Application/Hardware - Aware Operating System Design

Christian Dietrich, Daniel Lohmann
{dietrich,lohmann}@sra.uni-hannover.de

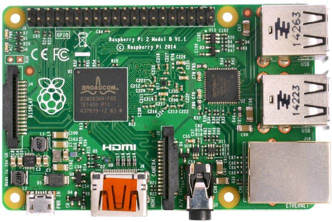
Leibniz Universität Hannover

7. February 2018

supported by DFG

Hardwired Control vs. Control Software

AUTOSAR



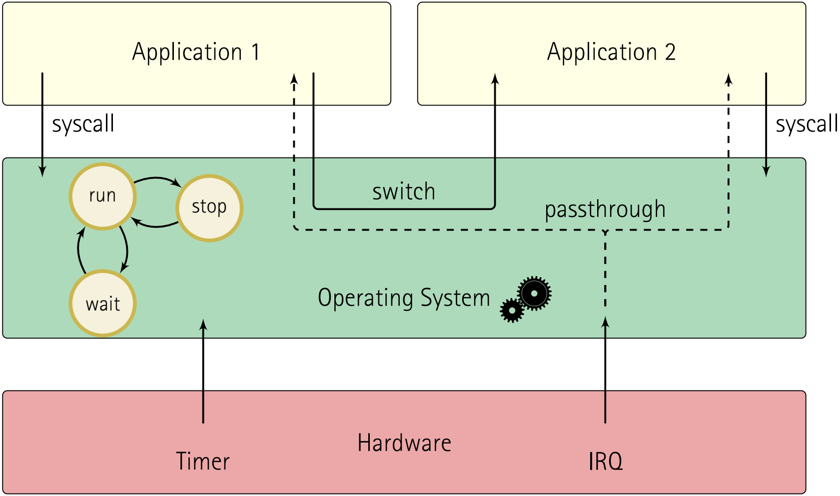
Hardwired Control

+ Low jitter, low latency
+ No organizational overhead
– Always start from scratch
– Expensive

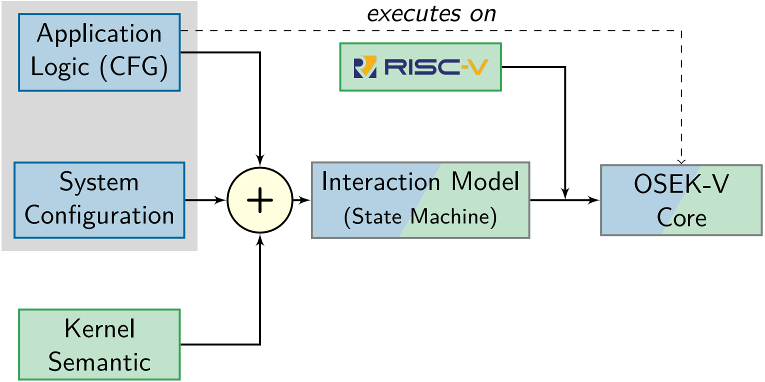
Control Software

+ Reusable parts: RTOS, libraries
+ Useful abstractions: e.g threads
+ Multiple interacting tasks
– Jitter through interference
– Overhead by abstractions (RTOS)

Between a Rock and a Hard Place



Our Idea in a Nutshell



Outline

- Motivation and Introduction
- **RTOS–Application Interaction Model**
- The OSEK-V Processor Pipeline
- Evaluation
- Conclusion

System Model: Event-Triggered Real-Time Systems

- System Model
  - Single-core or partitioned RTOS
  - Event-triggered real-time systems: execution threads, ISRs, etc.
  - Fixed-priority scheduling semantics
  - Ahead of time knowledge
    - System objects (thread, resources, periodic signals) and their configuration
    - Application structure including syscall locations and arguments

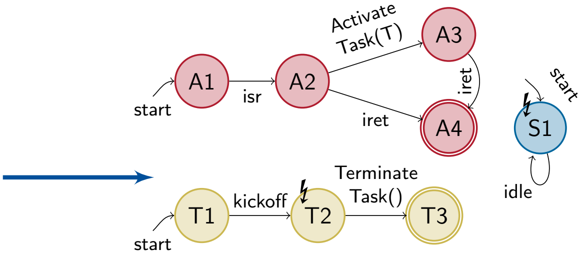
System Model: Event-Triggered Real-Time Systems

- System Model
  - Single-core or partitioned RTOS
  - Event-triggered real-time systems: execution threads, ISRs, etc.
  - Fixed-priority scheduling semantics
  - Ahead of time knowledge
    - System objects (thread, resources, periodic signals) and their configuration
    - Application structure including syscall locations and arguments

- Assumption apply to a wide range of systems: OSEK, AUTOSAR
  - Industry standard widely employed in the automotive industry
  - Static configuration at compile-time
  - Fixed-priority scheduling with threads and ISRs
  - Stack-based priority ceiling protocol (PCP) for resources

Application – Example System

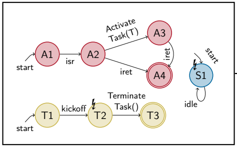

```
ISR(I1) {
   isr()
   if (cond)
     ActivateTask(T);
   iret();
}

TASK(T) {
   kickoff()
   computation();
   TerminateTask();
}
```



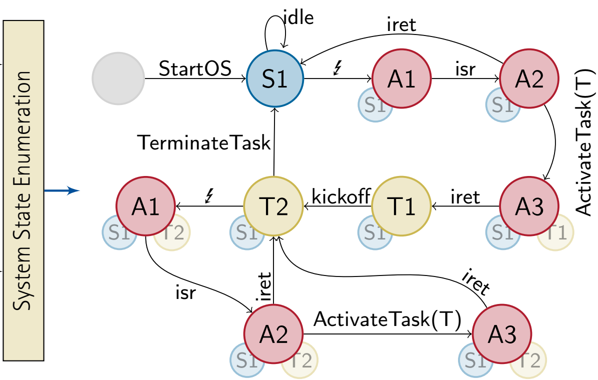
Step 1:

- Extract a finite state machine from the application code
- Application FSMs generate system-call “signals” towards the RTOS
- Computation code is ignored, since it cannot modify the RTOS state

The System State Machine




```
SYSTEM {
  TASK T {
    PRIO = 4;
    ACTIVATION = 1;
  }
  ISR I1 {
    DEVICE = 35;
  }
}
```



Step 2:

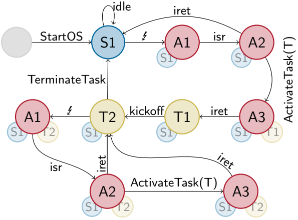
(LCTES'15, TECS'17)

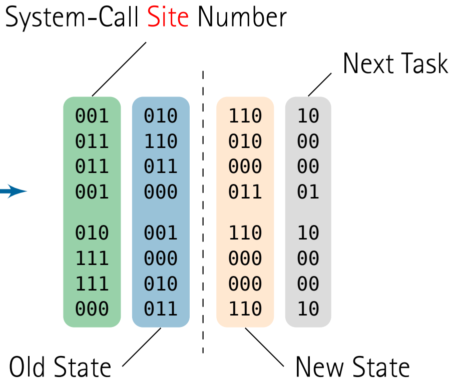
- Combine system-semantic, system configuration, and app FSMs
- Explicitly enumerate all possible system states
- Every state exposes one currently running thread

Outline

- Motivation and Introduction
- RTOS–Application Interaction Model
- **The OSEK-V Processor Pipeline**
- Evaluation
- Conclusion

From the model to the implementation

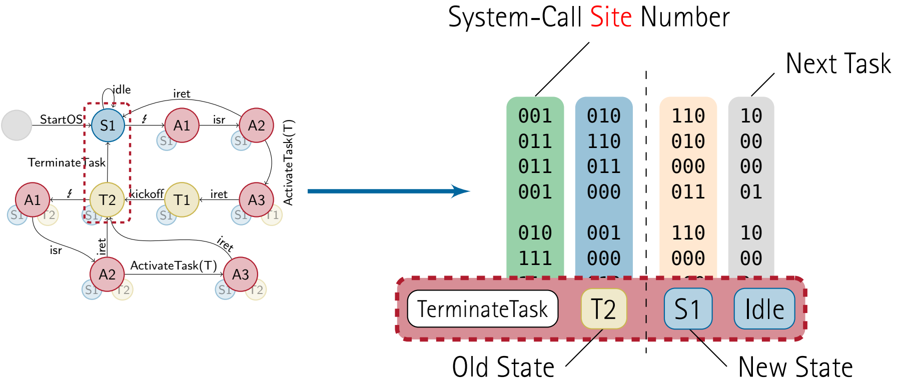




Step 3:

- Minimize the (deterministic) finite state machine
- Assign state and transition encodings
- Minimize the truth table for hardware implementation

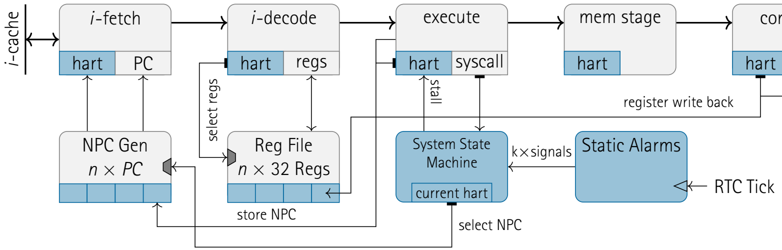
From the model to the implementation



Step 3:

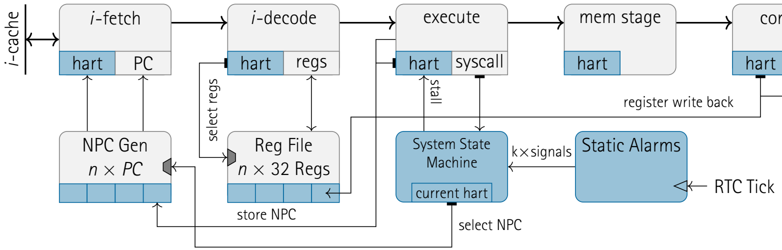
- Minimize the (deterministic) finite state machine
- Assign state and transition encodings
- Minimize the truth table for hardware implementation

The OSEK-V Pipeline



- RISC-V and the Rocket Core
  - New free and open research ISA (around since 2015)
  - Rocket is a 5-stage pipeline implementation written in Chisel
  - Multi-core capable, but no hardware multithreading

The OSEK-V Pipeline



- RISC-V and the Rocket Core
  - New free and open research ISA (around since 2015)
  - Rocket is a 5-stage pipeline implementation written in Chisel
  - Multi-core capable, but no hardware multithreading

Step 4:

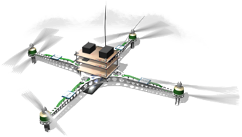
- Map every OS thread to one hardware thread
- System-state machine schedules hardware threads
- Execute stage sends system-call number to SSM (`osek` instruction)
- Component for alarms with constant period and phase (Static Alarms)

Outline

- Motivation and Introduction
- RTOS–Application Interaction Model
- The OSEK-V Processor Pipeline
- **Evaluation**
- Conclusion

Evaluation Scenario: Quadrotor Flight Control

*i4*Copter



- Realistic safety-critical real-time system
- 11 threads, 3 timers, 1 ISR, 53 system-call sites
- Used only task-setup (no actual compuation code)

*d*OSEK

- Framework for OSEK system analysis and kernel generator
- Extracts the system state machine for a given application
- Adapts application code to used custom `osek` instructions

OSEK-V

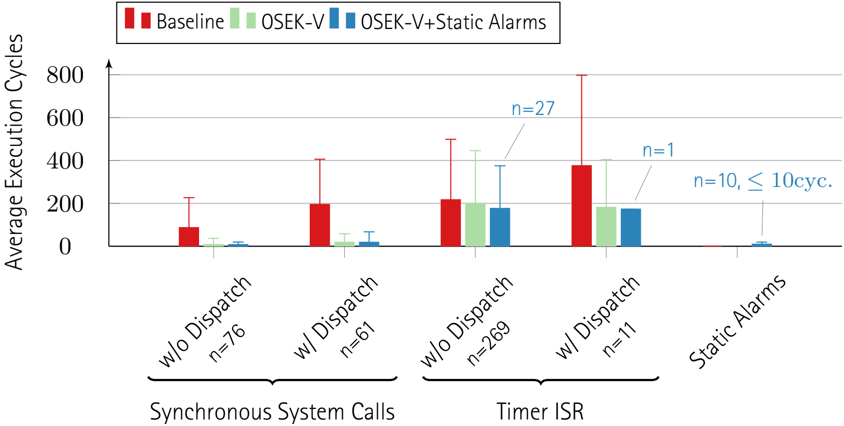
- CPU Pipeline provides the OSEK kernel functionality
- Generates Verilog code for the Zynq-7020 FPGA (ZedBoard)
- Generates cycle-accurate C++ simulator that runs application

FPGA Synthesis for ZedBoard (Zynq-7020)

- Generate System State Machine in 73.68 s (96 % state encoding)
  - Before state-machine minimization: 4834 states, 7479 transitions
  - After state-machine minimization: 701 states, 1246 transitions
  - Minimized Truth Table: 781 Rows/Clauses
- Synthesize the Rocket in less than 10 minutes with Xilinx toolchain
  - Memory LUTs go into register files (96 %)
  - Logic LUTs go into system state machine (76 %)

	Baseline	OSEK-V	+ Static Alarms
Kernel Text Segment (bytes)	14 386	8669	8393
Kernel Data Segment (bytes)	1908	410	354
Lookup Tables (LUT)	29 460	32 041	32 341
Lookup Tables for Memory (Mem-LUT)	1033	2016	2016
Flip-Flops	14 208	14 129	14 196

Simulation in Cycle-Accurate Simulator



- Synchronous syscalls at least 75% faster
- Interrupts are still more expensive, as not mapped to own hardware threads
- Static alarms offload timer handling

Outline

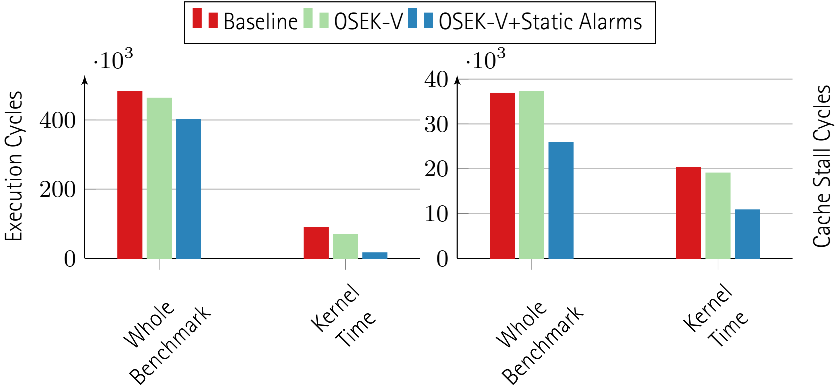
- Motivation and Introduction
- RTOS–Application Interaction Model
- The OSEK-V Processor Pipeline
- Evaluation
- **Conclusion**

Conclusion

- OSEK-V as a hybrid solution between hardwired control and software
  - RTOS behavior integrated into processor pipeline
  - Application logic is updatable to a certain degree

- OSEK-V has unique properties as an RTOS platform
  - **Automatic** application-specific pipeline derivation
  - **Fast** scheduling and thread context switches
  - **Predictable**: Operating System has minimal influence on hardware state
  - **Scales** with your application: Small systems result in low costs
  - **Easy** to verify actual implementation

Simulation in Cycle-Accurate Simulator



- Execute i4Copter system image in cycle-accurate C++ simulator
- IRQs are blocked during system calls and interrupt handling
- Average IRQ Blockade drops from 195 cycles to 41 cycles (with static alarms)