Semi-Extended Tasks: Efficient Stack Sharing Among Blocking Threads

Christian Dietrich, Daniel Lohmann

Leibniz Universität Hannover

December 14, 2018
Memory Consumption in Embedded Systems
98% of sold processors
98% of sold processors...
98% of sold processors

-0.01 € \Rightarrow +110\,000 \text{ €}
Memory Consumption in Embedded Systems

98% of sold processors

Quantized RAM Purchase: Microchip ATXmega C3 Series:

<table>
<thead>
<tr>
<th>Part</th>
<th>Flash (kB)</th>
<th>RAM (kB)</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATXMEGA64C3</td>
<td>64</td>
<td>4</td>
<td>4.05 EUR</td>
</tr>
<tr>
<td>ATXMEGA128C3</td>
<td>128</td>
<td>8</td>
<td>4.11 EUR</td>
</tr>
<tr>
<td>ATXMEGA256C3</td>
<td>256</td>
<td>16</td>
<td>5.06 EUR</td>
</tr>
<tr>
<td>ATXMEGA384C3</td>
<td>384</td>
<td>32</td>
<td>6.12 EUR</td>
</tr>
</tbody>
</table>
Quantized RAM Purchase: Microchip ATXMEGA C3 Series:

<table>
<thead>
<tr>
<th>Part</th>
<th>Flash RAM</th>
<th>Program RAM</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATXMEGA64C3</td>
<td>64 kB</td>
<td>4 kB</td>
<td>4.05 EUR</td>
</tr>
<tr>
<td>ATXMEGA128C3</td>
<td>128 kB</td>
<td>8 kB</td>
<td>4.11 EUR</td>
</tr>
<tr>
<td>ATXMEGA256C3</td>
<td>256 kB</td>
<td>16 kB</td>
<td>5.06 EUR</td>
</tr>
<tr>
<td>ATXMEGA384C3</td>
<td>384 kB</td>
<td>32 kB</td>
<td>6.12 EUR</td>
</tr>
</tbody>
</table>

98% of sold processors
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
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
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)
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
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 >:

1   ;; Function — Prologue
2    push    ebp                   ; Save old framepointer
3    mov     ebp, esp              ; Load new framepointer
4    mov     esp, [TOS_BTS]        ; Switch to shared stack
5    sub     esp, 16              ; Allocate local variables
```

**Shared Stack**
- `e1()`
- `foo()`
- `local 1 -12[esp]`
- `local 2 -8[esp]`
- `local 3 -4[esp]`
- `local 4 -0[esp]`
- `bar()`

**Private Stack**
- `e2()
  - arg 2 -12[ebp]
  - arg 1 -8[ebp]
  - return address -4[ebp]
  - old ebp -0[ebp]`

**CPU Register**
- `ebp`

**Shared Stack** and **Private Stack** are separated by the function prologue. The diagram illustrates the allocation of local variables and the switching between the stacks.

---

**LUH SET — Stack Memory as a Shared Resource**
Technical Detail: Function Prologue

1. \texttt{\textless f1 \textgreater :}
2. \hspace{1em} ;; Function — Prologue
3. \hspace{1em} push ebp \hspace{2em} ;; Save old framepointer
4. \hspace{1em} mov ebp, esp \hspace{2em} ;; Load new framepointer
5. \hspace{1em} mov esp, [TOS_BTS] \hspace{2em} ;; Switch to shared stack
6. \hspace{1em} sub esp, 16 \hspace{2em} ;; 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

T1

T2

\[ \text{BTS}=130 \]

\[ \leq 2 \]
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

- We suggest a combined approach with IPET/ILP solver
  - Model WCSC analysis as a maximum-flow problem
  - Search for costliest {preemption chain, function stacking}

- 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
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:
    
    
    $$\text{forbid}(T1 \rightarrow T2) \quad \text{forbid}(T1[S] \rightarrow T2)$$
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:
    
    $$
    \text{forbid}(T_1 \rightarrow T_2) \quad \text{forbid}(T_1[S] \rightarrow T_2)
    $$

- 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

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Base</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Threads</td>
<td>20</td>
<td>20 – 50</td>
</tr>
<tr>
<td>#blocking Threads</td>
<td>1</td>
<td>0 – 15</td>
</tr>
<tr>
<td>#IRQs</td>
<td>1</td>
<td>1 – 20</td>
</tr>
<tr>
<td>#Functionen</td>
<td>200</td>
<td>100 – 1000</td>
</tr>
<tr>
<td>#Critical Regions</td>
<td>1</td>
<td>1 – 10</td>
</tr>
</tbody>
</table>

- 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

#IRQs

Only Private Stacks

(d) Functions \( \cdot 100 \)
Stack-Consumption Factors

![Graph showing the relationship between IRQs and Stack Usage Factors for BTS and SET systems.](image)

- Less Application Structure = Less Stack Sharing

# IRQs

Stack Usage Factor

- BTS system
- SET system
Stack-Consumption Factors

- Less Application Structure
- Less Stack Sharing

(d) Functions \((\cdot 100)\)

Graph showing Stack Usage Factors for BTS and SET systems versus #IRQs and #Threads.
Stack-Consumption Factors

- Less Application Structure = Less Stack Sharing
- Decreasing Thread Complexity = Decreasing SET Savings

#IRQs

Less Application Structure

#Threads

Decreasing Thread Complexity
Stack-Consumption Factors

- Less Application Structure = Less Stack Sharing

- Decreasing Thread Complexity = Decreasing SET Savings

- #IRQs
- #Threads
- #Functions

LUH SET – Stack Memory as a Shared Ressource
Stack-Consumption Factors

Less Application Structure = Less Stack Sharing

Decreasing Thread Complexity = Decreasing SET Savings

Stable SET Savings
Stack-Consumption Factors

- Less Application Structure
  
  $\#\text{IRQs}$

- Decreasing Thread Complexity
  
  $\#\text{Threads}$

- Less Stack Sharing
  
  Stable SET Savings

- Decreasing SET Savings
  
  $\#\text{Functions}$

- $\#\text{Waiting}$
Stack-Consumption Factors

- Less Application Structure = Less Stack Sharing
- Decreasing Thread Complexity = Decreasing SET Savings
- Stable SET Savings
- Less Penalty for Passive Waiting

#IRQs

#Threads

#Functions

#Waiting

LUH SET – Stack Memory as a Shared Ressource
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

<table>
<thead>
<tr>
<th>g()</th>
<th>x()</th>
<th>l()</th>
<th>T#2</th>
<th>j()</th>
<th>k()</th>
<th>q()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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)