# SLOTH: Threads as Interrupts*

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

***Abstract*—Traditional operating systems differentiate between threads, which are managed by the kernel scheduler, and interrupt handlers, which are scheduled by the hardware. This approach is not only asymmetrical in its nature, but also introduces problems relevant to real-time systems because low-priority interrupt handlers can interrupt high-priority threads.**

**We propose to internally design all threads as interrupts, thereby simplifying the managed control-flow abstractions and letting the hardware interrupt subsystem do most of the scheduling work. The resulting design of our very light-weight SLOTH system is suitable for the implementation of a wide class of embedded real-time systems, which we describe with the example of the OSEK-OS specification. We show that the design conciseness has a positive impact on the system performance, its memory footprint, and its overall maintainability.**

## I. Introduction

One of the core responsibilities of an operating-system kernel is the management of control flows in the system. Traditionally, these encompass synchronously executed *threads*, and asynchronously triggered *interrupt handlers*. The latter ones are usually signaled by hardware devices and have an implicitly higher priority than synchronous control flows by being able to interrupt the CPU at any time. This bifid priority space—divided up into interrupt priorities and thread priorities—induces a problem termed *rate-monotonic priority inversion*: Interrupt-handler control flows that have a semantically lower priority than a real-time thread can interrupt and delay the execution of that real-time thread [4].

In previous work, we have tackled that problem by using a coprocessor that pre-handles all interrupts [18]. In this paper, we show how to overcome rate-monotonic priority inversion by making use of more sophisticated interrupt systems as available on many newer hardware platforms, without the need for a coprocessor. In our SLOTH[1] system, we propose to internally design every control flow in the system as an interrupt—even regular threads—by implementing thread-related system calls using the interrupt system. The SLOTH approach has the following advantages:

- It implements a unified priority space, allowing for arbitrary distribution of priorities to both thread and interrupt control flows.
- The kernel implementation can be kept extremely concise and is therefore well maintainable and subject to easy and comprehensive testing.
- By letting the hardware schedule the control flows, the performance of the system calls and context switches is very high compared to regular, purely software-based thread implementations, providing for both very low and deterministic overhead.

At the same time, the application programmer can still use the notion of a thread as a unit of decomposition; the API that SLOTH offers remains the same as in a traditional implementation, eliminating the need for porting.

We have implemented the conformance class BCC1 of the OSEK–operating-system specification [17], which targets event-driven embedded real-time systems, for the Infineon-TriCore microcontroller [5], which features an interrupt subsystem that fulfills the requirements for a SLOTH system. This way, we can show that our SLOTH design can be implemented using state-of-the-art commodity hardware, and we can evaluate the advantages of such a design.

## II. Design

In a seminal paper entitled *Interrupts as Threads* [7], Kleiman and Eykholt describe the implementation of control flows in the Solaris-2 kernel, in which interrupt handlers can become full-fledged threads if they need to block. We propose quite the opposite approach, which treats all threads as interrupt handlers and thereby lets the hardware handle most of the scheduling work implicitly.

### A. Overview of OSEK OS

Our kernel design targets an embedded, event-driven real-time system. In order to simplify the description, we use the terminology and system-service grouping as specified by the OSEK-OS standard [17], an operating-system specification widely used in the automotive domain. The feature diagram in Figure 1 gives an overview of the features of an OSEK system.

Among the offered control-flow abstractions, *tasks* (traditionally called threads) are managed by the OS scheduler, whereas interrupt service routines (ISRs) are triggered by the hardware. The OS is oblivious of *category-1 ISRs*, which are not allowed to use its system services, whereas *category-2 ISRs* have to be synchronized with the kernel since they *are* allowed to use system functions. Whether a task is preemptable by higher-priority tasks or not is configured globally

* This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4 and SCHR 603/7-1. Wanja Hofer was supported by the German Academic Exchange Service (DAAD) under grant no. D/09/40595.

[1] The name honors both the lazy animal breed and the deadly sin.

**Appeared in:** Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS '09), pages 215–228, IEEE Computer Society, 2009, ISBN 978-0-7695-3875-4

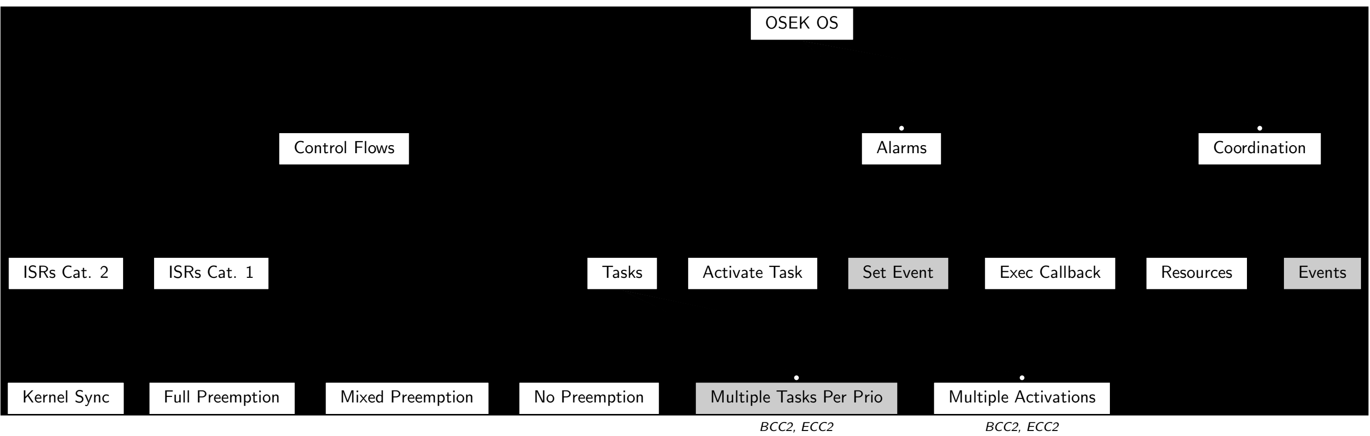


Fig. 1. Feature diagram of the OSEK–operating-system specification. Feature types include mandatory features (filled circle), optional features (hollow circle), minimum-one feature sets (filled arc), and exactly-one feature sets (hollow arc). Features not yet integrated in the SLOTH design are depicted in gray color. If a particular feature is mandatory only in conformance classes other than the basic BCC1, this information is given below that feature.

(full preemption or no preemption) or locally on a task-by-task basis (mixed preemption). Furthermore, whether multiple activations of a task can be stored by the OS and whether it supports multiple tasks with the same priority are optional system features. *Alarms* are timer abstractions that can activate a task, execute a callback function, or set an event upon expiry after a specified period of time. To wait for an *event* is the only possibility for a task to become blocked; it is unblocked when that event is set by another control flow. The other coordination abstraction—*resources*—is used to synchronize critical sections within the application by mutual exclusion.

OSEK also defines four conformance classes (BCC1, BCC2, ECC1, ECC2), which define minimum requirements on which of the optional features have to be provided (see also Figure 1). In our SLOTH design, we target the OSEK conformance class BCC1. Thus, we have a statically configured system with static task priorities (no task creation and altering of the task priorities at run time is possible) and run-to-completion tasks only (i.e., tasks cannot block by waiting for an event), supporting only one task per priority level. Apart from that, the application can be as complex as any other OSEK application, and it is configured and programmed using the same OSEK system API that any software implementation offers, so no porting is required.

### *B.* SLOTH *Design Overview*

An overview of our design is given in Figure 2. Tasks and ISRs are represented by an abstract interrupt source that has an appropriately configured priority. The corresponding request is triggered either synchronously by the CPU when the `ActivateTask()` system service is invoked, or asynchronously by connected hardware devices. Additionally, tasks that are configured to be activated by OSEK alarms after a specified time period are represented by interrupt sources that are connected to the timer system.

The scheduling of the system is done completely in hardware. First, an IRQ arbitration unit decides which of the attached interrupt sources (and, therefore, which of the attached control flows) has the highest priority. After that, the CPU is interrupted by an interrupt request, but only if its current priority is lower than the one of the requested control flow. In that case, the corresponding task or ISR is dispatched by looking it up in the vector table. Note that the current priority level of the CPU does not necessarily have to be the one of the executing task. The CPU priority level is also altered for synchronization purposes—for instance, in order to implement resources for mutual exclusion (see Section II-E).

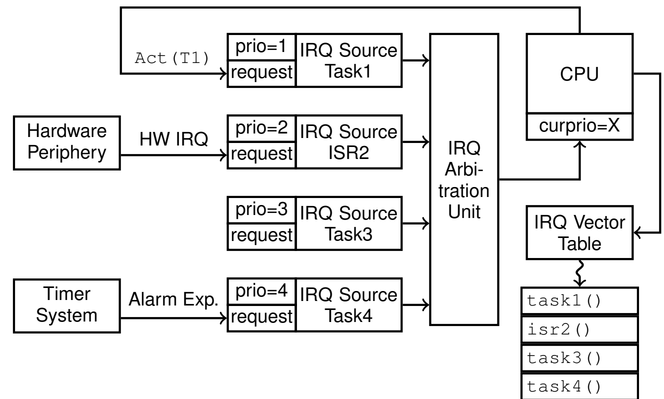


Fig. 2. Design of a SLOTH system, using interrupt handlers for the implementation of threads. The interrupt sources have a statically configured priority and are either triggered synchronously by the CPU through a system-service call (e.g., Task1), through hardware-periphery IRQs (e.g., ISR2), or through the timer system after setting a task alarm (e.g., Task4).

The rest of this section details the design of typical embedded–operating-system services on the example of the major system-service groups offered by the OSEK operating system. In parallel, refer to Figure 3 for an example control flow in a SLOTH system. It uses the application configuration as depicted in Figure 2; that is, Task1, ISR2, Task3, and Task4 have the priorities 1, 2, 3, and 4, respectively.

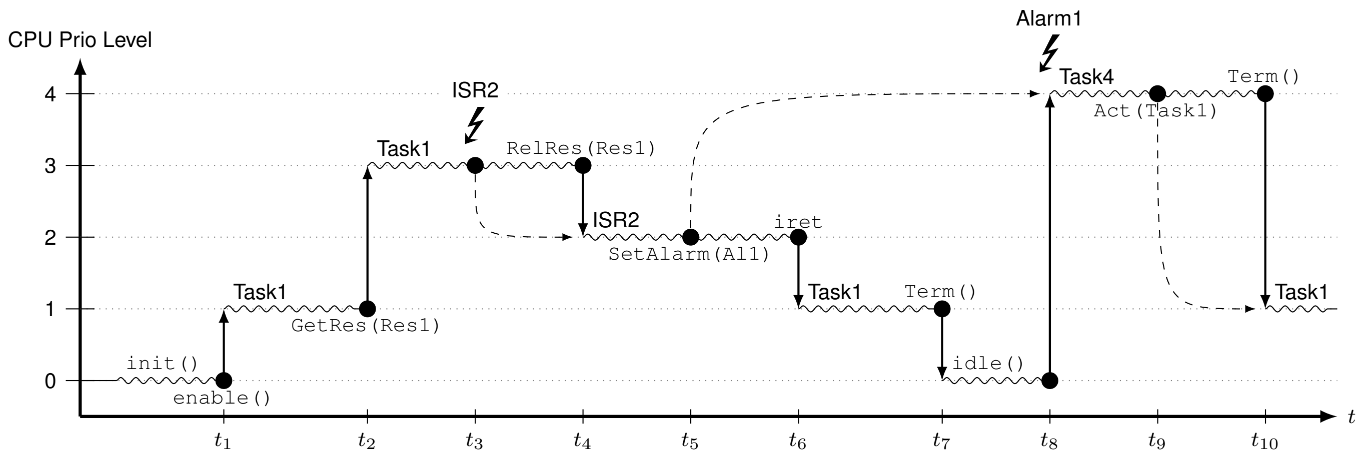


Fig. 3. Example control flow in a SLOTH system. The execution of most system calls leads to an implicit or explicit altering of the current CPU priority level, which then leads to an automatic and correct scheduling and dispatching of the control flows by the hardware.

### C. Task Management

In SLOTH, tasks (OSEK's name for threads) are identified by their priority; that is, a task's ID is the same as its priority. Activating a task corresponds to merely triggering the corresponding interrupt source. The resulting interrupt request is then immediately handled if the priority of the new task is higher than the one performing the activation, given that the currently running task is configured to be preemptable. Termination of a task is a simple return from the interrupt handler, which then leads to an automatic dispatch of the pending task with the next-highest priority. For task chaining, the specification demands that the task performing the chain operation is completed before the chained task starts to execute. In SLOTH, this behavior is ensured by disabling interrupts for a short and bounded time, then activating the task to be chained, and then returning from the interrupt handler, which implicitly re-activates interrupts.

In the example depicted in Figure 3, when Task1 terminates at $t_7$, it restores the previous priority 0 by executing a return-from-interrupt instruction. Likewise, when Task4 terminates at $t_{10}$, it also tries to restore the previous priority 0, which leads to an automatic scheduling of Task1 first, because it is still pending with priority 1. Its activation by Task4 at $t_9$ was automatically delayed, because of the lower priority of Task1.

### D. Interrupt Handling

In our SLOTH system, tasks and those kinds of ISRs that are allowed to perform system calls (named category-2 ISRs in OSEK) are completely identical, thereby unifying the priority space and allowing for mixed priorities between them. Only those ISRs that are guaranteed *not* to perform any system calls (category-1 ISRs) have priorities higher than all tasks and category-2 ISRs. Hence, category-2 interrupts can be suspended by setting the current CPU priority level to the highest priority of all category-2 ISRs. All interrupts (including the ones of category 1) can be suspended or disabled by the application by setting the CPU priority level to the highest priority of *all* ISRs, or by disabling interrupts completely.

Both kinds of ISRs are dispatched by the hardware whenever the CPU priority level is below the one of the interrupt request. In the example control flow in Figure 3, for instance, ISR2 is not dispatched until Task1 lowers its priority to 1 by releasing Resource1 at $t_4$, although ISR2 was already requested at $t_3$.When ISR2 terminates at $t_6$, it executes a regular return-from-interrupt instruction and thereby implicitly re-actives the pending control flow with the next-highest priority, Task1.

### E. Resource Management

Resources (OSEK's terminology for mutex synchronization objects) are used to protect critical sections. In order to avoid deadlocks and priority inversion, OSEK prescribes a stack-based priority ceiling protocol similar to the stack resource policy by Baker [1]. This protocol mandates immediately raising a task's priority to the resource ceiling priority upon acquiring the resource, and lowering it to the original priority upon releasing it. This way, tasks can never become blocked upon resource acquisition and the acquisition will always succeed; otherwise, another task with a higher priority (gained by acquiring that same resource) would be running instead.

In our SLOTH kernel, a resource ID is equal to its ceiling priority—that is, the highest priority of all tasks and category-2 ISRs that can acquire it. Thus, acquiring a resource means simply raising the current CPU priority level to the ceiling priority (i.e., the resource ID), and releasing it means re-setting the level to the original value. Since multiple resources can be acquired, the previous priority has to be saved on a stack. Because of the static system configuration, the stack usage induced by resource acquisition can be bounded at compile time.

In the example sketched in Figure 3, since Resource1 can be acquired by both Task1 and Task3 (not active in the example control flow), its ceiling priority is 3. Thus, when Task1 acquires it at $t_2$, it raises the CPU priority level to 3, and it tries to re-set it to the previous priority 1 upon releasing

it at $t_4$, leading to the dispatching of the pending ISR2 as described in Section II-D.

### *F. Alarms*

Alarms are offered by the OSEK operating system to enable the application to take action after a specified time budget has elapsed. If an alarm is configured to activate a task, an interrupt source that is connected to the hardware timer system is chosen for that task and configured with its priority. The service call setting an alarm can then be simply implemented by programming the connected hardware timer appropriately; the timing parameters have to correspond to the ones provided to the system call. When the timer expires, the configured task is then activated automatically by triggering the interrupt source, leading to preemption if the currently running task has a lower priority. Since most of these actions are done by hardware, the alarm-service implementation itself can be kept very light-weight.

In the example in Figure 3, ISR2 sets an alarm at $t_5$, which is configured to activate Task4 upon expiry. When the hardware timer fires at $t_8$, it automatically activates Task4, because the corresponding interrupt source has the priority 4 of Task4.

If an alarm is configured to execute a callback function, that function can be treated the same way—as a special, high-priority task. Callback functions were originally introduced in OSEK in order to offer a very light-weight reaction possibility, but with SLOTH's light-weight thread design, this is not an issue to be concerned about (see also Section V).

### *G. Nonpreemptive Systems*

The SLOTH design as described in this paper targets a preemptive system, in which each activation of a higher-priority task leads to a rescheduling and dispatching. In order to implement a nonpreemptive system, only a few details have to be adjusted in the design.

First, every nonpreemptable task starts at the priority level of the highest-priority task in the system instead of at its own priority. This way, when a task activates a higher-priority task, that task is not dispatched immediately. Second, an explicit point of rescheduling (e.g., the OSEK system service `Schedule()`) is implemented by lowering the priority to the original task priority before raising it again. This way, any pending tasks of higher priority are allowed to run and to complete at this point before the original task is executed again. Note that, in a preemptive system, `Schedule()` is effectively empty since rescheduling is always performed immediately anyway.

Using the same idea, the special scheduler resource `RES_SCHEDULER` is implemented by setting its ceiling priority to the one of the highest-priority task in the system. By acquiring this virtual resource for a limited period of time, preemptive tasks can delay preemption in critical sections until after releasing the resource—as demanded by the specification. Groups of tasks that *do not preempt* each other *within groups* but *do preempt* each other *between groups* can be designed the same way; for this purpose, OSEK offers *internal resources*. By letting each task run with the priority of the highest-priority task in its group (i.e., by acquiring this internal resource), preemption within the group is delayed until the task reschedules explicitly. This rescheduling system call temporarily lowers the current priority to the task's original priority—like in a completely nonpreemptive system as described above.

### *H. Multiple Task Activations*

The optional OSEK feature to support multiple activations of the same task can be easily integrated by an additional activation counter per task. When activating a task, in addition to requesting an interrupt, the corresponding counter is increased. Upon termination of the task, the counter is simply decreased, and—if the number of activations is greater than zero—the interrupt is requested again before really terminating the task.

This mechanism only works for tasks activated through the corresponding system call; it does not work for real ISRs that are triggered by hardware periphery, since to the best of our knowledge there is no interrupt controller that can store more than one activation. Because SLOTH implements tasks activated by alarms by letting the timer system simply set the interrupt-request bit (see Section II-F), those tasks have only limited multiple-activation support in SLOTH.

### *I. Summary of the* SLOTH *Thread Abstraction*

Compared to traditional OS thread implementations, SLOTH threads are different in several points.

First, SLOTH threads run to completion and are only preempted by higher-priority threads. Conventional threads can wait for an event and block, letting lower-priority threads run. This is a limitation that we want to tackle in future work (see also Section VII), but which still allows for a broad range of applications (see also Section V).

SLOTH's run-to-completion property leads to a strictly stack-like control-flow dispatching, which is also illustrated in Figure 3. This way, SLOTH can use only a single shared stack—the interrupt stack—for all its threads, and the preempted-thread context is stored on that stack. Traditional threads have a stack of their own and have their context saved by the kernel in an additional structure.

Traditional threaded OS kernels maintain a *software* ready queue and running pointer, and they need additional information in software, such as the priorities of the threads, to make scheduling decisions whenever the state of one the threads changes, possibly leading to a new thread being dispatched. SLOTH has all this information implicit in the interrupt hardware subsystem, with the ready queue being represented by the interrupt-pending bits of the hardware, relying on the hardware to do the scheduling and the dispatching.

To the application programmer, all of these differences are hidden beneath the same thread API; SLOTH currently offers the same OSEK task abstraction and system services like any other, software-based implementation.

### J. Requirements on the Hardware Interrupt System

For our approach to be feasible, we have two requirements on the interrupt subsystem of the hardware platform that our SLOTH kernel is implemented on:

1) Interrupt priorities: The interrupt system shall offer as many different interrupt priorities as there are threads and interrupt handlers in the system.
2) Interrupt triggering: The interrupt system shall support manual, software-based triggering of interrupts. This can be offered through a special instruction or through the modification of corresponding hardware registers.

Note that these are the only requirements for a SLOTH implementation. Some platforms fulfill these requirements natively (such as the Infineon TriCore detailed in Section III-A, or the ARM Cortex-M3), whereas others have an external interrupt controller that provides the corresponding functionality (such as the APIC present on all modern Intel-x86 systems).

## III. IMPLEMENTATION

We have implemented our SLOTH approach for the Infineon TriCore [5], an embedded microcontroller platform commonly used in the automotive domain. We shortly describe the relevant features of the platform before sketching our implementation.

### A. The Infineon-TriCore Platform

The TriCore platform has a sophisticated interrupt subsystem that fulfills our requirements as stated in Section II-J.

Interrupt sources are represented by *service request nodes* (SRNs), which encapsulate all the relevant properties such as priority, enable status, and request status. All SRNs are connected to an *interrupt arbitration unit* (IAU) through a special bus for exchanging priority information in order to find a precedence among the pending interrupts. This process, called *arbitration*, takes a defined number of system-bus cycles, which itself depends on the system clock frequency and the priority range of the SRNs actually competing in the arbitration. Hence, the fewer tasks and ISRs are configured in a system, the fewer arbitration cycles are needed to prioritize the concurrent requests.

Most of the SRNs are connected to an actual hardware source (e.g., the general-purpose timer array of the TC1796 derivative features 92 SRNs), but there are special SRNs available for software-only access (named `CPU_SRCx`). Additionally, hardware-connected SRNs that are not used in a given application can also be used to implement threads as interrupts, because every SRN has its registers memory-mapped, allowing for software-based interrupt triggering as required by SLOTH (see Section II-J).

### B. Task-Activation Implementation

The implementation of the SLOTH design as sketched in Section II is very straight-forward on the TriCore platform. However, special attention has to be paid to the synchronous task-activation mechanism.

Since a task is implemented as an interrupt handler, a prolog is included in the interrupt vector that saves the context of the interrupted task (which is a single instruction on the TriCore), re-enables interrupts, and then jumps to the actual task function. If a task wants to terminate, this corresponds to a simple return-from-interrupt instruction, which restores the previous CPU priority level and implicitly schedules and dispatches the pending control flows in the system. Before the actual return, the context of the interrupted task is restored first.

Synchronous task activation is performed by requesting the corresponding interrupt using the appropriate SRN. Basically, this is compiled to a single store instruction to a memory-mapped register. However, it takes a while until the interrupt request is propagated to the CPU, depending on the current state of the arbitration system. Since an activation of a higher-priority task is supposed to happen *synchronously* in a preemptive system, this activation has to be synchronized. This is done by first disabling interrupts, and by then reading back the request bit in order to synchronize the hardware and software [6]. After that, `nop` instructions are inserted to accommodate for the worst-case latency, which arises if an arbitration round has just begun. The number of `nop` instructions to be inserted is calculated and bounded statically, depending on the number of arbitration rounds and the number of cycles per arbitration round as demanded by the application configuration (i.e., number of tasks and system frequency)[2]. The subsequent enable-interrupts instruction is then the defined, synchronous point of preemption:

```
void ActivateTask(TaskType id)
{
    _disable();
    setr(id); /* set service request flag */
    srr(id); /* read back to sync HW/SW */
    /* worst case: wait for 2 arbitrations */
    nopsForOneArb();
    nopsForOneArb();
    _enable(); /* defined preemption point */
}
```

The same applies to the chaining of another task: The executing task relies on the chained task being executed immediately after it terminates if that new task has the highest priority in the system at that point. As described in Section II-C, interrupts are also disabled before the activation in order to prevent the new task from running until the old one has terminated.

Even when a *lower-priority* task is activated, this situation may require synchronization. Consider, for instance, that directly after the (nonsynchronized) activation of a lower-priority task, the priority level is lowered by terminating the running task. This has to be the defined point for the context switch, and not when the interrupt actually occurs at the CPU. If the activation is not synchronized, a lowest-priority task may execute for a few cycles *after* the termination of the high-priority task and *before* the interrupt dispatches the activated

[2]The timing properties of the TriCore platform that are needed for this calculation are exactly defined by Infineon in an application note [6].

task—which is a clear violation of the specification. Hence, *every* task activation is synchronized with `nop` timing as described above, independently of its priority.

All of the cases described in this section where interrupts need to be suspended temporarily for synchronization purposes only disable them for a *short* and *bounded* amount of time. That way, that time can be accounted for during the schedulability and latency analysis of the whole real-time system. The number of introduced `nop` instructions varies between 8 and 22, depending on the configuration, and is effectively time when the CPU cannot do useful work (although the interrupt system is performing the priority arbitration during that time). However, this is a small price to pay compared to the overhead of a traditional, software-based scheduler implementation (see also Section IV).

Note that after an interrupt request has been triggered, its source—a periphery device or the CPU itself—and the requested type of control flow—task, ISR, or callback—is completely oblivious to the CPU; it simply and automatically dispatches the corresponding control flow if the current CPU priority is below the requested priority.

### C. Application Configuration and System Generation

Since the system is statically configured and tailored to the needs of the application, this information can be used to generate static dispatching code that is highly optimized (see Figure 4). As the configuration describes the mapping from task IDs to interrupt sources, the essence of the `ActivateTask()` implementation (i.e., its subfunction `setr()`), for instance, is an `if–else` cascade that sets the request bit in the appropriate SRN depending on the task-ID parameter, which is also its priority. This implementation and the corresponding application calls can be statically analyzed and optimized by the compiler, resulting in an inlined piece of code consisting of a single instruction—namely the one that sets the correct bit. Similar code is generated for querying that bit to see if a task is in ready state, for setting an alarm that activates a specific task upon expiration, and for initializing the SRNs with the request bit already set, depending on the auto-start properties of the corresponding tasks. These implementations are also extremely light-weight since they are subject to the compiler's static analysis.

Furthermore, the interrupt vector table needs to be generated to jump to the correct task functions from the interrupt handlers of the different priorities as configured for the current application.

Additionally, a couple of system-relevant constants are extracted from the application configuration (see also Figure 4):

- The task IDs are set to their configured priorities, and the resource IDs are set to their ceiling priorities depending on the tasks that are configured to potentially acquire them.
- The ceiling priority of the virtual resource `RES_SCHEDULER` is set to the highest priority of all configured tasks.
- The number of needed arbitration cycles is derived from the configured system frequency and the priorities of the configured tasks, and the corresponding `nop` timing for synchronous task activation is calculated.

### D. System Startup

Upon startup of the SLOTH system (here, in `main()`, after the start-up code has initialized the stack, the interrupt vectors, and some platform-specific registers), the interrupt system needs to be initialized accordingly. This boot process basically encompasses the initialization of the SRNs according to the priorities in the application configuration; the corresponding code can easily be generated as described in Section III-C. If the configuration has any tasks declared to be auto-started upon startup, the request bit in the corresponding SRNs is set in addition to the priority. Note that the system is started with a CPU priority level of 0 but with interrupts still disabled; hence, these auto-start task activations will not take effect until interrupts are enabled after the system initialization is complete (see also $t_1$ in Figure 3).

The initial CPU priority level of 0 in `main()` leads to a fallback to that routine whenever there is no ISR or task ready to be scheduled—otherwise, the pending priority is greater than 0. Thus, appropriate idling action can be taken in an infinite loop in `main()`, putting the microcontroller unit to sleep or in a low-power mode until an interrupt (representing a control flow ready to be dispatched) requires servicing (see also $t_7$ and $t_8$ in Figure 3).

Additional initialization of the general-purpose timer array and the I/O-line–sharing unit of the Tricore is needed if the application uses alarms to activate tasks or execute callbacks.

## IV. EVALUATION

Since the design of our SLOTH system aims at making more use of existing hardware features than other operating systems, the software implementation is accordingly very concise.

### A. Lines of Code

The whole system implementing the OSEK conformance class BCC1 for the TriCore-TC1796 board as described above takes less than 200 source lines of code to be implemented[3]. This number includes code that is generated from the application configuration (see also Figure 4) with one instance per task, resource, and alarm configured; additional code for more instances is similar and adds to the number of lines of code, but not to its complexity. The start-up code for the platform is not included in those 200 lines of code; it was basically taken as supplied by the compiler (`tricore-gcc` by HighTec; programmed in assembly).

### B. Memory Footprint

Due to the concise system code base, the resulting footprint of the compiled system image is also small; the kernel implementing the conformance class BCC1 takes about 700

[3]Logical, semicolon-terminated lines; measured with CCCC [9], version 3.pre84.

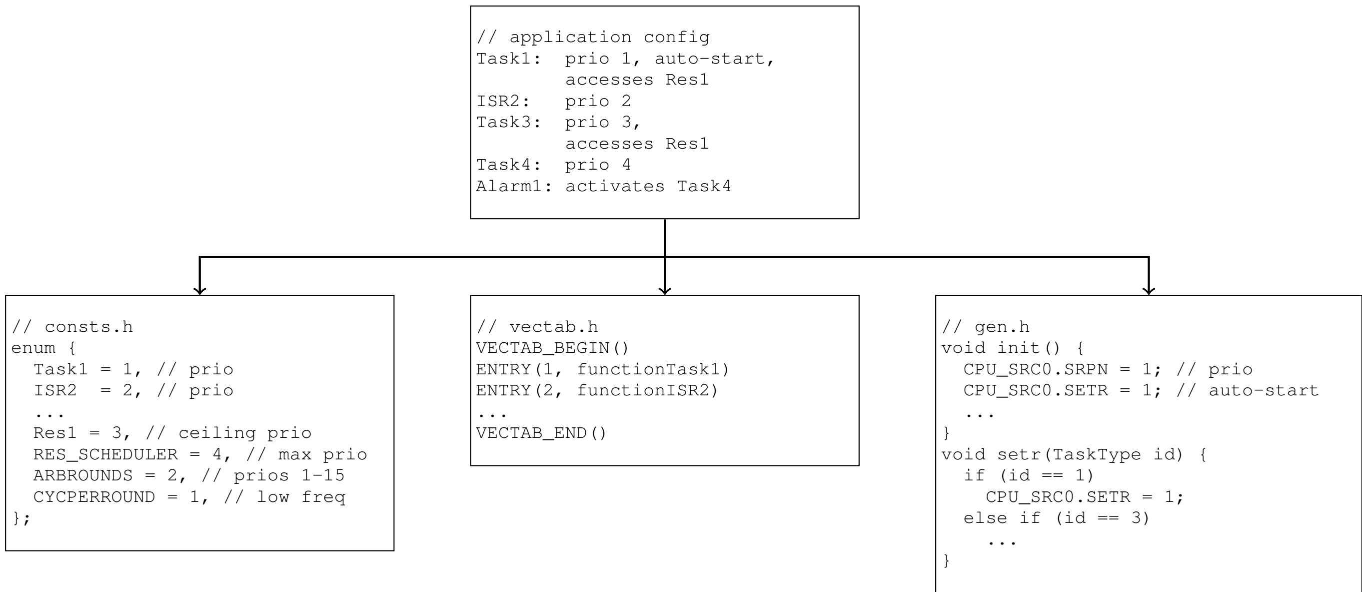


Fig. 4. SLOTH application configuration and system generation.

bytes[4]. This number again reflects the whole kernel with one task, resource, and alarm instance; additional instances can add to the memory footprint because additional interrupt handlers in the vector table are needed for additional tasks, for instance. The compiled start-up code as provided by the compiler takes up an additional 1,000 bytes, which can be reduced to about 500 bytes by tailoring its initialization functionality to the one actually needed by SLOTH.

Note that due to the system's hardware proximity, most system calls are very short and therefore subject to function inlining. Consider, for instance, the `setr()` function (see generated code in Figure 4), which is the essence of the system call `ActivateTask()` (see implementation sketch in Section II-C). Since in many static applications, the system-call parameter is constant at compile time, the dispatching through the `if–else` cascade can be statically optimized by the compiler. The result is a single store instruction to the corresponding memory-mapped register (without the following `nop` synchronization). Additionally, the functionality of the operating system is tailored to the application's needs by excluding system functions that are not referenced by the application; this is done through function-level–linking support by the compiler and linker.

### C. Execution Performance

In order to assess the quantitative effects of our SLOTH approach on the operating-system kernel, we have performed an analysis of run times of selected scenarios in a preemptive system with the features of the OSEK conformance class BCC1 (i.e., without events, without multiple tasks per priority, and without multiple activations). The selected scenarios encompass those system calls that are implemented differently in SLOTH because of its hardware-based nature. The other system calls will have similar performance as in a traditional, software-based kernel, as well as the application itself. The evaluated scenarios include:

1) Synchronously activating a task of *lower* priority, does *not* lead to dispatching: execution time of the `ActivateTask()` system service.
2) Synchronously activating a task of *higher* priority, *does* lead to dispatching: execution time from the point before `ActivateTask()` to the first user instruction of the activated task.
3) Terminating a task and returning to the previously running task: execution time from the point before `TerminateTask()` to the point after the task was dispatched.
4) Chaining a task: execution time from the point before `ChainTask()` to the first user instruction of the chained task.
5) Acquiring a resource: execution time of the `GetResource()` system service.
6) Releasing a resource *without* inducing another task to be dispatched: execution time of the `ReleaseResource()` system service.
7) Releasing a resource *with* inducing another task to be dispatched: execution time from the point before `ReleaseResource()` to the first user instruction of the dispatched task.

We have evaluated all of those scenarios with two different interrupt-system configurations that reflect the best case and the worst case regarding the interrupt-arbitration latency on the TriCore platform (see also Section III-A):

A) Best case (minimum number of arbitration cycles): 1 arbitration round (suitable for up to 3 interrupt priorities), 1 bus cycle per arbitration round (only good for lower

[4] Compiled with `tricore-gcc` by HighTec, version 3.4.5, with `-O3` optimizations.

| | 1) Act() w/o dispatch | 2) Act() w/ dispatch | 3) Term() w/ dispatch | 4) Chain() w/ dispatch | 5) GetRes() w/o dispatch | 6) RelRes() w/o dispatch | 7) RelRes() w/ dispatch |
|---|---|---|---|---|---|---|---|
| SLOTH A) (best case) | 34 | 60 | 14 | 67 | 19 | 14 | 36 |
| SLOTH B) (worst case) | 48 | 74 | 14 | 81 | 19 | 14 | 36 |
| CiAO | 75 | 206 | 107 | 139 | 19 | 66 | 204 |

TABLE I

SLOTH BEST-CASE AND WORST-CASE PERFORMANCE IN SELECTED SCENARIOS, COMPARED TO PERFORMANCE USING THE CIAO OS. DEPICTED IS THE NUMBER OF 20-NS CLOCK CYCLES NEEDED TO EXECUTE THE PARTICULAR TEST CASE.

system frequencies).

B) Worst case (maximum number of arbitration cycles); 4 arbitration round (suitable for up to 255 interrupt priorities), 2 bus cycles per arbitration round (also good for high system frequencies).

The measurement results for SLOTH are depicted in Table I[5]. For comparison purposes, we have deployed and measured the same application scenarios on CiAO, a configurable, OSEK-like embedded operating system for which an implementation for the TriCore platform is also available. CiAO has a traditional software scheduler, and its competitive performance compared to other commercial implementations has previously been published [12]. For the CiAO tests, we have configured the operating system to provide the minimal amount of features necessary for the scenarios so that it provides the same capabilities that SLOTH does. Since both CiAO and SLOTH have the same OSEK API, the test applications run are the same.

Because `nop` timing is required in SLOTH for synchronous task dispatching (see also Section III-B), the scenarios 1), 2), and 4) depend on the hardware-arbitration configuration, with the extremes being the best-case configuration A) and the worst-case configuration B). The other scenarios only alter the priority level of the CPU, which is independent of the number of arbitration cycles; hence, the run times for SLOTH are the same for both configurations. Note that scenario 1) differs in the configurations A) and B) although it does not lead to dispatching. This is because, as argued in Section III-B, situations may arise where synchronization is necessary nevertheless.

Compared to CiAO, SLOTH performs equally well or better in all scenarios. Especially the scenarios 2), 3), 4), and 7), all of which include a scheduling and dispatch operation, are significantly faster on SLOTH, which relies on the interrupt system to perform these tasks. Depending on the hardware configuration and whether a new task is activated or a running one terminates, SLOTH only needs between 280 ns and 960 ns for a task switch (including the actual context switch) on a 50-MHz system.

[5]Measurements were performed on a TriCore TC1796B with 50 MHz system frequency and CPU frequency (cycle time of 20 ns). The run times were obtained with a TRACE32 hardware debugging and tracing unit by Lauterbach and averaged over 5,000 iterations each.

## V. DISCUSSION

The SLOTH system design, relying on extensive use of the hardware interrupt system, leads to a small kernel bearing several advantages over traditional kernel designs, with a good range of application fields nevertheless.

In its current shape, SLOTH does not support blocking functionality for threads. It can therefore exploit the resulting strictly stack-based nature to implement its dispatcher using interrupt levels. As can be seen from the results of the evaluation in Section IV, this does not only lead to a concise system *design*, but also to a concise *implementation*. The small kernel code base is very well manageable and therefore maintainable with regard to possible requirement adaptations. Additionally, it is an ideal candidate to verification, a property of utmost importance to many real-time systems of the class targeted by SLOTH.

Furthermore, the evaluation revealed that the memory footprint of a SLOTH implementation is extremely small, which is another important property for the domain of deeply embedded systems, where single superfluous bytes in memory demand can lead to significant overall cost increase. Because of the strictly stacked nature of SLOTH, stack-sharing techniques can be used to reduce the stack part of the application's RAM demand to a minimum; the dispatcher only uses a single interrupt stack from the very beginning of the system startup. Moreover, the increased use of hardware functionality leads to a superior system performance compared to traditional, software-based implementations, which was shown in the evaluation in Section IV.

The fact that the SLOTH design maps control flows that are of different kinds in other systems (e.g., OSEK tasks, category-1 ISRs, category-2 ISRs, and callbacks) to a single abstraction has a major influence on the system conciseness, leading to the advantages described above. Additionally, the system *synchronization*—a major concern in all concurrent systems—is tremendously simplified, because the adjustment of the current CPU priority level is the single measure needed for all kinds of synchronization demands. This includes both demands by the application itself and demands internal to the system to keep its data structures from being corrupted by asynchronous control flows. The application demands are satisfied by raising the CPU priority level to the resource ceiling priority to acquire a resource, by raising it to the highest level of all configured tasks to disable preemption in a critical

section (this is prescribed in OSEK by the special resource `RES_SCHEDULER`), by raising it to the highest level of all category-2 ISRs to implement `SuspendOSInterrupts()`, and by raising it to the highest level of *all* ISRs to implement `SuspendAllInterrupts()`. The system-internal demands to keep the kernel synchronized are implemented by raising the level to the highest priority of all tasks and category-2 ISRs (both of which can access system data structures) configured in a given system.

SLOTH's unified control-flow design also introduces an additional degree of freedom for the system designer, who can decide upon the system's priority space independent of the synchronous/asynchronous nature of the distinct control flows. In other systems, asynchrononous interrupt handlers always have precedence over synchronous, scheduler-managed threads, which leads to a bifid priority space bearing the problem of rate-monotonic priority inversion [4], amongst others[6]. Furthermore, functionality that is described as optional in the specification because of its complexity can be offered along the way. For instance, the OSEK-OS specification says that the participation of category-2 ISRs in the priority ceiling protocol for resources (see Section II-E) is optional. If interrupts and threads are designed the same way like in SLOTH, they can automatically take part in that protocol, allowing for more complex application synchronization possibilities.

Additional types of control flows that were introduced to offer more light-weight alternatives to the traditional threads and ISRs (like OSEK callbacks and category-1 ISRs) are superfluous in SLOTH systems because the offered control-flow type already has a very low overhead to begin with. In fact, SLOTH can offer OSEK tasks and category-2 ISRs at the price of an OSEK callback or category-1 ISR.

Despite its simple design, SLOTH is suitable for the implementation of a wide range of real-time systems. This includes event-triggered systems with fixed priorities, as targeted by the widely-spread OSEK-OS specification, for instance. The missing blocking functionality can be tolerated by many real-world applications, which avoid making use of that feature because of reasons of memory demand (e.g., stack sharing is hampered) and analyzability of the system behavior. SLOTH is suitable to implement the most well-known fixed-priority scheduling algorithms—like the rate-monotonic algorithm [10] and the deadline-monotonic algorithm [11], for instance.

Legacy applications that are programmed using the API described in the OSEK standard can be used with SLOTH without modifications, since SLOTH implements the OSEK specification. The existing application configuration, including task priorities and other properties (also defined by OSEK, in its OSEK implementation language [16]), can also be used unmodified by the SLOTH generator to produce the configuration-dependent code (see also Section III-C). Hence, no porting is needed for an OSEK application to benefit from SLOTH's advantages, and the application programmer can rely on the programming model and abstractions he is used to.

[6]In fact, this problem is the reason why programmers are taught to keep ISRs short. In SLOTH, the ISRs can be long, since they reside in the same priority space as the system tasks.

## VI. Related Work

We are not aware of any work that is really similar to our approach in handling threads as interrupts.

Vice versa, Kleiman and Eykholt [7] proposed to handle *interrupts* as full *threads* so that interrupt handlers can use system services if they need to. They can even wait for an operating-system event and block, leading to the dispatching of another thread that is ready. This model is implemented in the Solaris kernel for desktop and server systems and was adapted by Lohmann et al. for an embedded-system kernel [13]. However, the overhead introduced by their approach leads to interrupt handlers having a performance overhead similar to that of threads, whereas our approach gives threads the (lower) overhead of interrupt handlers.

There are several approaches to aid the operating-system scheduler by using hardware abstractions; however, all of them rely on *customized* hardware. All of those approaches—including cs2 [14], FASTCHART [8], Silicon TRON [15], HW-RTOS [3], and Atalanta [19]—move operating-system functionality to the hardware level by synthesizing special circuits on FPGA boards and offering that functionality on a co-processor–like basis. Our approach, however, is applicable to commodity off-the-shelf hardware.

As previously mentioned, some of the implications of stack-like control-flow scheduling as used in SLOTH (and as prescribed by OSEK BCC1) were investigated by Baker. This includes the possibility for efficient process stack allocation by means of stack sharing and the stack resource policy to avoid priority inversion [1], [2]. Our implementation uses both techniques and can benefit from them.

## VII. Future Work

Our current system design supports the features of the OSEK conformance class BCC1; its main shortcoming is the missing support for blocking by events, which do not fit in with the current stack-oriented design. In order to be compatible to the classes ECC1, ECC2, and BCC2, we plan to carefully sketch a design for event support and support for multiple tasks per priority[7] (see also Figure 1). Both of those features are rather simple to implement in software—which we could do to integrate the functionality in our SLOTH system—, but we aim for a more sophisticated design that preserves the benefits of an operating system implementing threads as interrupts. For instance, the peripheral control processor of the Infineon-TriCore platform can be configured to be the primary service provider for all interrupts. This co-processor could then be used to implement part of that additional functionality by filtering the events and interrupting the main CPU only when needed.

Furthermore, we plan to investigate the applicability to and suitability of other hardware platforms for our SLOTH

[7]This is especially problematic together with multiple activations, since the activation order within a priority class is prescribed to be preserved.

approach. For instance, all modern Intel-x86 systems have an advanced programmable interrupt controller (APIC) available, which can compensate for the interrupt-system shortcomings of the x86 CPU architecture. By programming the I/O APIC accordingly and by using inter-processor interrupts sent through the processor's local APIC, we are positive that we can implement our design on that well-known platform. We also want to investigate what kinds of features hardware platforms have to offer to support our SLOTH concept *in an ideal way*. This includes the analysis of available hardware features and their shortcomings with respect to our approach.

Finally, we want to explore the feasibility to extend our SLOTH design to multiprocessor systems, and we want to investigate whether similar approaches can be used to implement time-triggered systems.

## VIII. SUMMARY

We have presented our SLOTH operating-system design, which uses interrupt handlers as its universal control-flow abstraction—also to implement synchronously activated threads. This model allows for a simple implementation of all major services expected from a statically configured, event-driven operating system, providing the same programming model and interface, which we have shown using the example of the OSEK-OS specification. As a side effect, the resulting unified priority space completely avoids the real-time problem of rate-monotonic priority inversion.

In order to evaluate the properties of the SLOTH design, we have implemented it on the Infineon-TriCore platform. We have shown that the unification of the control flows in the system has a significant impact on the operating system's conciseness in the design, in its implementation code, and in its compiled memory footprint. Furthermore, since SLOTH uses the hardware interrupt system instead of software-implemented routines to schedule the system's control flows, the resulting performance is more than competitive.

In our opinion, the results of our SLOTH work should encourage OS engineers to make better use of the hardware abstractions that a given platform offers. Especially in the domain of *embedded* OSes, where a small footprint and efficient execution are crucial, a small limitation in portability can often be traded for an improvement of those properties.

## REFERENCES


[1] Theodore P. Baker. A stack-based resource allocation policy for realtime processes. In *Proceedings of the 11th International Conference on Real-Time Systems (RTSS '90)*, pages 191–200, Washington, DC, USA, Dec 1990. IEEE Computer Society Press.

[2] Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems Journal*, 3(1):67–99, 1991.

[3] 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, New York, NY, USA, 2006. ACM Press.

[4] 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, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[5] Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *TriCore 1 User's Manual (V1.3.5), Volume 1: Core Architecture*, February 2005.

[6] Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *AP32009, TC17x6/TC17x7 – Safe Cancellation of Service Requests*, July 2008.

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

[8] 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, Jun 1991.

[9] Tim Littlefair. CCCC - C and C++ Code Counter homepage. http://cccc.sourceforge.net/.

[10] 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.

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

[12] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association.

[13] 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)*, New York, NY, USA, 2007. ACM Press.

[14] 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, New York, NY, USA, 2004. ACM Press.

[15] 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, Nov 1995.

[16] OSEK/VDX Group. OSEK implementation language specification 2.5. Technical report, OSEK/VDX Group, 2004. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2009-09-09.

[17] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2011-08-17.

[18] 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, New York, NY, USA, 2009. ACM Press.

[19] 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.