

Konfigurierbare Systemsoftware (KSS)

VL 5 – Generative Programming: The SLOTH Approach

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

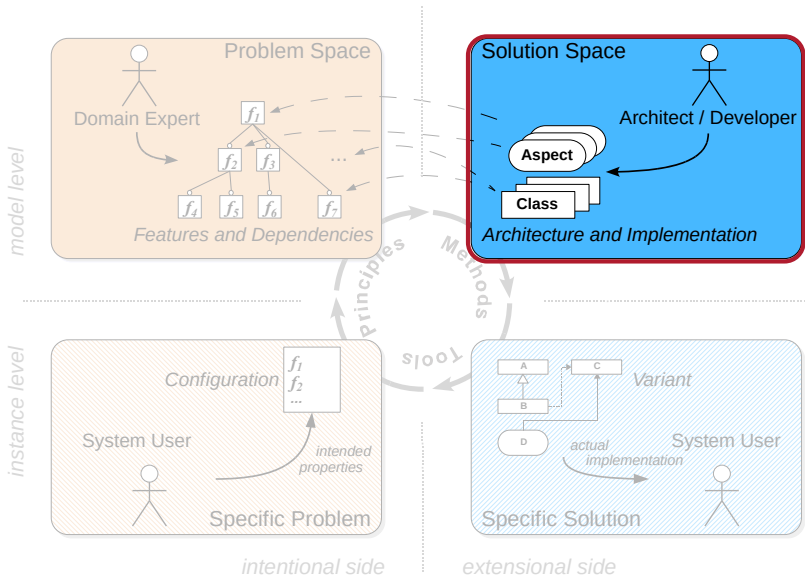
Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 16 – 2016-05-09

http://www4.informatik.uni-erlangen.de/Lehre/SS16/V_KSS

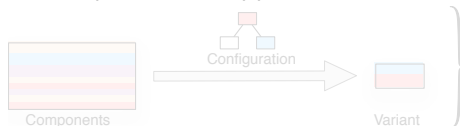


About this Lecture



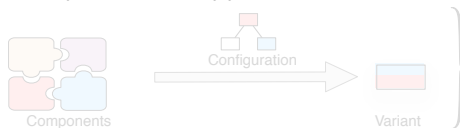
Implementation Techniques: Classification

■ Decompositional Approaches



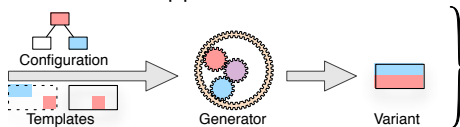
- Text-based filtering (untyped)
- Preprocessors

■ Compositional Approaches



- Language-based composition mechanisms (typed)
- OOP, **AOP**, Templates

■ Generative Approaches



- Metamodel-based generation of components (typed)
- MDD, C++ TMP, generators



Implementation Techniques: Classification

Decompositional Approaches



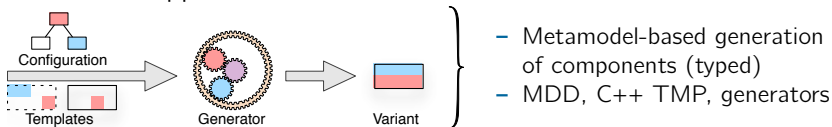
Compositional Approaches



“ I’d rather write programs to write programs than write programs. ”

Dick Sites (DEC)

Generative Approaches



- 5.1 Motivation: OSEK and Co
- 5.2 SLOTH: Threads as Interrupts
- 5.3 SLEEPY SLOTH: Threads as IRQs as Threads
- 5.4 SAFER SLOTH: Hardware-Tailored Isolation
- 5.5 SLOTH ON TIME: Time-Triggered Laziness
- 5.6 SLOTH* Generation
- 5.7 Summary and Conclusions
- 5.8 References



5.1 Motivation: OSEK and Co

- Background

- OSEK OS: Abstractions

- OSEK OS: Tailoring and Generation

- 5.2 SLOTH: Threads as Interrupts

- 5.3 SLEEPY SLOTH: Threads as IRQs as Threads

- 5.4 SAFER SLOTH: Hardware-Tailored Isolation

- 5.5 SLOTH ON TIME: Time-Triggered Laziness

- 5.6 SLOTH* Generation

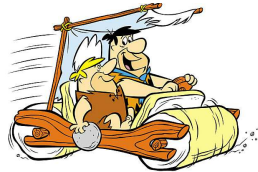
- 5.7 Summary and Conclusions

- 5.8 References



The OSEK Family of Automotive OS Standards

- **1995** OSEK OS (OSEK/VDX) [9]
- **2001** OSEKtime (OSEK/VDX) [11]
- **2005** AUTOSAR OS (AUTOSAR) [1]



■ OSEK OS

→ “Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen”

- **statically configured**, event-triggered real-time OS

■ OSEKtime

- **statically configured**, time-triggered real-time OS
- can optionally be extended with OSEK OS (to run in slack time)

■ AUTOSAR OS

→ “Automotive Open System Architecture”

- **statically configured**, event-triggered real-time OS
- real superset of OSEK OS ~ backwards compatible
- additional time-triggered abstractions (schedule tables, timing protection)
- intended as a successor for both OSEK OS and OSEKtime



- Control flows
 - Task: software-triggered control flow (strictly priority-based scheduling)
 - Basic Task (BT) run-to-completion task with strictly stack-based activation and termination
 - Extended Task (ET) may suspend and resume execution (↪ coroutine)
 - ISR: hardware-triggered control flow (hardware-defined scheduling)
 - Cat 1 ISR (ISR1) runs below the kernel, may not invoke system services (↪ prologue without epilogue)
 - Cat 2 ISR (ISR2) synchronized with kernel, may invoke system services (↪ epilogue without prologue)
 - Hook: OS-triggered signal/exception handler
 - ErrorHandler invoked in case of a syscall error
 - StartupHook invoked at system boot time
 - ...



- Coordination and synchronization
 - **Resource:** mutual exclusion between well-defined set of tasks
 - stack-based priority ceiling protocol ([12]):
 `GetResource()` $\rightsquigarrow$ priority is raised to that of highest participating task
 - pre-defined `RES_SCHED` has highest priority ($\rightsquigarrow$ blocks preemption)
 - implementation-optional: task set may also include cat 2 ISRs
 - **Event:** condition variable on which ETs may block
 - part of a task's context
 - **Alarm:** asynchronous trigger by HW/SW counter
 - may execute a callback, activate a task, or set an event on expiry



OSEK OS: System Services (Excerpt)

■ Task-related services

- `ActivateTask(task)` $\leadsto$ *task* is active ($\mapsto$ ready), counted
- `TerminateTask()` $\leadsto$ running task is terminated
- `Schedule()` $\leadsto$ active task with highest priority is running
- `ChainTask(task)` $\mapsto$ atomic $\left\{ \begin{array}{l} \text{ActivateTask}(task) \\ \text{TerminateTask}() \end{array} \right.$

■ Resource-related services

- `GetResource(res)` $\leadsto$ current task has *res* ceiling priority
- `ReleaseResource(res)` $\leadsto$ current task has previous priority

■ Event-related services (extended tasks only!)

- `SetEvent(task, mask)` $\leadsto$ events in *mask* for *task* are set
- `ClearEvent(mask)` $\leadsto$ events in *mask* for current task are unset
- `WaitEvent(mask)` $\leadsto$ current task blocks
until event from *mask* has been set

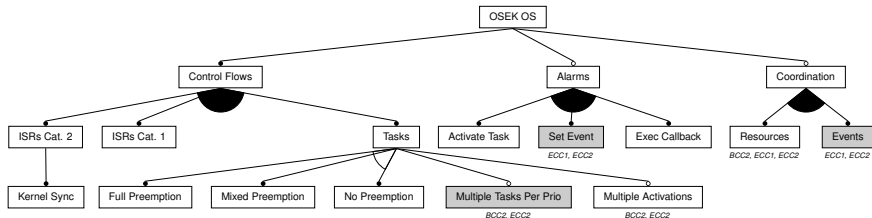
■ Alarm-related services

- `SetAbsAlarm(alarm, ...)` $\leadsto$ arms *alarm* with absolute offset
- `SetRelAlarm(alarm, ...)` $\leadsto$ arms *alarm* with relative offset



OSEK OS: Conformance Classes [9]

- OSEK offers predefined tailorability by four **conformance classes**
 - **BCC1** only basic tasks, limited to one activation request per task and one task per priority, while all tasks have different priorities
 - **BCC2** like BCC1, plus more than one task per priority possible and multiple requesting of task activation allowed
 - **ECC1** like BCC1, plus extended tasks
 - **ECC2** like ECC1, plus more than one task per priority possible and multiple requesting of task activation allowed for basic tasks
- The OSEK feature diagram



OSEK OS: System Specification with OIL [10]

- An OSEK OS instance is configured **completely statically**
 - all general OS features (hooks, ...)
 - all **instances** of OS abstractions (tasks, ...)
 - all **relationships** between OS abstractions
 - described in a **domain-specific language (DSL)**
- OIL: The OSEK Implementation Language
 - standard types and attributes (TASK, ISR, ...)
 - vendor/plattform-specific *attributes* (ISR source, priority, triggering)
 - task types and conformance class is deduced

OIL File for Example System (BCC1)

- Three basic tasks: Task1, Task3, Task4
- Category 2 ISR: ISR2 (platform-spec. source/priority)
- Task1 and Task3 use resource Res1 ~ ceiling pri = 3
- Alarm Alarm1 triggers Task4 on expiry

```
...
OS ExampleOS {
  STATUS      = STANDARD;
  STARTUPHOOK = TRUE;
};

TASK Task1 {
  PRIORITY    = 1;
  AUTOSTART   = TRUE;
  RESOURCE    = Res1;
};

TASK Task3 {
  PRIORITY    = 3;
  AUTOSTART   = FALSE;
  RESOURCE    = Res1;
};

TASK Task4 {
  PRIORITY    = 4;
  AUTOSTART   = FALSE;
};

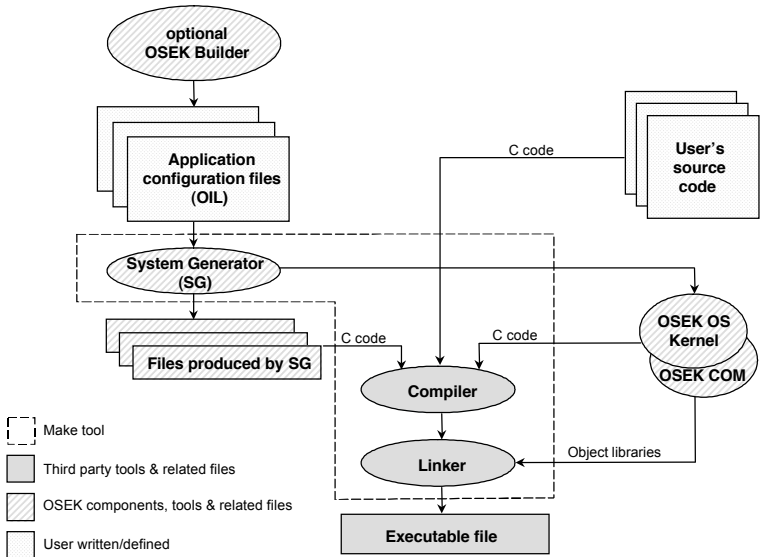
RESOURCE Res1 {
  RESOURCEPROPERTY = STANDARD;
};

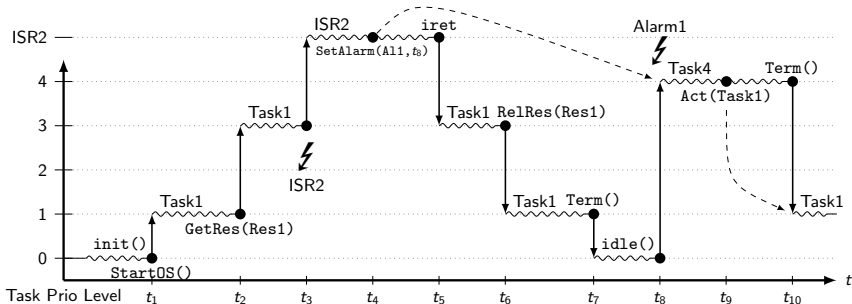
ISR ISR2 {
  CATEGORY      = 2;
  PRIORITY      = 2;
};

ALARM Alarm1 {
  COUNTER       = Timer1;
  ACTION        = ACTIVATETASK {
    TASK        = Task4;
  };
  AUTOSTART     = FALSE;
};
```



OSEK OS: System Generation [10, p. 5]





- Basic tasks behave much like IRQ handlers
(on a system with support for IRQ priority levels)
 - priority-based dispatching with run-to-completion
 - LIFO, all control flows can be executed on a single shared stack
- So why not dispatch tasks as ISRs?
 - ~> Let the hardware do all scheduling!
 - ~> Let's be a SLOTH!



5.1 Motivation: OSEK and Co

5.2 SLOTH: Threads as Interrupts

Basic Idea

Design

Results

Limitation

5.3 SLEEPY SLOTH: Threads as IRQs as Threads

5.4 SAFER SLOTH: Hardware-Tailored Isolation

5.5 SLOTH ON TIME: Time-Triggered Laziness

5.6 SLOTH* Generation

5.7 Summary and Conclusions

5.8 References

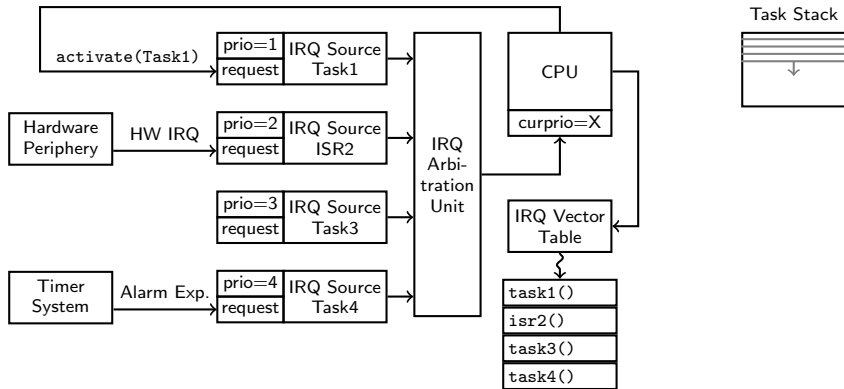


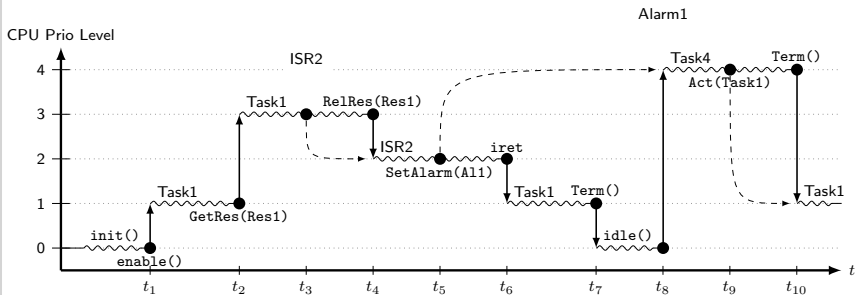
- **Idea: threads are interrupt handlers, synchronous thread activation is IRQ**
- Let interrupt subsystem do the scheduling and dispatching work
- Applicable to priority-based real-time systems
- Advantage: small, fast kernel with unified control-flow abstraction

Paper title of [6] is a pun to the approach taken by SOLARIS: “Interrupts as Threads”, ACM OSR (1995) [8]



- IRQ system must support priorities and software triggering





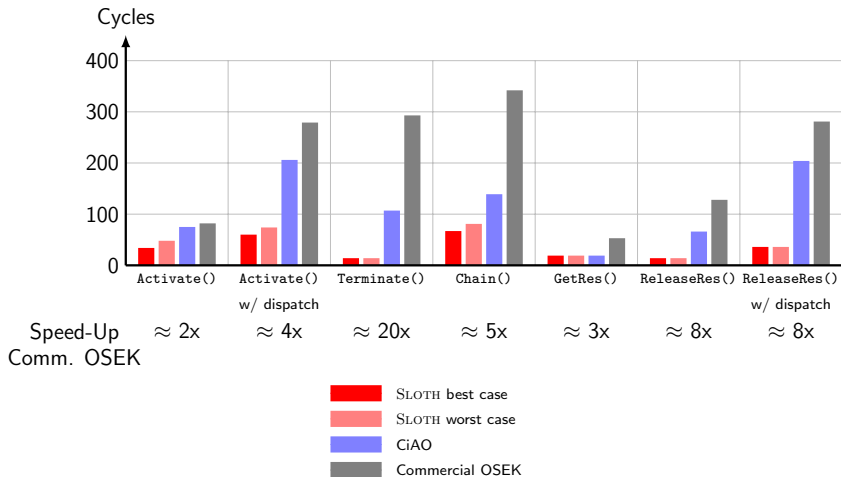
- Concise kernel design and implementation
 - < 200 LoC, < 700 bytes code memory, very little RAM
- Single control-flow abstraction for tasks, ISRs (1/2), callbacks
 - Handling oblivious to how it was triggered (by hardware or software)
- Unified priority space for tasks and ISRs
 - No rate-monotonic priority inversion [3, 4]
- Straight-forward synchronization by altering CPU priority
 - Resources with ceiling priority (also for ISRs!)
 - Non-preemptive sections with `RES_SCHEDULER` (highest task priority)
 - Kernel synchronization with highest task/cat.-2-ISR priority



- Reference implementation for Infineon TriCore
 - 32-bit load/store architecture
 - Interrupt controller: 256 priority levels, about 200 IRQ sources with memory-mapped registers
 - Meanwhile also implementations for ARM Cortex-M3 (SAM3U) and x86
- Evaluation of task-related system calls:
 - Task activation
 - Task termination
 - Task acquiring/releasing resource
- Comparison with commercial OSEK implementation and CiAO
- Two numbers for SLOTH: best case, worst case
 - Depending on number of tasks and system frequency

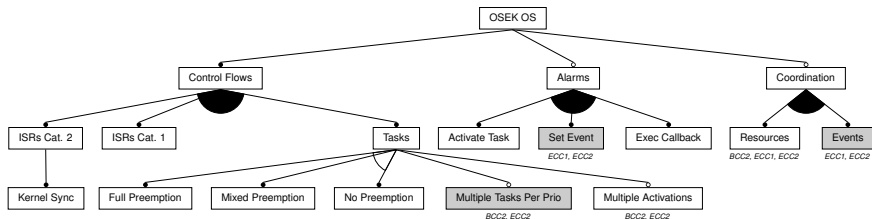


Performance Evaluation: Results



Limitations of the SLOTH Approach

- No multiple tasks per priority (→ OSEK BCC2 / ECC2)
↔ execution order has to be the same as activation order
- No extended tasks (that is, events, → OSEK ECC1 / ECC2)
↔ **impossible with stack-based IRQ execution model**
- No safety (that is, AUTOSAR-OS memory protection)
↔ impossible if everything runs as IRQ handler



Really?



5.1 Motivation: OSEK and Co

5.2 SLOTH: Threads as Interrupts

5.3 **SLEEPY SLOTH: Threads as IRQs as Threads**

Motivation

Design

Results

5.4 SAFER SLOTH: Hardware-Tailored Isolation

5.5 SLOTH ON TIME: Time-Triggered Laziness

5.6 SLOTH* Generation

5.7 Summary and Conclusions

5.8 References



Control Flows in Embedded Systems

	Activation Event	Sched./Disp.	Semantics
ISRs	HW	by HW	RTC
Threads	SW	by OS	Blocking
SLOTH [6]	HW or SW	by HW	RTC
SLEEPY SLOTH [7]	HW or SW	by HW	RTC or Blocking

(RTC: Run-to-Completion)



Main Goal

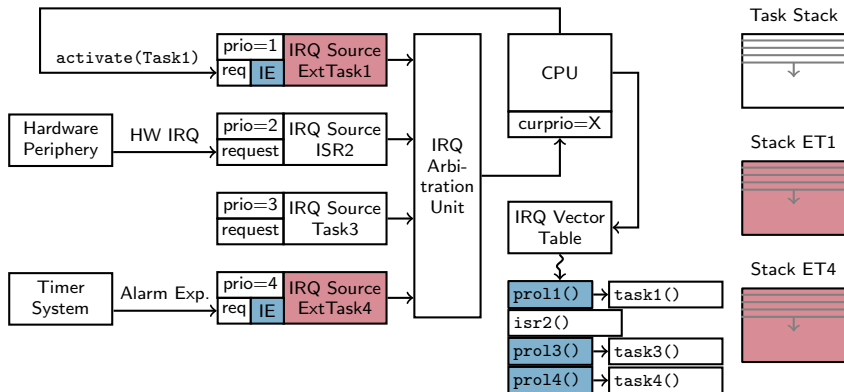
Support **extended blocking tasks** (with stacks of their own), while preserving SLOTH's **latency benefits** by having threads run as ISRs

Main Challenge

IRQ controllers do not support **suspension and re-activation** of ISRs



SLEEPY SLOTH Design: Task Prologues and Stacks

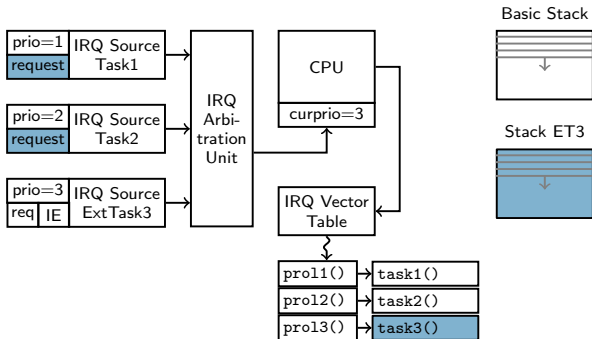
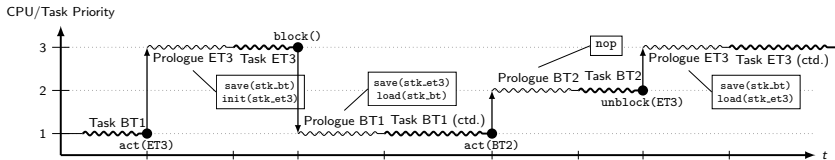


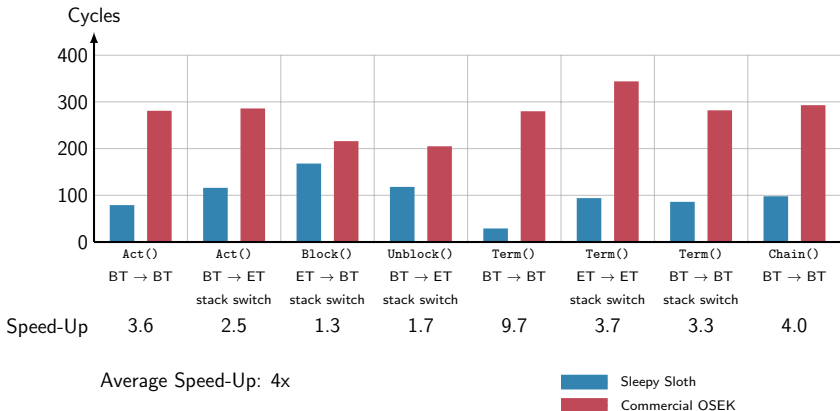
SLEEPY SLOTH: Dispatching and Rescheduling

- Task prologue: switch stacks if necessary
 - Switch *basic task* $\leftrightarrow$ *basic task* omits stack switch
 - On job start: initialize stack
 - On job resume: restore stack
- Task termination: task with next-highest priority needs to run
 - Yield CPU by setting priority to zero
 - (Prologue of *next* task performs the stack switch)
- Task blocking: take task out of “ready list”
 - Disable task’s IRQ source
 - Yield CPU by setting priority to zero
- Task unblocking: put task back into “ready list”
 - Re-enable task’s IRQ source
 - Re-trigger task’s IRQ source by setting its pending bit



SLEEPY SLOTH: Example Control Flow



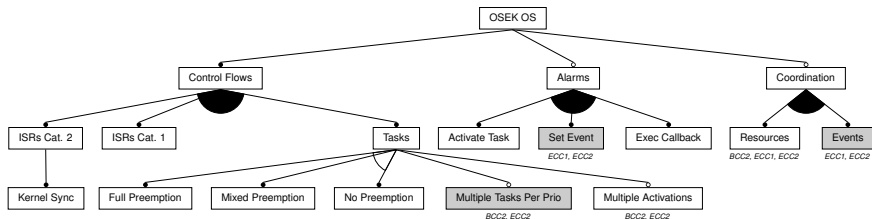


- Basic switches in a mixed system only slightly slower than in purely basic system



Limitations of the SLOTH Approach

- No multiple tasks per priority ($\mapsto$ OSEK BCC2 / ECC2)
 $\leftrightarrow$ execution order has to be the same as activation order
- ~~No extended tasks (that is, events, $\mapsto$ OSEK ECC1 / ECC2)~~
 $\leftrightarrow$ ~~impossible with stack-based IRQ execution model~~
- No safety (that is, AUTOSAR-OS memory protection)
 $\leftrightarrow$ **impossible if everything runs as IRQ handler**



Really?



5.1 Motivation: OSEK and Co

5.2 SLOTH: Threads as Interrupts

5.3 SLEEPY SLOTH: Threads as IRQs as Threads

5.4 **SAFER SLOTH: Hardware-Tailored Isolation**

Motivation

Design

Results

5.5 SLOTH ON TIME: Time-Triggered Laziness

5.6 SLOTH* Generation

5.7 Summary and Conclusions

5.8 References



Main Goal

Support **isolation** of **state and privileges**, while preserving SLOTH's **latency benefits** by having threads run as ISRs.

Main Challenge

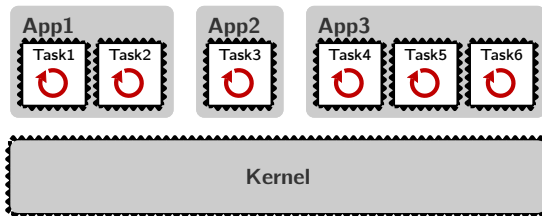
ISRs run in **supervisor mode** with full privileges. So how to

- Effectively isolate kernel and application
- Maintain design principles of Sloth



Memory Protection in Embedded Systems

- *Safety, but not security*
- *Protect the data, but not the code*
- Safety model based on AUTOSAR OS
- MPU-based isolation



- **Vertically:** Protect kernel state and MPU configuration
- **Horizontally:** Isolate applications or even tasks from each other



- Exploit as much knowledge about target hardware as possible
- Tailor kernel to fit both the platform and the application
- Taking into account:
 - Extent and layout of MPU configuration
 - Method for re-programming the MPU
 - Available hardware privilege levels
 - Is MPU active in all levels?
 - Degree of safety required by the application



Unsafe

- The original Sloth OS, without isolation

MPU

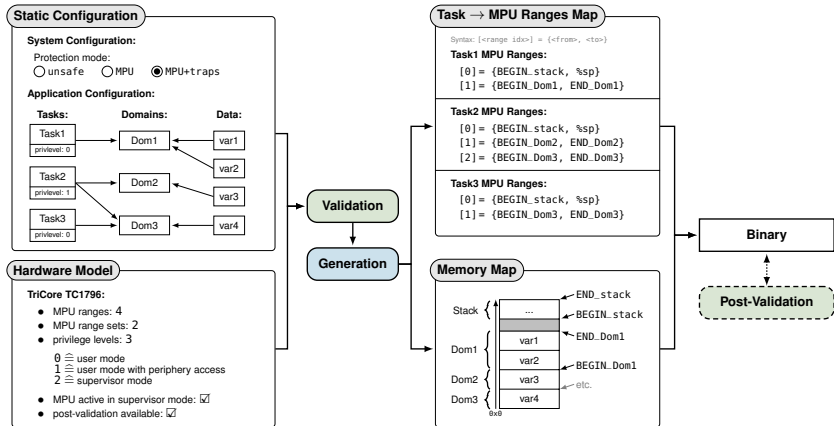
- MPU active, but tasks execute with supervisor privileges
- Vertical isolation ensured constructively in post-validation

MPU+traps

- Vertical isolation ensured by hardware privilege levels
- System services acquire kernel privileges via syscall mechanism



SAFER SLOTH: Architecture



MPU mode in SAFER SLOTH

unsafe Mode:

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

MPU Mode:

```
Task1:
...
; disable MPU
mfcrr %d15,$psw
insert %d15,%d15,0,12,1
mtcrr $psw,%d15
prio = getCurPrio();
pushResourceStack(prio);
if (Res1 > prio) {
    setCurPrio(Res1);
}
; enable MPU
mfcrr %d15,$psw
insert %d15,%d15,15,12,1
mtcrr $psw,%d15
...
```

User mode

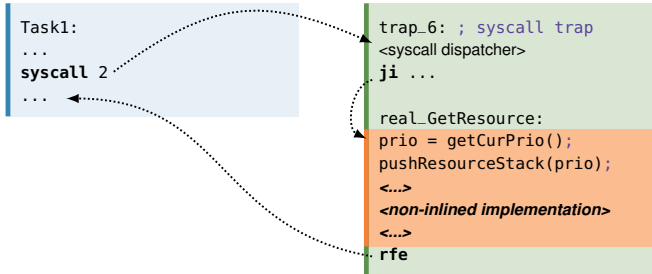
Supervisor mode

System service implementation



MPU+traps mode in SAFER SLOTH

MPU+traps Mode:



User mode

Supervisor mode

System service implementation



The Problem with Traps

- SLOTH gains a lot of its benefits through compiler optimizations
 - Inlining of system service calls
 - Removal of dead code
 - Constant propagation
- Traditional traps **prohibit** such optimizations
 - System services must be standalone functions
 - Jumped to via a syscall dispatcher

Solution Idea

Combine MPU and MPU+traps mode

↪ *Inline traps* as 4th protection mode (MPU+itraps)




```
Task1:
...
syscall 0
prio = getCurPrio();
pushResourceStack(prio);
if (Res1 > prio) {
    setCurPrio(Res1);
}

; load current pc
mfcrl %d15,$pc
add %d15,2
; overwrite return address
mov.a %a11,%d15
rfe
...
```

generates trap

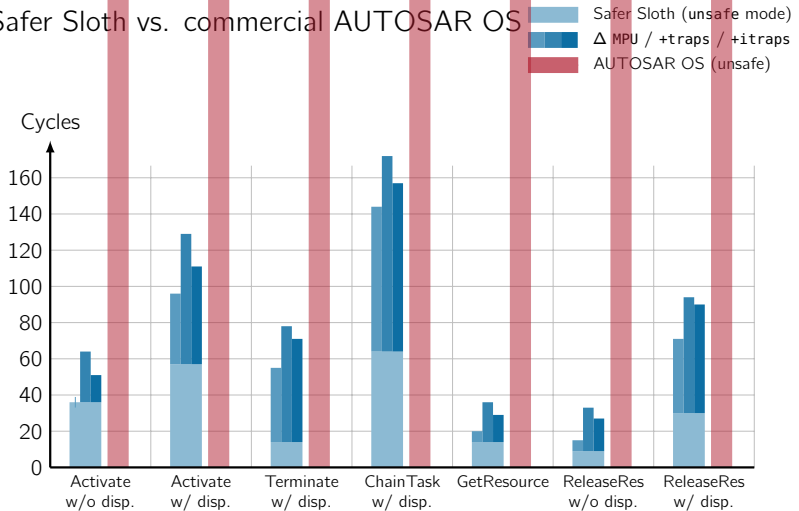
jumps back

returns to

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

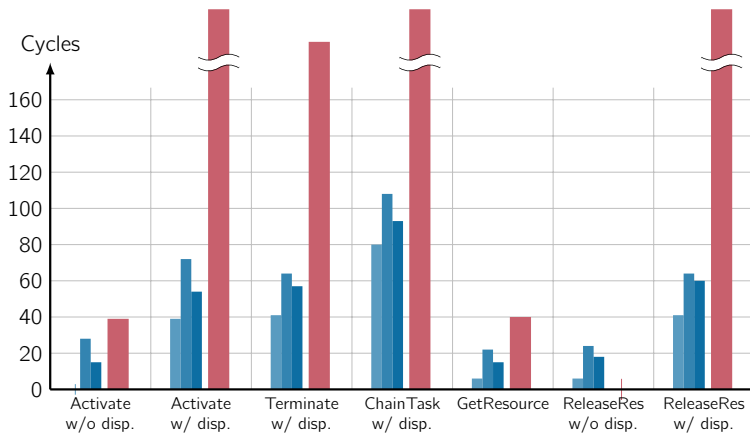


Safer Sloth vs. commercial AUTOSAR OS



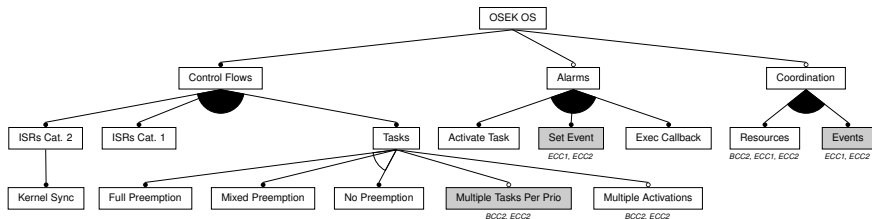
Safer Sloth vs. commercial AUTOSAR OS

- Safer Sloth (unsafe mode)
- Δ MPU / +traps / +traps
- AUTOSAR OS (Δ MPU+traps)



Limitations of the SLOTH Approach

- No multiple tasks per priority ($\mapsto$ OSEK BCC2 / ECC2)
 $\leftrightarrow$ execution order has to be the same as activation order
- No extended tasks (that is, events, $\mapsto$ OSEK ECC1 / ECC2)
 $\leftrightarrow$ impossible with stack-based IRQ execution model
- No safety (that is, AUTOSAR-OS memory protection)
 $\leftrightarrow$ impossible if everything runs as IRQ handler



5.1 Motivation: OSEK and Co

5.2 SLOTH: Threads as Interrupts

5.3 SLEEPY SLOTH: Threads as IRQs as Threads

5.4 SAFER SLOTH: Hardware-Tailored Isolation

5.5 SLOTH ON TIME: Time-Triggered Laziness

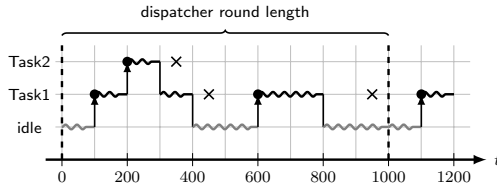
5.6 SLOTH* Generation

5.7 Summary and Conclusions

5.8 References

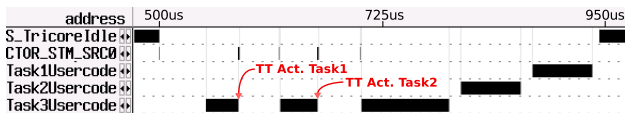


- **Idea: use hardware timer arrays to implement schedule tables**
- TC1796 GPTA: 256 timer cells, routable to 96 interrupt sources
 - use for task activation, deadline monitoring, execution time budgeting, time synchronization, and schedule table control
- SLOTH ON TIME implements OSEKtime [11] and AUTOSAR OS schedule tables [1]
 - combinable with SLOTH or SLEEPY SLOTH for mixed-mode systems
 - up to 170x lower latencies compared to commercial implementations

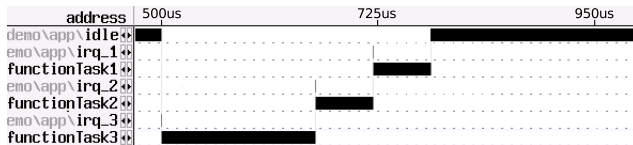


Qualitative Evaluation: AUTOSAR

Commercial AUTOSAR: **Priority inversion** with time-triggered activation (2,075 cycles each)



SLOTH ON TIME: avoids this *by design!*



“ Interrupts are perhaps the biggest cause of priority inversion in real-time systems, causing the system to not meet all of its timing requirements. ”

Stewart 1999: “Twenty-Five Most Common Mistakes with Real-Time Software Development” [13]

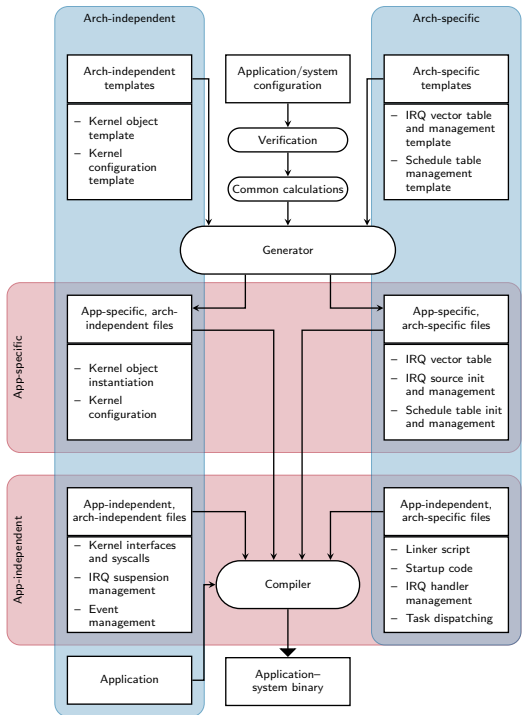


- 5.1 Motivation: OSEK and Co
- 5.2 SLOTH: Threads as Interrupts
- 5.3 SLEEPY SLOTH: Threads as IRQs as Threads
- 5.4 SAFER SLOTH: Hardware-Tailored Isolation
- 5.5 SLOTH ON TIME: Time-Triggered Laziness
- 5.6 SLOTH* Generation**
- 5.7 Summary and Conclusions
- 5.8 References

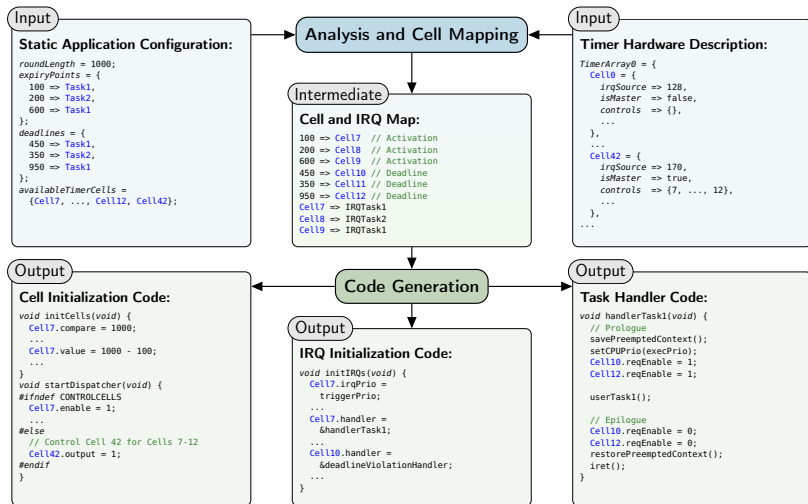


SLOTH* Generation

- Two generation dimensions
 - Architecture
 - Application
- Generator is implemented in Perl
 - Templates
 - Configuration



SLOTH ON TIME Generation



Agenda

- 5.1 Motivation: OSEK and Co
- 5.2 SLOTH: Threads as Interrupts
- 5.3 SLEEPY SLOTH: Threads as IRQs as Threads
- 5.4 SAFER SLOTH: Hardware-Tailored Isolation
- 5.5 SLOTH ON TIME: Time-Triggered Laziness
- 5.6 SLOTH* Generation
- 5.7 Summary and Conclusions**
- 5.8 References



Summary: The SLOTH* Approach

- Exploit standard interrupt/timer/mpu hardware to delegate core OS functionality to hardware
 - scheduling and dispatching of control flows
 - OS needs to be tailored to application *and* hardware platform
 - ↪ generative approach is necessary
- Benefits
 - tremendous latency reductions, very low memory footprints
 - unified control flow abstraction
 - hardware/software-triggered, blocking/run-to-completion
 - no need to distinguish between tasks and ISRs
 - no rate-monotonic priority inversion
 - reduces complexity
 - less work for the OS developer :-)



- [1] AUTOSAR. *Specification of Operating System (Version 4.1.0)*. Tech. rep. Automotive Open System Architecture GbR, Oct. 2010.
- [2] Daniel Danner, Rainer Müller, Wolfgang Schröder-Preikschat, Wanja Hofer, and Daniel Lohmann. “Safer Sloth: Efficient, Hardware-Tailored Memory Protection”. In: *Proceedings of the 20th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '14)*. Washington, DC, USA: IEEE Computer Society Press, 2014, pp. 37–47.
- [3] Luis E. Leyva-del Foyo, Pedro Mejia-Alvarez, and Dionisio de Niz. “Integrated Task and Interrupt Management for Real-Time Systems”. In: *Transactions on Embedded Computing Systems* 11.2 (July 2012), 32:1–32:31. ISSN: 1539-9087. DOI: 10.1145/2220336.2220344.
- [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)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2006, pp. 14–23. DOI: 10.1109/RTAS.2006.34.



- [5] Wanja Hofer, Daniel Danner, Rainer Müller, Fabian Scheler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS". In: *Proceedings of the 33rd IEEE International Symposium on Real-Time Systems (RTSS '12)*. (San Juan, Puerto Rico, Dec. 4–7, 2012). IEEE Computer Society Press, Dec. 2012, pp. 237–247. ISBN: 978-0-7695-4869-2. DOI: 10.1109/RTSS.2012.75.
- [6] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. "Sloth: Threads as Interrupts". In: *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS '09)*. (Washington, D.C., USA, Dec. 1–4, 2009). IEEE Computer Society Press, Dec. 2009, pp. 204–213. ISBN: 978-0-7695-3875-4. DOI: 10.1109/RTSS.2009.18.
- [7] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "Sleepy Sloth: Threads as Interrupts as Threads". In: *Proceedings of the 32nd IEEE International Symposium on Real-Time Systems (RTSS '11)*. (Vienna, Austria, Nov. 29–Dec. 2, 2011). IEEE Computer Society Press, Dec. 2011, pp. 67–77. ISBN: 978-0-7695-4591-2. DOI: 10.1109/RTSS.2011.14.
- [8] Steve Kleiman and Joe Eykholt. "Interrupts as Threads". In: *ACM SIGOPS Operating Systems Review* 29.2 (Apr. 1995), pp. 21–26. ISSN: 0163-5980.



- [9] OSEK/VDX Group. *Operating System Specification 2.2.3*. Tech. rep. <http://portal.osek-vdx.org/files/pdf/specs/os223.pdf>, visited 2014-09-29. OSEK/VDX Group, Feb. 2005.
- [10] OSEK/VDX Group. *OSEK Implementation Language Specification 2.5*. Tech. rep. <http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf>, visited 2014-09-29. OSEK/VDX Group, 2004.
- [11] OSEK/VDX Group. *Time-Triggered Operating System Specification 1.0*. Tech. rep. <http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf>. OSEK/VDX Group, July 2001.
- [12] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1175–1185. ISSN: 0018-9340. DOI: 10.1109/12.57058.
- [13] David B. Stewart. "Twenty-Five Most Common Mistakes with Real-Time Software Development". In: *Proceedings of the 1999 Embedded Systems Conference (ESC '99)*. 1999.

