Real-time Java and Real-time Garbage Collection

Tomas Kalibera
DISTRIBUTED SYSTEMS RESEARCH GROUP
http://dsrg.mff.cuni.cz

CHARLES UNIVERSITY PRAGUE
Faculty of Mathematics and Physics
Real-time Java: It could work

US Navy DDG-1000 Zumwalt class destroyer by Raytheon, programmed in Real-time Java, running on IBM’s WebSphere virtual machine

Boeing ScanEagle UAV, navigation system running on Purdue’s open-source real-time Java virtual machine
Real-time Java: Why?

- Automated memory management
  - Dynamic allocation
  - Automatic de-allocation
  - No memory access errors

- Managed platform
  - Integrated concurrency, dynamic loading
  - Many libraries with standard APIs

- Popularity
  - Good supply of programmers, tools
Real-time Java Challenges

- Automatic memory management
- Interfacing to (real) hardware
- Footprint
- Predictability, WCET
- Increased complexity of the system
  - Tool support for certification essential
Real-time Garbage Collection (RTGC)
Garbage Collector Essentials

local variables

heap

global variables

thread1

thread2
GC: Some Pointers are Overwritten

local variables

heap

global variables

thread1

thread2

Some pointers are overwritten by threads 1 and 2, crossing local and global variables and heap regions.
GC: Some Objects Become Unreachable

local variables

heap

global variables

thread1

thread2
GC: Objects Marked White

local variables

heap

global variables

thread1

thread2
GC: Directly Reachable not Processed Objects Marked Grey
GC: Processed (Reachable) Objects Marked Black

local variables

heap

global variables

thread1

thread2
GC: Processed (Reachable) Objects Marked Black
GC: Processed (Reachable) Objects Marked Black

local
variables

heap

thread1

global
variables

thread2
GC: Finally, All White Objects are Unreachable - Garbage
GC: Memory of Unreachable Objects is Reclaimed

local variables

thread1

thread2

heap

global variables
GC: Memory is Defragmented

local
variables

heap

global
variables

thread1

thread2
for(;;) {
  1. sleep until memory is exhausted
  2. do a new GC cycle
     identify unreachable objects
     reclaim memory
     defragment memory
}
Turning GC into RTGC

- **Incrementality**
  - Collection runs in short atomic increments to reduce pause times for the application
  - This includes object relocation in defragmentation

- **Scheduling**
  - It is too late to start a GC when memory is full; the system would crash in presence of incrementality
  - Giving too much CPU to GC leads to poor performance or even missed deadlines
Minuteman Real-time GC Framework

with

Jan Vitek, Filip Pizlo, Tony Hosking
Scope: Purdue Ovm Virtual Machine

• Focus on real-time embedded systems
• Ahead of time compilation via C, uniprocessor with user-managed threads, implemented in Java
• Real-time support
  ▪ RTSJ
  ▪ SCJ (work in progress)
  ▪ RTEMS/LEON, Xenomai
The GC must not reclaim a live object

Weak tri-color invariant
- “white object pointed to by a black object is also reachable from some grey object through a chain of white objects”

Incremental scanning of heap
- Mark grey old value of \( f.x \) (if it is white) \( f.x = g.y \)

Incremental scanning of threads
- Atomic for every single thread
- Mark grey \( g.y \) (if it is white)
Object relocation must be transparent to application

Two alternative implementations
- Brooks forwarding pointers with atomic copy
- Replication with incremental/atomic copy

Incremental Dynamic Defragmentation

new copy → old copy

\[ f.x = g.y \quad \Rightarrow \quad f.fwd.x = g.fwd.y \]
Incremental Dynamic Defragmentation

- Two alternative implementations
  - Brooks forwarding pointers with atomic copy
  - Replication with incremental/atomic copy

\[ \text{JTRES’09 paper: replication is faster than Brooks forwarding pointers} \]
Reducing the Need for Defragmentation: Arraylets

- Split representation of arrays
- Integrated with forwarding pointers

![Diagram showing split representation of arrays and integration with forwarding pointers.]

- array length
- header
- arraylet pointer
- arraylet (includes array data)
GC Scheduling: When to schedule the GC Thread?

- **Slack-based**
  - Run GC only when application is idle
- **Periodic**
  - Periodically suspend the application to run GC for a fixed-time quanta (i.e. 500 us)
- **Hybrid**
  - Use both idle time and periodic suspension

*RTSS’09 paper: certain systems are only schedulable with slack-based, while other only with periodic GC scheduler*...
• Goals – prove that
  ▪ System never runs out of memory
  ▪ System never misses a deadline

  *RTSS’09 paper: schedulability tests for periodically scheduled RTGC*

• Issues – builds on inputs that are not available
  ▪ Worst-case analysis of memory operations
  ▪ Worst-case execution time analysis
    • Which does not scale even to trivial C programs
    • C code generated by Java compiler is highly complex
Work in Progress

Pavel Parizek with
Tomas Kalibera and Jan Vitek

Finding Errors in Real-time Java Programs using Java PathFinder
Finding Errors with JPF

- Easier than for standard Java
  - Applications, platforms simpler by design
    - Fewer thread interleaving due to no time-slicing
    - Uni-processor schedulers further reduce interleavings
- Harder than for standard Java
  - At least some notion of execution time on the real platform is needed to explore all feasible paths
    - Strict priority enforcement makes it impossible to model timed-sleeps by immediate re-scheduling (highest priority thread would never leave the CPU)
Timed Sleeps Modeled around Thread Periods

• Assumptions
  ▪ Only periodic threads (common)
  ▪ No deadline misses (checked by other tools)

• Solution
  ▪ The highest priority thread is not scheduled if a lower priority thread would miss a deadline

A thread with a 10ms period should run at least once between two executions of a thread with 15ms period...