Java Language and Tools for Realtime and Safety Critical Applications

The Challenges and Opportunities of Using Java Technology

aicas incorporated

Dr. James J. Hunt, CEO
20th of June 2007

When faced with ever more complex requirements, greater cost pressure, and rigorous certification, the choice of languages matters!

The Importance of Language

- Available Expertise
- Tools Support
- Available Libraries
- Code Generation
- Resource Efficiency
- Robustness
- Understandability

- Traceability
- Analyzability
- Reusability
- Modularity
- Consistency
- Completeness
- Soundness

Language Types

- Specification languages
- Modelling languages
- Domain specific languages
- Programming languages
- Configuration languages
- Testing languages
- Byte and Machine Code
Critical Systems Development in Flux

- Ada losing market relevance
  - Ever fewer qualified developers
  - Perceived as career dead end
  - Decreasing vendor support
- C and C++ popular but provide little programmer support
- Java technology is increasingly interesting
  - Tools support
  - Implementation for critical systems maturing

Competitive Advantages of Java

- Time to Market
- Reliability
- Flexibility
  - Look and feel
  - Extensibility
- Reusability of code
- Platform independence

Advantage of Java over C and C++

- Clean syntax and semantics w/o preprocessor
  - wide ranging and better tool support
- Better support for separating of subtyping and reuse via limited inheritance and interfaces
- No explicit pointer manipulation
- Pointer safe deallocation
- Single dispatch style
- Strong, extensible type system
- Well defined tasking model

Risks of Using Java Technology

- High memory requirements
- Poor runtime performance
- Pauses during execution due to GC
  - Poor user interface feedback
  - Unpredictable execution time
- Undefined scheduling semantics
- No accurate clocks and timers
- Danger of priority inversion in shared code
- No direct hardware access
Alleviating Risks of Java Technology
- Code Reduction
- Static Compiler Technology with Profiling
  - Faster Code
  - Better time vs. space trade off
- Real-Time Specification for Java
- Real-time Garbage Collection
- Safety Critical Java

Reduction of Memory Demand
High memory demand of Java is mainly due to large libraries; but there are solutions.
- Reduction of the Java APIs through configurations and profiles (CLDC, CDC, etc.).
- Class file compaction
  - Use of a compact class format
  - Execution out of ROM
    Typically saves 50% of the code size
- Smart Linking: removal of unused code
  Typically saves 70-90% of the code size

Java bytecode is 2-3x smaller than compiled C Code!

Reduction of Memory Demand

Execution Speed
Java interpreters are slow; compiled code is faster, but
- Just-in-Time Compilation is not deterministic
- Load-time Compilation has high memory demand on target system
- Static Compilation with Profiling yields best results for small, closed systems.
  Embedded VM for dynamic loaded code

Typical compilation speed up: 10-30 times faster
Real-Time Specification for Java (RTSJ)

- Extends Java for realtime programming
- Many new features to manage
  - Scheduling
  - Threads
  - Memory
  - Synchronization
  - Control Flow

Realtime Scheduling

- New execution environment for Java
  - RealtimeThreads
  - AsyncEventHandler
  - At least 28 additional realtime priorities
- Scheduler
  - PriorityScheduler is required as default
  - Fixed priority, preemptive scheduling
  - Both RealtimeThread and AsyncEventHandler are Schedulable objects

Asynchronous Event Handlers

- Bind Schedulable object to an event for immediate processing upon reception
- Provides an additional processing paradigm besides RealtimeThreads
- Light weight (>10,000 possible)
- Executed in a thread context but need not be bound to one
- Easier to analyze

Threads without GC

No-Heap version of schedulable objects:

- NoHeapRealtimeThread
- AsyncEventHandler with noheap == true

Only no-heap versions are guaranteed not to be interrupted by garbage collection.

No-heap threads and event handlers must not access heap.

Runtime checks enforce this by throwing an IllegalAccessError if a heap object is used.
Memory Areas

Memory Area abstracted from the heap to generalize the concept of allocation context.

Heap Memory
- default allocation context
- controlled by GC
- not accessible by no heap schedulable objects

Immortal Memory
- default allocation context for static initializers
- memory is never reclaimed
- accessible by all schedulable objects

Access to Physical Memory

- RawMemoryAccess for getting and setting bytes of physical memory
  - Reading memory mapped sensors
  - Device control
- LTPhysicalMemory and VTPhysicalMemory for mapping Java object to special memory areas
  - Make special memory available for Java objects, e.g. fast memory.

Priority Inversion can be problematic

Deadline Task A missed!

Synchronization

Avoiding dangling and false references
- assignment rules prohibit creation of potential dangling references:
  - `a.f = b` for `b` in scope `s` is allowed only if `a` is in the same scope or in an inner scope of `s`.
- `IllegalAssignmentError` if assignment rule violated
**Priority Inheritance**
- Does not need any user configuration
- Deadlock can occur

**Priority Ceiling**
- Is inherently deadlock free
- Requires manual selection of ceiling priority
- May block threads more often than needed

**Deadline and Cost Monitoring**
- Notification when deadline is missed or cost for thread is too high
- Enables program to take corrective action

**Asynchronous Transfer of Control (ATC)**
- Enables a thread to interrupt another thread by throwing an exception in the other thread's context
- Terminates unneeded execution
The Challenge: Garbage Collection

**Garbage Collector**
- needs to clean up memory for the user.

**In classic Java systems**
- Dedicated thread for GC
- Regularly stops the application for GC work

**Consequences**
- Difficult to predict memory demand
- Unpredictable pauses
- **No realtime or safety critical code possible!**

Realtime Garbage Collection

- Paced garbage collector
  - Run GC at lower priority than realtime tasks
  - Scheduled to run at a given interval, for a given amount of time
  - Programmer must know both maximum memory use and maximum allocation rate

- Allocation time garbage collector
  - No GC thread; GC borrows application thread
  - Needs only maximum memory use

Classic Garbage Collection

GC can interrupt execution for long periods of time:

```
<table>
<thead>
<tr>
<th>Thread:</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC</td>
<td></td>
</tr>
<tr>
<td>User 1</td>
<td></td>
</tr>
<tr>
<td>User 2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

**Problem**
long, unpredictable pauses during execution

RTSJ with Classic Garbage Collection

No heap threads can interrupt garbage collector:

```
<table>
<thead>
<tr>
<th>Thread:</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt1</td>
<td></td>
</tr>
<tr>
<td>rt2</td>
<td></td>
</tr>
<tr>
<td>GC</td>
<td></td>
</tr>
<tr>
<td>User 1</td>
<td></td>
</tr>
<tr>
<td>User 2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

The application must be split into a realtime and a non-realtime part.
The JamaicaVM Solution

Deterministic Garbage Collection

All Java threads are realtime threads:

<table>
<thead>
<tr>
<th>Thread</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- GC work is performed at allocation time
- GC work must be sufficient to recycle enough memory before free memory is exhausted
- Execution time of all allocations must be bounded

RTSJ with Deterministic GC

- The RTSJ provides necessary features for realtime programming
- Memory area restrictions can be relaxed
  - Can use RealtimeThread instead of NoHeapRealtimeThread
  - Heap allocation possible in realtime code
- Synchronization possible with non realtime tasks without GC interference
- GC does not interrupt thread execution

Safety Critical Java (JSR 302)

- Java optimized for safety critical application, e.g. DO-178B levels A and B
- Based on a subset of the RTSJ
- Uses ScopedMemory for managing deallocation instead of garbage collection
- Uses extended typing through annotations to support static analysis
- Minimum set of supported classes

Safety Critical Java (JSR 302)

- Immortal memory and single mission scope
  - Three phases execution: initialize, run, recovery
  - Mission scope entered before initialization and exited after recovery (reinitializable immortal)
  - Allocation in mission scope only during initialization
- Dedicated scope for each execution context
  - Entered before release and exited on completion
  - Only state using primitive types saved in objects in mission scope can survive between releases
Three Phase Execution Model

- **Initialization Phase**
- **Run Phase**
- **Recovery Phase**

Restricted Scoping Model

- **ImmortalMemory** (class objects)
- **MissionMemory** (mission objects)

Possibly Three Levels

- **Level 0**
  - Cyclic Executive
  - No threading
- **Level 1**
  - Single Mission
  - Limited synchronization
- **Level 2**
  - Nested Missions
  - Flexible synchronization

Certification Issues

- Class initialization
- Dynamic dispatch
- Garbage collection
- Unchecked exceptions
- Dynamic class loading
- Just in time compilation
- Reflection
- Asynchronous transfer of control
Support and Analysis Tools

- Debugging
- Refactoring
- Data Flow Analysis (DFA)
- Thread monitoring
- Deductive Functional Verification
- Model checking
- Worst case memory analysis
- Worst case execution time analysis

Debugging and Monitoring in Java

Data Flow Analysis

- Full program flow analysis
- Fixed point algorithm
- Can detect all possible runtime exceptions

Class cast
Array Store
Index out of bounds
Negative array size
Divide by zero
Scope cycle
Illegal assignment
Null pointer
Sync needed
Deadlock
Illegal sync use
Illegal wait use
Detecting Runtime Errors

```java
if (device instanceof MyDevice) {
    MySensor s = (MySensor) device.sensor;
    int value = s.reading();
} ...
```

**NullPointerException**

```java
if (device instanceof MyDevice) {
    MySensor s = (MySensor) device.sensor;
    int value = s.reading();
} ...
```

**ClassCastException**

```java
if (device instanceof MyDevice) {
    MySensor s = (MySensor) device.sensor;
    int value = s.reading();
} ...
```

**NullPointerException**

```java
if (device instanceof MyDevice) {
    MySensor s = (MySensor) device.sensor;
    int value = s.reading();
} ...
```
Detecting Runtime Errors

```java
if (device != null) {
    MySensor s = (MySensor) device.sensor;
    int value = s.reading();
}
```

NullPointerException
ClassCastException
Detecting Runtime Errors

```java
if (device instanceof MyDevice) {
    MySensor s = (MySensor) device.sensor;
    int value = s.reading();
    ... } ...
```

NullPointerException

ClassCastException

```
if (device instanceof MyDevice) {
    MySensor s = (MySensor) device.sensor;
    int value = s.reading();
    ... }
```
Data Flow Analysis Tool

Model Checking
- Numerous Tools (e.g. SPIN, PROSPER, Uppaal)
- Used to check some extracted attribute of a program or program model
- Bases on state machines
  - Limited computational power
  - Subject to state explosion
- Good for state machine like programs or network traffic
- Limited value for checking complex algorithms

Deductive Formal Verification
- Formal Specifications
  - Preconditions
  - Postconditions
  - Invariants
- Liskov Substitution Principle
  - Defines a proper subtype
  - subclass = subtype
- Runtime Tools (JML)
  - Jmlc
  - Daikon
- Verification Tools
  - ESC/Java2 (Simplify)
  - JACK (B-Method, Simplify, PVS, Coq)
  - KeY (Dynamic Logic)

Conclusion
- Java technology offers significant advantages over alternative languages for critical systems.
- Standard IDE, debugging, and analysis tools ease development.
- Realtime garbage collection with RTSJ enables deterministic device control.
- Wide variety of tools available
- Standards are emerging for supporting certification to the highest level.