Static Analysis of Interrupt-driven Programs
Synchronized via the Priority Ceiling Protocol

Martin D. Schwarz, Helmut Seidl and Vesal Vojdani
Peter Lammich and Markus Müller-Olm

POPL’11, 26.01.2011, Austin, Texas, USA.
Sources of interrupts
What to execute first?

Priority preemptive scheduling

Schwarz et al.

Static Analysis of Interrupt-driven Programs
What to execute first?

Priority ceiling protocol

Schwarz et al. Static Analysis of Interrupt-driven Programs
Interrupts can occur anytime
Priority ceiling protocol

Priority: 3
Priority: 2
Priority: 1

release(r)

get(r)

Resources increase priority

Schwarz et al. Static Analysis of Interrupt-driven Programs
Priority ceiling protocol

Temporary protection from interrupts
Priority ceiling protocol

- Priority of $r = \text{maximal priority of all tasks using resource } r$
  $\quad = \text{ceiling priority of } r$

- Tasks with resources run at ceiling priority
  $\Rightarrow$ Mutually exclusive ownership of resources
  $\Rightarrow$ Resources can protect accesses to data structures
Priority ceiling protocol

- Priority of $r = \text{maximal priority of all tasks using resource } r$
  \[= \text{ceiling priority of } r\]
- Tasks with resources run at ceiling priority
  \[\Rightarrow \text{Mutually exclusive ownership of resources}\]
  \[\Rightarrow \text{Resources can protect accesses to data structures}\]
Introduction

Analysis of interrupt-driven programs

Flaws in interrupt-driven programs

- Run time errors
  - Array index out of bounds
  - Floating point overflows
  - …

- Concurrency bugs
  - Data race (lost updates)
  - Transactionality violations
  - …

Goal: Prove absence of certain concurrency bugs
Flaws in interrupt-driven programs

- Run time errors
  - Array index out of bounds
  - Floating point overflows
  - ...

- Concurrency bugs
  - Data race (lost updates)
  - Transactionality violations
  - ...

Goal: Prove absence of certain concurrency bugs
Data race

The update $X = 1;$ is lost
Violation of transactionality

Y = X;
Y = X - Y;

Priority: 3

Priority: 2

Priority: 1

Each access is protected
Violation of transactionality

The operation as a whole is not transactional
Introduction

Analysis of interrupt-driven programs

Transactionality analysis

Beyond concurrency bugs

Transactionality analysis

Resource analysis

- Determine sets of held resources for every program point
- Determine ceiling priority of every program point

Transactionality analysis

- Aggregate information of resource analysis
- Determine priorities of critical sections
Resource analysis

- Functions require context
  - Depend on resource set at the call
  - Return possibly changed resource set

- Preemptions do not influence resource sets
  - Tasks start with empty resource sets
  - Tasks terminate with empty resource sets

⇒ Ordinary interprocedural analysis
Resource analysis

- Functions require context
  - Depend on resource set at the call
  - Return possibly changed resource set

- Preemptions do not influence resource sets
  - Tasks start with empty resource sets
  - Tasks terminate with empty resource sets

⇒ Ordinary interprocedural analysis
**Assumption**: Resource acquisition does only depend on calling context and current program point.

⇒ Gen/kill resource analysis for summaries
**Assumption**: Resource acquisition does only depend on calling context and current program point.

⇒ Gen/kill resource analysis for summaries
Analyzing transactionality

Four accumulated priorities

Priority: 3

Priority: 2

Priority: 1

first access

last access
Analyzing transactionality

Four accumulated priorities

- Priority: 3
- Priority: 2
- Priority: 1

first access
last access

- Minimal priority in the **critical** section: 2
Analyzing transactionality

Four accumulated priorities

Priority: 3
Priority: 2
Priority: 1

first access  last access

• Minimal priority after the first access: 1
Analyzing transactionality

Four accumulated priorities

- Priority: 3
- Priority: 2
- Priority: 1

First access: Minimal priority before the last access: 2

Last access:
Analyzing transactionality

Four accumulated priorities

- Priority: 3
- Priority: 2
- Priority: 1

First access
Last access

- Minimal priority in the whole function: 1
Analyzing transactionality

Four accumulated priorities

Priority: 3
Priority: 2
Priority: 1

first access
last access

(2, 1, 2, 1)
Composition

( critical, after first, before last, whole )

\((w, x, y, z) \circ (a, b, c, d) = (y \land b, z \land b, y \land d, z \land d)\)
Composition

( critical, after first, before last, whole )

\[(w, x, y, z) \circ (a, b, c, d) = (y \land b, z \land b, y \land d, z \land d)\]

Schwarz et al. Static Analysis of Interrupt-driven Programs
Composition

( critical, after first, before last, whole )

\[(w, x, y, z) \circ (a, b, c, d) = (y \land b, z \land b, y \land d, z \land d)\]
(critical, after first, before last, whole)

\[(w, x, y, z) \circ (a, b, c, d) = (y \land b, z \land b, y \land d, z \land d)\]
Composition

( critical, after first, before last, whole )

\[(w, x, y, z) \circ (a, b, c, d) = (y \land b, z \land b, y \land d, z \land d)\]
Effect of edges

\[
\lbrack \text{edge} \rbrack^\#(p) = \begin{cases} 
(\infty, p, p, p) & \text{accesses in edge} \\
(\infty, \infty, \infty, p) & \text{no accesses in edge}
\end{cases}
\]

\(p\) denotes the priority at the edge with resources and calling context taken into account
Transactionality analysis

Transactionality summary:

\[ [f]^{\#}(R) \mapsto \text{(critical, first, last, whole)} \]

Theorem

Transactionality summaries can be computed in time

- linear in the size of the program
- quadratic in the number of priority levels
- exponential in the number of resources
Determining transactionality

Transactionality summary:

\[
\llbracket f \rrbracket^\#(R) \leftrightarrow (\text{critical, first, last, whole})
\]

Set of variables accessed by \( f \): \( V \)

**Theorem**

*Function \( f \) is transactional when called with resource set \( R \) if*

\[
\text{critical} \geq \text{maximal priority of interrupts}
\]

\[
\text{accessing a variable in } V
\]
Resource, data race and transactionality analysis have been implemented in the Goblint analyzer.

- Analyzes concurrent C
- Context sensitive
- Based on side-effecting fixpoint solving
Resource, data race and transactionality analysis have been implemented in the Goblint analyzer.

Performance:

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sunroof</td>
<td>40966 lines</td>
<td>15 s</td>
</tr>
</tbody>
</table>
Real benchmarks with resources

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>biped_robot</td>
<td>151 lines</td>
<td>0.02 s</td>
</tr>
<tr>
<td>pe_test</td>
<td>97 lines</td>
<td>0.06 s</td>
</tr>
<tr>
<td>res_test</td>
<td>74 lines</td>
<td>0.03 s</td>
</tr>
<tr>
<td>tt_test</td>
<td>101 lines</td>
<td>0.07 s</td>
</tr>
<tr>
<td>usb_test</td>
<td>140 lines</td>
<td>0.04 s</td>
</tr>
<tr>
<td>pingpong</td>
<td>53 lines</td>
<td>0.03 s</td>
</tr>
<tr>
<td>counter</td>
<td>58 lines</td>
<td>0.02 s</td>
</tr>
</tbody>
</table>

- Quite small
- Few resources
- Few priorities
# Real benchmarks with resources

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>biped_robot</td>
<td>151 lines</td>
<td>0.02 s</td>
</tr>
<tr>
<td>pe_test</td>
<td>97 lines</td>
<td>0.06 s</td>
</tr>
<tr>
<td>res_test</td>
<td>74 lines</td>
<td>0.03 s</td>
</tr>
<tr>
<td>tt_test</td>
<td>101 lines</td>
<td>0.07 s</td>
</tr>
<tr>
<td>usb_test</td>
<td>140 lines</td>
<td>0.04 s</td>
</tr>
<tr>
<td>pingpong</td>
<td>53 lines</td>
<td>0.03 s</td>
</tr>
<tr>
<td>counter</td>
<td>58 lines</td>
<td>0.02 s</td>
</tr>
</tbody>
</table>

- Quite small
- Few resources
- Few priorities
Synthetic benchmark

- One priority and resource per interrupt

![Graph](attachment:image.png)

- Runtime in seconds
- Number of interrupts

Schwarz et al. Static Analysis of Interrupt-driven Programs
Beyond concurrency bugs

- Finding run time errors requires value analyses
- Interrupts need to be taken into account
- Use a preemption operator summarizing possible interrupts
- Apply this operator at every program point

Example: Linear equalities
Beyond concurrency bugs

- Finding run time errors requires value analyses
- Interrupts need to be taken into account
- Use a preemption operator summarizing possible interrupts
- Apply this operator at every program point

Example: Linear equalities
Linear equalities

- Find linear equalities between variables
- Represented as vectors
- Summaries: Mappings from resource sets to vector spaces of matrices
Introduction

Analysis of interrupt-driven programs

Transactionality analysis

Beyond concurrency bugs

Example

Schwarz et al.  Static Analysis of Interrupt-driven Programs
**Example**

\[
\begin{align*}
J &\quad \text{get}(s) \quad z=20 \quad \text{rel}(s) \\
I &\quad \text{get}(r) \quad y++ \quad x-- \quad \text{rel}(r) \\
T &\quad \text{get}(s) \quad x=10 \quad y=0 \quad \text{rel}(s) \\
&\quad \text{get}(r) \quad t=x+y \quad x=t-x \quad \text{rel}(r) \\
&\quad z=t*2 \quad \text{get}(r) \quad y=t-y \quad \text{rel}(r)
\end{align*}
\]

**Invariant:**

\[
\begin{align*}
t &= 10 \\
z &= 20
\end{align*}
\]
Example

Invalid equations:

\[ x + y = 10 \]
\[ x = y \]
Linear equality analysis

Extending the approach of M. Müller-Olm and H. Seidl yields:

**Theorem**

*Linear equalities can be computed in time*

- linear in the size of the program
- quadratic in the number of priority levels
- exponential in the number of resources
- polynomial in the number of variables
Thank you for your attention!