



# Garbage Collection for Safety Critical Java

Martin Schoeberl  
Vienna University of Technology, Austria  
Jan Vitek  
Purdue University, USA



# Outline

- GC for RTS?
- Safety critical Java
- SCJ GC simplifications
- GC scheduling
- Implementation
- Experiments
- Conclusion

# Garbage-collection for RTS?

- Simpler programming model
- Can be safer – e.g. immutable objects
- Non GC solutions
  - Static allocation
  - Scoped memory (RTSJ)
  - Application managed memory (pool of objects)

# Garbage collection

- Traverse all live objects
  - Rest is garbage
- Recycle memory
- Properties
  - Stop-the-world / concurrent
  - Mark-sweep / copy
  - Compact or not
  - Exact / conservative pointers
  - Work or time based

# GC for Real-time Systems

- Concurrent
  - Short maximum blocking time
- Compacting
- Exact root scan (pointer)
- Time-triggered
  - Keep up with generated garbage



# Safety Critical Java

- Java for DO-178B applications
- Restricted subset of RTSJ
  - More worst case analysis friendly
  - JSR 302 on-going work
- 3 different levels
- More tomorrow by Doug Locke



# SCJ Warning

- **JSR 302 excludes GC!**
- This talk is about
  - Using the SCJ specification
  - With GC for non safety critical systems!



# Safety Critical Java Features

- L0 Single threaded
  - No GC thread possible
- L1 Static threads
  - Initialization and mission phase
- L2 Multiple missions
- Single run method for all tasks
  - No waitForNextX()

# Scoped Memory

- Good for simple dynamic memory needs
- Wrong usage generates Exceptions
- Producer/consumer pattern
  - Not directly supported in RTSJ
  - Not possible in SCJ
    - Scopes are not shared

# Periodic Tasks

## ■ RTSJ

- `waitForNextPeriod()`
- Split of logic possible

## ■ SCJ

- `Single run()` method
- Easier to analyze
- No local state preserving

# RTSJ Periodic Task

```
public void run() {  
  
    State local = new State();  
    doSomeInit();  
    local.setA();  
    waitForNextPeriod();  
  
    for (;;) {  
        while (!switchToB()) {  
            doModeAwork();  
            waitForNextPeriod();  
        }  
        local.setB();  
        while (!switchToA()) {  
            doModeBWork();  
            waitForNextPeriod();  
        }  
        local.setA();  
    }  
}
```

- Possible abuse
  - Local state
  - Initialization
  - Split logic
- WCET analysis harder

# SCJ Periodic Task

- `run()` *executed*
  - Periodic (time triggered)
  - Sporadic (event triggered)
- Single method to analyze
- No local state

```
new PeriodicThread(  
    new RelativeTime(...)) {  
  
    protected boolean run() {  
        doPeriodicWork();  
        return true;  
    }  
};
```

# GC Simplification by SCJ

- SCJ (Level 1)
  - No blocking (no `Object.wait()`)
  - No locals on the stack when waiting for the next period/event
  - Initialization and mission phase
- GC runs when all threads are blocked

# GC Scheduling

- Work based
  - Allocation pays for GC work
  - Jamaica
- Time based
  - High priority – dual period
    - Metronome
  - Low priority
    - Lund, Sun, Aonix, this work



# GC Thread Priority

- Which priority for the GC?
- Answer given by scheduling theory
  - Deadline monotonic
  - GC deadline = GC period
  - GC period  $>$  task deadlines
- GC has lowest priority

# Maximum GC Period

- Mutator thread  $i$
- $a_i$  allocated
- period  $T_i$
- Live one period
- Larger heap  $H$   
=> longer GC period
- Mark-compact  
similar to Copy

$$T_{GC} \leq \frac{H_{MC} - 3 \sum_{i=1}^n a_i}{2 \sum_{i=1}^n \frac{a_i}{T_i}}$$

$$T_{GC} \leq \frac{H_{CC} - 4 \sum_{i=1}^n a_i}{2 \sum_{i=1}^n \frac{a_i}{T_i}}$$

Details in ISORC 2006 paper

# Data Sharing

## ■ Producer/Consumer

- Allocated by producer
- *Freed* by consumer
- For producer  $\tau_i$  and consumer  $\tau_c$

live factor  $l_i = 2 \left\lceil \frac{T_c}{T_i} \right\rceil$

$$T_{GC} \leq \frac{H_{CC} - 2 \sum_{i=1}^n a_i l_i - 2 \sum_{i=1}^n a_i}{2 \sum_{i=1}^n \frac{a_i}{T_i}}$$



# Benefit from SCJ

- Simple root scanning
  - Stack is empty when GC runs
    - No atomic stack scanning
  - Only static references
- Initialization – Mission
  - Stop-the-world GC at Init
  - Simplified static memory

# Static Objects

- Objects that live *forever*
- Add size to maximum live
- Hurts copy collector
  - More memory needed
  - Are moved each GC cycle
- Solution:
  - Use a *static* memory



# Static Memory

- Special GC cycle
  - At mission start
  - All live data becomes static data
- Scanned for references
- Not collected
- $L_{\max}$  again only allocation from periodic threads



# Difference to RTSJ Immortal

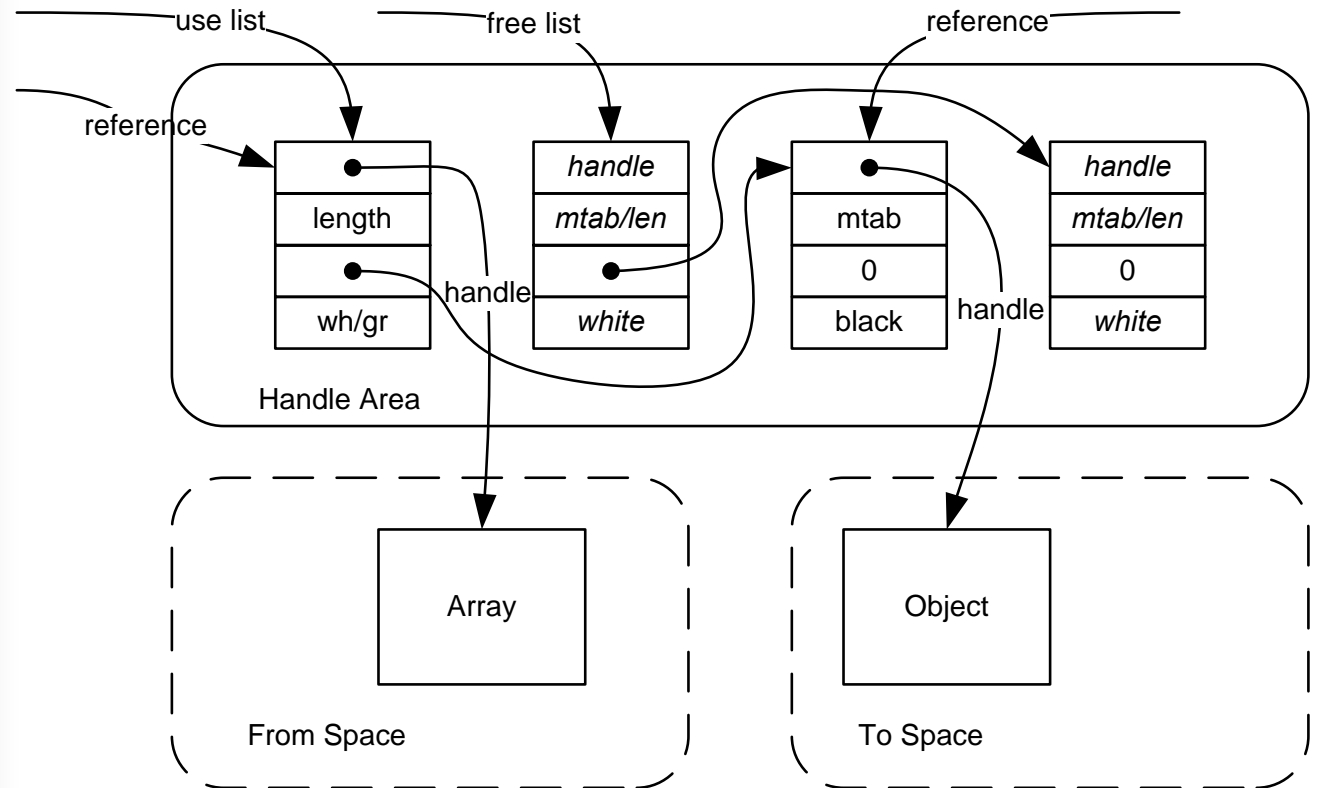
- No explicit allocation area
  - Information is implicitly gathered at mission start
- Automatically sized
- RTSJ Immortal
  - Scopes for dynamic memory
  - No references to scopes
- Static memory
  - References to heap are ok



# Implementation

- Two modes
  - Stop-the-world at initialization
    - Stack scanned for roots
  - Concurrent at mission
- Copy collector
  - Indirection through handles
    - Distinct handle area - not moved
  - Only static roots marked
    - No atomic stack scan

# Heap Layout



# The GC Algorithm

- Flip
- Mark roots
  - Atomic for a single static reference
- Mark and copy
  - Single mark operation atomic
  - Copy is atomic
- Sweep handle area
- Clear from-space



# Experiment Setup

- Concurrent Copy Collector
- 100 MHz Java processor (JOP)
  - 4 KB method cache
  - 512 byte stack cache
- 1 MB Heap
- Synthetic benchmarks
  - Producer/Consumer
- GC blocking time measured

# Measuring Release Jitter

```
public boolean run() {  
  
    int t =  
        Native.rdMem(Const.US);  
    if (!notFirst) {  
        expected = t+period;  
        notFirst = true;  
    } else {  
        int diff = t-expected;  
        if (diff>max) max = diff;  
        if (diff<min) min = diff;  
        expected += period;  
    }  
    work();  
  
    return true;  
}
```

- No notation of start time
- Relative measurements
- First release could be delayed
- Later correct release will be earlier
- Max. jitter = max - min

# Experiment 1

- Release jitter for single thread
  - No GC thread
  - 1 million releases measured
  - 100 us shortest practical period

Period	Jitter
200 us	0 us
100 us	0 us
50 us	17 us

# Task Set

- HF task
- Producer/Consumer pair
- GC with short period

Thread	$T_i$	$D_i$	Priority
HF	100 us	100 us	5
Prod.	1 ms	1 ms	4
Cons.	10 ms	10 ms	3
Logging	1000 ms	100 ms	2
GC	200 ms	200 ms	1

# Experiments

- Tests with various task sets
  - Run for hours
- Test 3 with stop-the-world GC
  - 4 seconds till first GC, 63 ms GC blocking
- Last test with prime numbers for periods

Threads	HF Jitter
HF	0 us
HF, Log	7 us
HF, Log, P, C	14 us
HF, Log, P, C, GC	54 us



# Conclusion

- GC is an option for RTS
- Time-triggered GC
- SCJ Simplification
  - No atomic stack scanning
  - Stop-the-world GC at initialization
  - Automatic immortal memory
- Short blocking time
  - Depends on copy for arrays
  - HW support as future work

A composite image for the top half of the slide. The left side shows a stack of papers with a blue tint, and the right side shows a clock face with a purple tint. A black-bordered box is overlaid on the center.

Thank You

A composite image for the bottom half of the slide. The left side shows a stack of papers with a green tint, and the right side shows a clock face with a yellow tint. A black-bordered box is overlaid on the center.

Questions & Suggestions