Efficient Region-Based Memory Management for Resource-limited Real-Time Embedded Systems

G. Salagnac, C. Nakhli, C. Rippert, S. Yovine

Vérimag
Université Joseph Fourier
Grenoble - France
Outline

- Introduction
- Pointer Interference Analysis
- Experimental results
- Conclusion
Motivation

The **Java** programming language

▶ Attractive language
▶ No manual dynamic memory management

Implementation pitfalls

▶ Non-determinism of Virtual Machines
▶ Garbage Collector *pause times*

⇒ difficult to use in a real-time embedded context
Our approach

Non-determinism of Garbage Collector pause times: the problem is in the JVM, not in the language!

Proposition

- Keep the **language**
  - no *manual* memory management

- Change the **implementation**
  - replace the GC by a *controllable* allocator
  - use region-based memory management
  - compute objects lifetimes at compile-time
  - find a reasonnable over-approximation
Non-determinism of Garbage Collector pause times: the problem is in the JVM, not in the language!

Proposition

- Keep the language
  - no *manual* memory management

- Change the implementation
  - replace the GC by a *controllable* allocator
    - use region-based memory management
  - compute objects lifetimes at compile-time
  - find a reasonable over-approximation
Memory management with regions

- **Benefits**
  - objects allocated side by side: no fragmentation, predictable times
  - region destroyed at once: predictable times

- **Drawbacks**
  - object placement issue: who decides?
  - region destroyed at once: space overhead
Memory management with regions

Benefits
- objects allocated side by side: no fragmentation, predictable times
- region destroyed at once: predictable times

Drawbacks
- object placement issue: who decides?
- region destroyed at once: space overhead
Memory management with regions

- Benefits
  - objects allocated side by side: no fragmentation, predictable times
  - region destroyed at once: predictable times

- Drawbacks
  - object placement issue: who decides?
  - region destroyed at once: space overhead
Outline

- Introduction
- Pointer Interference Analysis
- Experimental results
- Conclusion
Hypothesis: Objects of the same data structure will have similar lifetimes.

⇒ one region for each data structure

Static analysis:
- identify local variables that belong to the same data structure

Allocation Policy:
- place objects so that each structure is grouped in a region
Goal: find variables that belong to the same data structure
Group local variables by equivalence classes:

\[ v_1 := v_2 \lor v_1.f := v_2 \lor v_1 := v_2.f \]

\[ v_1 \sim_m v_2 \]

\[ \begin{align*}
v_1 & \mapsto p_1 \\
v_2 & \mapsto p_2 \\
\end{align*} \]

\[ p_1 \sim_{m'} p_2 \]

\[ v_1 \sim_{m'} v_2 \]

... and transitive-symmetric closure
Goal: find variables that belong to the same data structure
Group local variables by equivalence classes:

\[ v_1 := v_2 \lor v_1.f := v_2 \lor v_1 := v_2.f \]

\[ v_1 \sim m \, v_2 \]

\[ m \quad v_1 \longmapsto p_1 \quad v_2 \longmapsto p_2 \]

\[ p_1 \sim_{m'} p_2 \]

\[ v_1 \sim m \, v_2 \]

... and transitive-symmetric closure
Goal: find variables that belong to the same data structure

Group local variables by equivalence classes:

\[
\begin{align*}
   v_1 & := v_2 \lor v_1.f := v_2 \lor v_1 & := v_2.f \\
   v_1 \sim_m v_2
\end{align*}
\]

\[m
\]
\[
\begin{align*}
   v_1 & \mapsto p_1 \\
   v_2 & \mapsto p_2
\end{align*}
\]

\[
p_1 \sim_{m'} p_2
\]

\[
v_1 \sim_{m'} v_2
\]

... and transitive-symmetric closure
Goal: find variables that belong to the same data structure
Group local variables by equivalence classes:

\[
\begin{align*}
  v_1 & := v_2 \lor v_1.f := v_2 \lor v_1 := v_2.f \\
  v_1 \sim_m v_2
\end{align*}
\]

\[
\begin{array}{c}
  v_1 \mapsto p_1 \\
  v_2 \mapsto p_2 \\
  p_1 \sim_{m'} p_2
\end{array}
\]

... and transitive-symmetric closure
Allocation policy

Goal: Place each data structure in a region.

Question: For an allocation site $x = \text{new } C$, in which region must we put the allocated object?

Allocation policy:
- look for another local variable $y$ such that $y \sim x$ implies placing the object $x$ in the region of the object $y$
- if none, place the object in a new region, attached to $x$
Allocation policy

**Goal:** Place each data structure in a region.

**Question:** For an allocation site $x = \text{new } C$, in which region must we put the allocated object?

**Allocation policy:**

- look for another local variable $y$ such that $y \sim x$  
  $\implies$ place the object $x$ in the region of the object $y$

- if none, place the object in a *new region*, attached to $x$
Example

```java
void m1()
{
    a = new Container();
    m2(a);
    ...
}

void m2(Container x) {
    y = new Data;
    x.f = y ;
}
```

- $a$ is alone: it is allocated in its region.
- $x \sim_{m2} y \implies$ the object $y$ can be allocated in the region of the object $x$. 
Outline

- Introduction
- Pointer Interference Analysis
- Experimental results
- Conclusion
Experimental results (1)

G. Salagnac, C. Nakhli, C. Rippert, S. Yovine (Vérimag)
Experimental results (1)

<table>
<thead>
<tr>
<th>MM</th>
<th>Max Occupancy</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regions</td>
<td>540k</td>
<td>100%</td>
</tr>
<tr>
<td>GC</td>
<td>1M</td>
<td>100%</td>
</tr>
<tr>
<td>GC</td>
<td>800k</td>
<td>100%</td>
</tr>
<tr>
<td>GC</td>
<td>600k</td>
<td>120%</td>
</tr>
<tr>
<td>GC</td>
<td>550k</td>
<td>300%</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Garbage collector (800k threshold)
- Regions
Experimental results (2)

G. Salagnac, C. Nakhli, C. Rippert, S. Yovine (Vérimag)
Outline

- Introduction
- Pointer Interference Analysis
- Experimental results
- Conclusion
Conclusion and perspectives

Results:
- a simple pointer analysis algorithm
- a prototype memory manager
  - promising results

Work in progress:
- validation on industrial case-studies
- How to predict the runtime behaviour?