Semi-Extended Tasks:
Efficient Stack Sharing Among Blocking Threads

Christian Dietrich, Daniel Lohmann

Leibniz Universität Hannover

December 14, 2018

supported by DFG

Memory Consumption in Embedded Systems

Memory Consumption in Embedded Systems

98% of sold processors

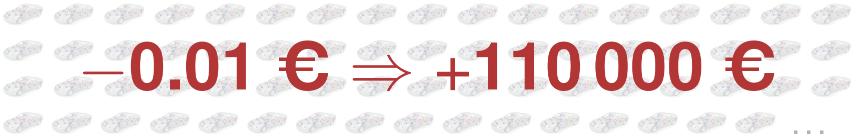
Memory Consumption in Embedded Systems

98% of sold processors

…

Memory Consumption in Embedded Systems

98% of sold processors

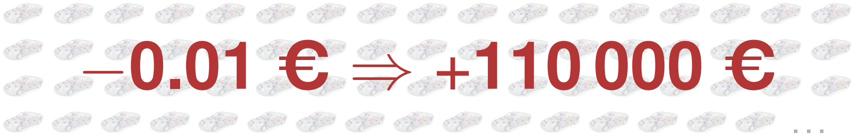


-0.01 € ⇒ $+110\,000$ €

. . .

Memory Consumption in Embedded Systems

98% of sold processors



−0.01 € ⇒ +110 000 €

. . .

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

98% ssors

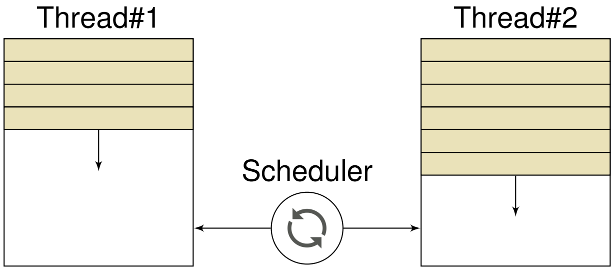
−0.0 00 €

. . .

Quantized R C3 Series:

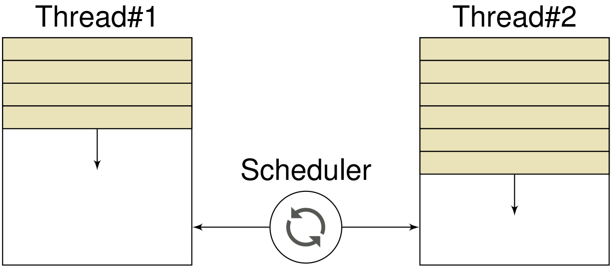
Part			e
ATXMEGA6			R
ATXMEGA1			R
ATXMEGA256C3	256 kB	16 kB	5.06 EUR
ATXMEGA384C3	384 kB	32 kB	6.12 EUR

Living in Private: Normal Threads



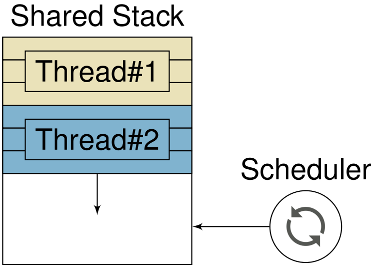
- Normal threads live on their private stack
  - Function calls push a new stack frame onto the private stack
  - Kernel switches arbitrarily between threads and stacks

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

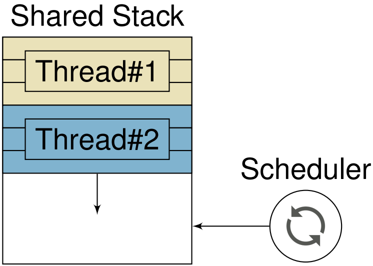
Living in a Commune: Basic Tasks



- 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



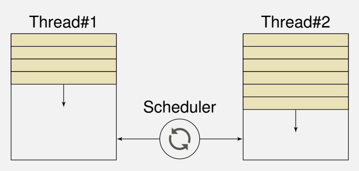
- OSEK/AUTOSAR has the concept of basic tasks
  - . . . live, tightly packed, on the same stack
  - . . . must have run-to-completion semantic and cannot wait

⇒ 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

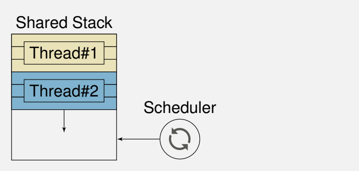
Problem Field

Extended Tasks



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

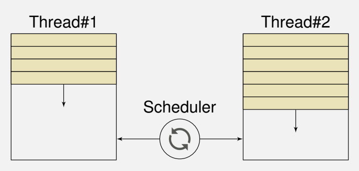
Basic Tasks



- – Cannot wait passively
- + Stack-sharing potential

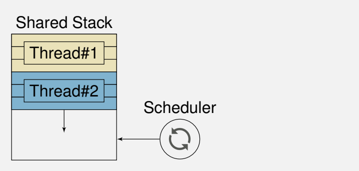
Problem Field

Extended Tasks



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

Basic Tasks



- – Cannot wait passively
- + Stack-sharing potential

Semi-Extended Tasks live on two stacks

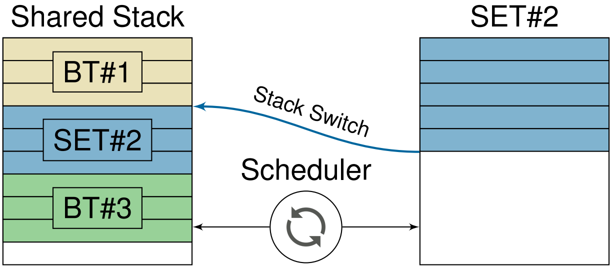
Approach

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

Approach

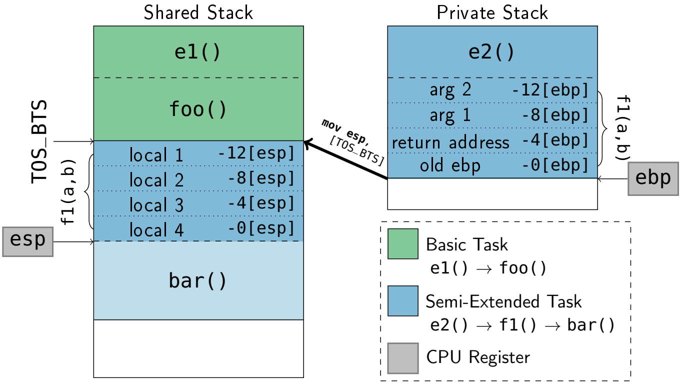
- **Semi-Extended Task Mechanism**
- Worst-Case Stack Consumption
- Optimize Stack Consumption with SETs

Semi-Extended Tasks (SET)



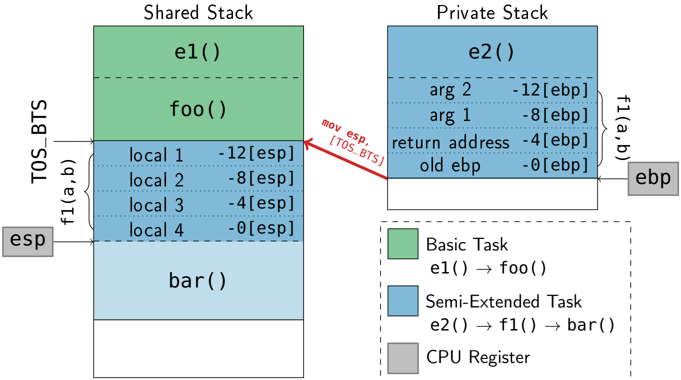
- SETs switch autonomously to the shared stack
  - Transition between stacks happens at stack-switch functions
  - SETs start as Extended Tasks and can become Basic Tasks
  - Special compiler-generated function prologue

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
```

Technical Detail: Function Prologue



Shared Stack

e1()

foo()

local 1 -12[esp]

local 2 -8[esp]

local 3 -4[esp]

local 4 -0[esp]

bar()

TOS_BTS

f1(a,b)

esp

mov esp, [TOS_BTS]

Private Stack

e2()

arg 2 -12[ebp]

arg 1 -8[ebp]

return address -4[ebp]

old ebp -0[ebp]

f1(a,b)

ebp

🟩 Basic Task e1() → foo()

🟦 Semi-Extended Task e2() → f1() → bar()

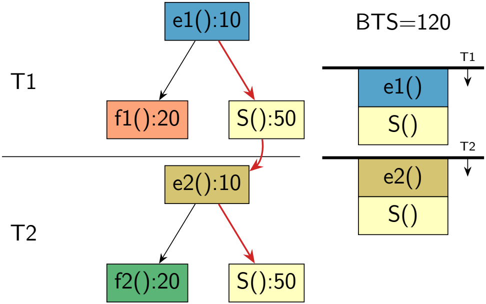
⬜ CPU Register

```
<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

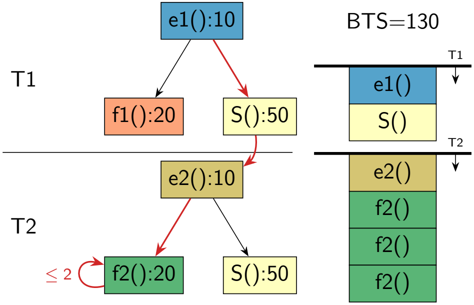
- Semi-Extended Task Mechanism
- **Worst-Case Stack Consumption**
- Optimize Stack Consumption with SETs

Worst-Case Stack Consumption (WCSC)



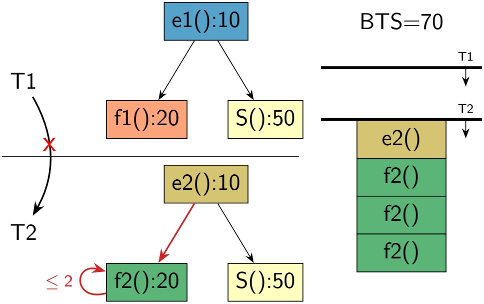
- WCSC analysis must consider different constraints
  - Intra-Thread Callgraphs

Worst-Case Stack Consumption (WCSC)



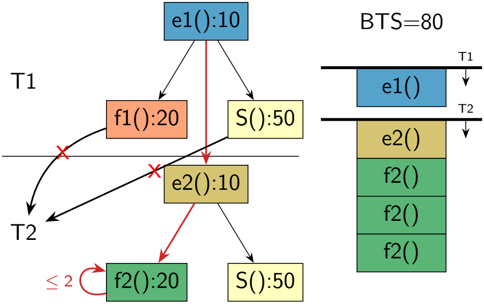
- WCSC analysis must consider different constraints
  - Intra-Thread Callgraphs
  - Recursion

Worst-Case Stack Consumption (WCSC)



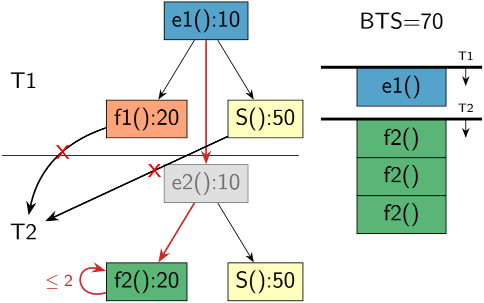
- WCSC analysis must consider different constraints
  - Intra-Thread Callgraphs
  - Recursion
  - Preemption Constraints

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:

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

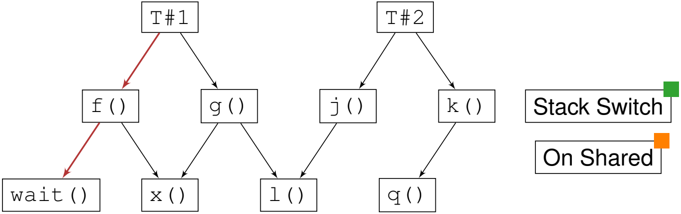
- Fine-Grained Preemption Constraints
  - Extract constraints from global control-flow graph
  - Flow-sensitive static analysis of application and RTOS
  - Presented in previous work: LCTES'15, TECS'17

Approach

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

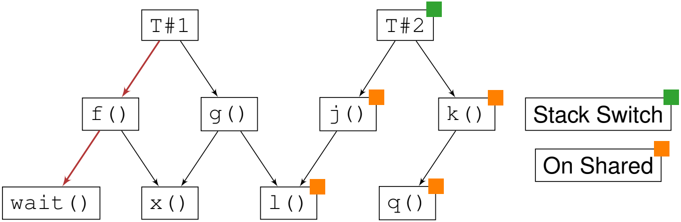
Where to Switch Stacks?

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



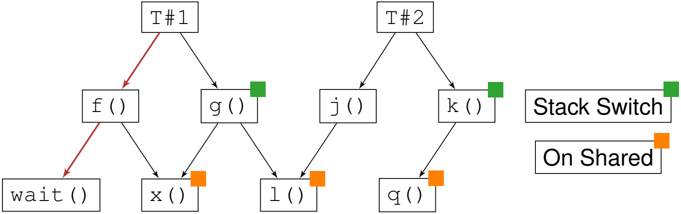
Where to Switch Stacks?

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



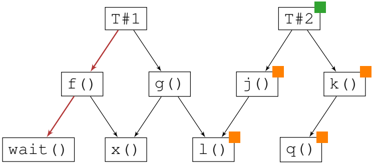
Where to Switch Stacks?

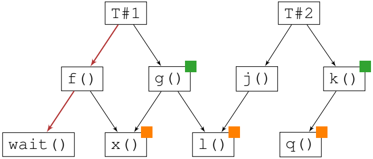
- Select stack-switch function to minimize the WCSC.
  - Parents of blocking system calls are forbidden
  - Children of stack-switch functions are forbidden
  - Possibilities: extended, basic, or semi-extended tasks



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

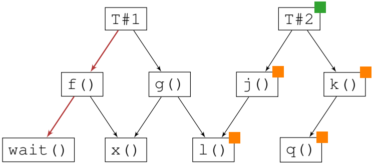


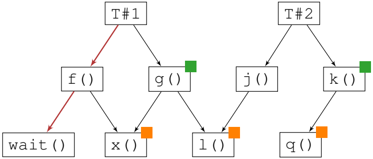


Minimizing the WCSC: Two-level Optimization

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

⇒ Genetic Algorithm with WCSC as Fitness Function

Results

- Generated Benchmark Scenarios
- Stack-space Savings

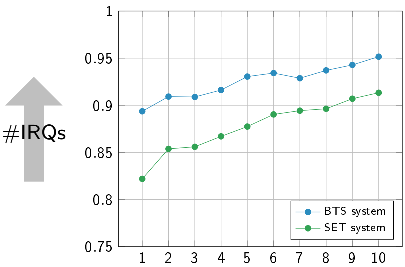
Generated Benchmark Scenarios

- Evaluation with $\geq$ 14000 generated systems
  - Based on a base configuration, scale in 5 dimensions
  - Compare ET-only, BT-only, and BT+SET systems

Dimension	Base	Range
#Threads	20	20 – 50
#blocking Threads	1	0 – 15
#IRQs	1	1 – 20
#Functionen	200	100 – 1000
#Critical Regions	1	1 – 10

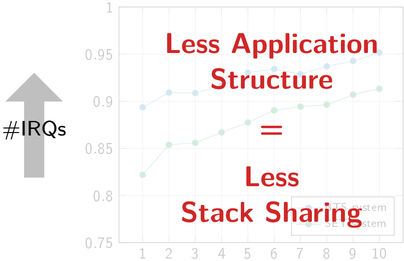
- Integration into Whole-System Generator
  - dOSEK: Python framework for system analysis and kernel generation
  - LLVM: Extract sizes of stack frames and stack-switch prologue
  - Gurobi: state-of-the-art ILP solver

Stack-Consumption Factors



Only Private Stacks

Stack-Consumption Factors



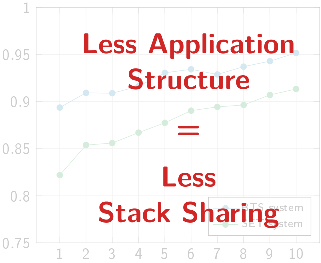
Less Application Structure

=

Less Stack Sharing

Stack-Consumption Factors

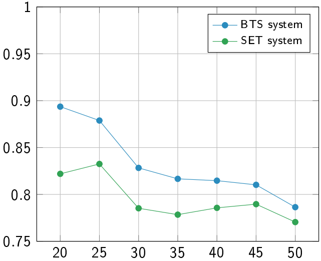
#IRQs



Less Application Structure

=

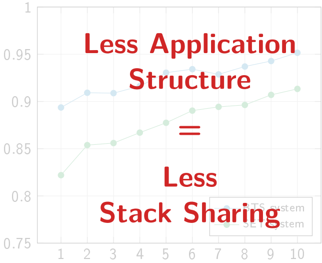
Less Stack Sharing



#Threads

Stack-Consumption Factors

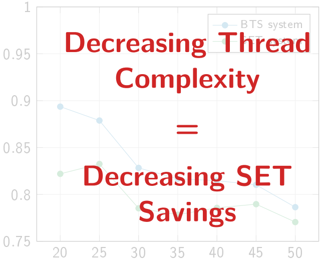
#IRQs



Less Application Structure

=

Less Stack Sharing



Decreasing Thread Complexity

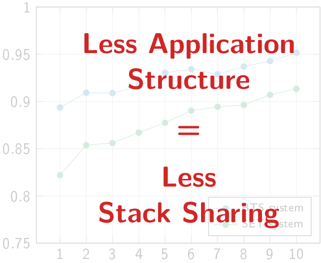
=

Decreasing SET Savings

#Threads

Stack-Consumption Factors

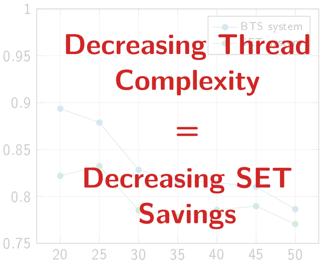
#IRQs



Less Application Structure

=

Less Stack Sharing



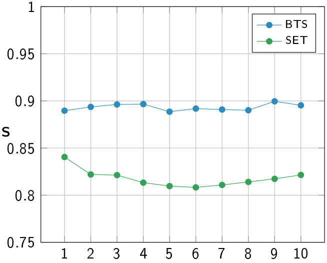
Decreasing Thread Complexity

=

Decreasing SET Savings

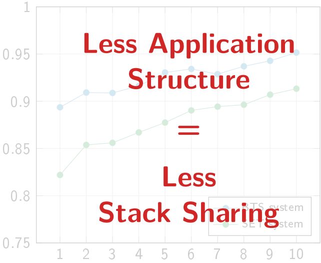
#Threads

#Functions



Stack-Consumption Factors

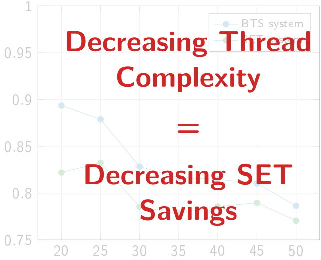
#IRQs



Less Application Structure

=

Less Stack Sharing



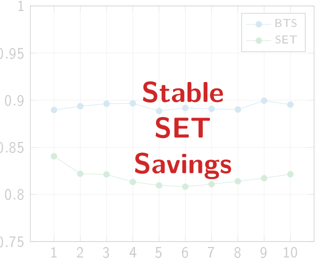
Decreasing Thread Complexity

=

Decreasing SET Savings

#Threads

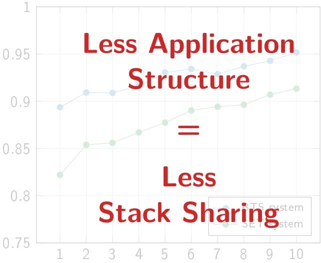
#Functions



Stable SET Savings

Stack-Consumption Factors

#IRQs

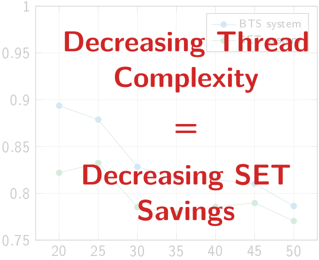


Less Application Structure

=

Less Stack Sharing

#Threads

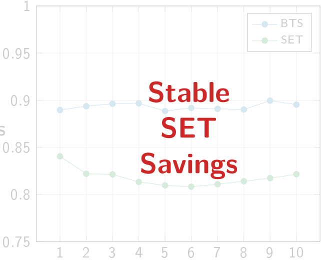


Decreasing Thread Complexity

=

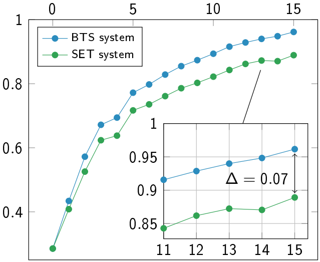
Decreasing SET Savings

#Functions



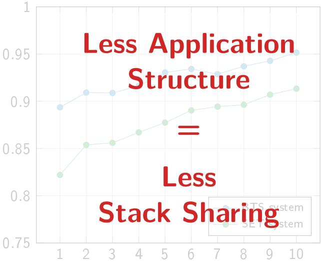
Stable SET Savings

#Waiting



Stack-Consumption Factors

#IRQs

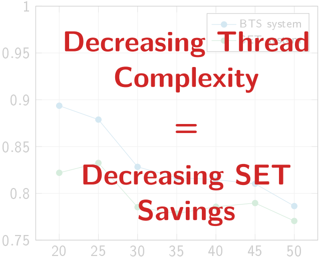


Less Application Structure

=

Less Stack Sharing

#Threads

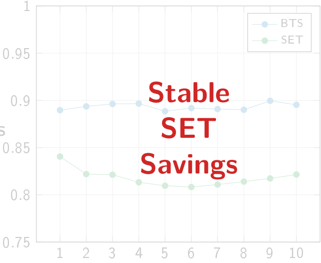


Decreasing Thread Complexity

=

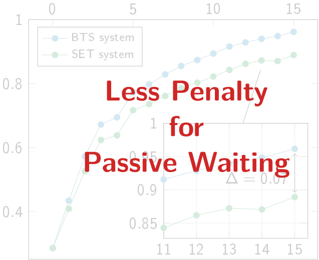
Decreasing SET Savings

#Functions



Stable SET Savings

#Waiting



Less Penalty for Passive Waiting

Conclusion

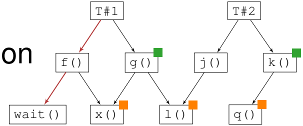
Conclusion

- Semi-Extended Tasks
  - SETs switch to shared stack if possible
  - Switching is efficient and does not involve the RTOS
  - Smaller penalty for passive waiting

- 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

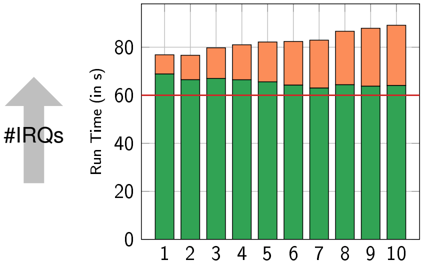


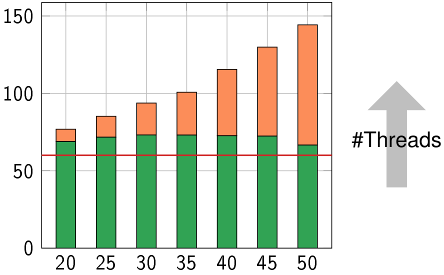
- Genetic algorithm to find a good configuration
  - Encode configuration as bit-vector
  - Bitmasks verify configuration
  - Configurations can be breed, mixed, and mutated

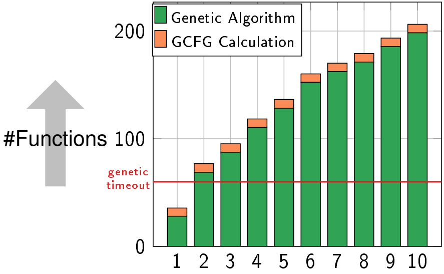
g()	x()	l()	T#2	j()	k()	q()
1	0	0	0	0	1	0

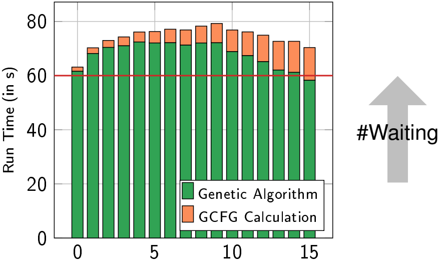
- Genetic Algorithm with Initial Population
  1. Generate new bit-vectors by mutation and cross-over
  2. Calculate fitness (WCSC) with IPET/ILP solver
  3. Select top 20 switch-configurations
  4. Goto 1, until satisfied (60 seconds of no progress)

Run-Time of Optimization


#IRQs
Run Time (in s)


#Threads


#Functions
genetic timeout
🟩 Genetic Algorithm
🟧 GCFG Calculation


#Waiting
Run Time (in s)
🟩 Genetic Algorithm
🟧 GCFG Calculation