High-level Programming of Embedded Hard Real-Time Devices

Lukasz Ziarek
Purdue University & Fiji Systems Inc.
Joint work with Filip Pizlo, Petr Maj, Ethan Blanton, and Jan Vitek

Eurosys, April 2010
Hard Real Time Systems

Range from satellite and flight control systems to high frequency trading algorithms
Commonalities

• Predictability is paramount
  • uniprocessor, simple RTOS kernels
• Fundamentally resource constrained
  • time and space; usually embedded
• High fidelity software
  • safety, validation, and verification
Commonalities

- Written in low level languages
- C/C++ and Ada
- Manual memory management (predictability)
  - static allocation, object pooling
- Simple control flow (analyzability)
Can we use Java in such a setting?

- Challenges:
  - Java specific overheads
  - null checks, type checks, array bounds check
  - automatic memory management (GC)
  - overheads, predictability, and fragmentation
  - other concerns (resources provide by the VM):
    - threads, locking, I/O, etc
Why Java?

• Embedded and real-time systems are increasing in complexity (fully autonomous UAVs, complex medical devices, etc)

• Real-time developers want to try Java.

• Ease of development, large availability of tools and programmers, and safety

• previous attempts:

• too slow, too big, too unpredictable
Compiling Java for Hard Real Time
Fiji VM

- A Java virtual machine that can run at full speed in any environment, including OS kernels
- Ahead-of-time compilation (no JIT)
- Predictability and performance comparable to C code
- Small footprint: choice of standard libraries
- Portable: x86, SPARC, Leon, PowerPC
The Fiji VM Overview

Java Application → Fiji VM compiler → Fast Native Code

everything else

Fiji VM CI → GCC

Bytecode Parser → Fiji IR → Transform & Optimize → Fiji IR → C Code Gen

Whole Program 0CFA → Lock, allocation, barrier inling → Whole Program Dead Code elim.

Wednesday, April 14, 2010
The runtime

- Code generated by Fiji Compiler
- Libraries
- Fiji Runtime

OS threads
OS locks
Memory
Thin OS interface

Full Java: 2.2MB
Fiji Core: 500 KB

Linux
Mac OS X
RTEMS
NetBSD
Performance/Predictability

- local assignments, simple arithmetic, casts, conditionals
- loops, method invocation, field/array access, static initialization
- allocation, locking, exceptions
- condition variables, threading, I/O

- same performance as C/C++
- slightly slower than C/C++
- faster than C/C++
- identical to C/C++
Garbage Collection

- Immix style concurrent on-the-fly gc
- Concurrent mode or slack based (Kalibera et al RTSS 09)
- Executes in a separate thread of control
  - No works is offload on application threads
- GC segregates memory into:
  - Regions: large contiguous chunks
  - Lines: fragments of memory
Allocation

- Maintains free list of regions and lines
- Regions:
  - bump pointer: fast and used for the majority of allocation
- Lines:
  - first-fit: filling lines quickly reduces fragmentation
Reclamation

- Runs either concurrently or as the lowest priority thread
- Objects are never moved!
- “Pauses” only occur when:
  - Unsuccessful memory request
  - Schedulability of GC (Kalibera et al)
  - Stack scanning
    - shallow stacks (Puffitsch and Schoeberl) - fast
CDx Benchmark

- Representative Real-time benchmark
- Aircraft detection based on simulated radar frames
- CDc - written in idiomatic C
- CDj - written in idiomatic Java
- Uses many arrays and is computationally intensive
CDx Benchmark

• The algorithm detects a collision whenever the distance between aircraft is smaller than a specified “proximity radius”

• Step 1: ← eliminates planes at large distances
  • split aircraft into clusters

• Step 2: ← closer examinations of potential collisions
  • for each cluster determine actual collisions
What if we run CDx on a real-time setup?

- **RTEMS 4.9.1** (hard RTOS microkernel: no processes or virtual memory)
- **40MHz LEON3** with 64MB RAM (radiation-hardened SPARC)
- This is the platform used by ESA and NASA
CDx Configuration

- 6 airplanes in our airspace
- execute over 10,000 radar frames
- runs take on average 45 minutes
- slight modification to generate frames
- 300ms period for the collision detector task
  - between 145ms - 275ms
- leaves less than 50% of the schedule for the GC
CDc Summary

Iterations w/ potential Collisions
CDj Summary

30% slower on average

Iterations w/ potential Collisions

Number of Iterations

Time (ms)

Wednesday, April 14, 2010
Source of overheads

measured using RTBx data logger

• Expect to see larger Java overheads when potential collisions are detected

• Array bounds checks

• Type checks

• Null checks

www.rapitasystems.com
Array Bounds Checks

more work when clusters are large or frequent

6 “modes”

Number of Checks correlated against execution time
Null Checks

More work when clusters are large or frequent

6 "modes"

Number of Checks correlated against execution time
Type Checks

more work when clusters are large or frequent

6 “modes”

Number of Checks correlated against execution time

Wednesday, April 14, 2010
Worst case C

Worst case Fiji VM

10% slower

Frame Number vs. Execution Time (ms)
Correlation Java vs C when running on RTEMS/LEON3

15 Full GC collection cycles!

10,000 samples
go no outliers
Future Work

- Massive Parallelism
- GPGPU
- V&V of runtime
- V&V of GC
- Multi-VM
- JIT

Fiji VM
Related Work

- OVM
  - first unmanned flight using Java
- RTSJ/SCJ
- IBM Metronome Collector
- Other VMs: Aonix PERC, AICAS Jamaica, Sun RTS, IBM WebSphere Real-time VM
Java in Real Time

• Even with GC Java is capable for hard real time tasks
• 15 full collections cycles
• Experiments indicate Java specific overheads are bounded and quantifiable
• CDj is only 30% slower than CDc
• CDj 10% slower in the worst case
Comments/questions
But what about fragmentation?

- Schism - fragmentation aware extension to CMR
- to be presented at PLDI 2010
- handles fragmentation
  - about 2% worse in practice 15% in worst case
CDj on x86

IBM WebSphere

Sun Hotspot
Client and Server

Fiji VM

Wednesday, April 14, 2010
## The Good

- Top defects in existing embedded code:*  
  - Buffer overflow/underflow
  - Null Object dereference
  - Uninitialized variable
  - Inappropriate cast
  - Division by zero
  - Memory leaks

Source: “Diagnosing Medical Device Software Defects” Medical DeviceLink, May 2009
SpecJVM98

- HotSpot 1.6 Server
- IBM J9
- Sun Java RTS 2.1
- IBM Metronome SRT
- Fiji VM CMR
- Fiji VM Schism/cmr/c
- Fiji VM Schism/cmr/a
- Fiji VM Schism/cmr/cw

Fastest Fiji collector

Fiji hard RT collector

Worst-case simulators

throughput relative to HotSpot 1.6 server

Wednesday, April 14, 2010