SAFER SLOTH: Efficient, Hardware-Tailored Memory Protection*

Daniel Danner, Rainer Müller,
Wolfgang Schröder-Preikschat, Wanja Hofer, Daniel Lohmann
Friedrich–Alexander–Universität (FAU) Erlangen–Nürnberg, Germany
{danner,raimue,wosch,hofer,lohmann}@cs.fau.de

Abstract—**The goal of the SLOTH family of operating system kernels is to provide a unified priority space to the real-time applications. By automated mapping of tasks to interrupts, we eliminate rate-monotonic priority inversion and increase execution determinism. In its standard implementation, however, SLOTH has been criticized for being unsafe, since interrupt service routines are executed in supervisor mode. SAFER SLOTH mitigates this shortcoming—while keeping the favorable properties of SLOTH—and provides a safe and isolated execution environment for application tasks. Adopting the SLOTH philosophy of embracing and exploiting hardware particularities, its generative approach automatically tailors the system to both the application and the target architecture. We achieve efficient MPU-based memory protection at reduced latency and low performance overhead by leveraging code inlining and compiler optimizations. In comparison to a commercial AUTOSAR OS, SAFER SLOTH achieves speedups between 8x (worst case) and 23x (best case) on kernel latencies while retaining the SLOTH advantages of strict priority obedience, excellent determinism and small memory footprints.**

I. INTRODUCTION AND MOTIVATION

Software implementations for embedded systems are usually judged based on their non-functional properties; besides properties related to efficiency (e.g., memory footprint or event latencies), real-time systems need to be executed in a deterministic way, respecting control flow priorities. In the automotive industry, embedded systems have lately been added an additional non-functional requirement: memory safety. Since in current automobiles several embedded devices by different suppliers are consolidated on a single hardware platform to reduce costs, the embedded operating system with a focus on safety needs to isolate the application tasks from each other. This way, malfunctions in one application do not necessarily bring the whole system down and can be contained, which is especially important in mission-critical systems.

The SLOTH embedded kernels, which implement the automotive OSEK OS interface [20], have proven to excel at deterministic execution by preventing certain kinds of priority inversion, and at efficiency-related non-functional properties [13], [14]. By utilizing the interrupt subsystem of the underlying hardware platform for kernel purposes and by executing application tasks as interrupt handlers, their software parts are really small. This conciseness is reflected in the code size (in a minimal feature-complete configuration, the kernel takes less than 200 LOC), resulting in good certifiability; in its small memory footprint, both code and data (less than 400 bytes of ROM and as few as 8 bytes in RAM); and in its low latencies in processing interrupts and system services (14 clock cycles for a task switch). Additionally, by executing tasks and ISRs in the same priority space—the interrupt priority space—, the execution of high priority tasks cannot be disturbed by low priority ISRs, eliminating the effect of rate-monotonic priority inversion [7] and increasing system predictability and determinism.

Concerning its safety, however, SLOTH has been criticized for executing application code in interrupt handler context, which uses supervisor mode privileges of the hardware. Additionally, AUTOSAR OS [1], the automotive successor standard to OSEK OS, now prescribes isolation to be enforced by the kernel in order to isolate applications from different suppliers running on the same platform. With the SAFER SLOTH design as described in this paper, we show that we can combine the determinism and efficiency properties of a hardware-tailored, interrupt-based kernel with the enforcement of application isolation. By using hardware-based memory protection together with different modes for privilege separation between the applications and the kernel depending on architecture-specific particularities, SAFER SLOTH is able to ensure system safety at minimal overheads.

A. Challenges

The main problem in the design of the SAFER SLOTH system is to guarantee application isolation requirements without sacrificing its deterministic and efficient hardware-centric nature. This includes several conceptual and technical challenges that we need to address:

1) Interrupts and ISRs are traditionally thought of as part of the trusted computing base, which would make SLOTH application tasks—implemented as ISRs internally—also trusted. To ensure system safety, however, we need to be able to isolate SLOTH tasks so that they cannot corrupt other application tasks or the kernel upon malfunctioning.
2) Traditional operating systems enforce isolation by hardware-supported privilege separation and system calls, which are dispatched by a supervisor-mode trap handler in the kernel. Part of SLOTH's run-time efficiency, however, is due to the fact that it inlines its system services into the application code to allow extensive compiler optimizations and dead-code elimination; these optimizations are not possible when using traditional system call traps. Thus, we need to find additional, resource-efficient ways to

*This work was partly supported by the German Research Foundation (DFG) under grants no. SCHR 603/8-1, LO 1719/1-2, and SFB/TR 89 (project C1).

Appeared in: Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium (RTAS '14), pages 37–47, IEEE Computer Society, 2014, ISBN 978-1-4799-4691-4

implement privilege separation in SLOTH by utilizing hardware properties.
3) The original and motivating property of SLOTH, preventing rate-monotonic priority inversion by using a unified priority space for all control flows, has to be preserved in a safe variant of the kernel. Only through constructive prevention of this kind of priority inversion does SLOTH reach its excellent determinism and therefore schedulability of real-time application control flows—properties that are also of utmost importance in safety-critical systems.

B. *Contributions*

The design of our SAFER SLOTH system addresses these challenges; in this paper, we provide the following contributions:

- We present an analysis of memory and privilege isolation requirements in embedded systems with the SLOTH architecture in mind (see Section IV).
- For the SAFER SLOTH system, we develop a sophisticated design that uses fine-grained application and system configuration options together with a model of hardware protection particularities to generate a memory-protected and privilege-separated kernel–application binary that is tailored to the application's safety requirements and the hardware properties (see Section V).
- Our reference implementation of SAFER SLOTH for the Infineon TriCore platform achieves small and constant overheads with 15 to 93 additional cycles per system service invocation (see Section VII).
- The applicability to other platforms is discussed and demonstrated by a port of the system to the ARM Cortex-M3 (see Section VIII). By following the SLOTH design principle of utilizing hardware peculiarities for kernel purposes on both implementations, SAFER SLOTH provides isolation while preserving a universal control flow abstraction in a unified priority space (see Section VI).

II. SAFETY MODEL

The main focus of a real-time operating system is to ensure that applications can fulfill their real-time requirements through timely execution. In order to additionally guarantee certain safety properties, the kernel has to introduce isolation boundaries between applications. These boundaries can be implemented by a memory protection unit (MPU) on the hardware platform, which, in contrast to a full-blown memory management unit (MMU), enforces access rights to physical memory segments but does not provide virtual address spaces.

With respect to the enforced access rights, SAFER SLOTH aims to protect write accesses to data but not execute accesses to code segments. We adopt this model of *protecting the data but not the code* from the automotive industry; the AUTOSAR OS standard [1] specifies this kind of isolation mechanism to focus on *safety, but not security*. We do, however, extend the protection semantics proposed in AUTOSAR OS by the possibility for sharing data between domains—a drawback that has been criticized by AUTOSAR OS users. Our model assigns application data to an intermediate abstraction called *protection domains*, which can then be configured to be allowed access by one or more tasks and ISRs. This way, both task-private data and shared data can be modeled by the application to be isolated from other control flows.

III. BACKGROUND

A. *Sloth Revisited*

SAFER SLOTH extends the design and implementation of the original SLOTH kernel, which is described in [13]. The main design idea in the SLOTH kernel is to have user tasks run as interrupt handlers internally—transparent to the application. By using interrupt sources and interrupt priorities for application tasks, the SLOTH kernel can rely on the interrupt subsystem of the commodity hardware to do the fixed-priority scheduling and dispatching for it, leaving the software part of the kernel very small.

From an interface point of view, SLOTH implements the OSEK OS API specification [20], the compatible predecessor of AUTOSAR OS [1], both developed by the automotive industry. OSEK (and, therefore, SLOTH as well) is a statically configured system; its system calls are also called with statically known parameters, such as task IDs by the application. In its BCC1 variant, which we use as a basis for the discussion of memory protection in this paper, OSEK offers basic tasks, which run to completion (i.e., they cannot block) and are scheduled based on their statically configured priorities. The OSEK system service to activate a task is implemented in SLOTH by setting the request bit in the interrupt source corresponding to the given task. Terminating the running task is implemented by simply executing a return-from-interrupt instruction. Both services trigger a priority arbitration of the interrupt subsystem, which will then schedule and dispatch the task with the next-highest priority automatically. A small task prologue at the beginning of each interrupt handler is responsible for saving the comprehensive context of the interrupted task, which will be restored on task termination.

B. *Hardware Requirements*

SLOTH as well as SAFER SLOTH run on commodity off-the-shelf microcontroller systems; however, the hardware-centric nature of their designs poses certain requirements on the implementation to be feasible:

1) Both SLOTH and SAFER SLOTH need the platform to provide as many interrupt priorities and interrupt sources as there are application tasks and ISRs in the system.
2) Both SLOTH and SAFER SLOTH need the platform to provide means to trigger an interrupt manually from within software—either through a special instruction or through memory-mapped hardware registers.
3) SAFER SLOTH additionally requires a memory protection unit (MPU) on the platform. Depending on the application configuration (see Section V), the implementation requires the platform to offer different privilege levels (typically user and supervisor mode) and a sufficient number of memory ranges with configurable access rights.

Many 32-bit microcontroller platforms fulfill those requirements, including the two target platforms we implemented SAFER SLOTH on; the ARM Cortex-M3 offers up to 256 priority levels and an MPU with 2 privilege levels and 8 memory ranges; the Infineon TriCore TC1796, offers 256 priority levels with almost as many IRQ sources with memory-mapped registers, plus an MPU with 3 privilege levels and 2 protection sets with 4 memory ranges each.

IV. MEMORY PROTECTION IN EMBEDDED SYSTEMS

From a functional perspective, our primary goal in SAFER SLOTH is to isolate application tasks in terms of write access to memory. In the area of embedded real-time systems, this is commonly achieved by a memory protection unit (MPU) provided by the hardware platform. To enforce memory protection boundaries, an MPU transparently compares the address of each memory access to a configured set of ranges that are usually stored in dedicated hardware registers. The memory ranges can be organized in multiple sets of ranges to allow faster switching between pre-programmed configurations. A memory range is defined by a lower and upper bound and attributes that allow or deny reading, writing, and executing code from or to the enclosed memory region. Other approaches such as language-based software safety do exist; in the automotive domain, however, MPU-based protection as specified by AUTOSAR OS is the state of the art (see discussion in Section VIII), which is why SAFER SLOTH focuses on this type of memory protection.

In order for the *horizontal isolation* between application tasks enforced by the MPU to be effective, the isolated control flows must be prohibited from manipulating the state of the MPU itself; otherwise, there would be no actual safety. The common, straightforward implementation of this *vertical isolation* between the application and the kernel is to demote application code to a lower privilege level at which the platform guarantees that no reconfiguring or disabling of the MPU can take place. Whenever the kernel, being the trusted computing base, then needs to change the MPU configuration, some mechanism—usually a trap used to dispatch the requested system call—is used to temporarily equip it with the suitable high privilege level.

In the SLOTH kernel, however, all control flows are implemented as interrupt service routines, which usually implies execution at the highest privilege level. On the one hand, it is perfectly possible inside an ISR to switch to a lower privilege level before executing the application code and then use a traditional system call mechanism for kernel operations. On the other hand, however, such a mechanism often comes with a high overhead and, more importantly, interferes with a core advantage of SLOTH, which is the inlining of system services and the subsequent optimizations based on static knowledge in the code. For this reason, we examine how we can exploit the properties and particularities of the underlying hardware in such a way that we can find alternatives to the traditional approach of executing user code at lower privileges and using traps for kernel operations, while still providing the same degree of safety and also benefit from less run-time overhead.

V. SAFER SLOTH DESIGN

The SAFER SLOTH framework shown in Figure 1 follows the design of the original SLOTH generation architecture, which already achieves highly optimized code paths by static analysis. Since the whole system is configured statically, the configuration is used to tailor the memory protection mechanisms to the application requirements and, most importantly, to a particular hardware platform.

A. *Hardware Architecture Model*

As SLOTH tries to exploit the features of the target hardware in an efficient way, it is essential to customize the system to the needs of the application as well as to the peculiarities of the hardware. Different protection modes in SAFER SLOTH have different requirements on the specific properties of the hardware that also influence the applicability. This includes the number of memory ranges and their organizational layout in the MPU, support for different privilege levels, and whether memory protection is active in all of these levels.

As shown in the analysis in Section IV, hardware support is required to implement memory protection in order to isolate tasks from each other. For the purpose of SAFER SLOTH, only the memory protection attributes for data access are relevant, as we focus on denying data write accesses outside the configured protection domains to isolate tasks from each other according to our safety model presented in Section II, which is adopted from AUTOSAR.

For proper isolation in SAFER SLOTH, tasks need to have a lower privilege level, as they should not be allowed to execute privileged instructions, which could potentially re-program or disable memory protection. Although this separation of privileges is required to prohibit access to the MPU configuration, in a statically configured system this can be achieved by different means than hardware-enforced privilege levels. If the instructions to re-program the MPU can be unambiguously identified in the machine code, a static analysis on the compiled binary can detect them outside of system services. Therefore, if the MPU is active in supervisor mode, the costly traps can be avoided, while ensuring execution of privileged operations by the kernel only without additional measures at run time.

Depending on the hardware, the MPU is usually configured either by dedicated privileged instructions or through memory-mapped registers. Dedicated instructions are easy to identify in the machine code, while address calculations to memory-mapped registers are harder to detect. We discuss further implications of such a protection mode in Section VIII.

B. SAFER SLOTH *Protection Modes*

The original SLOTH implements an `unsafe` mode, without horizontal isolation between tasks. For SAFER SLOTH, we present two modes of MPU-based memory protection that introduce different approaches for the isolation of tasks.

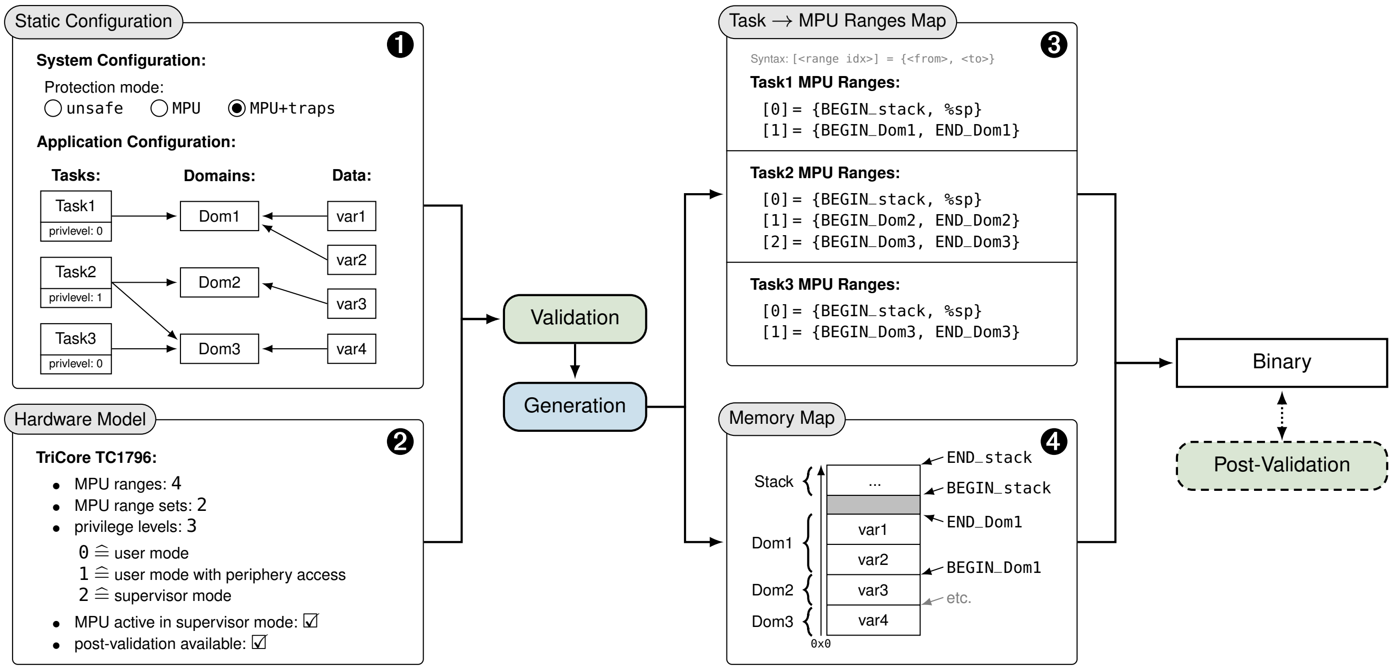


Fig. 1: SAFER SLOTH architecture: The application goes through several steps starting at the configuration, followed by the validation and system generation, to the compiled binary. Depending on the mode, a post-validation step is required.

In the `MPU+traps` mode, the separation of privileges is achieved by support of hardware privilege levels that enforce the boundary between the privileged kernel and the unprivileged tasks at run time. As all tasks are dispatched as interrupt handlers, the privilege level will be lowered in a small prologue in the interrupt service routine before the actual task function is called. System call instructions trap into the kernel, where the system services are executed at the highest hardware privilege level.

Furthermore, we propose an `MPU` mode for SAFER SLOTH, in which the whole system runs in a privileged mode including the tasks provided by the application. In this mode the costly switches between different privilege levels can be avoided, while memory accesses conducted by the tasks are still isolated within their defined boundaries.

C. Application Configuration

In order to support the different protection modes, the system configuration ❶ in Figure 1, specified by the application developer, controls whether tasks run with memory protection enabled at all and whether privileges are separated by running tasks in an unprivileged user mode.

For memory protection in SAFER SLOTH, the application configuration associates tasks to one or more protection domains, which define the boundaries of data write accesses allowed at run time later on. For this purpose, the application programmer assigns each variable to a specific domain either in the configuration or by adding annotations directly in the source code:

```
int var1 DOMAIN(Dom1);
int var2 DOMAIN(Dom1);
int var3 DOMAIN(Dom2);
int var4 DOMAIN(Dom3);
```

In addition to what is specified by AUTOSAR, the domains may be allocated to multiple tasks, which allows them to share data using the contained variables. Depending on the underlying hardware platform, the number of task–domain assignments is limited by the number of MPU ranges provided. For instance, if there are four MPU ranges available, each task may have at most three domains assigned, as the fourth range is used to protect the stack. Note that this does not limit the total number of tasks, domains, or variables in a system, but only constrains the complexity and fragmentation of data sharing between tasks. In the example shown in Figure 1 at most three MPU ranges are used at the same time, as seen in the range map of `Task2` shown at ❸.

Particularly if `MPU+traps` mode is selected in ❶, application code is executed at a lower hardware privilege level than the kernel. The second input box ❷ in Figure 1 reflects the hardware-specific properties, which includes the available privilege levels that can be configured on a per-task basis.

D. Validation

The validation step in Figure 1 analyzes the applicability of the chosen configuration for the target hardware platform as specified in the hardware model.

Additionally, to avoid compromising safety in `MPU` mode, the use of privileged instructions reprogramming the MPU or disabling memory protection altogether has to be limited to system services only. As all tasks run with supervisor privileges in this mode, the hardware itself does not restrict execution of privileged instructions. Thus, additional measures need to be deployed in the form of a post-validation step.

This post-validation step identifies and reports use of forbidden privileged instructions in the application code outside of the inlined kernel system services after compilation of the binary. The identification method for privileged instructions in the compiled binary using an offline analysis is dependent on the hardware architecture and instruction set. Later modifications of the code must not be allowed by the hardware platform by either an inherent design feature as loading the code from ROM, or with memory protection preventing writes to the code segment.

E. Generation and Compilation

After the inputs have been validated and the applicability of the configuration has been checked, the generation step tailors the system to the needs of the application and to the target platform.

The assignment of domains to tasks is represented by references to the lower and upper bound of each domain for each task. Implicitly, one domain is always reserved for accesses to the stack. Based on this mapping ❸ in Figure 1, a linker script is generated that is used to link the compiled code to the final binary. The annotated variables from the source code are grouped by their associated domains and placed into the corresponding memory locations forming the addresses to be used as lower and upper bounds of the domains, as shown as an example in the memory map ❹ in Figure 1.

Furthermore, switches between the different protection domains only take place at certain transitions. For the dispatching of new tasks, the lower and upper bounds and access rights of each memory protection domain of the dispatched task have to be configured in the MPU. This code modifying the memory protection registers will be generated to adhere to the overall static configuration of the system. As it does not need to make dynamic decisions, it will be generated once per task and will reside completely in ROM.

VI. Implementation

We have implemented Safer Sloth on two platforms, the Infineon TriCore TC1796—which is widely used in the automotive industry—and the ARM Cortex-M3. We provide evaluation results for both platforms in VII, but due to space constraints, the presentation of implementation details in this section is limited to the TriCore architecture.

The TC1796 offers three privilege levels, namely one supervisor mode and two user modes, whereas the slightly more privileged level `User1` still allows access to the periphery and the lowest level `User0` does not. The chip also comes with an MPU that can switch between two protection configurations (referred to as *sets*) with range registers to specify data access rights to four different memory ranges.

The hardware requirements for Safer Sloth established in Section III-B are therefore satisfied. Furthermore, the MPU operates without regarding the current privilege level, and thus enforces its limits within supervisor mode as well. This is an essential prerequisite for the implementation of `MPU` mode to be possible on this platform.

A. Horizontal Isolation of Tasks

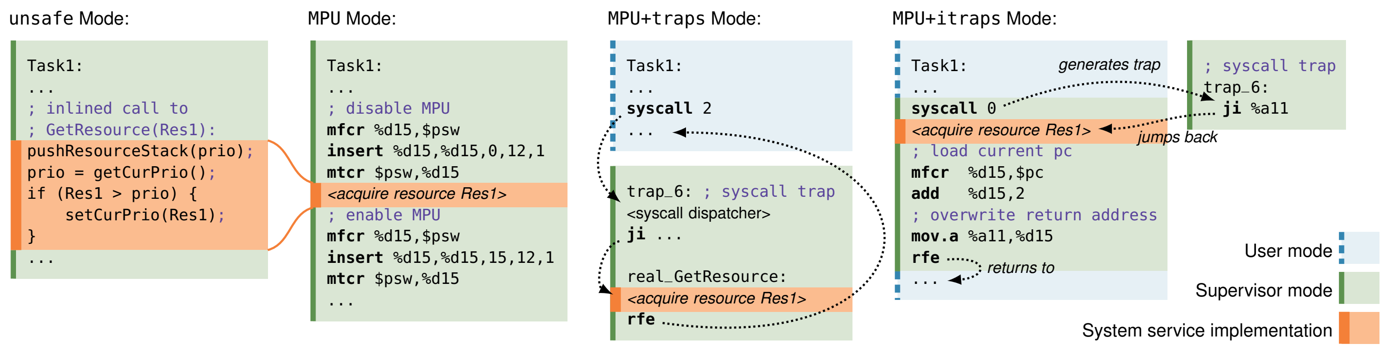
In all modes, the horizontal isolation is enforced by configuring the MPU in such a way that write accesses of a particular task are limited to the memory domains explicitly assigned to it. This means that the range registers of the MPU that define the permitted memory areas need to be updated whenever a different task gets dispatched. As a consequence, the MPU must also be deactivated whenever an application invokes some kind of kernel operation that involves write access to kernel state residing in memory.

For setting up task-specific MPU configurations when a task is dispatched, both the task prologues and the termination routines are extended to reconfigure the MPU. Since each task prologue is specifically generated for a single task, the appropriate range register values are statically known at compile time. Here, the introduced overhead amounts to the instructions required to load the MPU configuration, which corresponds to the number of domains assigned to this particular task. Equally, when terminating a task, the MPU configuration of the previously preempted—now restored—task needs to be restored as well. In contrast to the individual task prologue, it is not statically known at this point which task this will be. As a result, the code inserted into the termination routine needs to perform a dynamic decision at run time by looking up the MPU configuration belonging to the restored task ID from ROM.

Furthermore, basic tasks are considered *run-to-completion* in OSEK. Sloth takes advantage of that fact by having tasks share a single stack for the entire application. Due to this, however, Safer Sloth can not simply consider a certain stack region to be a static part of a task's MPU configuration in order to isolate tasks from stacks of other tasks. Instead, the MPU range set up for granting a task access to its own stack—therefore prohibiting access to foreign stacks—is determined dynamically whenever a task is newly dispatched. This is realized in the task prologue by updating the upper limit of the MPU region to the current stack pointer, prohibiting access to the stack frames of the preempted task. The lower limit remains unchanged to keep each task unrestricted in its maximum stack usage. Upon termination of a task, the previous upper limit is restored to the stack region in the MPU configuration.

B. Vertical Isolation between Kernel and Tasks

MPU reconfiguration is also required for any system services that involve writing to kernel memory. These are extended with instructions that switch the MPU to the second *set* at the beginning of the system service. This second set is initialized once during startup and configured to allow access to the kernel state. Correspondingly, code at the end of the system service is inserted to switch back to the previous set and thereby re-enabling the protection. Figure 2 shows a representation



```
Task1:
...
; inlined call to
; GetResource(Res1):
pushResourceStack(prio);
prio = getCurPrio();
if (Res1 > prio) {
    setCurPrio(Res1);
}
...
```

```
Task1:
...
; disable MPU
mfcr %d15,$psw
insert %d15,%d15,0,12,1
mtcr $psw,%d15
<acquire resource Res1>
; enable MPU
mfcr %d15,$psw
insert %d15,%d15,15,12,1
mtcr $psw,%d15
...
```

```
Task1:
...
syscall 2
...

trap_6: ; syscall trap
<syscall dispatcher>
ji ...

real_GetResource:
<acquire resource Res1>
rfe
```

```
Task1:
...
syscall 0
<acquire resource Res1>
; load current pc
mfcr   %d15,$pc
add    %d15,2
; overwrite return address
mov.a %a11,%d15
rfe
...
```

```
; syscall trap
trap_6:
  ji %a11
```

Fig. 2: Comparison of an example system service call in the four different protection modes of SAFER SLOTH.

of a compiled task named `Task1` which contains a call to the OSEK system service `GetResource()` for obtaining the resource `Res1`:

```
TASK(Task1) {
    ...
    GetResource(Res1);
    ...
}
```

The `unsafe` listing in Figure 2 shows how this call is inlined in the original SLOTH implementation with no memory protection enabled.

`MPU` mode—Constructive Vertical Isolation. As illustrated by second-to-left in Figure 2, `MPU` mode inserts instructions for disabling and enabling the MPU around the actual system service implementation. Note that this does not affect the compiler's ability to inline and optimize the entire function in the application code.

In the case of the `GetResource()` call in this example, disabling the MPU is mandatory because the system service requires to write to the kernel state in memory. Due to the design of SLOTH, however, this does not necessarily apply to all kinds of system services. For example, the `ActivateTask()` implementation on the TC1796 merely disables interrupts, triggers the appropriate IRQ, executes a few no-operations to ensure a completed interrupt arbitration, and then enables interrupts. No memory access is involved; therefore, this system service remains exactly the same as when memory protection is disabled in the system configuration.

Since in `MPU` mode, the application tasks still execute with full privileges, they are not prohibited from accidentally disabling or reconfiguring the MPU on their own, which would compromise the safety of the system. Fortunately, on the TriCore platform, the instructions for manipulating the MPU configuration consist of dedicated opcodes. This allows the detection of unintended, safety-compromising instructions in the application code within the post-validation step of the system, which is further discussed in Section VIII.

C. Vertical Isolation by Hardware Privilege Levels

In contrast to `MPU` mode, the modes `MPU+traps` and `MPU+itraps` enforce vertical isolation by executing application tasks at a lower privilege level, at which disabling or reconfiguring the MPU is prohibited by the hardware. Since all control flows in SLOTH start out being an interrupt service routine, they normally run in the fully privileged supervisor mode. In order to achieve the execution at a lower privilege level, the task prologue—a small piece of code which is responsible for, amongst others, saving the context of the preempted task—is extended so that the supervisor privileges are explicitly dropped before entering the actual application code of the task. On the TriCore platform, this is implemented by resetting a single flag to zero. Depending on the application configuration for this task, the selected privilege level is either `User0` or `User1` for the remaining execution of the task.

With tasks now running at a low privilege level, it is not possible anymore for system services to be directly invoked and disable the MPU as they do in `MPU` mode. Instead, at the transition from the task into the kernel it is required to elevate the privilege level to allow the system services to execute privileged instructions. System services therefore need to perform a privilege switch by generating a trap, for which we present two alternative implementations. The first one matches the traditional approach of using a trap handler that dispatches the desired system call based on a parameter passed along with the trap. As a second variant, we devised a new mechanism called *inline traps* (or `MPU+itraps` mode), which implements a system call trap that preserves the advantages of inlining system services into the application code.

`MPU+traps`—Traditional System Calls. This mode employs a traditional scheme using a trap handler and a table of system services for dispatching. On the TriCore platform, the `syscall` instruction causes a dedicated trap handler to be invoked, to which parameters can be passed in general purpose registers. The trap handler is implicitly executed in supervisor mode and, thus, the code of the real system service can be executed at this privilege level. The execution sequence of this implementation is shown in the `MPU+traps` mode in Figure 2. Incidentally, when entering a trap handler, the MPU automatically switches to the kernel domain set (and back to the previous set when leaving), eliminating the need for explicitly changing the set as in `MPU` mode.

By using a dispatch table for the different system calls, the invocation of a system service is bound late to their corresponding implementation. This inhibits the elimination of dead code, which is a strong feature of the SLOTH system. The resulting code paths in the system services are taken on conditions only known at run time. Although some parameters are statically known at compile time, optimizations by the compiler are not possible in this mode.

`MPU+itraps`—Inline Traps. The disadvantages of using a traditional system call interface led to the conception of an alternative, which interleaves the trap mechanism with the inlined code of the system service. Due to their shortness and the statically known parameters, most system services benefit greatly from code inlining made possible with this approach. The resulting machine code and the execution sequence as illustrated in the rightmost section of Figure 2 is as follows.

The user code begins the system call using the usual `syscall` instruction, which transfers control to the trap handler. At the entry of the trap handler, we instantly jump back to the return address of the trap stored in the `%a11` register. Note that, unlike a return-from-exception instruction, this does not leave the trap context but retains the supervisor privileges and the kernel domain set as active in the MPU. At the return address, the machine code following the `syscall` instruction is simply the inlined implementation of the actual system service. As the insertion of the `syscall` instruction and the implied manipulation of the control flow is transparent to the C compiler, it correctly assumes the same context in terms of register contents for the inlined code, which is not disturbed by the short detour through the trap handler. The system service implementation can therefore fully benefit from compiler optimizations such as the resolution of statically known expressions and dead code elimination.

Since the execution of the system service takes place in the call frame of the trap handler, it needs to be terminated with an `rfe` return-from-exception instruction inserted at its end. However, the return address recorded in `%a11` still points to the *beginning* of the sequence. For `rfe` to work as expected and continue executing at the next instruction, the return address is patched first by loading the current program counter (adjusted by 2) into `%a11`. The `rfe` instruction then eventually performs the exit from the trap handler context and resumes execution at the next instruction of the application code.

This setup essentially transforms the `syscall` instruction into an acquire-supervisor-privileges instruction. Due to the `syscall` instruction being inserted as an `asm volatile` statement, it effectively serves as a barrier between memory access of the application code and the system service code, preventing any reordering of unprivileged code into the privileged execution of the system service. Nevertheless, to ensure the system safety of this mode, additional post-validation is necessary to rule out false use of the `syscall` instruction. The implications of this mode are discussed in Section VIII.

VII. EVALUATION

For evaluating SAFER SLOTH, we want to assess both the impact it has in terms of additional overhead in comparison to the original, unprotected SLOTH kernel and also compare these observations to the effects that enabling memory protection in a traditionally designed system has on their respective performance. In this section, we first focus on the reference implementation of SAFER SLOTH for the Infineon TriCore TC1796 and will then provide a brief presentation of the measurement results obtained on the ARM Cortex-M3.

A. Evaluation Setup

We carried out measurements on a TC1796 clocked at 50 Mhz, using a hardware trace unit by Lauterbach, which allows to analyze the precise number of cycles spent between two given lines of code. This way, we measured benchmarks repeatedly at least 1,000 times and we could observe that the deviations from the average have been negligible in all cases. The final results were calculated by averaging all measured intervals within the sample.

We devised a small, platform-independent example application to be used as a test and measure environment; it consists of three tasks and two shared resources. The user code performs a sequence of interactions between tasks so that all system services and transitions relevant to SAFER SLOTH take place and can be measured. In cases where a system service possibly—depending on the preemption circumstances—entails the dispatching of another task, both scenarios were measured separately. If a dispatch is included in the test case, the trace unit was set up to measure up to the first instruction of the user code belonging to the dispatched task. For new activations—as distinguished from resuming a preempted task—, this means that the entire prologue of the new task is part of the measured interval.

B. Quantitative Evaluation of Overhead Composition

Table I(a) provides a comprehensive overview of the measurement results obtained for SAFER SLOTH on the TC1796. For each of the seven test cases, the first column lists the baseline overhead for the original SLOTH kernel configured in `unsafe` mode. The following three columns contain the additional overhead that is introduced by enabling protection by each of the three modes, one at a time.

Comparing the delta values of the various system services separately in each column reveals that the impact varies strongly with the system service. Qualitatively, these differences persist throughout the different protection modes; for instance, `ChainTask()` exhibits the highest delta in all modes. This can be attributed to the different actions introduced in each test case for controlling the MPU. As an example of low overhead, the `ActivateTask()` call without dispatch is not affected at all in `MPU` mode, since no additional measure was inserted here (see Section VI-A). In contrast, the 80 additional cycles for `ChainTask()` with dispatch in `MPU` mode reflect the fact that this test case covers both the restoration of the previous task's range set in the MPU *and* the MPU reconfiguration that takes place in the prologue of the newly dispatched task. This measurement is also consistent with the results for the other three dispatching calls, which each yield half the

	Protection Mode			
a) **Safer Sloth**	`unsafe`	`MPU`	`MPU+traps`	`MPU+itraps`
`ActivateTask()` without dispatch	36	+0	+28	+15
`ActivateTask()` with dispatch	57	+39	+72	+54
`TerminateTask()` with dispatch	14	+41	+65	+57
`ChainTask()` with dispatch	64	+80	+108	+93
`GetResource()`	14	+6	+22	+15
`ReleaseResource()` without dispatch	9	+6	+24	+18
`ReleaseResource()` with dispatch	30	+41	+64	+60
b) **Commercial AUTOSAR OS**	`unsafe`	`MPU+traps`		
`ActivateTask()` without dispatch	459	+39		
`ActivateTask()` with dispatch	768	+273		
`TerminateTask()` with dispatch	536	+192		
`ChainTask()` with dispatch	856	+273		
`GetResource()`	393	+40		
`ReleaseResource()` without dispatch	342	+0		
`ReleaseResource()` with dispatch	740	+235		

TABLE I: SAFER SLOTH (a) and commercial AUTOSAR OS (b) measurements on the Infineon TriCore TC1796: The first column provides the baseline of the regular SLOTH in `unsafe` mode (in number of cycles; all bars drawn to the same scale). The other three columns contain the delta values indicating the additional overhead introduced by each protection mode. For the commercial OS, only an equivalent of `MPU+traps` mode is available.

amount of additional cycles. The test cases `GetResource()` and `ReleaseResource()` without dispatch also stand out for adding only 6 cycles in `MPU` mode, which correspond to the comparatively cheap disabling and re-enabling of the MPU.

The results for the `MPU+itraps` mode show that our *inline traps* mechanism yields a consistent improvement over the traditional trap implementation in the `MPU+traps` mode.

C. Comparison to Commercial AUTOSAR OS

We also measured a widely used commercial AUTOSAR OS implementation[1] for the Infineon TriCore to compare it to SAFER SLOTH. This system is an implementation of the AUTOSAR OS standard, which specifies a safety model that is similar to the one we use in SAFER SLOTH. We used the same application and measurement setup as in SAFER SLOTH.

Table I(b) lists the measurement results, contrasting a system configuration with disabled memory protection against one with enabled memory protection. For simple comparison, the increases in overhead are depicted with bars at the same scale as in Table I(a). For all but one test case, both the baseline costs and their increase when enabling protection surpasses what we achieve in SAFER SLOTH. The only exception—with zero additional cost—is the `ReleaseResource()` call without dispatch. Since we do not have access to the system's source code, it is difficult to speculate about the cause for this. It is remarkable, though, that already the unsafe variant of the commercial system entails a significantly higher run time of system services than any of the SAFER SLOTH variants.

[1]tresosECU by Elektrobit Automotive, used by VW and BMW in their standard cores, amongst others.

D. Evaluation on the ARM Cortex-M3

Additionally to the measurements carried out on the TC1796, we also performed measurements on the ARM Cortex-M3. Unfortunately, no commercial AUTOSAR OS implementation for this platform was available to us, therefore limiting this evaluation to a comparison of the `unsafe` mode to the three different protection modes of SAFER SLOTH on the ARM Cortex-M3. The measurements were obtained using the same test application as for the TriCore platform, except that an internal clock cycle counter was used instead of a hardware trace unit. As you can see in Table II, the overhead dealt for reconfiguring the MPU turns out to be significantly higher than on the TC1796; it amounts to around 130 cycles. However, the `MPU+itraps` protection mode achieves favorable overheads compared to `MPU+traps` throughout all test cases. Overall, the results exhibit a high similarity to the TC1796 evaluation in, for instance, the zero overhead for `ActivateTask()` in `MPU` mode and the low costs of disabling the MPU in the resource services.

VIII. DISCUSSION

With SAFER SLOTH, the interrupt-driven SLOTH kernels now also provide efficient means for memory isolation. In the following, we discuss some particularities of our design and how the general SLOTH philosophy—to embrace and exploit hardware particularities instead of blindly abstracting from them—fits to the requirements of memory isolation in safety-critical real-time systems.

A. Effect on Real-Time Characteristics

First and foremost: As its predecessors, SAFER SLOTH provides excellent predictability and priority obedience, and all

	Protection Mode			
SAFER SLOTH	`unsafe`	`MPU`	`MPU +traps`	`MPU +itraps`
`ActivateTask()` w/o dispatch	7	+0	+71	+34
`ActivateTask()` w/ dispatch	39	+118	+207	+180
`TerminateTask()` w/ dispatch	27	+148	+174	+162
`ChainTask()` w/ dispatch	57	+254	+333	+309
`GetResource()`	19	+20	+80	+35
`ReleaseResource()` w/o dispatch	20	+21	+70	+35
`ReleaseResource()` w/ dispatch	49	+126	+207	+180

TABLE II: SAFER SLOTH measurement results on the ARM Cortex-M3 in number of cycles.

issues of rate-monotonic priority inversion [7] are prevented by construction. However, depending on the chosen protection level, the latencies for system services go up, caused by the run-time overhead for privilege switches and MPU reconfigurations. Even though the relative overhead is high (up to a factor of four for `unsafe` versus `MPU+traps` mode), the absolute overhead with a maximum of 93 clock cycles on TriCore is remarkably low and, furthermore, constant and computable at compile time. By its interrupt-driven design, SAFER SLOTH in `MPU+traps` mode is still five to nine times faster than the commercial AUTOSAR OS in `unsafe` mode and eight to twelve time faster than the commercial AUTOSAR OS in `MPU+traps` mode.

B. Unprivileged Interrupt Handlers

In personal discussions at previous RTSS conferences where we presented our work on SLOTH, the most frequently expressed concern was the fact that by modeling all tasks as interrupt handlers they implicitly run with kernel privileges—hence, strong isolation does not appear to be feasible with our approach.

From the technical point of view, we considered this concern as somewhat surprising: Also with a software-based scheduler, a thread would initially start in kernel mode—it is up to the kernel to ensure that the thread executes some kernel code to lower its privilege level before leaving the kernel. In SAFER SLOTH this works in a very similar manner: The interrupt handler does not map directly to the task function, but to a small prologue, which is generated for each task and which sets up the task-specific privileges before invoking the actual task function.

From the conceptual point of view, however, we assume that the frequent expression of this concern underlines that tasks and interrupts are still perceived as two fundamentally different abstractions—which is not true: At its core, the only difference between interrupts and tasks is their activation by hardware versus software. With current microcontrollers, *all* further differences (prioritization, privileges, latencies, blocking/non-blocking execution, and so on) are just a matter of system-software design! SLOTH has always featured a unique control-flow abstraction that is independent from the kind of activation [13] and blocking semantics [14]; with SAFER SLOTH this now also holds for memory isolation and privilege separation.

C. How Safe is Safe?

As pointed out in Section II, the major motivation for memory isolation and privilege separation in embedded control systems is safety, not security: In the automotive industry, for instance, the OEMs co-locate control applications provided by different suppliers on a single microcontroller. These applications are generally considered trustworthy—but not bug free. Hence, memory isolation is applied as a means to isolate the effects of potential bugs and, eventually, the liability issues that may be caused by them.

Aside from the "traditional" `unsafe` and `MPU+traps` modes, SAFER SLOTH in its `MPU+itraps` and `MPU` modes provides intermediate modes that delegate parts of the enforcement of privilege separation from run-time checks to compile time checks and constructive software development measures. The feasibility, benefits, and resulting safety of these modes, however, depend to a high degree on the architecture and the application—the SLOTH philosophy is to automatically tailor the kernel to *both*.

In the `MPU+itraps` mode, for instance, all system services get inlined into the application code—embraced by the explicit raising and lowering of the privilege level to supervisor mode, which on the TriCore is technically implemented with the `syscall` and `rfe` instructions and a "jump-back" trap handler (see Section VI). To guarantee safety with this setup, one has to make sure that the application code cannot accidentally raise its privilege level (i.e., invoke `syscall` outside of a system service). As (a) we can assume immutable code (see Section II) and (b) `syscall` on TriCore and `svc` on ARM are dedicated opcodes, it is straightforward to check the application's binary against the direct invocation of the corresponding opcode in the post-validation step at compile time (see Section V). This still leaves the (much smaller) potential danger of an accidental *indirect* invocation via an erroneous indirect jump that may be caused by (c) an accidentally overwritten return address or (d) an invalid function pointer. Again, this depends on the architecture and application: On the TriCore, for instance, the call stack is managed in protected memory and separately from the data stack so that (c) is impossible; on the Cortex-M3 as well as on many other platforms, the compiler can be instructed to insert code for extra return address validation, rendering (c) virtually impossible. The use of function pointers in application code is uncommon in embedded control applications—and even explicitly forbidden by many safety-related development standards, such as MISRA-C. Thus, in many cases, (d) can be ruled out by construction.

In the even more efficient `MPU` mode, *all* application code runs in supervisor mode. System services get inlined—embraced by the explicit reconfiguration of the MPU. This requires, additionally to the above, an architecture where (e) the MPU is not implicitly disabled in supervisor mode and (f) it is possible to check at compile time that the application code cannot accidentally access the MPU control registers. On the TriCore, (e) is given and the MPU registers are accessible by dedicated opcodes only, so for (f), the same line of argument applies as for the `syscall` instruction above. The same holds for other (potentially harmful) privileged instructions, such as `disable` or `sleep`. On the ARM Cortex-M3, (e) is given as well. Regarding (f), however, MPU registers are memory-mapped, so post-validation will be much more difficult—although not infeasible, considering the recent advances in applying model checking techniques to prove properties of low-level system software [2]. Other memory-mapped hardware registers, for instance for programming a DMA-capable device, do also imply a danger, but can often be protected by the MPU itself.

In addition to such platform-specific limitations of the `MPU` and `MPU+itraps` modes, they also imply a safety trade-off if the fault model includes transient faults or potentially malicious applications. Since both modes rely on validated properties of the user code, they are susceptible to, for instance, bit flips in address registers or even the program counter, possibly leading to destructive behavior in the user code regardless of its validation. In case post-validation is deemed too difficult on a particular platform, or the user's fault model rules out relying on validated code, the more conservative `MPU+traps` mode therefore might be the preferred choice despite higher costs in terms of latency and memory footprint.

D. Hardware Limitations

As outlined in Section V, an application configuration is not entirely unrestricted with regards to the degree of data sharing between tasks. Since MPUs commonly only provide a finite number of configurable memory ranges, the underlying hardware imposes an upper limit to the number of domains assigned to each task in a SAFER SLOTH application. Other than requiring the system designer to work around this issue by, for instance, merging domains where it is feasible, a possible remedy could be to offer a kind of virtualization layer between tasks and domains. This layer could allow tasks with many assigned domains to switch between subsets of these domains—each subset small enough to fit into a set of MPU ranges—whenever necessary. This would, of course, deal additional overhead for reconfiguring the MPU during the execution of tasks that have many domains assigned.

E. Applicability and Tailorability

The SAFER SLOTH design proves to be well portable to different hardware platforms, whereas the degree of safety that can be offered to the application depends on external factors such as the specific properties of the underlying hardware. The optimal choice of the protection mode in a SAFER SLOTH configuration depends on both the requirements of the application and the target platform at hand. Instead of hiding hardware peculiarities behind layers of abstraction, our generative approach allows to tailor the system specifically to meet the application's demands in terms of costs and safety guarantees by embracing and exploiting the particularities of the concrete architecture. In doing so, SAFER SLOTH achieves memory safety at significantly lower costs than traditional designs.

F. Related Work

In earlier work on the CiAO operating system [18], we have already suggested a "semi-trusted" isolation mode that corresponds to the `MPU` mode of SAFER SLOTH. CiAO, however, employs a traditional scheduler and is subject to issues of rate-monotonic priority inversion. Furthermore, it does not provide the `MPU+itraps` mode of SAFER SLOTH.

The clever exploitation of particularities of memory protection hardware for the efficient implementation of isolation and virtualization has a long tradition in the domain of general-purpose operating systems. A cornerstone was Multics [6] with its single level store by means of which the clear distinction between "external" (e.g., files) and "internal" memory of processes was discarded. Based on its concept of *inline traps* to support the implementation of system calls, SAFER SLOTH is similar with respect to its handling of user and kernel space. This concept differs from the VDSO (virtual dynamic shared object) technique of Linux 2.6 to accelerate switching between user and kernel mode of operation of a process. VDSO makes use of the *fast system call* facility (`sysenter/sysexit`) as introduced with the Pentium II processor [15]. In contrast to SAFER SLOTH, which thoroughly benefits from static program analysis at configuration time to optimize (1) parameter passing and (2) system call dispatching, and despite of the fast system call facility, Linux still has to take the conventional line through a system call dispatch table in the kernel. SAFER SLOTH shares properties with single-address-space operating systems (SASOS), such as the Mach-based Opal [3] or the L4-based Mungi [12]. However, SASOS are currently fixed to general-purpose operation modes and far too complex for special-purpose systems both in functional and non-functional terms. Commodity operating systems today tend to abstract from these differences for the sake of portability. In contrast, the SAFER SLOTH approach is to not hide such architecture-specific features, but to employ generators and static checkers to map the more generic abstractions (protection domains and privilege separation) to them.

A lot of related work exists with respect to constructive (language/compiler-based) memory protection: In software-based fault isolation (SFI) [21], the binary code is patched at compile time to interpret (and check) potentially critical instructions at run time. The original motivation behind SFI was—similar to our work—not to replace hardware-based memory protection, but to reduce its overhead (here, on DEC Alpha). In recent years, SFI has re-gained attention, but now with a focus on platforms that are not equipped with an MPU, especially 8-bit sensor nodes. Examples include XFI [9], the t-kernel [10], TinyOS [5], or SOS/Harbor [11], [16]. These SFI approaches induce a (significant) extra run-time overhead and impair predictability, which partly can be mitigated by customized hardware [17] or by developing all application code in type-safe dialects/subsets of C [19], [8], [4]. The `MPU+traps` and `MPU` modes of SAFER SLOTH differ from them by (a) only delegating privilege separation to constructive means, which is (b) motivated by (architecture-dependent) run-time efficiency and predictability gains on (c) commodity microcontroller systems.

IX. Conclusion

The SLOTH embedded kernels have proven to excel at deterministic execution by preventing certain kinds of priority inversion and at efficiency-related non-functional properties. However, SLOTH has also been criticized for executing application code in interrupt handler context, which uses supervisor mode privileges of the hardware. With the SAFER SLOTH design described in this paper, we have shown that we can combine a hardware-tailored, interrupt-based kernel with the enforcement of application isolation. With its *inline traps* and *MPU only* modes for privilege separation between the applications and the kernel, SAFER SLOTH continues the SLOTH philosophy of exploiting and embracing particularities of commodity hardware instead of abstracting from them. SAFER SLOTH is able to ensure memory safety at minimal overheads. In comparison to a leading commercial AUTOSAR OS implementation, SAFER SLOTH provides, depending on the protection mode, a speedup of 8x (worst case) up to 23x (best case) on kernel latencies, while still providing excellent determinism, strict priority obedience, and small memory footprints.

Acknowledgements

We would like to thank our students Markus Müller and Clemens Lang for their help on SAFER SLOTH, as well as our shepherd, Frank Mueller, for his guidance.

References

[1] AUTOSAR. Specification of operating system (version 5.0.0). Technical report, Automotive Open System Architecture GbR, November 2011.

[2] Bernhard Blackham and Gernoth Heiser. Sequoll: A framework for model checking binaries. In *19th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS '13)*, pages 97–106, Washington, DC, USA, 2013. IEEE.

[3] Jeffrey S. Chase, Henry M. Levy, Michael J. Freeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comp. Syst.*, 12(4):271–307, November 1994.

[4] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 520–535, Heidelberg, Germany, 2007. Springer.

[5] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. Efficient memory safety for TinyOS. In *5th Int. Conf. on Embedded Networked Sensor Systems*, pages 205–218, New York, NY, USA, 2007. ACM.

[6] Robert C. Daley and Jack Bonnell Dennis. Virtual memory, processes, and sharing in MULTICS. *CACM*, 11(5):306–312, May 1968.

[7] Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *12th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS '06)*, pages 14–23, Los Alamitos, CA, USA, 2006. IEEE.

[8] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without garbage collection for embedded applications. *Transactions on Embedded Computing Systems*, 4(4):73–111, February 2005.

[9] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *7th Symp. on OS Design and Implementation (OSDI '06)*, pages 75–88, Berkeley, CA, USA, 2006. USENIX.

[10] Lin Gu and John A. Stankovic. t-kernel: Providing reliable os support to wireless sensor networks. In *4th Int. Conf. on Embedded Networked Sensor Systems*, New York, NY, USA, 2006. ACM.

[11] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor networks. In *3rd Int. Conf. on Mobile Systems, Applications, and Services (MobiSys '05)*, pages 163–176, New York, NY, USA, June 2005. ACM.

[12] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russel, and Jochen Liedtke. The Mungi single-address-space operating system. *Softw. Pract. Exper.*, 18(9), July 1998.

[13] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *30th IEEE Int. Symp. on Real-Time Systems (RTSS '09)*, pages 204–213. IEEE, December 2009.

[14] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Sleepy Sloth: Threads as interrupts as threads. In *32nd IEEE Int. Symp. on Real-Time Systems (RTSS '11)*, pages 67–77. IEEE, December 2011.

[15] Intel Corporation, Santa Clara, California, USA. *Intel Architecture Software Developer's Manual*, 1999.

[16] Ram Kumar, Eddie Kohler, and Mani Srivastava. Harbor: Software-based memory protection for sensor nodes. In *IPSN '07: 6th Int. Conf. on Information Processing in Sensor Networks*, pages 340–349, New York, NY, USA, 2007. ACM.

[17] Ram Kumar, Akhilesh Singhania, Andrew Castner, Eddie Kohler, and Mani Srivastava. A system for coarse grained memory protection in tiny embedded processors. In *Proceedings of the 44th annual Design Automation Conference*, pages 218–223, New York, NY, USA, 2007. ACM.

[18] Daniel Lohmann, Jochen Streicher, Wanja Hofer, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Configurable memory protection by aspects. In *4th W'shop on Progr. Lang. and OSes (PLOS '07)*, pages 1–5, New York, NY, USA, October 2007. ACM.

[19] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL '02: 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 128–139, New York, NY, USA, 2002. ACM.

[20] 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.

[21] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *14th ACM Symp. on OS Principles (SOSP '93)*, pages 203–216, New York, NY, USA, 1993. ACM.