Semi-Extended Tasks:
Efficient Stack Sharing Among Blocking Threads

Christian Dietrich, Daniel Lohmann

Leibniz Universität Hannover

September 2018

supported by DFG

Memory Consumption in Embedded Systems

- Development of embedded systems is highly price sensitive
  - High number of deployed processors: > 100 MCUs per car
  - High overall yield: 11 million cars (2017: VW)

⇒ Small savings have huge impact: -0.01€/part $\approx$ 110 k EUR for VW

Memory Consumption in Embedded Systems

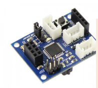
- Development of embedded systems is highly price sensitive
  - High number of deployed processors: > 100 MCUs per car
  - High overall yield: 11 million cars (2017: VW)

⇒ Small savings have huge impact: -0.01€/part ≈ 110 k EUR for VW

- Quantized RAM Purchase: Microchip ATXMega C3 Series:

Part	Flash	RAM	Price
ATXMEGA64C3	64 kB	4 kB	4.05 EUR
ATXMEGA128C3	128 kB	8 kB	4.11 EUR
ATXMEGA256C3	256 kB	16 kB	5.06 EUR
ATXMEGA384C3	384 kB	32 kB	6.12 EUR

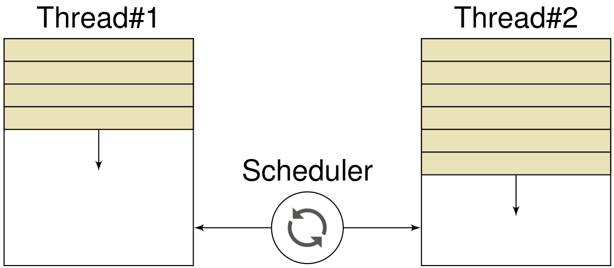
Memory Consumption in Embedded Systems



- Developmen ce sensitive
  - High numbe s per car
  - High overal
  - ⇒ Small savin 10 k EUR for VW
- Quantized R C3 Series:

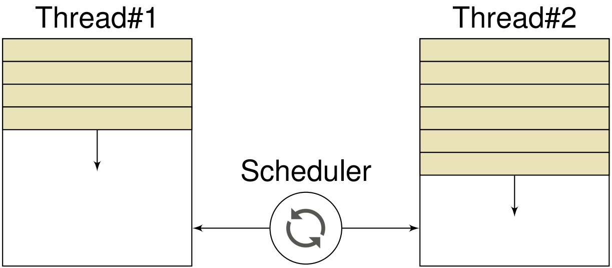
			rice
			EUR
			EUR
ATXMEGA256C3	256 kB	16 kB	5.06 EUR
ATXMEGA384C3	384 kB	32 kB	6.12 EUR

Living in Private: Normal Threads



- Normal threads live on their private stack
  - Function calls push a new stack frame onto the private stack
  - Kernel switches arbitrarily between threads and stacks

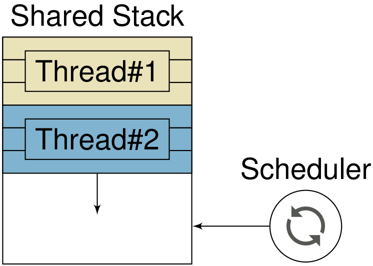
Living in Private: Normal Threads



- Normal threads live on their private stack
  - Function calls push a new stack frame onto the private stack
  - Kernel switches arbitrarily between threads and stacks

- Real-time schedules are much more restricted
  - Not all preemptions/resumptions are possible at any point
  - Stack reusable if two threads are never simultaneously ready

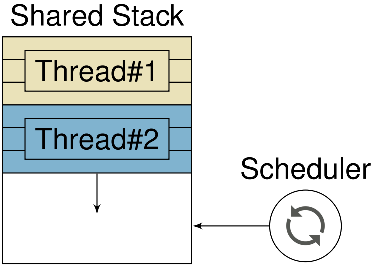
Living in a Commune: Basic Tasks



- OSEK/AUTOSAR has the concept of basic tasks
  - . . . live, tightly packed, on the same stack
  - . . . must have run-to-completion semantic and cannot wait

⇒ Only the top-most basic task can be running (by construction)

Living in a Commune: Basic Tasks



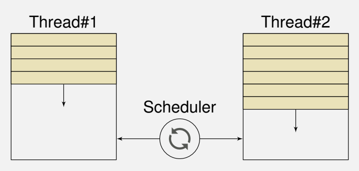
- OSEK/AUTOSAR has the concept of basic tasks
  - . . . live, tightly packed, on the same stack
  - . . . must have run-to-completion semantic and cannot wait

  ⇒ Only the top-most basic task can be running (by construction)

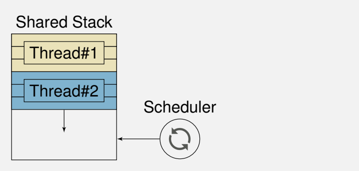
- Worst-case stack consumption depends on real-time parameters
  - Preemption thresholds, non-preemptability, priority-ceiling protocol

Problem Field

Extended Tasks



Basic Tasks

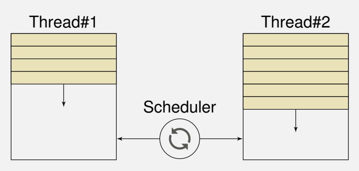


- \+ Fully flexible (can wait)
- – High static stack usage

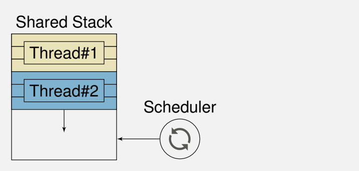
- – Cannot wait passively
- \+ Stack-sharing potential

Problem Field

Extended Tasks



Basic Tasks



- \+ Fully flexible (can wait)
- – High static stack usage

- – Cannot wait passively
- \+ Stack-sharing potential

Semi-Extended Tasks live on two Stacks

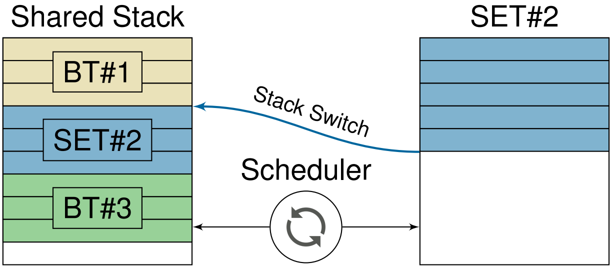
Approach

- Semi-Extended Task Mechanism
- Worst-Case Stack Consumption
- Optimize Stack Consumption with SETs

Approach

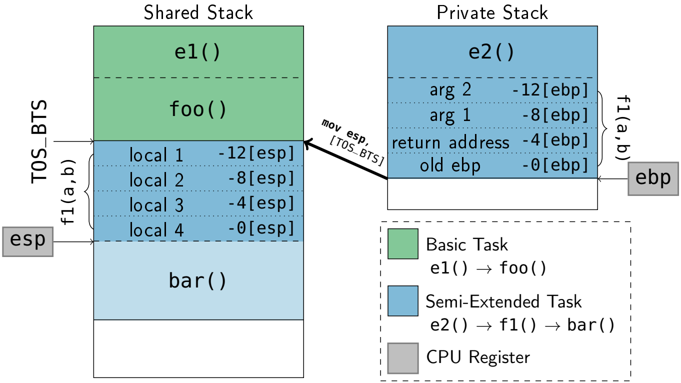
- **Semi-Extended Task Mechanism**
- Worst-Case Stack Consumption
- Optimize Stack Consumption with SETs

Semi-Extended Tasks (SET)



- SETs switch autonomously to the shared stack
  - Transition between stacks happens at stack-switch functions
  - SETs start as Extended Tasks and can become Basic Tasks
  - Special compiler-generated function prologue

Technical Detail: Function Prologue

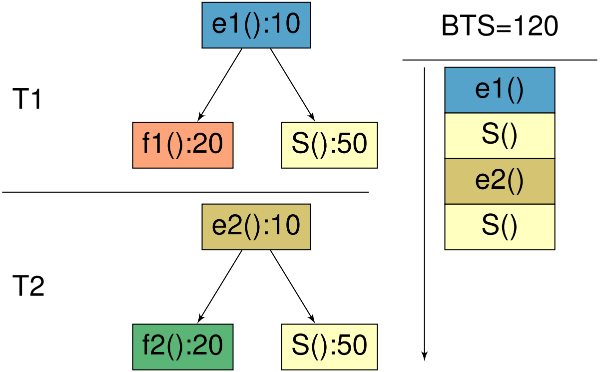


```
<f1>:
    ;; Function - Prologue
    push    ebp                  ; Save old framepointer
    mov     ebp, esp             ; Load new framepointer
    mov     esp, [TOS_BTS]       ; Switch to shared stack
    sub     esp, 16              ; Allocate local variables
```

Approach

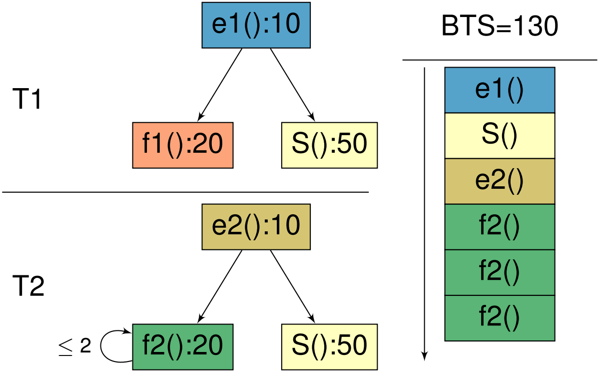
- Semi-Extended Task Mechanism
- **Worst-Case Stack Consumption**
- Optimize Stack Consumption with SETs

Worst-Case Stack Consumption (WCSC)



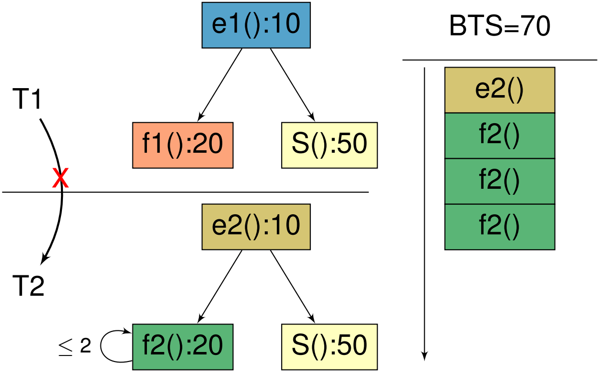
- WCSC analysis must consider different constraints
  - Intra-Thread Callgraphs

Worst-Case Stack Consumption (WCSC)



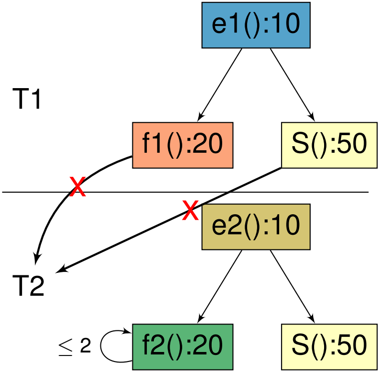
- WCSC analysis must consider different constraints
  - Intra-Thread Callgraphs
  - Recursion

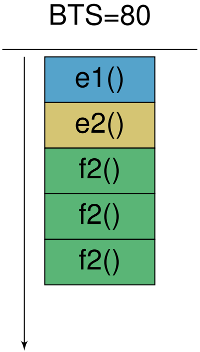
Worst-Case Stack Consumption (WCSC)

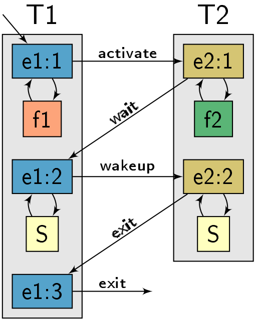


- WCSC analysis must consider different constraints
  - Intra-Thread Callgraphs
  - Recursion
  - Preemption Constraints

Worst-Case Stack Consumption (WCSC)

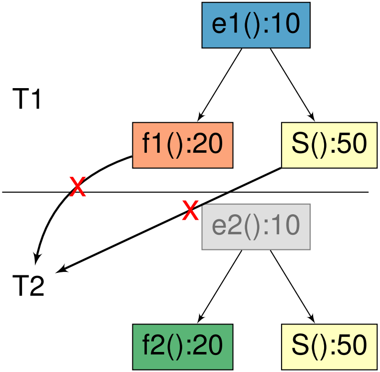


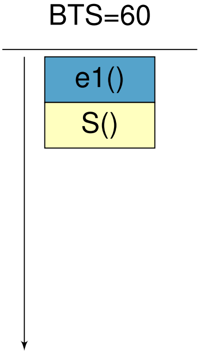


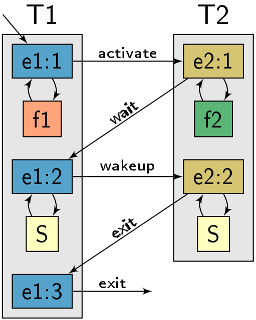


- WCSC analysis must consider different constraints
  - Intra-Thread Callgraphs
  - Recursion
  - Preemption Constraints
  - Global Control Flow

Worst-Case Stack Consumption (WCSC)







- WCSC analysis must consider different constraints
  - Intra-Thread Callgraphs
  - Recursion
  - Preemption Constraints
  - Global Control Flow
  - SET Stack Switches

Worst-Case Stack Consumption (WCSC)

- Current WCSC Analyses for Shared Stack are Coarse-Grained
  - Analyse each Task in Isolation
  - Combine Stack Consumption According to Preemption Rules

Worst-Case Stack Consumption (WCSC)

- Current WCSC Analyses for Shared Stack are Coarse-Grained
  - Analyse each Task in Isolation
  - Combine Stack Consumption According to Preemption Rules

- We suggest a combined Approach with IPET/ILP Solver
  - Model WCSC analysis as a maximum-flow problem
  - Search for costliest {preemption chain, function stacking}
  - Allows for fine-grained preemption constraints:

$$\text{forbid}(\text{T1} \longrightarrow \text{T2}) \quad \text{forbid}(\text{T1[S]} \longrightarrow \text{T2})$$

Worst-Case Stack Consumption (WCSC)

- Current WCSC Analyses for Shared Stack are Coarse-Grained
  - Analyse each Task in Isolation
  - Combine Stack Consumption According to Preemption Rules
- We suggest a combined Approach with IPET/ILP Solver
  - Model WCSC analysis as a maximum-flow problem
  - Search for costliest {preemption chain, function stacking}
  - Allows for fine-grained preemption constraints:

$$\text{forbid}(T1 \longrightarrow T2) \quad \text{forbid}(T1[S] \longrightarrow T2)$$

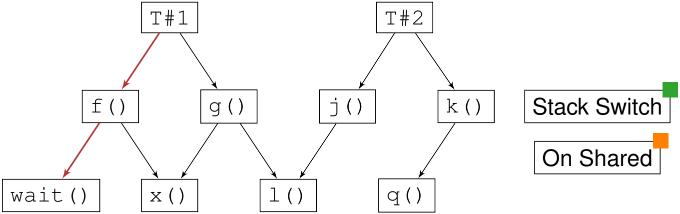
- Fine-Grained Preemption Constraints
  - Extract Constraints from Global Control-Flow Graph
  - Flow-Sensitive Static Analysis of Application and RTOS
  - Presented in previous work: LCTES'15, TECS'17

Approach

- Semi-Extended Task Mechanism
- Worst-Case Stack Consumption
- **Optimize Stack Consumption with SETs**

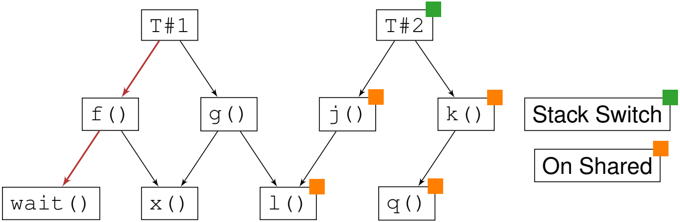
Where to Switch Stacks?

- Select stack-switch function to minimize the WCSC.
  - Parents of blocking system calls are forbidden



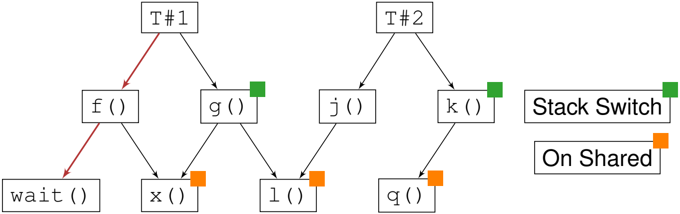
Where to Switch Stacks?

- Select stack-switch function to minimize the WCSC.
  - Parents of blocking system calls are forbidden
  - Children of stack-switch functions are forbidden



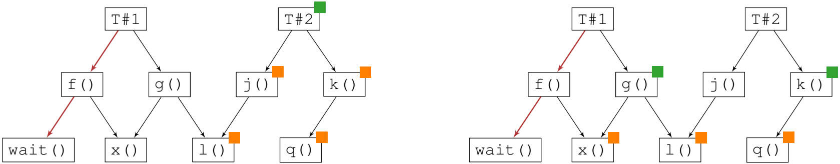
Where to Switch Stacks?

- Select stack-switch function to minimize the WCSC.
  - Parents of blocking system calls are forbidden
  - Children of stack-switch functions are forbidden
  - Possibilities: extended, basic, or semi-extended tasks



Where to Switch Stacks?

- Select stack-switch function to minimize the WCSC.
  - Parents of blocking system calls are forbidden
  - Children of stack-switch functions are forbidden
  - Possibilities: extended, basic, or semi-extended tasks



Minimizing the WCSC: Two-level Optimization

Results

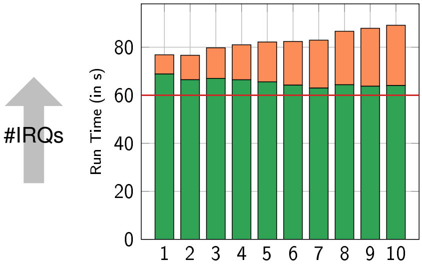
- Synthetic Benchmark Scenario
- Run-Time of the Optimization
- Stack-space Savings

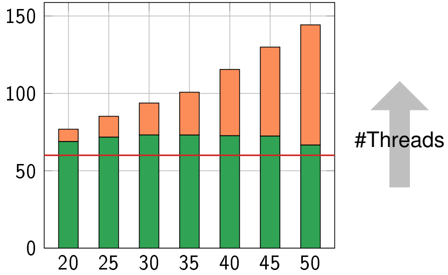
Synthetic Benchmark Scenarios

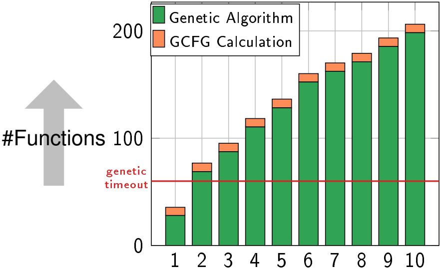
- Evaluate the stack-optimization with > 14000 systems
  - #Threads: 20 – 50
  - #IRQs: 1 – 20
  - #Waiting Threads: 0 – 15
  - #Functions: 100 – 1000
  - #Priority-Ceiling Resources: 1 – 10

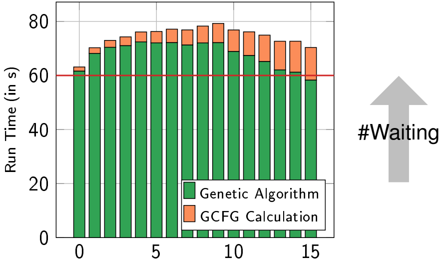
- Integration into Whole-System Generator
  - dOSEK: Python framework for system analysis and kernel generation
  - LLVM: Extract sizes of stack frames and stack-switch prologue
  - Gurobi: state-of-the-art ILP solver

Run-Time of Optimization

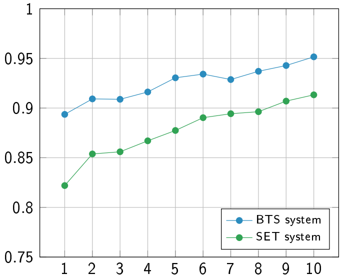




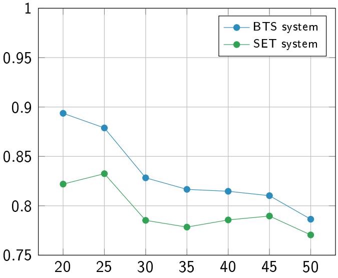




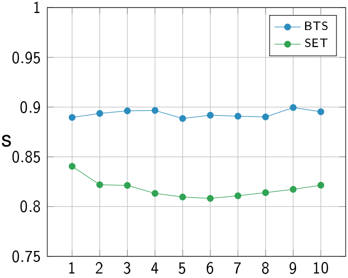
Stack-Usage Factors



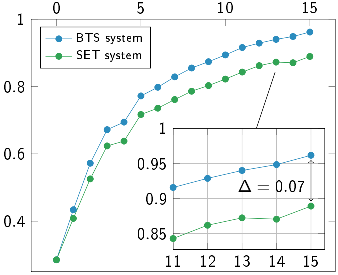
#IRQs



#Threads



#Functions



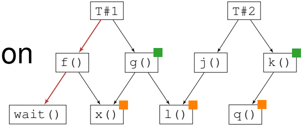
#Waiting

Conclusion

Conclusion

- Semi-Extended Tasks
  - SETs switch to Shared Stack if possible
  - Switching is efficient and does not involve the RTOS
- Fine-Grained Worst-Case Stack Consumption Analysis
  - Real-Time Properties (Priorities, Preemption Thresholds)
  - Flow-Sensitive Preemption Constraints
  - Supports Semi-Extended Tasks
- Stack-Space Saving compared to pure BTS systems
  - 7 percent on average, up to 52 percent
  - 80 percent of all systems used less stack space

Genetic Algorithm as a Higher-Level Optimization



- Genetic algorithm to find a good configuration
  - Encode configuration as bit-vector
  - Bitmasks verify configuration
  - Configurations can be breed, mixed, and mutated

g()	x()	l()	T#2	j()	k()	q()
1	0	0	0	0	1	0

- Genetic Algorithm with Initial Population
  1. Generate new bit-vectors by mutation and cross-over
  2. Calculate fitness (WCSC) with IPET/ILP solver
  3. Select top 20 switch-configurations
  4. Goto 1, until satisfied (60 seconds of no progress)