Real-Time Programming in Java

Real-Time in the Age of Complex Systems

Invited Speaker: David F. Bacon
IBM Research
Classical Real-Time Control

SENSE

↓

COMPUTE

↓

ACTUATE

Anti-lock Braking System
Complex Real-Time Systems
What’s Changing?

• Processing Power (instructions per 10ms)

• Ubiquitous Sensing and Actuation
  – e.g. video stream per cell phone

• More and more computing is real-time
Implications for Real-Time Systems

• Much Larger (in code and scope)

• Highly Dynamic

• Non-deterministic
  – non-determinism of underlying system desirable

• Different Kind of Software Engineering
Memory Management

**STATIC**
(arrays)
- Fortran
- Esterel
- Verilog

**DYNAMIC**
(malloc/free)
- C
- C++
- Ada
- Pascal

(new/garbage collect)
- Java
- Lisp
- C#
- Smalltalk

---

Determinism  Verifiability

Flexibility  Software Engineering
COMMITTED TO PRODUCTION

Telco SIP Switch

DDG-1000 Destroyer

Trade Execution

TARGETED

Playstation/Xbox etc

Automotive Electronics

RESEARCH

Java-based Synthesizer

JAviator (w/ Salzburg)

Air Java (w/ Berkeley CE)
Three Approaches in Java

RTSJ Standard:
Scoped/Immortal Memory

Metronome:
Real-Time Garbage Collection

Flexotasks:
Time-Portable Java
Why is Real-Time Difficult in Java?

- No real-time scheduling APIs
- Garbage Collection
- Dynamic Class Loading
- Just-in-Time (JIT) Compilation
- Dynamic, average-case-based optimization
Real-Time Specification for Java

Two primary additions to the Java language:

– Real-Time Scheduling APIs

– Semi-manual memory management

(real-time garbage collection “known” impossible)
Basic Java Memory Architecture

class Foo {
    Foo a;
    Foo b;
}

RTSJ Memory Architecture

Heap

Control Loop Iteration

Stack

Immortal
f(a,b) \{ a.x = b; \}
RTSJ Memory Management Issues

• Complex Programming Model

• Non-compositional

• Run-time Failures

• Checking Overhead

• Storage Leaks in Immortal Memory

• Often Poorly Suited (e.g. Producer/Consumer)
Metronome:
Real-Time Garbage Collection
GC: A Simple Problem (?)

- Transitive Graph Closure
- Application is Stopped During Collection
- Memory is only freed at the end
Basic Approaches: Mark/Sweep

- $O(live)$ mark phase but $O(heapsize)$ sweep
- Usually requires no copying
- Mark stack is $O(maxdepth)$
Basics II: Semi-space Copying

- \( O(\text{live}) \)
- If single-threaded, no mark stack needed
- Wastes 50% of memory
Demo: Synthesizer in Java

• Human performance:
  – Latency < 8ms end-to-end
  – Jitter < 10us

• MIDI/soundcard latency: 2.5ms
Kinds of “Concurrent” Collection

- “Stop the World”
- Parallel
- Concurrent
- Incremental
Our Subject: Metronome-2 System

- Parallel, Incremental, and Concurrent
- No increment exceeds 450us
- Real-time Scheduling
- Smooth adaptation from under- to over-load
- Implementation in production JVM
What Does “Real-time” Mean?

• Minimal, predictable interruption of application

• Collection finishes before heap is exhausted

• “Real space” - bounded, predictable memory

• Honor thread priorities
The Cycle of Life

- Not really a “garbage collector”…
- … but a memory management subsystem
Metronome Memory Organization

- Page-based
- Segregated free lists
- Ratio bounds internal & page-internal fragmentation
Large Objects: Arraylets

- (Almost) eliminates external fragmentation
- (Almost) eliminates need for compaction
- Very large arrays still need contiguous pages
- Extra indirection for array access
Handling the Concurrency

GC Threads

- Mark
- Collect
- Format

Application ("mutator") Threads

- Load pointer
- Store pointer
- Allocate

Threads

Application

 Laden pointer

 GC

 Threads

 Mark

 Collect

 Format

 Load pointer

 Store pointer

 Allocate

 Application

 Laden pointer
Yuasa Snapshot Algorithm (1990)

• Logically
  – Take a “copy-on-write” heap snapshot
  – Collect the garbage in that snapshot

• Physically
  – Stop all threads
  – Copy their stacks (“roots”)
  – Force them to save over-written pointers
  – Trace roots and over-written pointers

* Dijkstra’75, Steele’76
1: Take Logical Snapshot
2(a): Copy Over-written Pointers
2(b): Trace

* Color is per-object mark bit
2(c): Allocate “Black”
3(a): Sweep Garbage
3(b): Allocate “White”
4: Clear Marks
Yuasa Algorithm Phases

- Snapshot stack and global roots
- Trace
- Flip
- Sweep
- Flip
- Clear Marks

* Synchronous
Metronome-2 Concurrency

- GC Master Thread
- GC Worker Threads
- Application Threads (may do GC work)
Metronome-2 Phases

- **Initiation**
  - Setup
    - turn double barrier on

- **Root Scan**
  - Active Finalizer scan
  - Class scan
  - **Thread scan**
    - switch to single barrier, color to black
  - Debugger, JNI, Class Loader scan

- **Trace**
  - Trace*
  - Trace Terminate***

- **Re-materialization 1**
  - Weak/Soft/Phantom Reference List Transfer
  - **Weak Reference clearing** (snapshot)

- **Re-Trace 1**
  - Trace Master
  - (Trace*)
  - (Trace Terminate***)

- **Re-materialization 2**
  - Finalizable Processing

- **Clearing**
  - Monitor Table clearing
  - JNI Weak Global clearing
  - Debugger Reference clearing
  - JVMTI Table clearing
  - Phantom Reference clearing

- **Re-Trace 2**
  - Trace Master
  - (Trace*)
  - (Trace Terminate***)
  - Class Unloading

- **Flip**
  - **Move Available Lists to Full List** (contention)
    - turn write barrier off
  - **Flush Per-thread Allocation Pages**
    - switch allocation color to white
    - switch to temp full list

- **Sweeping**
  - **Sweep**
  - **Switch to regular Full List**
  - **Move Temp Full List to regular Full List** (contention)

- **Completion**
  - Finalizer Wakeup
  - Class Unloading Flush
  - **Clearable Compaction**
  - Book-keeping

* Parallel
** Callback
*** Single actor symmetric