

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

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 reasonable over-approximation

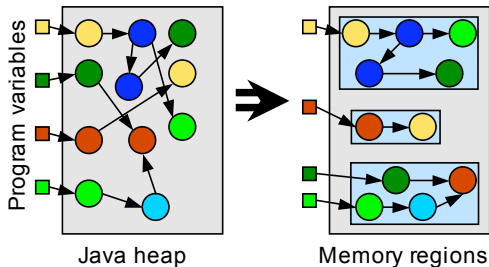
Our approach

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 reasonable over-approximation

Memory management with regions



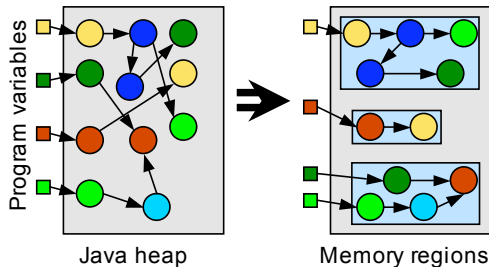
► Benefits

- 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



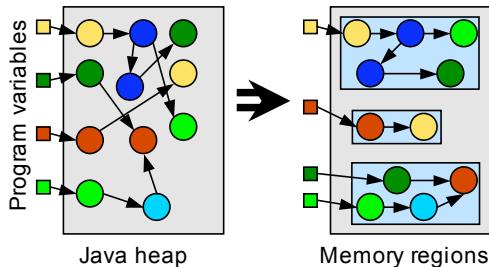
► Benefits

- 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

Pointer Interference Analysis

Goal: find variables that belong to the same **data structure**

Group local variables by equivalence classes:

$$\frac{V_1 := V_2 \quad \vee \quad V_1.f := V_2 \quad \vee \quad V_1 := V_2.f}{V_1 \sim_m V_2}$$

$$\frac{\begin{array}{c} m \\ \downarrow \\ V_1 \mapsto p_1 \\ V_2 \mapsto p_2 \\ m' \end{array} \quad p_1 \sim_{m'} p_2}{V_1 \sim_m V_2}$$

... and transitive-symmetric closure

Pointer Interference Analysis

Goal: find variables that belong to the same **data structure**

Group local variables by equivalence classes:

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

$$\frac{\begin{array}{c} m \\ \downarrow \\ v_1 \mapsto p_1 \\ v_2 \mapsto p_2 \\ m' \end{array} \quad p_1 \sim_{m'} p_2}{v_1 \sim_m v_2}$$

... and transitive-symmetric closure

Pointer Interference Analysis

Goal: find variables that belong to the same **data structure**

Group local variables by equivalence classes:

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

$$\frac{\begin{array}{ccc} m & & \\ \downarrow & v_1 \mapsto p_1 & \\ & v_2 \mapsto p_2 & p_1 \sim_{m'} p_2 \\ m' & & \end{array}}{v_1 \sim_m v_2}$$

... and transitive-symmetric closure

Pointer Interference Analysis

Goal: find variables that belong to the same **data structure**

Group local variables by equivalence classes:

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

$$\frac{\begin{array}{c} m \\ \downarrow \\ v_1 \mapsto p_1 \\ v_2 \mapsto p_2 \\ m' \end{array} \quad p_1 \sim_{m'} p_2}{v_1 \sim_m v_2}$$

... 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 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 $x = \text{new } C$, 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

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

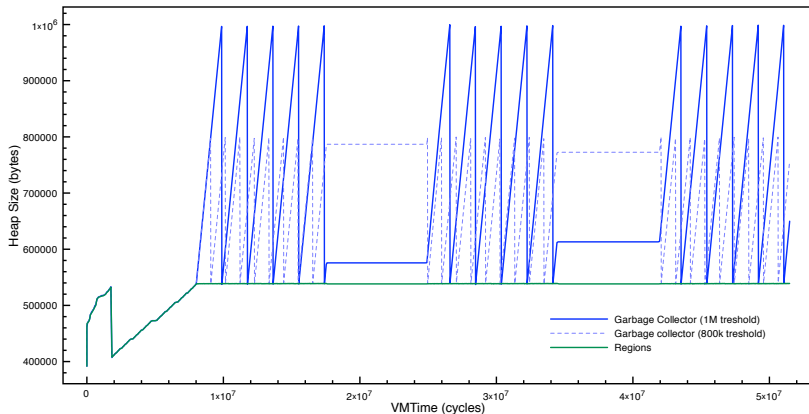
```
void m2(Container x) {  
  y = new Data;  
  x.f = y ;  
}
```

- ▶ **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 .

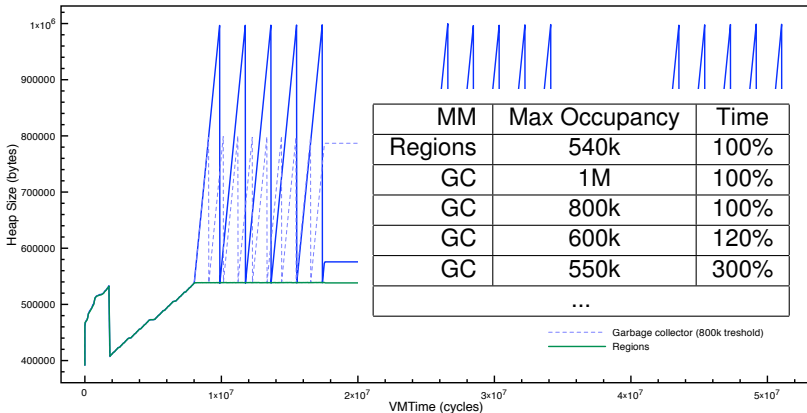
Outline

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

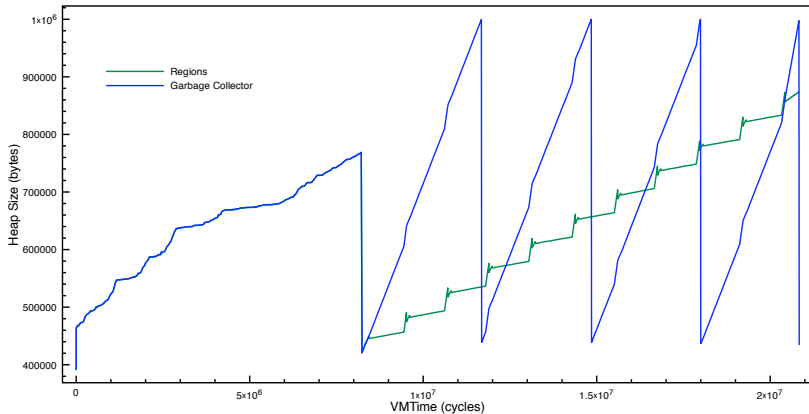
Experimental results (1)



Experimental results (1)



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 ?