

The Real-Time Specification for Java

Mike Elliott
Michael.R.Elliott2@boeing.com

Just what does real-time mean?

What is real-time?

- Hard real-time means a violation of timing constraints means system failure
- Soft real-time means timing constraints are stringent performance goals
- Timing windows may vary in magnitude (microseconds to minutes)
- Not about performance – about predictability

Two kinds of real-time

- Hard real-time
 - Analytical techniques to determine absolute bounds on worst-case resource needs
- Soft real-time
 - Use empirical techniques to estimate resource needs
- Hard real-time is hard. Soft real-time may be harder!
 - Soft real time usually deals with greater complexity and more dynamic behavior

Characterizing RT

- Very informally: Real-time SW has timing constraints that are distinct from the general desire to offer high throughput and/or user responsiveness
- Usually real-time constraints are associated with:
 - Monitoring and control of the physical environment
 - Time-constrained communication and networking protocols

So . . . Why Java?

Why Java?

- Top seven stated reasons:
 - Reduced development costs (38%)
 - Availability of open-source objects and modules (30%)
 - Quicker development (29%)
 - Availability of qualified developers (21%)
 - Improved software reuse (19%)
 - Increased system functionality (18%)
 - Reduced maintenance costs (18%)

Source: Embedded Market Forecasters (2005) (108 total responses)

Java All Over



**Mobile Solutions,
3G Networks**



**RFID,
Sensor-
boardroom**



**Remote
Devices,
Edge-Edge**



**Next
Generation
Devices**

Findings in industry

- Embedded RT developers who move from C++ to Java are roughly twice as productive during development of new functionality
- Maintenance and integration efforts are five to ten times less costly using Java than C++
 - This is largely because traditional embedded real-time development is so miserably non-portable

Testimonials

- Intel project: developed fault-tolerant distributed Java demonstration in 3 days
 - Their assessment: “It would have taken 3 solid months to develop this demonstration without Java [i.e., VxWorks and C]”
- Nortel experience: Java developers are more productive and their code more trouble-free than C++ developers

Extrinsic factors

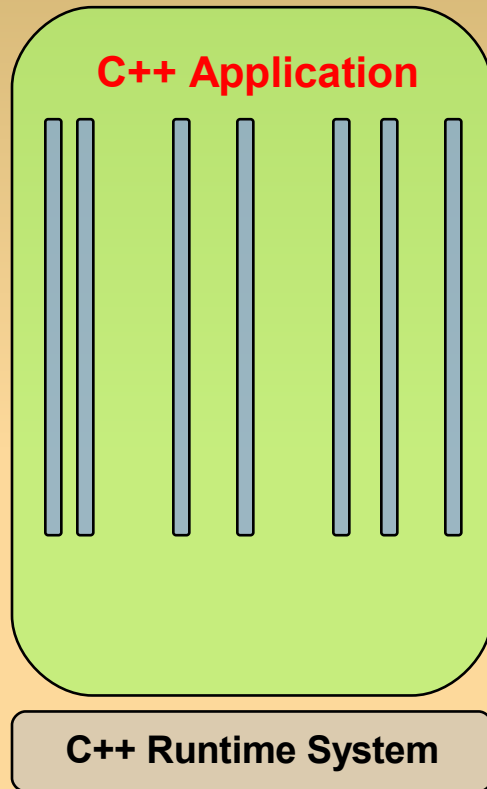
- Support from
 - Institutions (e.g., university curricula)
 - Industry (corporate endorsements, guidelines, adoption)
 - Government (research funding, procurement guidelines)
 - Organizations (e.g., JUG)
 - Grass roots (how many count Java as a “primary or favorite language”?)
 - Technology (vendor support, 3rd party involvement)

And finally . . .

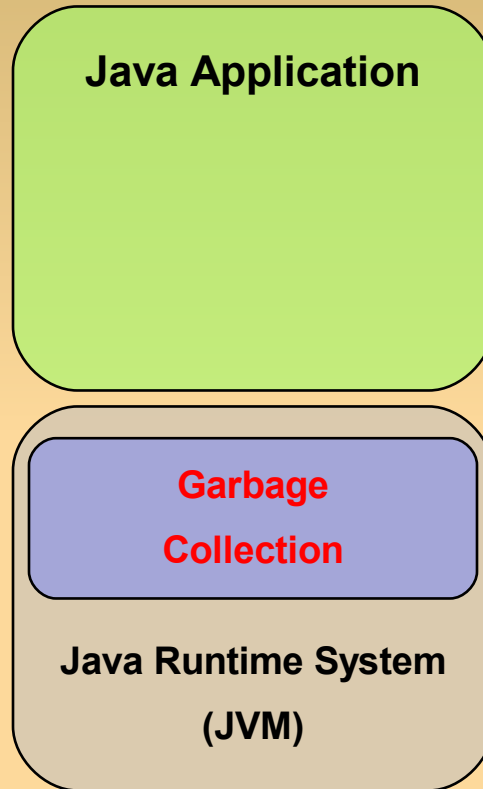
It really annoys the C and
C++ crowd!

So . . . Why real-time
Java?

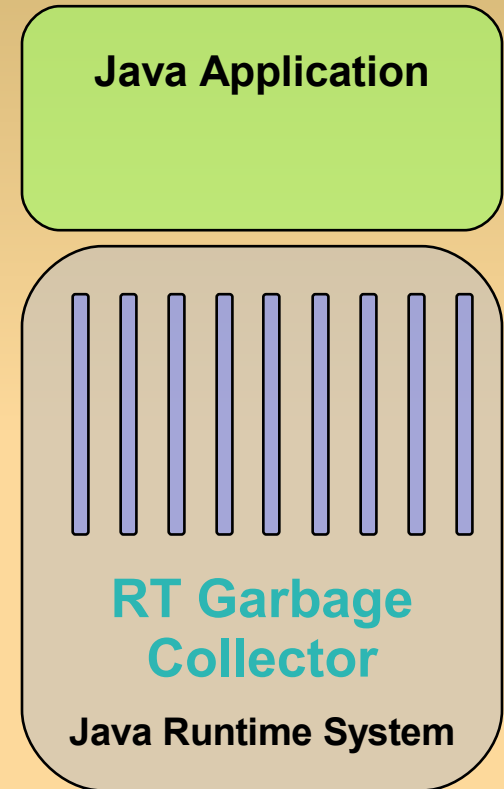
Real-time environments



Manual, Unsafe
Predictable



Automatic, Safe
Unpredictable



Automatic, Safe
Predictable

Real-world experience

- Using Java is commercially proven
 - Thousands of successfully deployed systems
 - Millions of hours of proven 5-9's and higher reliability
 - Ease of portability, maintainability, scalability
 - Improved developer productivity
- Performance comparable with C++ for the large, complex, dynamic applications.
- GC preemption latency as low as 100 μ s.

Guiding principles

- Applicability to particular Java environments
- Backward compatibility
- Write once, run anywhere
- Current practice vs. advanced features
- Predictable execution
- No syntactic extension
- Allow variation in implementation decisions

Extended semantics

- Scheduling
- Memory management
- Synchronization
- Asynchronous event handling
- Asynchronous transfer of control
- Asynchronous thread termination
- Physical memory access
- Exceptions

Conversion – step 1

Converting an existing Java program to real-time:

- Step 1:

```
Class ThreadUser {  
    Thread myThread = new Thread();  
    . . .  
}
```

changes to

```
Class RTThreadUser {  
    RealtimeThread myThread = new RealtimeThread();  
    . . .  
}
```

Conversion – step 2

Converting an existing Java program to real-time:

- Step 2:

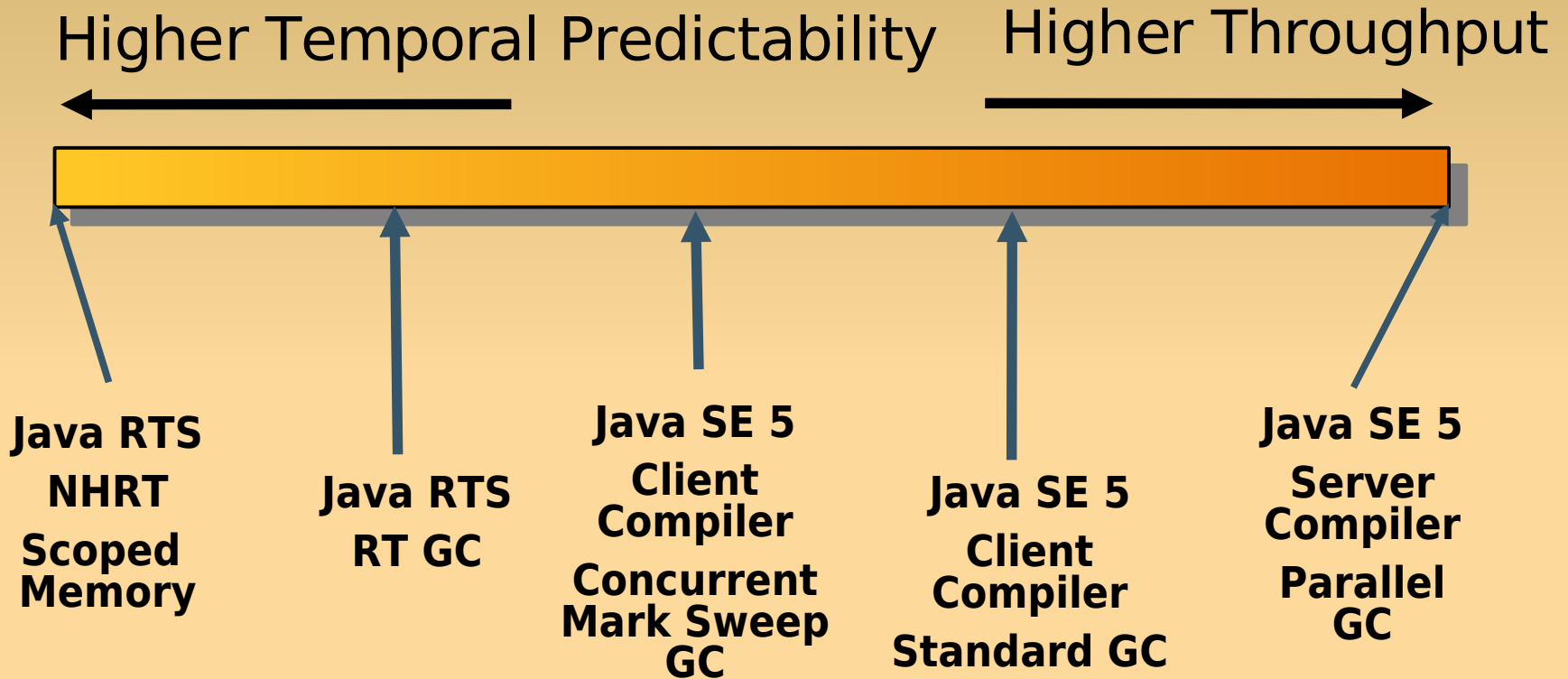
There is no step 2. You're done!

What's all this stuff about
garbage collection?

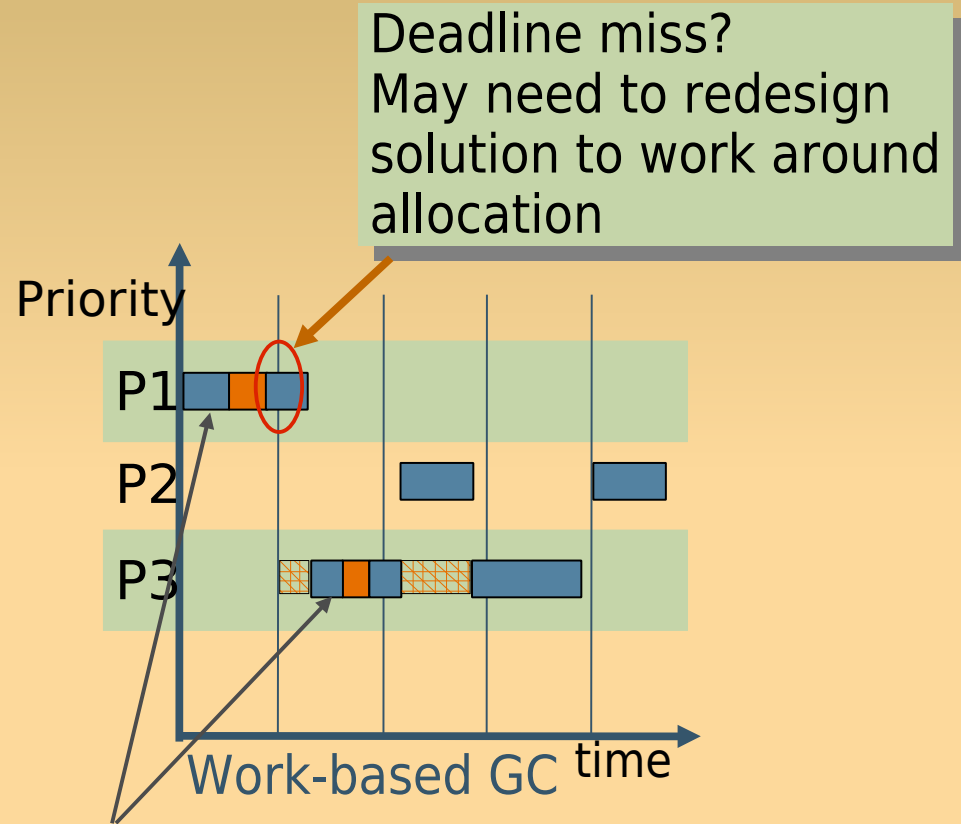
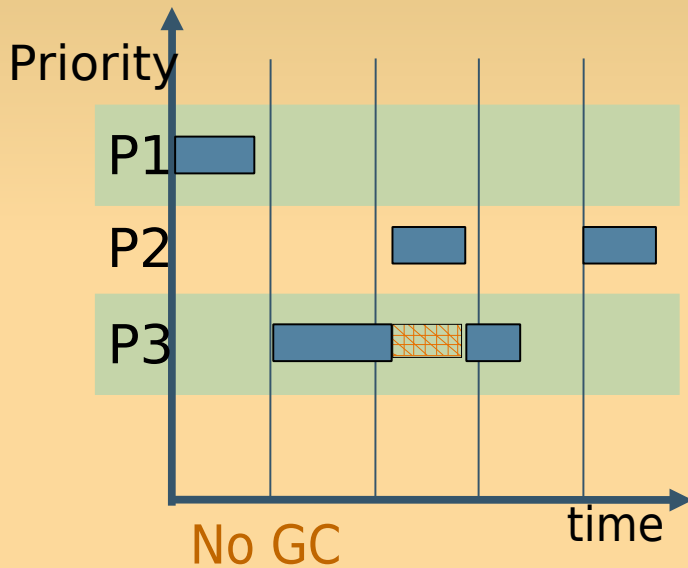
Garbage collection

- Typical GC language runtimes use visibility to determine when an object can no longer be referenced by application logic
- If no reference to an object exists in application logic then the application can no longer 'see' that object and it can be collected.

Performance Continuum



Work-based GC



Deadline miss?
May need to redesign
solution to work around
allocation

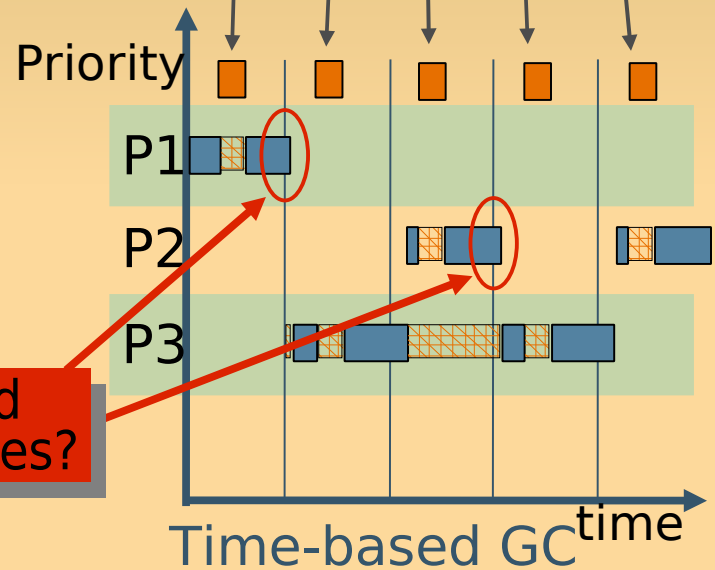
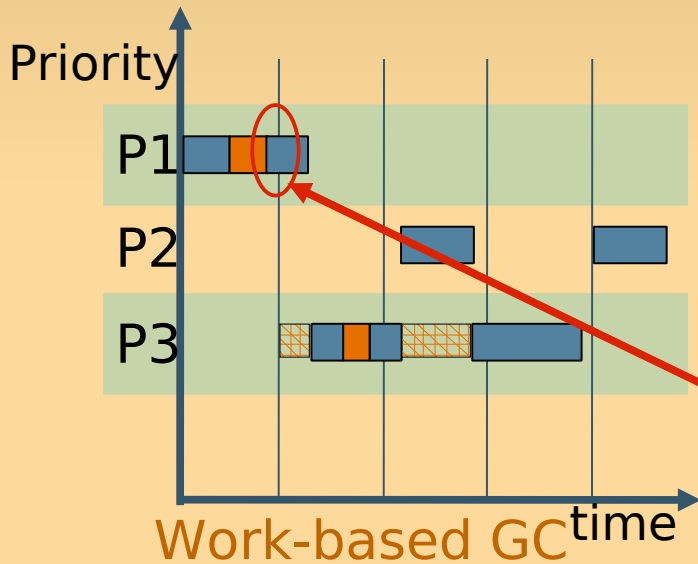
Allocations, followed by GC work

- Thread doing work
- ▨ Thread pause – higher priority interruption
- GC based on allocation

- Work-based garbage collector
- Lund (Henriksson) GC
 - High priority threads not interrupted but low priority threads could be starved
- SUN RTGC
 - Henriksson + policies
 - Base Policy: work-to-block
 - Tunable
 - Capable of running GC on a separate processor

Time Based GC

Garbage Collection allocation spread across each period and run at highest priority



Missed Deadlines?

- Thread doing work
- ▨ Thread pause – priority interruption
- GC active/running

IBM Metronome

- Time-based garbage collector
- Real-time garbage collection with 1ms worst case pause time and providing assured minimum application CPU utilization
- Garbage collection is scheduled as just another periodic real-time task
- Compiler integrated read-barrier
- Arraylets

How does it work?

RT System Model

Non real-time

- Regular Java threads
- Maximized throughput

Real-time code and regular app code can share memory, state, overall environment – something not possible in current RT solutions.

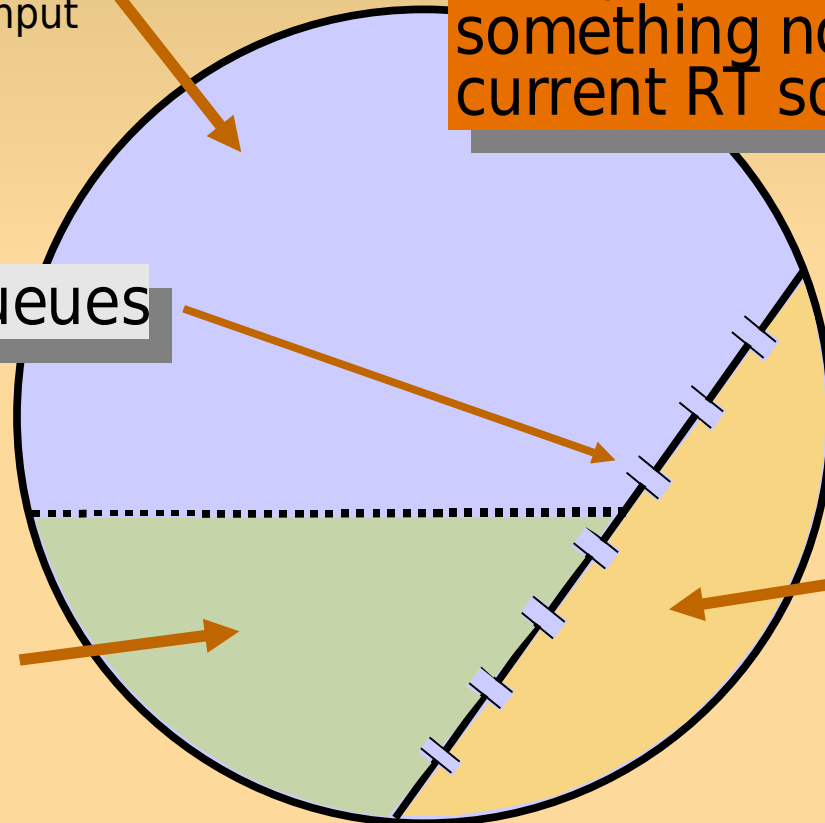
Data transfer queues

Soft real-time

- Realtime threads
- RT GC

Hard real-time

- NoHeapRealtime threads
- Bounded jitter



- Real-time scheduling theory is a fundamental principle
 - Development of programs which have temporal correctness requirements
 - Environment which enhances the ability of developers to write more portable real-time code
- The RTSJ approaches the solution to writing real-time code from a different direction than that of a typical RTOS

Schedulability

- Abstracted so as to allow application to imbue schedulable objects with values that characterize their expected behavior and requirements with respect to time
- Not just threads are schedulable
- Event handlers
 - No longer just execute whenever
 - Execute under control of the Scheduler

Feasibility set

- Initial set of instances of Schedulable along with their appropriate characteristics
- A set of things which the scheduler manages
- Also contains the feasibility algorithm
 - (RMA) Rate Monotonic Assignment
 - (DMA) Deadline Monotonic Algorithm

Feasibility algorithm

- Analyzes the set of schedulables
- Uses the values of the characteristics given to the instances of Schedulable
- Determines if all of the schedulables will meet their given deadlines

Scheduling subsystem

- Definition of schedulable objects (SO)
- Manage the execution eligibility of SO
- Perform feasibility analysis for set of SO
- Control admission of new SO
- Manage AsyncEventHandler, RealtimeThread
- Assign release characteristics to SO
- Assign execution eligibility values to SO
- Manage execution of groups of SO

Execution contexts

- `java.lang.Thread` (JLT)
 - Backward compatibility, non-timely
- `javax.realtime.RealtimeThread` (RTT)
 - Soft real-time or latency $>$ GC interference
 - Must follow assignment rules
- `javax.realtime.NoHeapRealtimeThread` (NHRT)
 - Hard real-time requirements
 - Must follow assignment rules and disallowed access to objects on regular heap

Event handler contexts

- `javax.realtime.AsyncEventHandler` (AEH)
 - Runs some time after an `AsyncEvent` (AE) occurs
 - Expectation is there may be thousands of events, with corresponding handlers
- `javax.realtime.BoundAsyncEventHandler` (BAEH)
 - An event handler which is permanently bound to a thread
 - Provides added timeliness

Memory management

- RTSJ can allocate objects in regions of memory other than the regular Java heap
 - Immortal Memory (IM)
 - Scoped Memory (SM)
- The lifetime of objects allocated in these memory regions is managed differently than those objects allocated on the heap

Non-heap memory

- Objects allocated in IM never die
- They can be referenced by all logic at any time without concern
- Objects allocated in an SM region live until no thread is executing in the scope to which the SM has been assigned
- The RTSJ also introduces the idea of a current memory area (CMA)
- All logic has an associated memory 'area'

Current memory area

- An object, through new, has a CMA
- CMA can be changed
- NHRTs must at all times have IM or an SM
- JLTs may have only the heap or the IM
- RTTs may have all three types

- Four Schedulables:
 - RealtimeThread (RTT)
 - NoHeapRealtimeThread (NHRT)
 - AsyncEventHandler (AEH)
 - BoundAsyncEventHandler (BAEH)
- All Schedulable are visible to and managed by the scheduler (e.g., PriorityScheduler)
 - PriorityScheduler is required
- Can be given a set of characteristics

Priority inversion

- Mars Pathfinder
 - July 4 1997
- High priority
 - bus message controller
- Medium priority
 - communications
- Low priority
 - meterological measurement

Release characteristics

- `javax.realtime.ReleaseParameters`
- Indicate to the scheduler whether the SO is periodic, sporadic or aperiodic
- Include cost per release, start time, handlers for missed deadlines or cost overruns
- Cost is the time taken for one release to execute to completion on a uniprocessor
- Cost value is used by the cost enforcement algorithms and feasibility tests

Scheduling releases

Let t_i be time of the i^{th} occurrence of task T

- T is periodic if

$$t_{i+1} - t_i = C_0, i = 1 \dots +, C_0 > 0$$

- T is sporadic if

$$t_{i+1} - t_i \Rightarrow C_1, i = 1 \dots +, C_1 > 0$$

- T is aperiodic if

$$t_{i+1} - t_i \Rightarrow 0 \dots +$$

Definitions

- Periodic releases are time triggered
- Sporadic releases are event triggered
 - Irregular but with a minimum interarrival time
- Aperiodic releases are event triggered
 - No minimum inter-arrival assumption

- Traditional priority is the dispatch time metric used by the (required) priority scheduler
- This scheduler may define a subclass of SchedulingParameters to communicate to the scheduler any necessary values
- The priority scheduler uses PriorityParameters which simply holds the priority of the SO

Cost enforcement

- Although optional, cost enforcement is probably the most important feature for portability and analysis in the scheduling subsystem
- Cost enforcement turns the exchange of information into a contract between the application and the system
- System will de-schedule any SO which exceeds its cost and reschedule for the start of the next release time

Feasibility (cont.)

- A sequence of releases of a set of tasks that ensures that all releases of all tasks meet their deadlines
 - When looking at systems with more than one task, determining feasibility more difficult.
 - In general solving this problem is NP-Hard
 - There are assumptions that can be made which allow a closed form computation to determine feasibility
 - The most widely known is the rate monotonic priority assignment algorithm (RMA)

Feasibility analysis

- During the initialization phase, instances of Schedulable, RTT, NHRT, AEH and BAEH are created.
- Each is added to the feasibility set
- Feasibility analysis is performed on existing set and newly added instance
- Result (boolean) is returned to application
- Application need know nothing about the idiosyncrasies of the particular RT-JVM

- By relying on the RTSJ feasibility analysis an application can obtain some measure of portability among different hardware, OS and RT-JVM platforms
- Estimation of cost value remains difficult
 - But that's not specific to real-time Java
- Applications need only create instances of `Schedulable`, give them appropriate characteristics, add them to the set and get the result of the feasibility analysis

Memory management

- Regions of memory outside the heap
 - Scoped memory, Immortal memory
 - Memory mapped to a physical address
 - Allow for the specification of maximum memory consumption and maximum allocation rates for individual real-time threads
 - Allow for external querying of the behavior of the GC
 - Allow for the altering of that behavior (maybe)

Immortal memory

- Objects allocated in IM never die
- They can be referenced by all logic at any time without concern
- Never garbage collected
 - So they will persist, even when there are no more references to them
- Visible from all threads
- An IM object can only contain references to other IM objects or heap objects

Scoped memory

- Regions of memory not subject to garbage collection
- NHRTs do not interact with the heap
- Cannot interfere with the garbage collector
- Cannot be interfered with by the garbage collector
- This is sufficient to protect hard real-time tasks from experiencing GC related jitter

- Provide predictable allocation and deallocation performance
- Insure that hard real-time threads need not block when memory is being reclaimed by the GC
- Hard real-time threads must co-exist with soft real time threads and ordinary Java threads
 - Both of these may experience garbage collection pauses

SM stacking

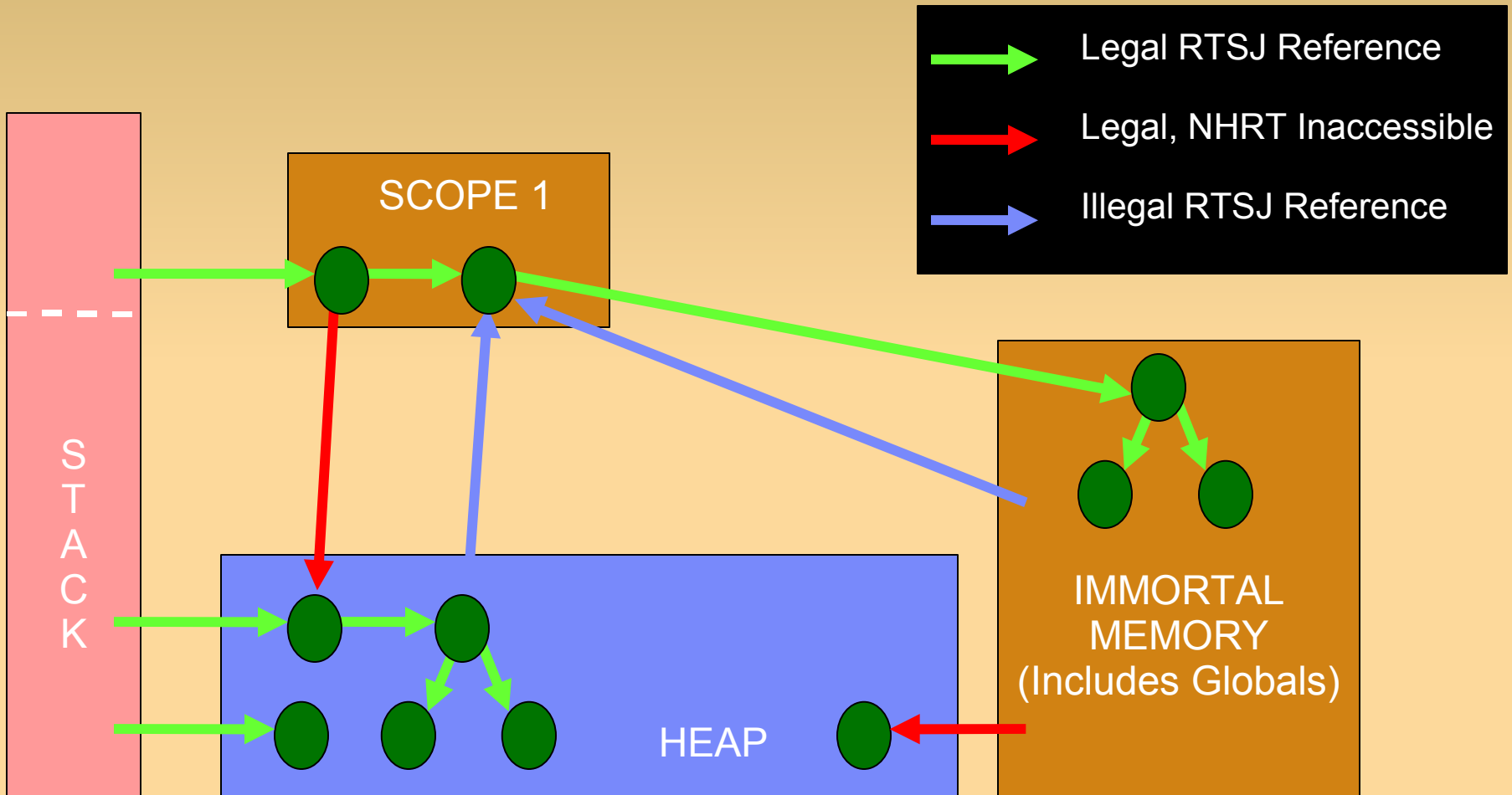
- Similar in principle to stack-based allocation
- Individual objects in a scope can not be deallocated – the entire scope is deallocated
- Each SM region can be entered by multiple threads which will allocate objects from the same pool and communicate by shared variables
- Physical memory used by a scope can be reused when all threads exit that scope

- Scopes can be nested
- When a thread enters another scope, the original scope becomes parent of the entered scope.
- A scope can be the parent of multiple scopes
- The scope hierarchy forms a tree rather than a simple stack

Cross references

- References are allowed across scopes
- Dangling references are disallowed
- Reference counting is used to ensure that objects are reclaimed only after all threads have finished the region
- It is not permitted for a region with a longer lifetime to hold a reference to an object allocated in an area with a shorter lifetime or a more deeply nested region

Scoped references



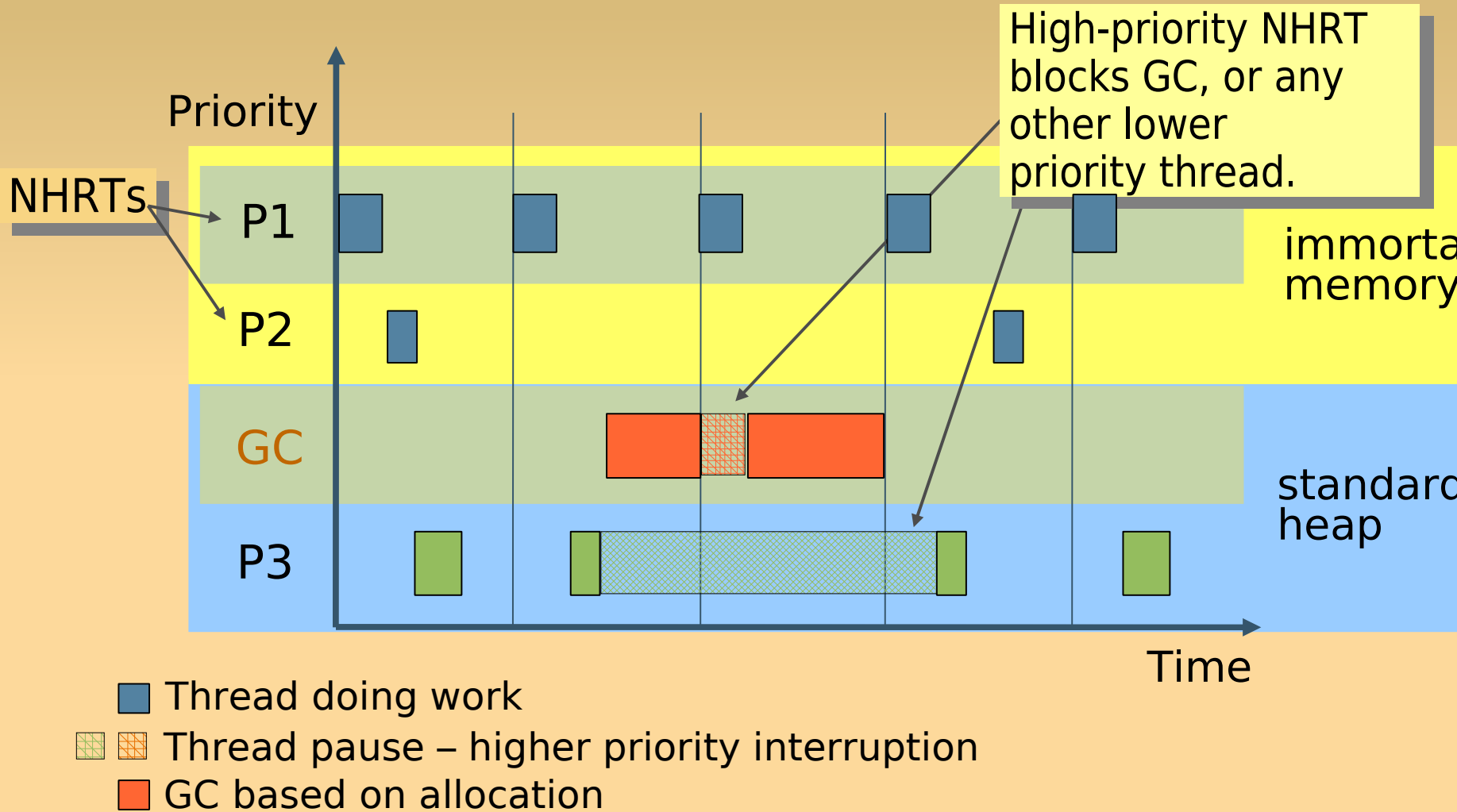
Memory class

- ScopedMemory is the abstract parent class of all scoped memory classes
- LTMemory
 - Allocation guaranteed to take linear time with size of object
- VTMemory
 - Allocation may vary
- Portal objects are used for communicating between threads

Scoped threads

- A RealtimeThread provides
 - getCurrentMemory()
 - getMemoryAreaStackDepth()
 - getOuterMemoryArea()
- A NoHeapRealtimeThread must be associated with scoped memory or immortal memory
 - It can never allocate an object on the heap and is forbidden from reading or writing any reference to an object that is heap allocated

No-Heap Threads



Active scope stack

- A thread's stack is subset of scope stack
- enter() method
 - Scope is pushed onto the thread's scope stack and reference count incremented
- executeInArea() method
 - Scope to be entered must already be a parent
 - Current scope is updated
- exit() method
 - Pops scope stack

Synchronization

- Strengthen Java synchronization by mandating monitor execution eligibility control (priority inversion control)
- Allow the application of priority ceiling emulation to individual objects
- Allow the setting of system default priority inversion algorithm
- Allow wait-free communication between real-time threads and regular Java threads

Wait-free queue

- When locked objects are accessed by JLT or NHRT instances, the implementation must provide alternate means for protected concurrent access.
 - WaitFreeDeque
 - WaitFreeReadQueue
 - WaitFreeWriteQueue
- No enhancement of synchronized is necessary to implement any known priority inversion algorithm

- Allow time up to nanosecond accuracy
 - HighResolutionTime
 - RelativeTime
 - RationalTime
- Allow distinctions between
 - Absolute points in time
 - Times relative to some starting point
- New construct: rational time
 - Allows expression of occurrences per some interval of relative time

Time-real

- A method of expressing time with sub-millisecond accuracy is an absolute minimum requirement
- Whatever precision underlying system supports
- Relative times allow for time-based metaphors such as deadline-based periodic scheduling where cost of task is expressed as relative time
- Deadlines are usually represented as times relative to the beginning of the period

- Allow creation of a timer whose expiration is either periodic or set to occur at a particular time
- Trigger some behavior to occur on expiration of a timer, using the asynchronous event mechanism
 - Clock
 - PeriodicTimer
 - OneShotTimer

Using timers

- OneShotTimer is used for timeout behavior
 - Triggering of the timer is guaranteed
- Problem for periodic timers: importance of periodic triggering outweighs precision of start time
 - A periodic task may be represented as a frequency and the period remain synchronized
 - In these cases a relatively simple correction can be enforced by the implementation

Asynchrony

- Provide mechanisms that bind program logic to the occurrence of internal or external events
- Provide mechanisms that allow the asynchronous transfer of control
- Provide mechanisms that allow the asynchronous termination of threads

Asynchronous event handling

- AsyncEvent (non abstract)
 - Expressed as invocation of fire().
- AsyncEventHandler (abstract)
- BoundAsyncEventHandler (abstract)
- Handlers act like threads
 - Scheduled on the occurrence of an event
 - Expressed as an invocation of run()
- The interrupt() method in a JLT provides rudimentary asynchronous communications

- Dateless
- fire() method passes no data to the handler
 - Intentional – simpler
- If data needs to be associated with an event the application can set up a buffer to do so
 - And worry about buffer overflows as necessary
- Can be fired across threads
 - But be careful!

Event handlers

- In some real-time systems there may be a large number of potential events and handlers (thousands) although at any time only a small number will be used
- It would not be appropriate to dedicate a thread to each event handler
- The programmer can specify that a handler is bound to a specific thread or not bound to a specific thread

Exceptions

- Additional exception classes are required by the additional behavior of `javax.realtime.*`.
- Provide for the ability to asynchronously transfer the control of program logic
- `AsynchronouslyInterruptedException`
 - Only well-defined code blocks are subject to having their control asynchronously transferred

What about safety-critical?

Safety critical feasibility

- Ben Brosgol (ADACore) and Andy Wellings:
 - What is “safety critical” software?
 - Failure can cause loss of human life or have other catastrophic consequences
 - Why Java?
 - Addresses insecurities of C / C++
 - Run-time checks for array index out of bounds, etc.
 - Automatic garbage collection (but this interferes with predictability, analyzability)

Safety considered

- The RTSJ was never intended for safety-critical applications
- But it does address some of the problems with full Java
 - Garbage collection latency
 - Underspecified semantics for thread scheduling
 - Priority inversion management

Safety goals

- Work in progress to define a safety-critical real-time Java profile “based on” the RTSJ
- Started in July 2003 - The Open Group’s Real-Time Embedded Systems Forum
- 7/06: accepted as JSR-302
- Goal is to create a capability, based on RTSJ, that can be certified, e.g. DO-178b
- <http://jcp.org/en/jsr/detail?id=302>

Items to consider

- Static initialization – elimination of circularities and race conditions
- All class loading done at initialization time – as opposed to “mission” time
- Byte code verifier to check for constraint problems
- No garbage collection whatsoever

Overall objectives

- Satisfy DO-178b Level A certification
- Performance, memory, latency of C, C++
- Portability and scalability of SE
- Standardize low-level services, interrupts, I/O
- Efficient, safe stack memory with compile-time enforcement of referential integrity
- Possible integration with traditional Java
- Same toolset as SE

Two proposals

- Ravenscar-Java
 - Kwon, Wellings, King – York University
- Scalable-Java
 - Nilsen - Aonix
- Platform hierarchy – SC most restrictive
- Subset of RTSJ
- Garbage collection replaced by SM
- Java 5.0 meta-data annotations

Use hard real-time guidelines

- In general, hard real-time guidelines are appropriate for SC development
- However, some practices must be restricted
- Following slides refer to Scaleable Java proposal

Use only 28 priority levels

- RTSJ states that 28 levels must be provided
- SC should restrict itself to 28
- Vendors can support this range as a standard SC platform

NoHeapRealtimeThread

- Use only instances of COMPLIANT-mode NoHeapRealtimeThread
- Sharing of data and control between native code and SC code should be avoided
- Native code cannot be scrutinized by the byte-code verifier

Subscript Checking

- Prohibit use of `@omitSubscriptChecking`
- Turning off subscript checking is discouraged, even though static analysis has been done
- Benefit is prevention of an error from propagating to other components

Scope checking

- Prohibit use of `@OmitScopeChecking`
- Again, static analysis may well have determined that this is unlikely
- Prevent any attempt to make assignments that would violate nested scoping rules

Checked scope links

- Cannot be certified safe by static theorem provers
- Risk that program may abort with a run-time exception
- Can be lifted if developers can provide absolute proof that run-time exceptions cannot be thrown

Static analysis

- Use of `@StaticAnalyzable` is required
- All relevant modes of analysis to be true for
 - `enforce_time_analysis`
 - `enforce_memory_analysis`
 - `enforce_non_blocking`

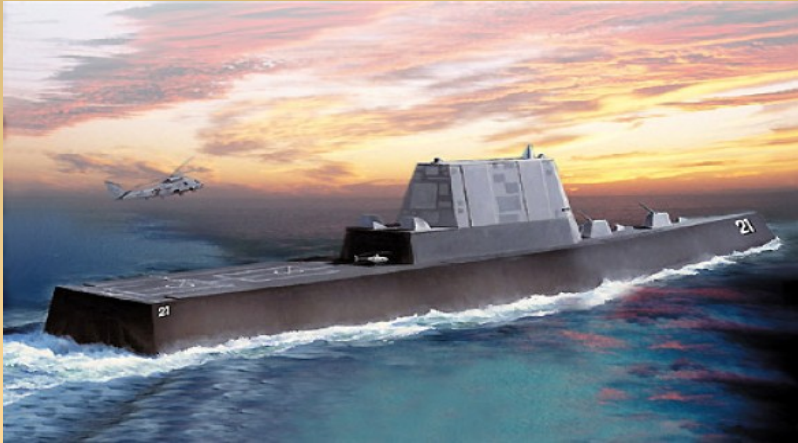
Class loading

- Prohibit dynamic class loading
- Hard real-time can allow this
- SC should avoid it

No priority inheritance

- Priority inheritance is difficult to certify
- Use Priority ceiling emulation
- Introduces synchronization overhead delays proportional to complexity of the application

Does this stuff actually
work?



Total Ship Computing Environment Infrastructure (TSCEI)

an integrated suite of standardized, open architecture hardware, operating system, middleware and infrastructure services.

- Based upon OMG RT CORBA specification
- Prism Technologies provides the RT Java ORB
- Enables java applications to become part of TSCEI
- Both RTSJ compliant JVMs were competed
 - IBM Websphere Real-time
 - Sun Java RTS

ScanEagle UAV



- Purdue and Boeing enhanced ScanEagle
 - OVM – Open Virtual Machine?
 - BSD License
 - ovmj.org
 - Demonstration:
 - Added “Payload board” with RTSJ app running on OVM
 - Autonomous route planning and navigation
 - battle damage assessment
 - "offset stare" from a no-fly zone during missile launches.

N-UCAS / X45 UAV

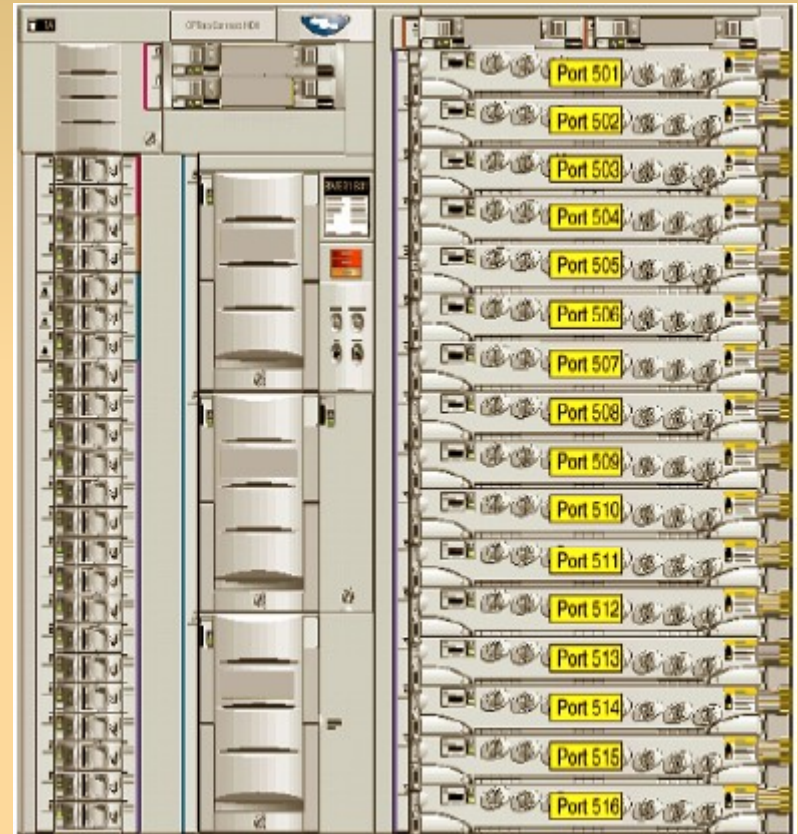
- Mission Planning System written in Java by Boeing and BAE for X-45A
- 9/05 Aonix selected as partner for X-45C
- X-45C OS is VxWorks 6.x
- PERC VM



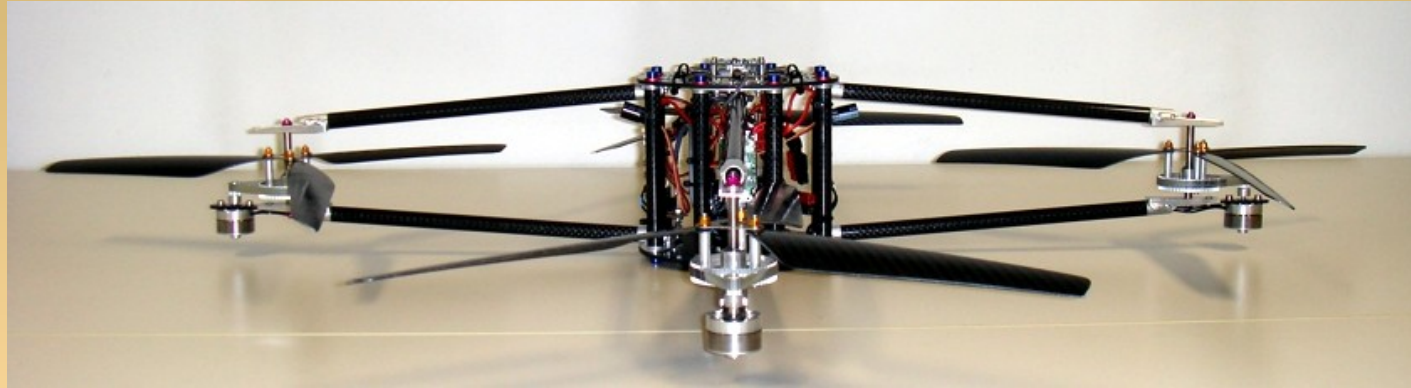
- PERC VM
 - PERC is a clean-room virtual machine created for embedded and real-time systems requiring J2SE.
 - Customized Toolset
 - Supports VxWorks 6.x.
 - Not RTSJ compliant

Communications

- Nortel Networks
- SONET Fiber Switch
- Java APP runs on line cards using VM
- VxWorks w/ PPC core
- 1 million lines of java code
- SONET Protocol has 40ms timing constraints



Quad-Rotor autonomous helicopter



- Runs with IBM Metronome GC
- Rate Requirements
 - gyroscopes, accelerometers: up to 350Hz
 - ultrasonic sensors: ~ 12 Hz
 - motors: ~ 100 Hz
- IBM's real-time GC (Metronome) has a worst case latency of $700\mu\text{s}$

Any drawbacks?

Low latency tasks

- There are limits to the minimum latency for garbage collecting
 - Some work, after all, needs to get done in every pause
 - Limit expected to be greater than 100 microseconds on current hardware
 - Some threads ultimately need to run at higher priority than the garbage collector

NHRT problems

- Both reads and writes can fail
- Can be costly in performance (checking overhead)
- Presents problems for modularity (scoping rules)
- Architecture storage leak (immortal memory)

All done?

Are we done yet?

Toys to play with

The reference implementation is available from
<http://www.timesys.com/java/>

Ovm Java Virtual Machine

<http://www.cs.purdue.edu/homes/jv/soft/ovm/>

Web resources

<http://www.rtsj.org>

<http://www.alphaworks.ibm.com/topics/realtimejava>

<http://java.sun.com/javase/technologies/realtime.jsp>

Special Thanks and Credit



Michael Genewich
Michael.genewich@sun.com
Tim Bardwell
Tim.Bardwell@sun.com



Brad Burns
bradbur@us.ibm.com
Mike Fulton
fultonm@ca.ibm.com

Other Sources

Prism Technologies
Aonix

Purdue University
Universität Salzburg