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

$\Longrightarrow$ 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

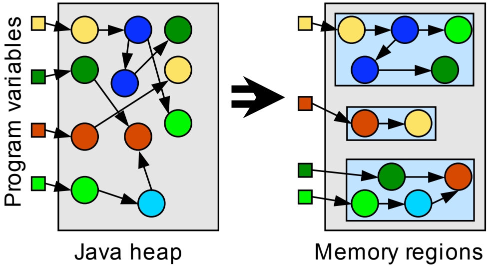
Our approach

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

Proposition

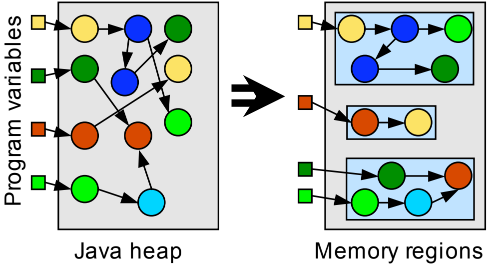
- Keep the language
  - no *manual* memory management
- Change the implementation
  - replace the GC by a *controllable* allocator
    - use region-based memory management
  - compute objects lifetimes at compile-time
    - find a reasonnable over-approximation

Memory management with regions



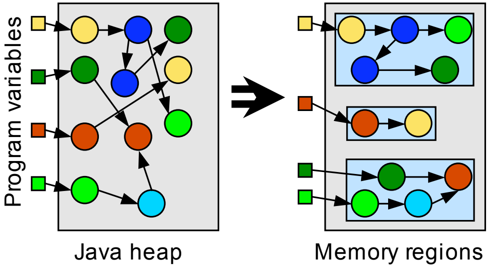
- Benefits
  - objects allocated side by side: no fragmentation, predictable times
  - region destroyed at once: predictable times
- Drawbacks
  - object placement issue: who decides ?
  - region destroyed at once: space overhead

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

Analysis design

Hypothesis: Objects of the same data structure will have similar lifetimes.

$\Longrightarrow$ 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

Pointer Interference Analysis

Goal: find variables that belong to the same data structure
Group local variables by equivalence classes:

$$\frac{v_1 := v_2 \quad \vee \quad v_1.f := v_2 \quad \vee \quad v_1 := v_2.f}{v_1 \sim_m v_2}$$

$$\frac{m \xrightarrow{\;v_1 \longmapsto p_1,\; v_2 \longmapsto p_2\;} m' \qquad p_1 \sim_{m'} p_2}{v_1 \sim_m v_2}$$

... and transitive-symetric closure

Pointer Interference Analysis

Goal: find variables that belong to the same data structure
Group local variables by equivalence classes:

$$\frac{v_1 := v_2 \quad \vee \quad v_1.f := v_2 \quad \vee \quad v_1 := v_2.f}{v_1 \sim_m v_2}$$

$$\frac{m \xrightarrow[v_2 \longmapsto p_2]{v_1 \longmapsto p_1} m' \qquad p_1 \sim_{m'} p_2}{v_1 \sim_m v_2}$$

... and transitive-symetric closure

Pointer Interference Analysis

Goal: find variables that belong to the same data structure
Group local variables by equivalence classes:

$$\frac{v_1 := v_2 \quad \vee \quad v_1.f := v_2 \quad \vee \quad v_1 := v_2.f}{v_1 \sim_m v_2}$$

$$\frac{\begin{array}{l} m \\ \big\downarrow \begin{array}{l} v_1 \longmapsto p_1 \\ v_2 \longmapsto p_2 \end{array} \qquad\qquad p_1 \sim_{m'} p_2 \\ m' \end{array}}{v_1 \sim_m v_2}$$

... and transitive-symetric closure

Pointer Interference Analysis

Goal: find variables that belong to the same data structure
Group local variables by equivalence classes:

$$\frac{v_1 := v_2 \quad \vee \quad v_1.f := v_2 \quad \vee \quad v_1 := v_2.f}{v_1 \sim_m v_2}$$

$$\frac{\begin{array}{l} m \\ \Big\downarrow \begin{array}{l} v_1 \longmapsto p_1 \\ v_2 \longmapsto p_2 \end{array} \\ m' \end{array} \qquad p_1 \sim_{m'} p_2}{v_1 \sim_m v_2}$$

... and transitive-symetric closure

Allocation policy

Goal: Place each data structure in a region.

Question: For an allocation site `x = new C`, in which region must we put the allocated object ?

Allocation policy:

- look for another local variable `y` such that `y`$\sim$`x`
  $\Longrightarrow$ place 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 = new C`, in which region must we put the allocated object ?

Allocation policy:

- look for another local variable `y` such that `y`$\sim$`x`
  $\Longrightarrow$ place the object `x` in the region of the object `y`
- if none, place the object in a *new region*, attached to `x`

Example

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

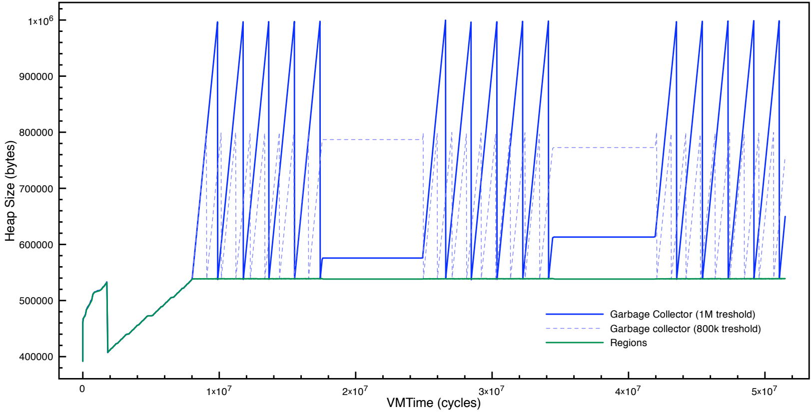
```
void m2(Container x) {
  y = new Data;
  x.f = y ;
}
```

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

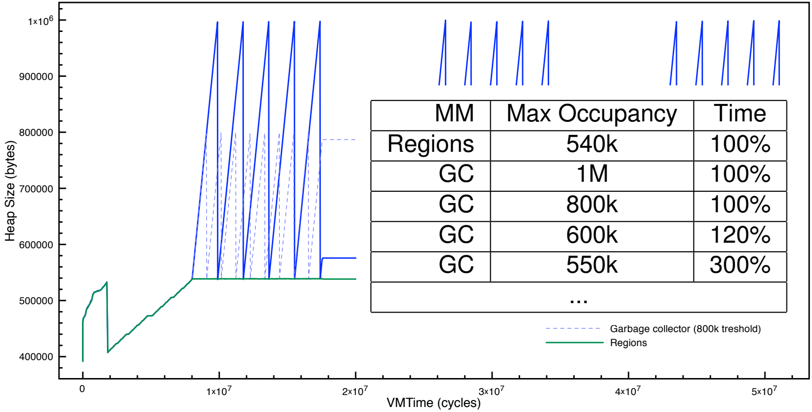
Outline

- Introduction
- Pointer Interference Analysis
- Experimental results
- Conclusion

Experimental results (1)

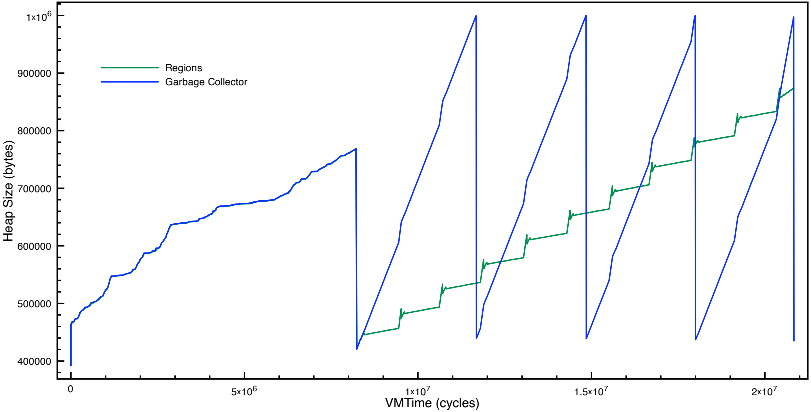


Experimental results (1)



MM	Max Occupancy	Time
Regions	540k	100%
GC	1M	100%
GC	800k	100%
GC	600k	120%
GC	550k	300%
...		

Experimental results (2)



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 ?