RT-LAGC: Fragmentation-Tolerant Real-Time Memory Management Revisited

Isabella Stilkerich, Michael Strotz, Christoph Erhardt, Michael Stilkerich
Real-Time Embedded Java

- Productivity
- Safety
  - Software-based memory protection
- Efficiency
  - Ahead-of-time compilation
  - Static configuration
Real-Time Embedded Java

- **Productivity**
- **Safety**
  - Software-based memory protection
- **Efficiency**
  - Ahead-of-time compilation
  - Static configuration
- **… Memory management?**
  - In particular: predictability?
Real-Time Memory Management

• **Pseudo-static allocation** (*ImmortalMemory*)
  - Bump pointer
  - No memory release
Real-Time Memory Management

- **Pseudo-static allocation** (*ImmortalMemory*)
  - Bump pointer
  - No memory release

- **Region-based allocation** (*ScopedMemory*)
  - Bump pointer
  - Region released as a whole
Real-Time Memory Management

- **Pseudo-static allocation** *(ImmortalMemory)*
  - Bump pointer
  - No memory release
- **Region-based allocation** *(ScopedMemory)*
  - Bump pointer
  - Region released as a whole

Simple, predictable, special-purpose
Real-Time Memory Management

- **Pseudo-static allocation** (*ImmortalMemory*)
  - Bump pointer
  - No memory release
- **Region-based allocation** (*ScopedMemory*)
  - Bump pointer
  - Region released as a whole
- **Automatic memory management**
  - Complex, list-like heap structure
  - Garbage collection
    - Heap fragmentation, unpredictable allocation times
    - Unpredictable latencies

Simple, predictable, special-purpose
Agenda

- **Goal:** real-time-capable automatic memory management
  - Bounded allocation times
  - Bounded latencies
  - Good throughput

- **Implementation in the KESO JVM**
  - *Real-Time Latency-Aware Garbage Collection*

- **Evaluation**
The KESO JVM

- Ahead-of-time compilation to C code
- Static configuration
- “Pay only for what you use”
Garbage Collectors in KESO

1. No GC

2. Slack-based mark-and-sweep GC
   a) Stop-the-world
      • Not preemptible
   b) Incremental
      • Preemptible by application tasks
      • Synchronisation, write barriers
• Configurable per domain
Garbage Collectors in KESO

1. No GC

2. Slack-based mark-and-sweep GC
   a) Stop-the-world
      • Not preemptible
   b) Incremental
      • Preemptible by application tasks
      • Synchronisation, write barriers

• Configurable per domain

Extend for better predictability
Fragmented Objects

- Traditional heap
  - Variable-size chunks → heap fragmentation
  - Unpredictable allocation time
Fragmented Objects

- **Traditional heap**
  - Variable-size chunks → heap fragmentation
  - Unpredictable allocation time

- **Fragmented allocation**
  - Fixed-size slots → no heap fragmentation possible
  - Fragmented objects
  - Predictable allocation time

![Diagram of Fragmented Objects]
Fragmented Arrays

- **Spines**
  - Constant access complexity
  - Allocated in separate heap space
    - Bump pointer
    - Replicating or generational GC
  - **Optimisation**: omit spine if all fragments are contiguous
Object Layout

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| a) | final class A { | }
|   |  | }
|   |  | next
|   |  | A.r1
|   |  | A.r0
|   |  | Header
|   |  | A.i0
| b) | class A { | }
|   |  | next
|   |  | A.r1
|   |  | A.r0
|   |  | Header
|   |  | A.i0
| c) | class B extends A { | }
|   |  | next
|   |  | B.r2
|   |  | A.r1
|   |  | B.i1
|   |  | B.i2

- Reference grouping for efficient scanning by GC
- Fragmented Bidirectional Object Layout
Scanning the Object Graph

- **Static references**
  - Grouped together in `.data` section

- **Non-static reference fields**
  - Grouped together in object fragments

- **Local references**
  - Henderson’s linked stack frames
    - Only generated for blocking methods
  - Only scan stacks of currently blocked tasks
Write Barriers

• Whenever a reference is written, mark its pointee grey
  • Significant application overhead
Write Barriers

• Whenever a reference is written, mark its pointee grey
  • Significant application overhead

• Trade-off: latency vs. performance
  • Omit write barriers for local references
  • During stack scanning, raise GC’s priority above stack owner’s
    • Short, bounded time (sans recursion)
    • GC still preemptible by higher-priority tasks
  • Also possible: omit write barriers for static fields
Improving GC Predictability

- **Observation**: GC runtime dominated by mark phase
  - Object-graph traversal expensive
  - Only surviving heap objects contribute

- **Survivability analysis**
  - Per-object lifespan categorisation
  - Based on escape analysis and call-graph analysis
  - Object may survive GC if reference …
    - … escapes globally
    - … is live beyond blocking system calls
Survivability Classification

- Method-local
  - Automatic stack allocation
Survivability Classification

- **Method-local**
  - Automatic stack allocation

- **Region-local**
  - Automatic region-based allocation
Survivability Classification

- **Method-local**
  - Automatic stack allocation

- **Region-local**
  - Automatic region-based allocation

- **Task-local**
  - Per-task heaps
Survivability Classification

- **Method-local**
  - Automatic stack allocation

- **Region-local**
  - Automatic region-based allocation

- **Task-local**
  - Per-task heaps

- **Task-escaping**
  - Regular heap
Evaluation

- **Collision Detector (CD$_j$) 1.2**
  - Real-time air-traffic simulator and collision detector
  - CiAO OS
  - TriCore TC1796 @ 150 MHz
Evaluation

• Collision Detector (CD$_j$) 1.2
  • Real-time air-traffic simulator and collision detector
  • CiAO OS
  • TriCore TC1796 @ 150 MHz

• Object sizes
  • Regular objects are small due to by-reference semantics
  • Chain length determines execution time of
    • Allocation
    • Field access

![Bar chart showing object size and execution time]
Memory Footprint / Heap Usage

• Memory footprint

<table>
<thead>
<tr>
<th></th>
<th>.text</th>
<th>.data + .bss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental</td>
<td>50 KiB</td>
<td>660 KiB</td>
</tr>
<tr>
<td>RT-LAGC</td>
<td>58-60 KiB</td>
<td>820 KiB</td>
</tr>
<tr>
<td>Δ</td>
<td>+ 16-20 %</td>
<td>+ 23 %</td>
</tr>
</tbody>
</table>

• Heap usage

• Generational vs. replicating spines: spine memory reduced by 50 %
• Stack allocation reduces heap usage by 43 %
• 32-byte vs. 16-byte fragments: usage raised by 45 % (internal fragmentation)
Runtime

- RT-LAGC16 (Rep/F16/FFA)
- RT-LAGC32 (Gen/F32)
- RT-LAGC16 (Gen/F16)
- RT-LAGC32 (Gen/F32/EEA)
Runtime

Baseline

ms

RT-LAGC16 (Rep/F16/FFA)

RT-LAGC32 (Gen/F32)

RT-LAGC16 (Gen/F16)

RT-LAGC32 (Gen/F32/EEA)
Runtime

Deactivate stack alloc. (+25 %)

Baseline

RT-LAGC16 (Rep/F16/FFA)
RT-LAGC32 (Gen/F32)
RT-LAGC16 (Gen/F16)
RT-LAGC32 (Gen/F32/EEA)
Runtime

Deactivate stack alloc. (+25 %)
Fragment size 16 (+2%)
Baseline

RT-LAGC16 (Rep/F16/FFA)
RT-LAGC32 (Gen/F32)
RT-LAGC16 (Gen/F16)
RT-LAGC32 (Gen/F32/EEA)
Runtime

- Deactivate stack alloc. (+25 %)
- Force fragmented arrays (+10 %)
- Fragment size 16 (+2%)
- Replicating spines (+3 %)
- Baseline

- RT-LAGC16 (Rep/F16/FFA)
- RT-LAGC32 (Gen/F32)
- RT-LAGC16 (Gen/F16)
- RT-LAGC32 (Gen/F32/EEA)
Runtime

![Runtime Graph]

- RT-LAGC
- IRRGC
- CBGC
- RDS
Runtime

Baseline

ms

RT-LAGC, IRRGC, CBGC, RDS
Runtime

No fragmented objects (-16.7%)

Baseline

- RT-LAGC
- IRRGC
- CBGC
- RDS

ms
Runtime

- No fragmented objects (-16.7%)
- Remove synchronisation (-6.8%)

Baseline
Runtime

- Baseline
- No fragmented objects (-16.7%)
- Remove synchronisation (-6.8%)
- No GC (-15.4%)
Runtime

RT-LAGC still provides good throughput: CD_j configuration is 17% slower than CD_c.

No fragmented objects

No GC (-15.4%)
Conclusion

- **RT-LAGC**
  - Cooperative approach
  - Predictable allocation times
  - Low reaction time to external events
  - Good throughput
  - Fine-grained synchronisation code
Conclusion

- **RT-LAGC**
  - Cooperative approach
  - Predictable allocation times
  - Low reaction time to external events
  - Good throughput
  - Fine-grained synchronisation code

- **Developer assistance during integration**
  - Survivability analysis
  - Object sizes