 4). Next, the prologue checks whether the interrupted task was either preempted or blocked, or whether it terminated (2). If it did *not* terminate, the prologue re-activates the task to be continued later by triggering the task's interrupt source at the priority it was running at (2a). This is what ET3's prologue does at t_1 and t_7 in the example control flow in Figure 2. Note that the interrupted task's priority might have been raised at the time of preemption due to the possession of a resource in combination with the stack-based priority ceiling protocol; in that case, the continuation of the task needs to be treated specially (see Section V-E). The check for that condition itself is done via a kernel-maintained bit array (`hasSeenCPU[]`). After that, the kernel variable holding the current task is set to the dispatched task ID, which corresponds to the current IRQ number (3).

If the dispatched task has run before (checked by comparing its `hasSeenCPU` property; Step 4 in Figure 4), its context is restored from the kernel context array (entailing its stack and registers; 5a), IRQs are enabled (6a), and execution of the task is continued by returning via the return address in the saved context (7a). If it has *not* run before, its context is initialized (entailing re-setting its stack pointer; 5b) and its `hasSeenCPU` property is set to true to be considered by

[4]Note that, with appropriate compiler support, the prologue can save only those registers that are actually *used* by the dispatched task.

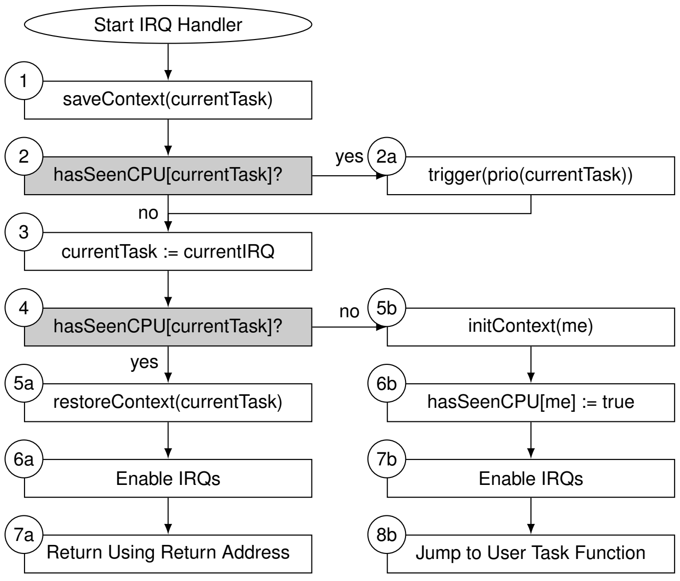

Fig. 4. State diagram of the SLEEPY SLOTH task prologue for extended tasks in its maximum version. Depending on the results of SLEEPY SLOTH's static analysis, each task prologue is tailored to the functionality actually needed at run time.

further preemptions by other tasks (6b). Eventually, the task prologue enables IRQs (7b) and jumps to the actual user task function to start executing user code (7b).

B. Task Termination

Task termination in SLEEPY SLOTH differs from the way it works in the original SLOTH since a stack switch might be needed after termination—for instance, if another task (with the next-highest priority) was unblocked and needs to be continued in its execution after termination of the current task. Again, SLEEPY SLOTH solely relies on the prologue of the next task to determine if a context switch is needed or not. The next task is scheduled and dispatched by the hardware by letting the terminating task set the CPU priority to zero. The interrupt system then determines the next-highest priority task that is ready to run. Before that, the terminating task indicates to the next prologue that it has terminated by re-setting its `hasSeenCPU` flag to false. This way, the interrupt source of the terminating task will not be re-triggered for continued execution (see Steps 2 and 2a in Figure 4).

C. Task Blocking

The main additional system call that SLEEPY SLOTH provides over the original SLOTH kernel is `WaitEvent()`. The implementation compares the event mask of the current task to the event mask to be waited for, and, if they do not match, blocks the task. This is done by disabling the task's IRQ source; this way, it will not be considered in the interrupt arbitration, which determines the highest-priority interrupt to be handled. After that, the CPU is yielded by setting the CPU priority to zero (much in the same way that a task terminates; see also Section V-B) and letting pending interrupts (corresponding to tasks that are ready to run) trigger (see also t_3 in the example control flow in Figure 2). The whole blocking mechanism is synchronized against preemption by other tasks and interrupt handlers by disabling all IRQs at the beginning and re-enabling them at the end:

```
void WaitEvent(EventMaskType mask)
{
  disableIRQs();
  if ((eventMask[currentTask] & mask) == 0) {
    /* none of the events has already been set */
    eventsWaitingFor[currentTask] = mask;
    /* block task */
    disableIRQSource(currentTask);
    setCPUPrio(0);
    waitForArbitration();
  }
  enableIRQs(); /* point of preemption */
}
```

Note that, like when activating a task in the original SLOTH kernel [5], modifying the hardware interrupt priority state of current and pending priorities—in this case, disabling an IRQ source and setting the CPU priority to zero—might require synchronization with the interrupt arbitration system. This is due to latencies during the arbitration in the interrupt system; for instance, for the TriCore microcontroller platform, those latencies are defined by Infineon in an application note [6]. The maximum number of clock cycles that the arbitration process takes depends on several system properties like the system frequency and the number of involved IRQ sources—and, therefore, the number of SLEEPY SLOTH tasks in the system. Thus, this number can be calculated statically by the static analyzer (see Section IV-C) and is inserted in the form of `nop` instructions in `waitForArbitration()` before enabling the IRQs again[5]. This way, SLEEPY SLOTH ensures that the defined point for preemption after blocking the current task will always be the point after the IRQ enable instruction, independent of the current state and latency of the interrupt arbitration system. Note that this arbitration delay makes up for most of the hardware-induced costs, which are significantly lower than any software-induced costs (see also evaluation in Section VI).

D. Task Unblocking

In OSEK, tasks are unblocked by setting one of the events that the task has been waiting for. SLEEPY SLOTH's system call `SetEvent()` therefore first checks whether that condition is met, and then it unblocks the task by re-enabling its IRQ source and triggering its IRQ (see also t_7 in the example control flow in Figure 2). This makes the interrupt controller consider the task in its priority arbitration mechanism and schedule the task according to the system's priority state:

```
void SetEvent(TaskType id, EventMaskType mask)
{
  eventMask[id] |= mask;
  if ((eventMask[id] & eventsWaitingFor[id]) != 0) {
    /* at least one of the events that
```

[5]Before waiting for the arbitration by executing `nop` instructions, the TriCore SLEEPY SLOTH implementation also reads back the interrupt register to synchronize hardware and software as demanded by Infineon's specification [6].

```
     * the task has been waiting for is set */
    eventsWaitingFor[id] = 0;
    /* unblock task */
    disableIRQs();
    enableIRQSource(id);
    waitForArbitration();
    enableIRQs(); /* point of preemption */
  }
}
```

As elaborated in the description of the task blocking mechanism (see Section V-C), due to the synchronization with the interrupt arbitration system, the defined preemption point will be the enable-IRQ instruction at the end of the system call.

E. Resources in SLEEPY SLOTH

OSEK resources are used to synchronize accesses of applications to critical sections by raising their priorities according to a stack-based priority ceiling protocol. In SLEEPY SLOTH, these resources require special handling. Consider a task that, having acquired a resource, is preempted by a higher-priority extended task, which therefore performs a stack and context switch (see example control flow in Figure 5). When the preempted task continues execution, it has to do so *at the raised priority* of the resource (t_d in Figure 5). In the original SLOTH kernel, a resource's ceiling priority was set to the highest priority of all tasks that can acquire that resource. In order for SLEEPY SLOTH to be able to distinguish between an activation of that highest-priority task and a re-activation of a task that had acquired that resource, the resource is given its own dedicated priority, one higher than the ceiling. The resource is therefore also assigned a dedicated IRQ source.

The resource's IRQ source is only triggered after a task that had acquired the resource is preempted by a higher-priority task (t_b in Figure 5). The preempting task's prologue will re-trigger the resource IRQ (see Step 2a in Figure 4 and t_b in Figure 5). This ensures that the dedicated resource IRQ handler will be dispatched once the CPU priority is lowered again—when the higher-priority task terminates or blocks (t_d in Figure 5). The resource IRQ handler then loads a reference to the preempted task (recorded by the original `GetResource()` system call) and restores its context. The execution priority is left unchanged, as it is already at the resource priority (t_e in Figure 5).

F. Basic Tasks in SLEEPY SLOTH

The main goal in designing SLEEPY SLOTH is to support blocking extended tasks while preserving SLOTH's advantages and performance as far as possible for non-blocking basic tasks. Since basic tasks run to completion in a strictly priority-ordered stacked way, SLEEPY SLOTH, just like SLOTH, lets all basic tasks run on a single stack. This makes context switches between basic tasks—both on preemption and on termination—extremely lightweight and fast, since the hardware automatically saves and restores part of the register set upon interrupt entry and return and no additional stack switch is needed.

In SLEEPY SLOTH, however, additional overhead is incurred to *determine* whether a stack switch is needed—that is, whether either the interrupted or the newly dispatched task or both are extended tasks. This property is configured by the application programmer at compile time and stored in a bit field for fast access. Apart from that, during times in the application when only basic tasks are scheduled and dispatched, the overhead incurred by the SLEEPY SLOTH kernel is minimal and comparable to the one incurred by SLOTH (see also the empty prologue at t_5 in the example control flow in Figure 2 and the evaluation in Section VI). Additionally, if the static configuration permits, run time checks can be omitted altogether in the tailored basic task prologues, further reducing overhead (see Section IV-C).

VI. EVALUATION

We have evaluated our SLEEPY SLOTH reference implementation on the Infineon TriCore platform, a 32-bit microcontroller widely used in the automotive industry, featuring a RISC load/store architecture and a Harvard memory model. The interrupt system has 256 priority levels and the TC1796 chip that we use has about as many interrupt sources with memory-mapped registers, enabling SLEEPY SLOTH to modify their enable and pending bits for its purpose. A specialty of the TriCore platform is its separation of the data stack from the call stack, which is managed in separate so-called context save areas; SLEEPY SLOTH therefore has to save and restore both stacks when switching between two extended tasks. We clocked the chip at 50 MHz (clock cycle of 20 ns), although we state our measurements in numbers of clock cycles to be frequency-independent. All measurements were performed using only zero-wait-state internal memories (both code and data), so caching effects did not apply.

To assess the performance gain achieved by implementing thread scheduling using interrupt hardware, we have performed several microbenchmarks comparing SLEEPY SLOTH to a leading commercial OSEK implementation. We set up several test applications for that purpose and compiled them unaltered for both SLEEPY SLOTH and the commercial OSEK, measuring only the latencies of the involved system calls. All numbers were obtained using a Lauterbach hardware trace unit and averaged over at least 10,000 samples.

A. Basic Task System

In prior work [5], we published the execution time numbers of the basic SLOTH kernel for seven microbenchmarks related to task switching. Table II reproduces those numbers and additionally shows the measurements for the SLEEPY SLOTH kernel and the commercial OSEK implementation.

For one, the results show that the numbers for SLOTH and SLEEPY SLOTH are almost identical. This is because SLEEPY SLOTH can be entirely tailored to the functionality that the application actually needs; thus, in a setting with *only* basic tasks, SLEEPY SLOTH will behave identically to SLOTH. The occasional and small deviations are due to bug fixes since the old measurements were performed.

On the other hand, both SLEEPY SLOTH and SLOTH outperform the commercial OSEK implementation, which uses a

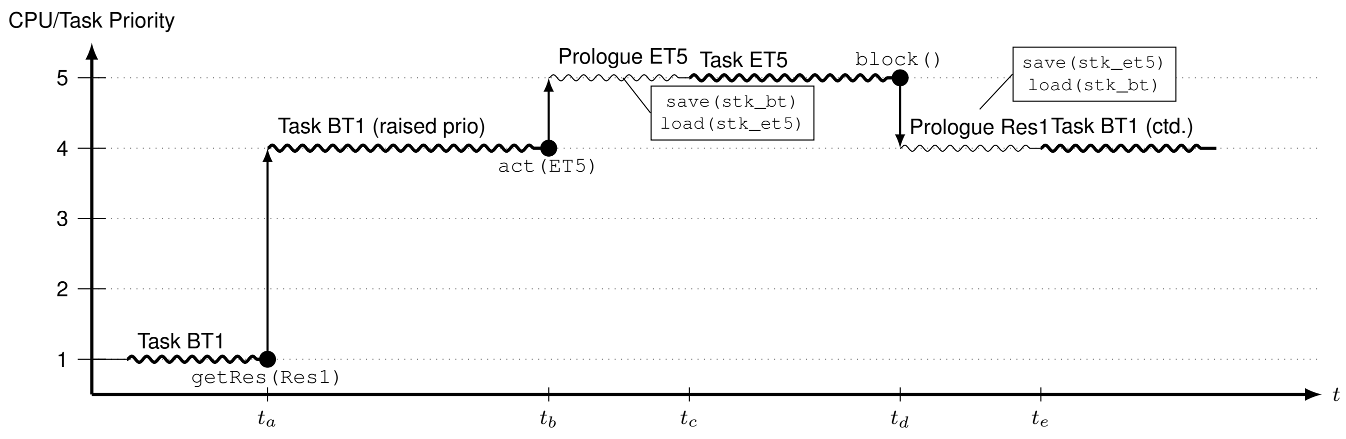

Fig. 5. Example control flow with an OSEK resource in a SLEEPY SLOTH system with one basic task BT1 (with priority 1) and one extended task ET5 (with priority 5). BT1 accesses a resource it shares with extended task ET3 (with priority 3); in SLEEPY SLOTH, the resource is therefore assigned the dedicated priority slot 4 due to the stack-based priority ceiling protocol.

Test Case	SLOTH [5]	SLEEPY SLOTH	OSEK
A1) Task Activation w/o Dispatch	34	38	75
A2) Task Activation w/ Dispatch	60	60	273
A3) Termination w/ Dispatch	14	14	266
A4) Chain w/ Dispatch	67	67	327
A5) Resource Acquisition	19	18	66
A6) Resource Release w/o Dispatch	14	16	128
A7) Resource Release w/ Dispatch	36	38	280

TABLE II
EVALUATION OF **BASIC** TASK SWITCHING MICROBENCHMARKS WITHOUT STACK SWITCHES (EXECUTION TIME IN CLOCK CYCLES), COMPARING THE ORIGINAL SLOTH KERNEL TO SLEEPY SLOTH AND A COMMERCIAL OSEK IMPLEMENTATION.

Test Case	SLEEPY SLOTH	OSEK	Speed-Up
B1) Extended Task Activation w/ Dispatch	121	286	2.4
B2) Blocking w/ Dispatch	143	224	1.6
B3) Unblocking w/ Dispatch	120	205	1.7
B4) Event Mask Clearing	6	32	5.3
B5) Extended Task Termination w/ Dispatch	82	275	3.4
B6) Extended Task Chain w/ Dispatch	113	392	3.5

TABLE III
EVALUATION OF **EXTENDED** TASK SWITCHING MICROBENCHMARKS WITH STACK SWITCHES (EXECUTION TIME IN CLOCK CYCLES), COMPARING THE SLEEPY SLOTH KERNEL TO A COMMERCIAL OSEK IMPLEMENTATION.

software scheduler and dispatcher for operation. The achieved speed-up is between 2.0 and 19, with higher speed-ups for those test cases that not only involve a scheduling decision but also dispatching a new task.

B. Extended Task System

To assess the performance of SLEEPY SLOTH's extended task features, we configured an application that consists only of extended tasks. Hence, every task switch needs to perform a stack switch; the microbenchmark numbers are shown in Table III.

Although SLEEPY SLOTH uses the interrupt controller, whose run-to-completion model does not perfectly fit SLEEPY SLOTH's blocking threads, it still outperforms the commercial OSEK implementation for all test cases. SLEEPY SLOTH's extended task switches are considerably slower (82–143 cycles) than its basic task switches (14–67 cycles) due to the stack switches and involved decisions, but it is still considerably faster than the commercial OSEK, which has a software scheduler (205–392 cycles, resulting in a speed-up of 1.6 to 3.5).

C. Mixed Task System

One of SLEEPY SLOTH's original goals and challenges was to be able to provide the flexibility of blocking tasks without influencing the task switch performance for basic tasks too much (see Section III). On application granularity, we have shown this by measuring a purely basic task system on SLEEPY SLOTH (see Section VI-A). To evaluate this property on task granularity, we configured a single application with two basic tasks and two extended tasks, and we measured different types of preemptions between those tasks (see Table IV).

As expected after analyzing the results from the basic-only and extended-only benchmarks, SLEEPY SLOTH is also faster than the software-based commercial OSEK in the mixed task case (speed-up between 1.3 and 9.7). The numbers for transitions between two basic tasks are on SLEEPY SLOTH's low end (79, 29, and 98 cycles for C1, C5, and C8) and compare to the corresponding benchmarks in the basic-only version (60, 14, and 67 cycles for A2, A3, and A4). Thus, in a mixed task system, basic task scheduling in SLEEPY SLOTH is burdened by an overhead of 15 to 31 cycles, added to an already low base overhead of 14 to 67 cycles.

D. Evaluation Summary

The main goals in the design and implementation of SLEEPY SLOTH were to provide the flexibility of blocking tasks while relying on hardware as much as possible in order to provide good performance for extended tasks without harming performance for basic task scheduling (see Section III). The

Test Case	Task Type Transition	Stack Switch	SLEEPY SLOTH	OSEK	Speed-Up
C1) Task Activation	Basic → Basic	w/o Stack Switch	79	281	3.6
C2) Task Activation	Basic → Extended	w/ Stack Switch	116	286	2.5
C3) Blocking	Extended → Basic	w/ Stack Switch	168	216	1.3
C4) Unblocking	Basic → Extended	w/ Stack Switch	118	205	1.7
C5) Task Termination	Basic → Basic	w/o Stack Switch	29	280	9.7
C6) Task Termination	Extended → Extended	w/ Stack Switch	94	344	3.7
C7) Task Termination	Extended → Basic	w/ Stack Switch	86	282	3.3
C8) Task Chain	Basic → Basic	w/o Stack Switch	98	393	4.0

TABLE IV
EVALUATION OF **MIXED**—THAT IS, BOTH EXTENDED AND BASIC—TASK SWITCHING MICROBENCHMARKS (EXECUTION TIME IN CLOCK CYCLES), COMPARING THE SLEEPY SLOTH KERNEL TO A COMMERCIAL OSEK IMPLEMENTATION.

execution time measurements on the TriCore platform show that the SLEEPY SLOTH implementation is able to meet these goals.

First, no performance penalty at all is incurred for systems that only need basic run-to-completion tasks. In contrast, the software-based commercial implementation shows about the same overhead for switches between basic tasks as for switches between extended tasks.

Second, although the SLEEPY SLOTH implementation is not as simple as is SLOTH's, extended task scheduling is not slower than in the commercial implementation with a software scheduler. In fact, SLEEPY SLOTH is able to outperform the commercial OSEK by a factor of 1.6 to 3.5.

Third, in a mixed task system, the task switch overhead scales with the demand of the involved tasks. Task switches between basic tasks in a mixed SLEEPY SLOTH system are cheaper than task switches between extended tasks, which need additional stack switches.

The real-time application benefits from SLEEPY SLOTH by suffering lower system call latencies compared to a software scheduler kernel, positively affecting the response times it asserts itself to the user of the system. The actual performance gain depends on the actual application and the ratio of executed application code to executed system code. Additionally, since all tasks and ISRs run in the same priority space, the analysis of the system's real-time properties is facilitated (see also discussion in Section VII-A). Note that the numbers discussed in this section include all hardware-related preemption costs such as waiting for the bus arbitration (see also Section V-C); nevertheless, the SLEEPY SLOTH system still outperforms the software-based commercial kernel.

VII. DISCUSSION

SLEEPY SLOTH combines the flexibility of an off-the-shelf embedded kernel with the efficiency of a purely interrupt-driven system. In this section, we discuss the necessity for different control flow types in embedded systems, and we discuss the general applicability of the SLEEPY SLOTH approach.

A. Control Flows in Embedded Systems

As outlined in the introduction and Table I, current operating systems distinguish between two kinds of control flows. If blocking ability is needed, the control flow must be represented by a *thread*, if it requires asynchronous activation by a hardware device, it has to be an *ISR*. SLEEPY SLOTH threads, however, provide both of those properties in one generalized abstraction, leading to several advantages for the real-time application.

For one, SLEEPY SLOTH threads can interact and synchronize freely using common synchronization and notification mechanisms, such as OSEK resources and events. In traditional systems, communication between threads and ISRs and synchronization of threads and ISRs is complicated or even impossible to achieve.

Additionally, since SLEEPY SLOTH threads run in one common priority space—the interrupt priority space—, the conditions for rate-monotonic priority inversion [3] are eliminated. In traditional kernels, it is impossible to have a thread run at a higher priority than an ISR (since interrupt priorities are implicitly higher than thread priorities), even though the application might demand it. SLEEPY SLOTH enables the application to freely distribute priorities among its control flows depending solely on its *semantic* requirements.

B. General Applicability of the Approach

The nature of both SLOTH and SLEEPY SLOTH implies that the actual *implementation* on a platform with a specific interrupt controller is highly hardware-dependent, since the kernels rely on efficient hardware mechanisms to perform the scheduling work for them. However, the matter of the fact is that the actual kernel code is very small and has a clear abstraction boundary that needs to be mapped to the hardware platform, resulting in a manageable porting effort. The fact that a kernel can reach new levels of efficiency by tailoring its implementation to the target hardware has been exploited many times before—for instance, when making microkernel inter-process communication more efficient [10].

Our SLEEPY SLOTH prototype implements the OSEK operating system specification to be able to compare it to other implementations without adapting the benchmark applications (see also evaluation in Section VI). The approach to implement blocking threads as interrupt handlers with a tailored prologue, however, is applicable to any event-driven real-time operating system with static priorities. *Dynamic-priority* systems, on the other hand, which need to re-prioritize threads at run time, are not worthwhile implementing using the SLEEPY SLOTH approach, since dynamic re-configuration of interrupt sources and preempted interrupt handlers is usually very costly. Generally speaking, SLEEPY SLOTH is suitable to implement the most well-known fixed-priority scheduling algorithms such as rate-monotonic [12] and deadline-monotonic scheduling [13]. Advanced scheduling mechanisms such as the general priority ceiling protocol, the priority inheritance protocol, and aperiodic servers can still be implemented in SLEEPY SLOTH, since they only require *occasional* re-prioritization of control flows by re-configuring the corresponding interrupt sources.

Despite running all threads as ISRs, exception traps can also be accommodated in SLEEPY SLOTH. Traps cannot be masked and do not interfere with the interrupt priorities that SLEEPY SLOTH uses for its purposes. Thus, they can simply be executed in the context of the currently running thread or ISR; the trap return will then restore the context appropriately. Note that the SLEEPY SLOTH system calls themselves are *not* implemented as traps but as simple functions that can even be inlined by the compiler.

VIII. RELATED WORK

The approach to having basic run-to-completion tasks run as ISRs and to mapping task activations to IRQ source triggering was new when we first published SLOTH [5]. SLEEPY SLOTH extends this work by providing this functionality to more complex blocking tasks. We are not aware of any work on a comparable embedded kernel that runs on commodity hardware; here, we list work on kernels with scheduling enhanced by *customized* hardware and work on the thread–ISR boundary.

Control flow scheduling and dispatching is the core responsibility of any operating system kernel and thus important to be efficient. That is why ways to move software task scheduling into hardware have been researched, although all of the approaches—like Atalanta [23], cs2 [15], HW-RTOS [2], FASTCHART [11], and Silicon TRON [16]—use *customized* hardware synthesized on an FPGA or a similar component to achieve this. This allows implementing arbitrary control flow semantics, including blocking threads. The Responsive Multithreaded Processor (RMT) [25], [24] is customized to integrate real-time functions into the processing unit, including very fast task switching by providing eight hardware contexts plus 32 contexts in an on-chip context cache [17]. SLEEPY SLOTH, on the other hand, is designed to run on *commodity off-the-shelf hardware* with any modern interrupt controller to achieve its boost in performance. Note that some architectures provide hardware support for fast context switching; however, that support only targets fast *dispatching* but not *scheduling* of tasks as the SLEEPY SLOTH approach does.

Overcoming the strict distinction between OS-operated threads and hardware-operated interrupt handlers has also been the work of Kleiman et al. [8]. In their work on the Solaris kernel for desktops and servers, they investigated ways to enhance interrupt service routines to become threads under the control of the OS kernel—in order for them to be able to block. Lohmann et al. [14] showed that this concept is also feasible for embedded system kernels and made this kind of interrupt synchronization a configurable property of their CiAO system. Before those two systems, early microkernels and microkernel-like systems (including AX [22], L3 [9], and L4 [10]) started implementing interrupt handlers as user threads, having very small stubs send the corresponding threads an IPC message once an interrupt is triggered; this way, regular threads and ISR threads also run in a single priority space. However, during the execution of the interrupt handler stubs, those systems still exhibit rate-monotonic priority inversion by potentially delaying high-priority threads. Intel's early real-time operating systems, such as iRMX [20] and iDCX 51 [7], also included concepts of special interrupt tasks, which are scheduled and dispatched as high-priority OS tasks in the OS priority space. However, all of the discussed kernels merely upgrade their interrupt handlers to full-blown threads (which implies the *overhead* for software-managed threads), whereas SLEEPY SLOTH goes the other way and makes threads run as ISRs, resulting in a significant performance gain.

IX. CONCLUSIONS

We have presented our SLEEPY SLOTH operating system design, which exploits standard interrupt hardware to implement an efficient generalized thread abstraction that can be triggered by hardware and software events. SLEEPY SLOTH control flows are scheduled and dispatched with low latency by the interrupt controller; additionally, they can be blocked and resumed in their execution like threads. They share a single priority space, facilitating arbitrary interactions between hardware- and software-induced control flows and avoiding the real-time problem of rate-monotonic priority inversion.

As SLEEPY SLOTH uses the hardware interrupt system instead of software-implemented routines for scheduling and dispatching, the resulting performance boost is convincing. We have evaluated our approach by implementing the conformance classes BCC1 and ECC1 of the OSEK OS standard, which is omnipresent in the automotive industry. With respect to event latencies, SLEEPY SLOTH outperforms a leading commercial implementation of this standard by a factor of 1.3 to 19.

The SLEEPY SLOTH approach abolishes the artificial distinction between threads and ISRs: *threads can be interrupt handlers* and *interrupt handlers can be threads*. Developers of event-driven systems can forget about the differences, choosing priorities and semantics freely based solely on the requirements of the application.

X. FUTURE WORK

Having designed and implemented a generalized thread abstraction that is scheduled and dispatched by the interrupt controller, we aim at accommodating future multi-core platforms as well. Since SLEEPY SLOTH is already running on an ARM Cortex-M3 microcontroller, we want to investigate ways how to run it on the open source Pandaboard [19], which features a dual-core ARM Cortex-A9 MPCore. Both the M3 and the A9 have the same interrupt controller, ARM's nested vectored interrupt controller (NVIC), which fulfills the requirements as stated in Section III-B. Since SLEEPY SLOTH is based on the OSEK OS standard [18], we will evaluate its successor standard, AUTOSAR OS, and in particular its multi-core specification [1] to be able to present a sophisticated multi-core SLOTH design.

XI. ACKNOWLEDGMENTS

We thank the RTSS 2011 reviewers for their valuable feedback and especially our shepherd Gernot Heiser for his efforts in helping us improve this paper.

REFERENCES

[1] AUTOSAR. Specification of multi-core OS architecture (version 1.1.0). Technical report, Automotive Open System Architecture GbR, November 2010. Available at http://www.autosar.org/download/R4.0/AUTOSAR_SWS_MultiCoreOS.pdf.

[2] Sathish Chandra, Francesco Regazzoni, and Marcello Lajolo. Hardware/software partitioning of operating systems: A behavioral synthesis approach. In *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI '06)*, pages 324–329, Philadelphia, PA, USA, April 2006.

[3] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '06)*, pages 14–23, San Jose, CA, USA, April 2006.

[4] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt scheduling with low overhead for real-time kernels. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '06)*, pages 385–394, Sydney, Australia, August 2006.

[5] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS '09)*, pages 204–213, Washington, DC, USA, December 2009.

[6] Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *AP32009, TC17x6/TC17x7 – Safe Cancellation of Service Requests*, July 2008. Available at http://www.infineon.com/dgdl/?folderId=db3a304412b407950112b409a9ae0337&fileId=db3a304412b407950112b41be4652cbf.

[7] Intel Corporation, 5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA. *iDCX 51 Distributed Control Executive User's Guide for Release 2*, April 1987. Available at http://www.alfirin.net/flamer/bitbus/dcx51.zip.

[8] Steve Kleiman and Joe Eykholt. Interrupts as threads. *ACM SIGOPS Operating Systems Review*, 29(2):21–26, April 1995.

[9] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 175–188, Asheville, NC, USA, December 1993.

[10] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 237–250, Copper Mountain Resort, CO, USA, December 1995.

[11] Lennart Lindh and Frank Stanischewski. FASTCHART – a fast time deterministic CPU and hardware based real-time-kernel. In *Proceedings of the 1991 Euromicro Workshop on Real-Time Systems*, pages 36–40, Paris-Orsay, France, June 1991.

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

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

[14] Daniel Lohmann, Jochen Streicher, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Interrupt synchronization in the CiAO operating system. In *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '07)*, Vancouver, BC, Canada, March 2007.

[15] Andrew Morton and Wayne M. Loucks. A hardware/software kernel for system on chip designs. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*, pages 869–875, Nicosia, Cyprus, March 2004.

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

[17] Amos R. Omondi and Michael Horne. Performance of a context cache for a multithreaded pipeline. *Journal of Systems Architecture*, 45(4):305–322, 1998.

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

[19] Pandaboard homepage. http://pandaboard.org/.

[20] RadiSys Corporation, 5445 NE Dawson Creek Drive, Hillsboro, OR 97124, USA. *Introducing the iRMX Operating Systems*, December 1999. Available at http://www.tenasys.com/support/files/IntroducingiRMX.pdf.

[21] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudi Pfister, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system. In *Proceedings of the 2009 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES '09)*, pages 59–67, Grenoble, France, October 2009.

[22] Wolfgang Schröder. *A Family of UNIX-like Operating Systems — Use of Processes and the Message-Passing Concept in Structured Operating-System Design*. Dissertation, Technical University of Berlin, 1986. In German.

[23] 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.

[24] Nobuyuki Yamasaki. Responsive multithreaded processor for distributed real-time systems. *Journal of Robotics and Mechatronics*, 17(2), 2005.

[25] Nobuyuki Yamasaki, Ikuo Magaki, and Tsutomu Itou. Prioritized SMT architecture with IPC control method for real-time processing. In *Proceedings of the 13th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '07)*, pages 12–21, Bellevue, WA, USA, April 2007.