High-level Real-time Programming in Java

Josh Auerbach   David Bacon   Perry Cheng
Dave Grove      V.T. Rajan    Michael Hind
IBM T.J. Watson Research Center
Matthias Hauswirth U. della Svizzera Christoph Kirsch
U. Salzburg
Daniel Spoonhower Martin Vechev
Carnegie Mellon U. U. of Cambridge
Why Real-Time Java??

- **Traditional methodologies**
  - Highly restricted programming models with verifiable properties
  - And/Or low-level languages for explicit control
  - “ad-hoc low-level methods with validation by simulation and prototyping”

- **But: these methodologies do not scale**
  - Halting problem
  - Low productivity (low-level languages, hand-optimization)

- **And: complexity of real-time systems are growing extremely fast**
  - From isolated devices to integrated multi-level networked systems
  - Traditional methodologies break down
Why Not Real-time Java?

- **Garbage Collection**
  - Non-deterministic pauses from 100 ms to 1 second
  - Requirement for real-time behavior is 100 us to 10 ms

- **Dynamic (JIT) Compilation**
  - Unpredictable interruptions
  - Large variation in speed (10x)

- **Dynamic Loading and Resolution**
  - Semantics determined by run-time ordering

- **Optimization technology optimizes average case**
  - Thin locks, speculative in-lining, value prediction, etc.
  - Sometimes cause non-deterministic slowdowns

- ...
Not just “Soft” versus “Hard” Real-time

Compare: video in a web browser versus video master broadcast
Our Vision for Real-time Java

- Change the programming model...
  - By changing the domain in which the language can be applied;
  - Not by changing the language

- Comprehensive programming methodology & environment
  - For the new generation of complex real-time systems
From Seconds to Nanoseconds...

- STRATEGY
- TACTICS
- COORDINATION
- ACTUATION
- SENSING
- MODULATION
- SIGNALING
- CUSTOM HARDWARE

- PERCEPTION
- REACTION
- COGNITION
From Sensors to Supercomputers...
IBM Real-Time Java (J9 Virtual Machine)

- **Metronome Real-time Garbage Collection**
  - Provides real-time without changing the programming model

- **RTSJ (Real-Time Specification for Java) - existing standard**
  - Scheduling
  - Scopes

- **Ahead-of-Time Compilation**
  - Ahead-of-time (AOT) compilation and JXE Linking
  - Removes JIT non-determinism, allows code to be moved into ROM

- **Real-time Linux**
  - Maximize use of existing patches; stabilize; add needed features
  - Contribute to open-source community

- **Status**
  - Being tested by major customer in defense missions sector
  - Other industries: telco, sensors & actuators, financial, gaming
  - Fourth alpha version delivered to customers, university partners 7/05
What’s Next for Real-time Java

• Background: RTSJ Scopes

• Metronome Real-time Garbage Collection

• Eventrons

• E-Tasks

• Conclusions

(lots more in the paper)
RTSJ Scopes
RTSJ Scoped Memory

SCOPE 1

SCOPE 2

STACK

HEAP

LEGEND
- Green: Legal RTSJ Pointer
- Red: Legal, NHRT Inaccessible
- Grey: Illegal RTSJ Pointer

IMMORTAL MEMORY (Includes Globals)
Problems with Scopes

- Change to fundamental Java semantics
  - Both reads and writes can fail ("safe SEGFAULT")
  - Smells like Java, but isn’t
- Expensive to implement (read and write barriers)
  - And hard to optimize
- Violates modularity
  - Incompatible with pre-existing code; no re-use
  - Huge problem for builders of large systems
RTSJ Real-time Programming Abstractions

RealTimeThread with Heap

NoHeapRealTimeThread with Scopes

10 s
1 s
100 ms
10 ms
1 ms
100 µs
10 µs
Metronome
Real-time Garbage Collection
What is Metronome?

- **A true real-time garbage collector**
  - 1 ms worst-case pause
  - Sufficient for majority of real-time applications
  - Guaranteed utilization (typically 70-80% at 10 ms resolution)
  - Guaranteed ≠ Proved (system is too complex)

- **True to the Java programming model**
  - No change to memory semantics
  - Bounds based on simple application characterization

- **Originally a uniprocessor algorithm...**
  - Embedded systems heritage
  - Forces highly accurate analysis, but simplifies concurrency
  - But now extended to medium-scale SMP’s

[Bacon, Cheng, Rajan - POPL’03]
Scheduling Collection: Redistributing Work

Example Application

Allocates half as fast as the collector can collect
\[ c = -2a \]

Note: collector frees no memory until done!

Application

Collector

Base Application Memory

Resulting Schedule
Metronome Execution: TuningFork Demo
Metronome Programming Abstractions

- RealTimeThread with Heap
- NoHeapRealTimeThread with Scopes

100 ms
10 ms
1 ms
10 s
1 s
100 µs
10 µs
Eventrons
Eventrons: Very Low Latency Operations

- **General principle:**
  - The higher the frequency, the simpler the task
- **Many high-frequency tasks do buffer processing**
  - Don’t create new data structures, just move data

- **Eventrons**
  - Data structure must be allocated in advance
    - Usually includes some buffers
  - When Eventron is created, code and data are checked
    - If OK, then guaranteed to be free of memory exceptions
  - More complex processing can be done by low-frequency task
    - Builds data structures collected by real-time garbage collector
Eventrons vs. Raw Linux Capability

Period Histogram (50us)

- Raw usleep
- Eventron
Metronome + Eventron Abstractions

RealTimeThread with Heap

10 s
1 s
100 ms
10 ms
1 ms
100 µs
10 µs

- Metronome
- Eventrons
- RealTimeThread with Heap

David F. Bacon
EMSOFT'05 Presentation
18 September 2005
E-Tasks
Task-based Model with L.E.T.

[Henzinger & Kirsch – EMSOFT’01; Henzinger et al – PLDI’02]
Time Portability

- Compelling Java feature: “write once, run anywhere”
  - As long as you don’t care about timing
- Time-portability is a critical problem
  - Otherwise, must re-test on every platform change
- Logical execution time (LET) provides a framework
  - Specify external timing
  - Compile to infinitely fast abstract virtual machine (E-code)
  - Validate when loading application in a particular virtual machine
E-Tasks: A Functional Abstraction in Java

- Task memory is private
  - No pointers to or from heap
  - Global access restricted to constants
- Dynamic allocation allowed
  - Tasks may be garbage collected internally
- Complex data structures may be sent and received
  - “Send by collection”
- Validated – no run-time errors
  - Less precise than Eventrons, since data structures unknown
  - Dependent on order of validation
Automated Space/Time Tradeoffs

- Logical execution time of E-code provides framework
  - Garbage collection within tasks allows space/time tradeoff
- Scheduling as a space/time optimization problem
  - Dynamic adaptation to CPU speed, memory size
  - Deterministic, portable behavior of logical execution time
## Comparing Real-time Abstractions

<table>
<thead>
<tr>
<th></th>
<th>Eventrons</th>
<th>Scopes</th>
<th>E-Tasks</th>
<th>Metronome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Preemption</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Access to rest of Heap</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Access from Heap</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Allocation</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Memory Safety</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Memory Reclamation</td>
<td>GC on finalize</td>
<td>Free on exit</td>
<td>Local GC</td>
<td>Global GC</td>
</tr>
<tr>
<td>Latency Limit</td>
<td>2 us</td>
<td>10 us</td>
<td>50 us</td>
<td>250 us</td>
</tr>
</tbody>
</table>
Metronome + Eventron + E-Task Abstractions

RealTimeThread with Heap

- 10 s
- 1 s
- 100 ms
- 10 ms
- 1 ms
- 100 µs
- 10 µs

E-Tasks

Eventrons
Conclusions
Conclusions

- **New generation of complex real-time systems**
  - Require software engineering benefits of Java

- **Real-time Java has come a long way**
  - Scheduling
  - Real-time garbage collection
  - Deterministic compilation

- **Many challenges remain**
  - Latencies competitive with C
  - Predictable scheduling
  - Time-portability

- **But real-time Java is happening now**
  - Huge interest across many industries
  - State of the art product-quality implementation
  - Major improvements in programming models and tools
Questions?

http://www.research.ibm.com/metronome
Don’t Avoid Garbage Collection – Fix It!

- Garbage collection invented by McCarthy [1960]
  - Work on real-time collection begins in 1978 [Baker]
- But fundamental problems were not solved
  - Space overhead 4-8x
  - Fragmentation (impractical worst-case space bounds)
  - Extra CPU required (collector run on separate CPU)
  - No guaranteed time bounds
- So: not credible in real-time and systems communities
  - Led to design of Scopes in RTSJ
Can Application be Modeled So Simply?

Collector

\( a = \text{allocation rate??} \)

\( c = \text{collection rate} \)

Space

Time
Allocation Stability vs. Time Scale

![Bar Chart]

- **Time Window (microseconds)**:
  - 100
  - 500
  - 1000
  - 5000
  - 10000
  - Average

- **Peak Allocation Rate**
  - MB/s (peak)
  - 0
  - 50
  - 100
  - 150
  - 200
  - 250
  - 300

**Graph Description**
- The bar chart illustrates the peak allocation rate over different time windows. The y-axis represents the MB/s (peak) of allocation, while the x-axis denotes the time window in microseconds.
- The chart shows a significant peak at 100 microseconds, with a gradual decrease as the time window increases.
- The average allocation rate is also indicated for comparison.
Tuning Fork
Tracing, Analysis, and Visualization

- **Trace facility components**
  - Cycle-accurate record of JVM events (GC, JIT, scheduler, etc)
  - Linux kernel modification allows integration of OS events
  - User-level (Java) events

- **Visualization and Analysis**
  - Currently have static tools; need “always on” monitoring
  - Trace analyzer/visualizer must itself be a real-time system
  - Detect and diagnose real-time faults
  - Oscilloscope/TiVO user interface; zoom in and out
  - Dynamic statistical analysis of behavior for FAWCET
## Comparing Real-time Abstractions

<table>
<thead>
<tr>
<th></th>
<th>Eventrons</th>
<th>Scopes</th>
<th>Metronome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Preemption</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Access to rest of Heap</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Access from Heap</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Allocation</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Memory Safety</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Memory Reclamation</td>
<td>GC on finalize</td>
<td>Free on exit</td>
<td>Global GC</td>
</tr>
<tr>
<td>Latency Limit</td>
<td>2 us</td>
<td>10 us</td>
<td>250 us</td>
</tr>
</tbody>
</table>
BACKUP
Minimum Mutator Utilization

- Metric for “real-timeness”
- What it’s not
  - Throughput
  - Latency
- What it is
  - Worst-case utilization over a time interval
  - Interval may contain multiple short interruptions
    - Upper bound on latency (interval x utilization)
  - Other intervals may have higher utilization (100% when GC off)
    - Lower bound on throughput (utilization)

\[ \Delta t = 4 \text{ms} \]
\[ u = 50\% \]

Latency = 1ms
Throughput = 75%

[Cheng, Blelloch – PLDI’01]
Not just “Soft” versus “Hard” Real-time

TIME SCALE

DETERMINISM

RECOVERY

1 us

1 min

1 us
Garbage Collection as a Periodic Task

Why it wasn’t used

- Short period: low CPU utilization
  - Missed deadlines

- Long period: low memory utilization
  - Memory overflows → Synchronous Collection → Missed deadlines
Space γ Time

**Scheduler**

- \( u = \text{utilization} \)
- \( s = \text{used space} \)
  - 50% 75%
  - 45 MB 100 MB
- \( \Delta t = \text{period} \)
  - 5 ms

**Application (Mutator)**

- \( a^*(\Delta GC) = \text{Per-GC Allocate Rate} \)
- \( m = \text{Live Data} \)
  - 50 MB/s
  - 30 MB

**Collector**

- \( R_T = \text{Trace Rate} \)
  - 50 MB/s
- \( R_S = \text{Sweep Rate} \)
  - 300 MB/s
Limits of Real-time Garbage Collection

- Changes to heap require synchronization
  - Application modifies pointers and allocates objects
  - Collector moves objects to compact memory
- Synchronization is expensive
  - To keep cost reasonable, done in quanta (Metronome “beats”)
  - Quantization has limit (250-500 us)
- Real-time collection works for many tasks, but not all
  - Currently, only alternative is Scopes
Multiprocessor
Metronome
Multiprocessor Metronome

- **Multiprocessor support required for large systems**
  - Coordination layer supervises numerous real-time subsystems

- **Metronome uses safe-point architecture**
  - Stops mutator threads at predictable points
  - Makes low-level mutator and collector operations atomic

- **On SMP, atomicity is lost unless all threads are stopped**
  - Leads to race condition when GC moves an object (defragment)
Multiprocessor Metronome (1)

- Synchronous collector increments ("stop-the-worldlet")
  - Limits concurrency; makes problem tractable
  - Barrier synchronizations limit scalability to ~8 CPUs
- Real-time-constrained load balancing
Multiprocessor Metronome (2) - Staccato

- To scale, must have fine-grained concurrency
  - But cost is too high
- Staccato uses an asymmetric abort protocol
  - Low cost synchronization
  - But not guaranteed to make progress
- Apply probabilistic (FAWCET) approach
  - Bound variance; Reduce correlation
  - Progress more likely than hardware failure
- Benefits
  - Very high degree of SMP scalability
  - Arbitrary pre-emptibility means microsecond response
    - Metronome is 100x more responsive than competitors; Staccato is 100x additional
Deterministic Compilation
Eliminating JIT Non-determinism (1) - AOT

- Java is a dynamic language
  - Compilation is “just-in-time”
  - So unfortunately, meaning of “compile” is order-dependent
- Ahead-of-time Compilation (AOT)
  - Compiles jar files into loadable binary modules
  - Splits modules into read-only (ROM) and read-write portions
- Supports all Java features:
  - Dynamic class loading; Reflection
- Inhibited optimizations
  - Optimizations with high variability
  - Just-in-time (data dependent) optimizations
  - 80-110% speed of JITted code
Eliminating JIT Non-determinism (2)

- **AOT Compilation sacrifices significant performance**
- **Current JIT techniques are ad-hoc and non-incremental**
  - Sometimes introduce slowdowns
- **Extend Metronome methodology to JIT compilation**
  - Collector “beats” become JVM beats - use for GC, JIT, etc.
  - Distribute JIT over time without impacting deadlines
- **Instrument/update code in time-safe manner**
  - Use slack for trial runs
  - If bad optimization, abandon before deadline missed

[w/ Jeremy Lau, Matthew Arnold]
FAWCET: Probabilistic Real-time
Faster Hardware, Less Deterministic (?)
FAWCET: Probabilistic Real-Time Analysis

- **Real-time Truism:** “Real-Time is Not Real Fast”
  - But we say: *Real Slow is Not Real Good*
  - Faster programs are more likely to meet deadlines
  - Worst-Case Execution Time (WCET) analysis inhibits optimization

- **We propose a probabilistic approach**
  - FAWCET: Frequency Analyzed Worst-Case Execution Time
    - Embrace non-determinism instead of avoiding it
  - Require sources of non-determinism statistically independent
    - Cache miss, hash table collision, buffer overflow, devirtualization failure
  - Allows over-provisioning to be amortized over event types
    - Drive probability of timing failure below hardware MTBF
FAWCET vs. WCET

- Short Time Scale
  - WCET: Few events.
  - FAWCET: Many events. Modest over-provisioning

- Long Time Scale
Synthesis and Verification of Concurrent Collectors

- Steele
- Dijkstra
- Vechev et al
- Yuasa

Abstract Collector

Memory Overhead (Floating Garbage)

Worst-case Completion Time

[ Vechev, Bacon, Cheng, Grove - ECOOP’05]
Syncopation: Generational Real-Time GC

- Allocation rate is single biggest factor controlling utilization
  - Nursery reduces heap allocation rate
  - With arraylet pre-tenuring, survival rate even lower (up to 3x reduction)
- Collect nursery synchronously
  - On ARM, we can collect 64K in 4ms; on Pentium, collect 256K in 500 us
  - So, collect small nursery synchronously
  - But spike in allocation rate or survival rate can hurt utilization
  - Divide real-time interval (measure) into beats
  - If too many nursery beats, syncopate: spill allocation into heap
- Early results: up to 30% increase in utilization possible

[Bacon, Cheng, Grove, Vechev – LCTES’05]
GC Pause Times (ms)
Instantaneous Utilization: 10 ms (100 Hz)