EMPRESS: an Efficient and effective Method for PREdictable Stack Sharing

Sebastian Altmeyer, Reinder J. Bril, Paolo Gai
Reconciling **Predictability** and **Performance**

**RTOS Stack Implementation:**

- enables precise **timing verification**
- enables **optimization of memory/cache layout**
- reduces **stack usage**
- limits **runtime overheads**
Content

1) System Assumptions
2) Current Stack Implementations
3) EMPRESS
4) Evaluation/Case Study
5) Conclusions
System Assumptions

- **single processor** system
- **direct address-mapping** memory to cache (no MMU)
- *n* tasks: $\tau_1, \tau_2, \ldots, \tau_n$ (precedence constraints allowed)
- **priority driven scheduling** (FPPS, FPTS, EDF…)
- **early blocking** resource access protocol
- **no suspension**/no data left on the stack between task instances
- **maximum stack usage** $SU_i$ per task $\tau_i$
Content

1) System Assumption
2) **Current Stack Implementations**
3) EMPRESS
4) Evaluation/Case Study
5) Conclusions
Dedicated Stacks

- each task is given **dedicated stack area**
- **unique stack pointer** per task
- total stack usage: $\sum SU_i$
  - potentially wasted memory
Shared Stack

- all tasks share stack area
- variable stack pointer per task
  - potentially reduced stack usage
Dedicated Stacks

- Stack pointer **independent of preemption point/scenario**
- Stack pointer **statically known**
- Potentially **memory gaps** during runtime
Shared Stack

- Stack pointer depend on preemption point/scenario
- Stack pointer dynamically computed
- no memory gaps
Timing Analysis and **Static** Stack Pointer

![Diagram of a stack with static stack pointer τ_i]
Timing Analysis and **Static** Stack Pointer

![Diagram showing stack and cache with stack pointer τ_i mapping to cache]

- Stack
- Cache
- Stack pointer τ_i
- Maps to
Timing Analysis and **Static** Stack Pointer

Stack

Cache

\[ \tau_i \]

maps to
Timing Analysis and **Static** Stack Pointer

Static Timing Analysis

- **Stack**
- **Cache**

- \( \tau_i \)
- Maps to

- Classify
  - Cache Hit
  - Cache Miss
Timing Analysis and **Variable** Stack Pointer

![Diagram showing stack with variable $\tau_i$]
Timing Analysis and **Variable** Stack Pointer

![Diagram showing stack with variable stack pointer $\tau_i$]
Timing Analysis and **Variable** Stack Pointer
Timing Analysis and **Variable** Stack Pointer
Timing Analysis and Variable Stack Pointer
Timing Analysis and **Variable** Stack Pointer

Static Timing Analysis
Timing Analysis and **Variable** Stack Pointer

- Stack
- Cache
- Classify
  - Cache Hit
  - Cache Miss

Static Timing Analysis
Timing Analysis and **Variable** Stack Pointer

Access to an unknown address:
- Assumed a **cache miss**
- **Pollutes** abstract cache state
- Results in unknown **access times** (NUMA Architecture)
- Prevents optimization of the cache/memory layout

Absint's aiT Timing analyzer guesses a stack pointer if none is provided [12]

**Static timing analysis implies dedicated stacks**
Content

1) System Assumptions
2) Current Stack Implementation
3) EMPRESS
4) Evaluation/Case Study
5) Conclusions
**EMPRESS:** an **E**fficient and effective **M**ethod for **PRE**dictable **S**tack **S**haring

**Idea:**

Static Stack Pointer = Worst-Case Address in Shared Stack

- Static stack pointer
- Stack sharing of mutually non-preemptive tasks
- Stack usage of shared stack under worst-case assumptions
Building the Preemption Graph

• Preemption graph

\[ \tau_i \rightarrow \tau_l \quad \leftrightarrow \quad \tau_i \text{ can preempt } \tau_l \]

using:
- priorities
- periods/deadlines
- precedence constraints
- etc.
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \text{max Stack usage of a task}$
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \max$ Stack usage of a task

**ALGORITHM**

for all: $i = n \ldots 1$  // iterate over all tasks, from lowest priority
    $SA_i = 0$  // set initial stack address to 0
    for all: $j > i$  // iterate over all tasks with lower priority
        if $\tau_i$ can preempt $\tau_j$ then
            $SA_i = \max (SA_i, SA_j + SU_j)$
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \text{max Stack usage of a task}$

**ALGORITHM**

```plaintext
for all: $i = n \ldots 1$  // iterate over all tasks, from lowest priority
    $SA_i = 0$  // set initial stack address to 0
for all: $j > i$  // iterate over all tasks with lower priority
    if $\tau_i$ can preempt $\tau_j$ then
        $SA_i = \text{max} (SA_i, SA_j + SU_j)$
```
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \text{max Stack usage of a task}$

ALGORITHM

\begin{verbatim}
for all: $i = n \ldots 1$    // iterate over all tasks, from lowest priority
$SA_i = 0$              // set initial stack address to 0
for all: $j > i$        // iterate over all tasks with lower priority
    if $\tau_i$ can preempt $\tau_j$ then
        $SA_i = \text{max} \ (SA_i, SA_j + SU_j)$
\end{verbatim}
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on

- preemption graph
- $SU = \text{max Stack usage of a task}$

**ALGORITHM**

```plaintext
for all: $i = n \ldots 1$  // iterate over all tasks, from lowest priority
    $SA_i = 0$    // set initial stack address to 0
for all: $j > i$  // iterate over all tasks with lower priority
    if $\tau_i$ can preempt $\tau_j$ then
        $SA_i = \text{max} (SA_i, SA_j + SU_j)$
```

Select worst-case stack pointer $SA$ based on

- preemption graph
- $SU = \text{max Stack usage of a task}$
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \text{max Stack usage of a task}$

**ALGORITHM**

```plaintext
for all: i = n ... 1  // iterate over all tasks, from lowest priority
  $SA_i = 0$  // set initial stack address to 0
  for all: j > i  // iterate over all tasks with lower priority
    if $\tau_i$ can preempt $\tau_j$ then
      $SA_i = \text{max} (SA_i, SA_j + SU_j)$
```

Diagram illustrating preemption graph and stack usage.
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \text{max Stack usage of a task}$

**ALGORITHM**

```plaintext
for all: $i = n \ldots 1$  // iterate over all tasks, from lowest priority
   $SA_i = 0$  // set initial stack address to 0
for all: $j > i$  // iterate over all tasks with lower priority
   if $\tau_i$ can preempt $\tau_j$ then
      $SA_i = \text{max} (SA_i, SA_j + SU_j)$
```

![Preemption Graph and Stack Addresses](image-url)
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \max$ Stack usage of a task

**ALGORITHM**

```plaintext
define for all: $i = n \ldots 1$ // iterate over all tasks, from lowest priority
   $SA_i = 0$ // set initial stack address to 0
for all: $j > i$ // iterate over all tasks with lower priority
   if $\tau_i$ can preempt $\tau_j$ then
      $SA_i = \max (SA_i, SA_j + SU_j)$
```

**Diagram**

- Tasks $\tau_1$, $\tau_2$, $\tau_3$, $\tau_4$, $\tau_5$
- Preemption graph showing relationships between tasks
- Stack usage levels for each task:
  - $\tau_3$: high
  - $\tau_5$: medium
  - $\tau_4$: low
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \text{max Stack usage of a task}$

**ALGORITHM**

```plaintext
for all: $i = n \ldots 1$  // iterate over all tasks, from lowest priority
    $SA_i = 0$  // set initial stack address to 0
for all: $j > i$  // iterate over all tasks with lower priority
    if $\tau_i$ can preempt $\tau_j$ then
        $SA_i = \text{max} (SA_i, SA_j + SU_j)$
```
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \text{max Stack usage of a task}$

**ALGORITHM**

for all: $i = n \ldots 1$ // iterate over all tasks, from lowest priority  
$SA_i = 0$ \hspace{1em} // set initial stack address to 0  
for all: $j > i$ // iterate over all tasks with lower priority  
if $\tau_i$ can preempt $\tau_j$ then  
$SA_i = \text{max} (SA_i, SA_j + SU_j)$
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \text{max Stack usage of a task}$

ALGORITHM

\begin{align*}
\text{for all: } & i = n \ldots 1 & // \text{iterate over all tasks, from lowest priority} \\
& SA_i = 0 & // \text{set initial stack address to 0} \\
\text{for all: } & j > i & // \text{iterate over all tasks with lower priority} \\
& \text{if } \tau_i \text{ can preempt } \tau_j & \text{then} \\
& SA_i = \text{max} (SA_i, SA_j + SU_j) &
\end{align*}
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \max$ Stack usage of a task

**ALGORITHM**

\[
\text{for all: } i = n \ldots 1 \quad \text{// iterate over all tasks, from lowest priority} \\
SA_i = 0 \quad \text{// set initial stack address to 0} \\
\text{for all: } j > i \quad \text{// iterate over all tasks with lower priority} \\
\text{if } \tau_i \text{ can preempt } \tau_j \text{ then} \\
SA_i = \max (SA_i, SA_j + SU_j)
\]
Computing the Stack Addresses

Select worst-case stack pointer $SA$ based on
- preemption graph
- $SU = \text{max Stack usage of a task}$

**ALGORITHM**

```plaintext
for all: $i = n \ldots 1$  // iterate over all tasks, from lowest priority
    $SA_i = 0$  // set initial stack address to 0
for all: $j > i$  // iterate over all tasks with lower priority
    if $\tau_i$ can preempt $\tau_j$ then
        $SA_i = \text{max} (SA_i, SA_j + SU_j)$
```

Similar algorithm for computing worst-case stack usage for shared stacks [8]
Difference to other Implementations (1)

Shared Stack

\[ \tau_2 \]

\[ \tau_5 \]

Dedicated Stack

\[ \tau_2 \]

\[ \tau_5 \]

EMPRESS

\[ \tau_2 \]

\[ \tau_5 \]
Difference to other Implementations (2)

Shared Stack

Dedicated Stack

EMPRESS

possible schedule
Difference to other Implementations (3)

Shared Stack

Dedicated Stack

EMPRESS

possible schedule
Content

1) System Assumptions
2) Current Stack Implementations
3) EMPRESS
4) Evaluation/Case Study
5) Conclusions
Case Study: PapaBench

Control software of an UAV [28]
➢ C-Code available
➢ Complete task-set definition (deadlines, periods, precedence)
➢ 2 task-sets Fly-by-wire (5 tasks) / Autopilot (8 tasks)

Evaluation of
1) Reduction of Stack Usage
2) Impact on Predictability
## Case Study: PapaBench

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
<th>Period</th>
<th>Priority</th>
<th>Stack Usage (Byte)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Receive Radio-Command</td>
<td>25ms</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>T2</td>
<td>Send Data to MCU0</td>
<td>25ms</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>T3</td>
<td>Receive MCU0 values</td>
<td>50ms</td>
<td>3</td>
<td>48</td>
</tr>
<tr>
<td>T4</td>
<td>Transmit Servos</td>
<td>50ms</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>T5</td>
<td>Check Failsafe</td>
<td>50ms</td>
<td>5</td>
<td>48</td>
</tr>
<tr>
<td>T6</td>
<td>Managing Radio orders</td>
<td>25ms</td>
<td>1</td>
<td>120</td>
</tr>
<tr>
<td>T7</td>
<td>Stabilization</td>
<td>50ms</td>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td>T8</td>
<td>Send Data to MCU1</td>
<td>50ms</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>T9</td>
<td>Receive GPS Data</td>
<td>250ms</td>
<td>5</td>
<td>128</td>
</tr>
<tr>
<td>T10</td>
<td>Navigation</td>
<td>250ms</td>
<td>6</td>
<td>188</td>
</tr>
<tr>
<td>T11</td>
<td>Altitude Control</td>
<td>250ms</td>
<td>7</td>
<td>56</td>
</tr>
<tr>
<td>T12</td>
<td>Climb Control</td>
<td>250ms</td>
<td>8</td>
<td>72</td>
</tr>
<tr>
<td>T13</td>
<td>Reporting Task</td>
<td>100ms</td>
<td>4</td>
<td>44</td>
</tr>
</tbody>
</table>

* Precedence Constraints

* derived by Absint's Static Stack Analyzer [17,22]
Case Study: PapaBench
Reduction of Stack Usage

Preemption Graph

Fly-By-Wire

Autopilot
Case Study: PapaBench
Reduction of Stack Usage

Preemption Graph

EMPRESS Stack Layout
Case Study: PapaBench
Reduction of Stack Usage

Preemption Graph

Fly-By-Wire

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Autopilot

<table>
<thead>
<tr>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
<th>T10</th>
<th>T11</th>
<th>T12</th>
<th>T13</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EMPRESS Stack Layout

<table>
<thead>
<tr>
<th>Fly-By-Wire</th>
<th>Autopilot</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>96</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dedicated Stacks</th>
<th>Shared Stack</th>
<th>EMPRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fly-By-Wire</td>
<td>184</td>
<td>144 (-21%)</td>
</tr>
<tr>
<td>Autopilot</td>
<td>680</td>
<td>424 (-37%)</td>
</tr>
</tbody>
</table>
Case Study: PapaBench
Reduction of Stack Usage

General Case (Dedicated Stack vs. Shared Stack/EMPRESS)

- No reduction if fully preemptive/no precedence constraints
  ... unrealistic
- Real systems plenty of precedence constraints/resource sharing (ECRTS 2017 Industrial Challenge [25])
  - 21% and 37% probably on the low side
- Precedence constraints/FPTS/Non-preemptive regions to reduce stack size up to 75% [10]
Case Study: PapaBench
Improved Predictability

Timing Analysis + Shared Stack/Variable Stack pointer

What could we do to analyze with unknown Stack Pointer?

1) Perform $n$ analyses:
   • one for each potential stack pointer
   • not scalable; for PapaBench: 300 vs 13 analyses

2) Perform one imprecise analysis
Case Study: PapaBench
Improved Predictability

Architecture: ARMv7
- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via Absint’s Timing Profiler
1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)
Case Study: PapaBench
Improved Predictability

Architecture: ARMv7
- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via Absint’s Timing Profiler
1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)

Relative increase in WCET bound due to variable stack pointer
Case Study: PapaBench
Improved Predictability

Architecture: ARMv7
- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via Absint’s Timing Profiler
1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)

small stack usage/no reuse of stack data

Relative increase in WCET bound due to variable stack pointer
Case Study: PapaBench
Improved Predictability

Architecture: ARMv7
- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via **Absint’s Timing Profiler**
1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)

![Relative increase in WCET bound due to variable stack pointer](chart.png)
Case Study: PapaBench

Improved Predictability

Architecture: ARMv7
- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via **Absint’s Timing Profiler**
1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)

**highest increase: 30% (10 cyc), 60% (20 cyc), 2nd highest stack usage**
Case Study: PapaBench
Improved Predictability

Architecture: ARMv7
- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via Absint’s Timing Profiler
1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)
Case Study: PapaBench
Improved Predictability

Architecture: ARMv7
- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via Absint’s Timing Profiler
1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)

average increase: 6% (10 cyc), 18% (20 cyc)

Relative increase in WCET bound due to variable stack pointer
Implementation of EMPRESS

Is EMPRESS interesting for RTOS vendors/industry?

Can we implement EMPRESS within an RTOS?

What is the implementation overhead?
Implementation of EMPRESS

Is EMPRESS interesting for RTOS vendors/industry?

Can we implement EMPRESS within an RTOS?

What is the implementation overhead?
Implementation of EMPRESS

Is EMPRESS interesting for RTOS vendors/industry? Yes.

Can we implement EMPRESS within an RTOS? Sure.

What is the implementation overhead? Depends.
EMPRESS within Erika RTOS [13]

Implementation requirements:
- Absence of blocking primitives
- Mutexes handled via Immediate Priority Ceiling Protocols [29]
- Run-to-completion semantics

**ERIKA Enterprise v2**
- Additional stack saving/stack pointer modification

**Overhead:**
- 2 instructions at task activation

**ERIKA Enterprise v3**
- Already offers possibility to overlay stack regions with shared stacks

**Overhead:**
- No additional costs/zero overhead
Content

1) System Assumptions
2) Current Stack Implementations
3) EMPRESS
4) Evaluation/Case Study
5) Conclusions
Conclusions

EMPRESS: an Efficient and effective Method for PREdictable Stack Sharing

Reconciling Predictability and Performance:
• enables precise timing verification
• enables optimization of memory/cache layout
• reduces stack usage
• limits runtime overheads

Idea:
• Stack sharing of mutually non-preemptive tasks
• Static Stack Pointer = Worst-case Stack Pointer of shared stack
Bibliography


