Java Memory Management for Real-time Systems

- Java is increasingly used for embedded systems
  - Cell phones, automotive command-and-control, avionics
- Garbage collection is a key feature of Java
  - + Simplifies development
  - + Avoids error-prone explicit memory management (malloc/free)
    - - Traditionally not suitable for real-time requirements
- RTSJ proposes an alternative memory management solution
  - + Allows threads that are completely independent of GC
  - - Complex model
Outline

- Problems with RTSJ
  - Runtime Barriers, Difficult Usage, Fragmentation
- Overview of Metronome
- Integration of Metronome into a Real-time System
- Reducing Context Switch Times
- Metronome vs. RTSJ
  - A Hybrid Proposal
RTSJ: Scopes and Real-time Threads

$\text{Memory Areas}$
- Heap - the regular Java heap
- Immortal - all objects in this area are permanently kept live
- Scoped memory regions - models LIFO lifetimes

$\text{Threads}$
- Normal Java threads - lowest priority
- RealTimeThread - higher priority than regular threads and GC
- NoHeapRealTimeThread - highest priority
RTSJ: Scoped Memory

Accessible by all threads
Forbidden Edges
Inaccessible by NoHeapRealTimeThread
RTSJ Runtime Barriers

$ Read Barrier
  - Prevent NoHeapRealTimeThread from seeing heap object
  - Hard to coalesce/optimize because of dependence on
    - Particular field
    - Time of access
    - Thread type

$ Write Barrier
  - Expensive
    - Determine scopes of 2 objects
    - Determine ancestor relationship of 2 scopes
  - Hard to coalesce/optimize
    - Type system does not guarantee where objects are
    - Dependent on contents of written value (cf generational barrier)
Difficult Usage

- Only NoHeapRealTimeThread has true pre-emption
- NoHeapRealTimeThread can only manipulate immortal memory and scoped memory
- RTSJ has wait-free queues to avoid priority inversion
- How does NoHeapRealTimeThread deal with non-LIFO patterns?
- Scenario: Consider a real-time server where a NoHeapRealTimeThread accepts incoming requests in the form of a String. Where is the String object allocated?
  - Immortal - leak
  - Heap - forbidden
  - Scoped - leak since pattern is FIFO, not LIFO
- What about library code???
How many scopes should one use?

- **Few scopes**
  - + Floating garbage
  - - Less fragmentation
  - - Less work

- **More scopes**
  - + More quickly recycles memory
  - - Fragmentation from over-provisioning
  - - More meta-data and entry/exit overhead

- **Issue**
  - Choice may be restricted by application algorithm
Metronome: Core Features

$ Mostly Non-copying
  • Avoids minimum 2X space overhead of copying collector

$ Segregated Free Lists
  • Segregated by size

$ By need defragmentation
  • Trigger when there is a possibility of fragmentation before GC cycle complete

$ Read Barrier
  • Permits incremental defragmentation

$ Incremental Mark-Sweep
  • Snapshot-at-the-beginning

$ Arraylets
  • Bound fragmentation and amount of work to defragment
  • Permits incremental defragmentation
Metronome: Pause Times Histogram (SPECjvm javac)
**Metronome: Utilization Level (SPECjvm javac)**

- **Y-axis:** Utilization
- **X-axis:** Execution Time (s)
- The graph shows periodic spikes in utilization with a consistent pattern.
Read Barrier (Java-modified Brooks-style)

Case 1
Case 2

Normal: *(x+8)
Read Barrier: *(*(x-4)+8)

Optimizations
- Coalescing
- Code motion
- Combine with Java’s null-check
- Overhead reduced from 15-25% to an average of 4-6%
**Time-Based Scheduling**

- **Work-Based Scheduling**
  - + Simple guarantee of completion on time
  - - Uneven mutator utilization due to bursty allocation

- **Time-Based Scheduling**
  - + Very predictable mutator utilization
  - - Potentially hard to determine termination point (space usage)

- **Per-quantum vs Per-GC parameters**
  - Per-quantum are not stable
  - Per-GC characteristics are stable
  - Time-Based works well in practice because it depends on per-GC
Provable Real-time Bounds

Tuner

\[ u_T = \text{utilization} \]

30% \hspace{1cm} 50%

\[ s = \text{used space} \]

45 MB \hspace{1cm} 60 MB

\[ \Delta t = \text{time resolution} \]

5 ms

Program (Mutator)

\[ a^*(\Delta GC) = \text{Per-GC Allocate Rate} \]

10 MB/s

\[ M = \text{Live Data} \]

30 MB

Collector

\[ R = \text{Tracing Rate} \]

50 MB/s
Integrating Metronome with a Real-Time System

- **Benefits:**
  - A single memory address space
  - Elimination of run-time memory access exceptions
  - Simple sharing of objects between threads of different priorities
  - Real-time Threads can call any standard library routines

- **Integration is easy**
  - Add GC as a new periodic task
  - On each thread creation, increase the GC’s requirement by $1 - u_T$ for feasibility analysis
Metronome’s Current Switch Times and Pause Times

- Maximum pause time: 4 ms
- Maximum context switch time: 1 ms

Factors:
- Max Pause > Max context switch
- Underlying OS/scheduler’s real-time characteristics
- Algorithmic decisions

Current: Pause time down to under 1 ms with tuning
Goal: Reduce context switch time to < 100 µs
Reducing Context Switch Times

- **Read Barrier (Stack Fixup)**
  - Eager - lower overhead, higher context switch time
  - Lazy - higher overhead, lower context switch time

- **Abortable Copy Operation**
  - Abort or defer out of copying larger objects (stacklet/arraylet)
  - Careful scheduling within a quantum

- **Deferred Root Scanning**
  - Use a modified write barrier
  - Stack frame work is bounded and deferrable
  - No termination problem
Metronome vs. RTSJ

- **Metronome**
  - + Simpler programming model
  - - Greater context switch time

- **RTSJ**
  - - Complex memory management
  - + Possibly better memory usage with ScopedMemory

- **Note:**
  - Lack of intra-scope GC can make ScopedMemory worse than GC
  - Over-provisioning may be unavoidable with ScopedMemory
A hybrid approach: Metronome + ScopedMemory

- Intent: Capture benefits of both Metronome and ScopedMemory
- Change: Eliminate NoHeapRealTimeThread
- Benefits
  - Associated read barrier is eliminated
  - MemoryAccessError exception is eliminated
  - Less brittle, can use library code, graceful use of needed features

- Development Cycle:
  1. Program without ScopedMemory at all
  2. Measure all parameters and check if requirements are met
     - Use ScopedMemory when the fit is intuitive and tight
       - the lifetime is LIFO and there is little floating garbage
     - Reduce allocation rates particularly of high-priority threads.
     - Increase use of scoped memory.
Conclusions

- RTSJ memory management is complex and may not fit certain programs
- Incremental GC is feasible and can be integrated into a real-time system
- A hybrid model can capture key benefits
  - Ease of programming use
  - Simpler (though not simplest) model
  - Graceful introduction of advanced feature as real-time requirements are tightened