Memory Management for Real-time Java

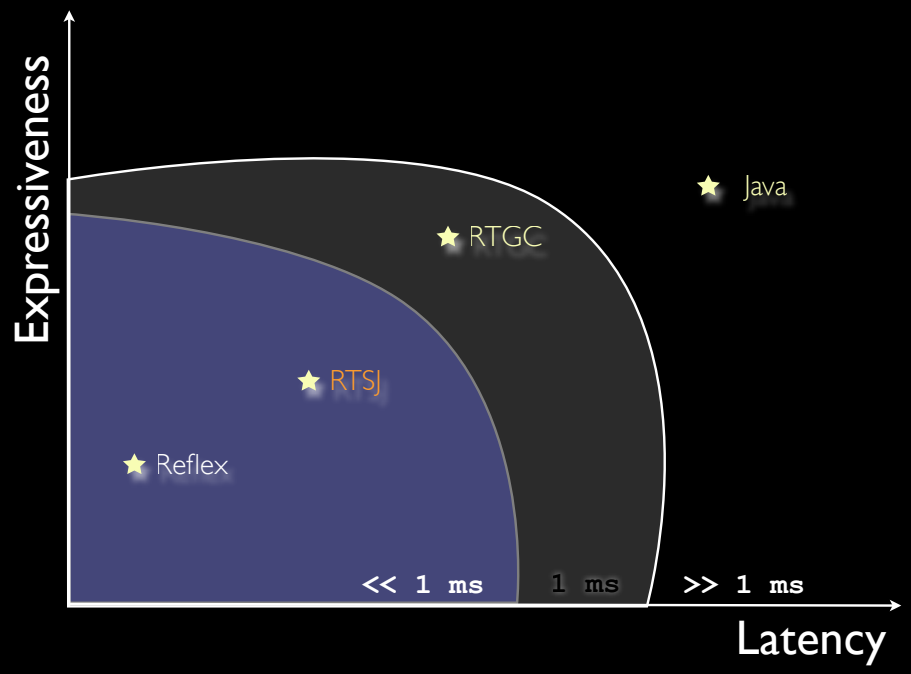
Jan Vitek

(S3) PURDUE UNIVERSITY IBM

Overview

- Memory management & Java
- Scoped Memory
- Real-time Collectors:
  - AICAS Jamaica, Sun McKinak, IBM Metronome
- Lock-free Collectors
- Conclusions

Design space



Manual Memory Management

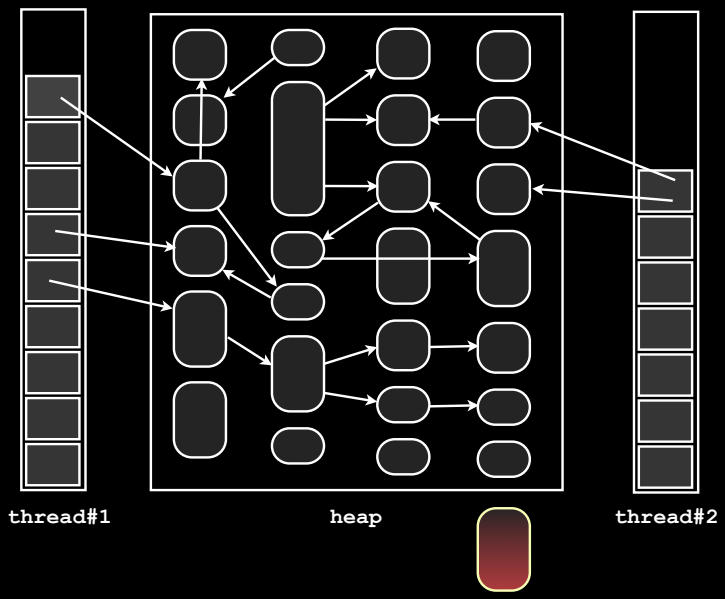
- Memory management is the source of many errors (dangling pointers, memory leaks) in traditional software systems due to the non-local nature of the allocation/deallocation protocol
- One can sometimes prevent such errors by avoiding allocation, at the cost of over provisioning, but not in general.
- Programmers often use custom 'allocators' in conjunction with object pools
- *... but rich data structures and concurrency make manual management hard*
- *... requires synchronization on multicores*
- *... must deal with fragmentation*

Garbage Collection

Garbage Collection

- A Garbage Collector (GC) is an algorithm that automatically finds unused objects in the memory of an application and prepares them for reuse
- GC frees programmers from worrying about the exact lifetime of objects and ensures that the heap will not be corrupted by access to previously freed data
- *... but introduces pauses that may be* ***O****(heap) and can increase the memory required.*
  Moreover, pauses occur at unpredictable times, especially in concurrent programs

Garbage Collection

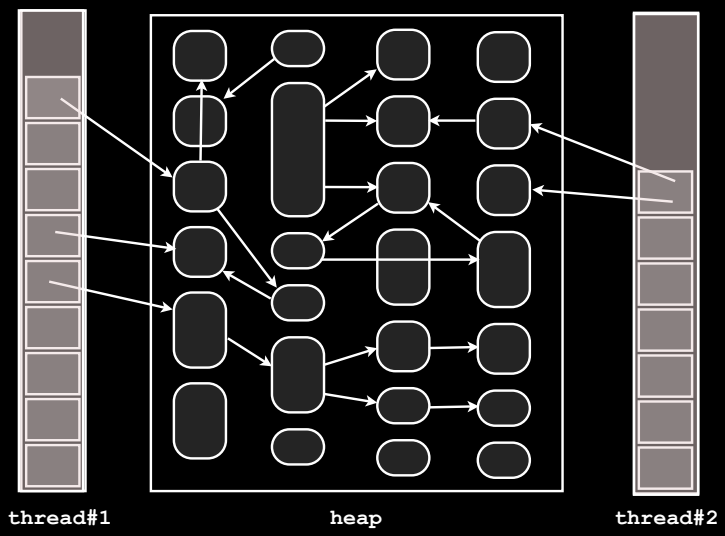


Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

ECS 2009

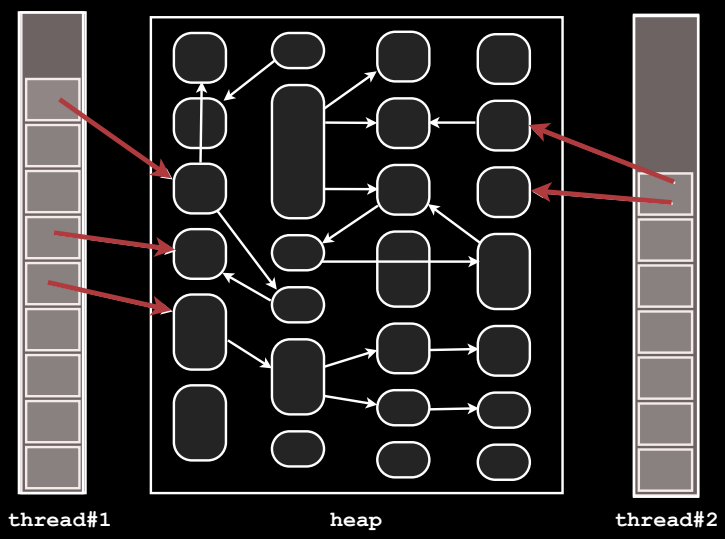
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

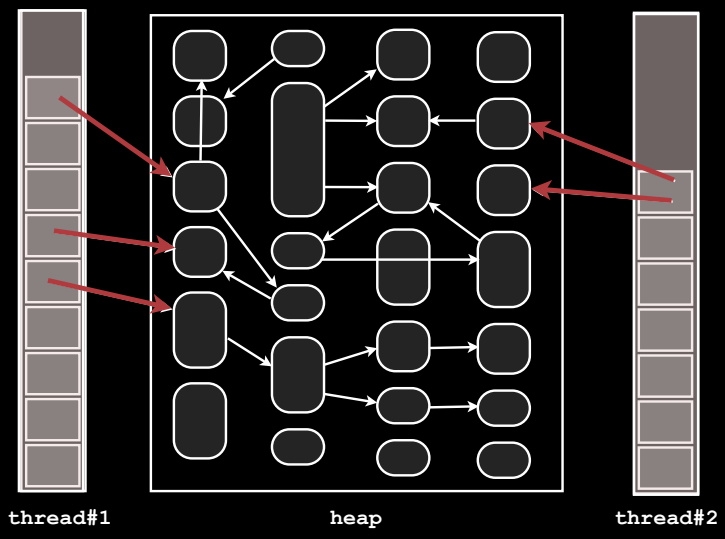
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

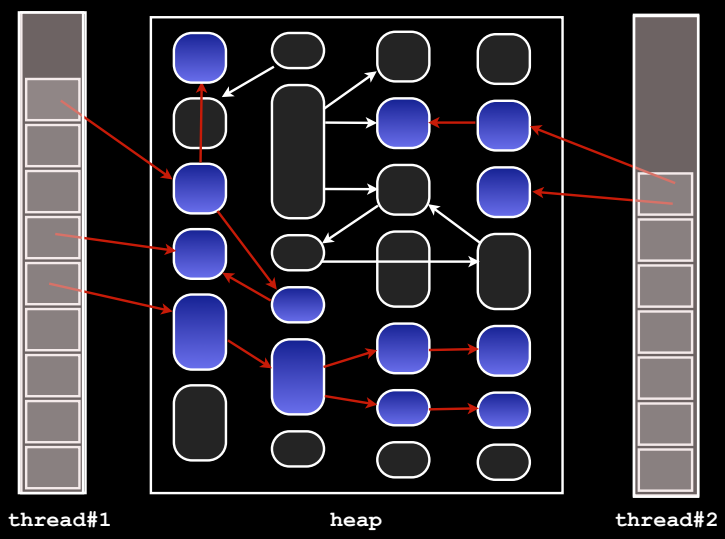
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

Garbage Collection

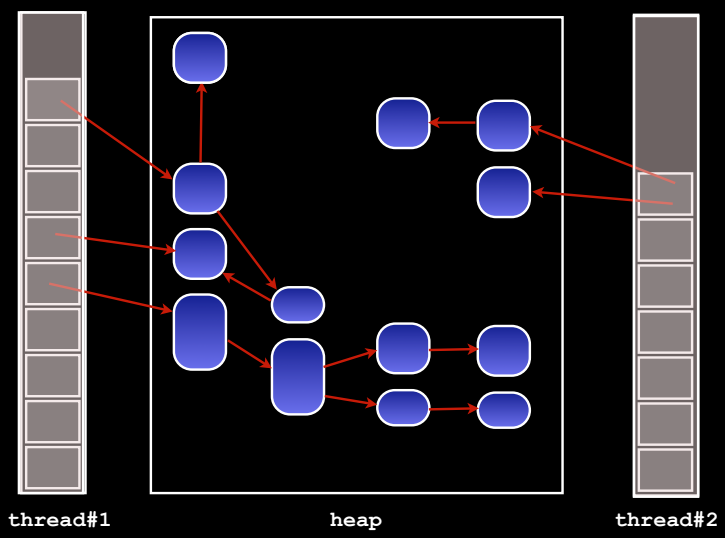


Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

ECS 2009

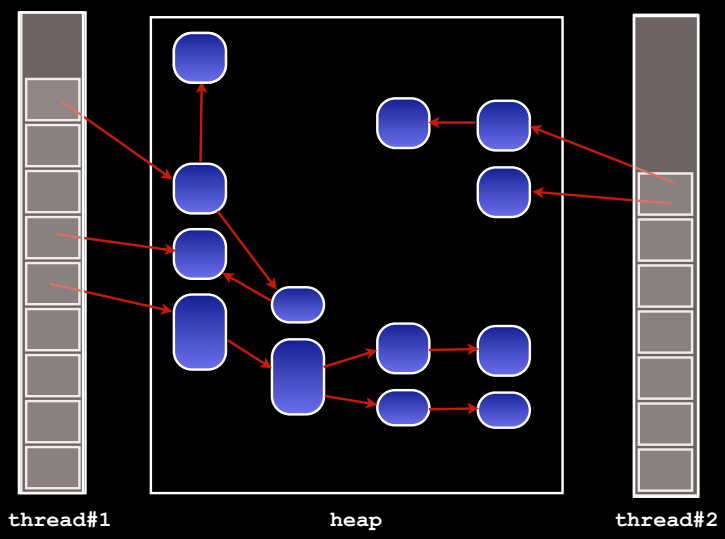
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

Garbage Collection

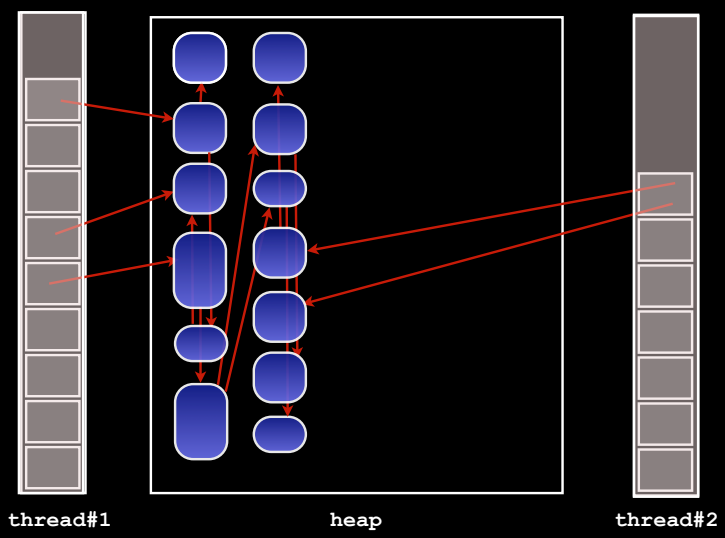


Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

ECS 2009

Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

GC is Easy

- If responsiveness is not an issue,
  the GC can complete in one long pause under the assumption that there is no interleaved application activity
- Marking is easy
  if the graph does not change while you are searching it.
- Copying/compacting objects and fixing up the heap is easy
  if the application is prevented from accessing the heap

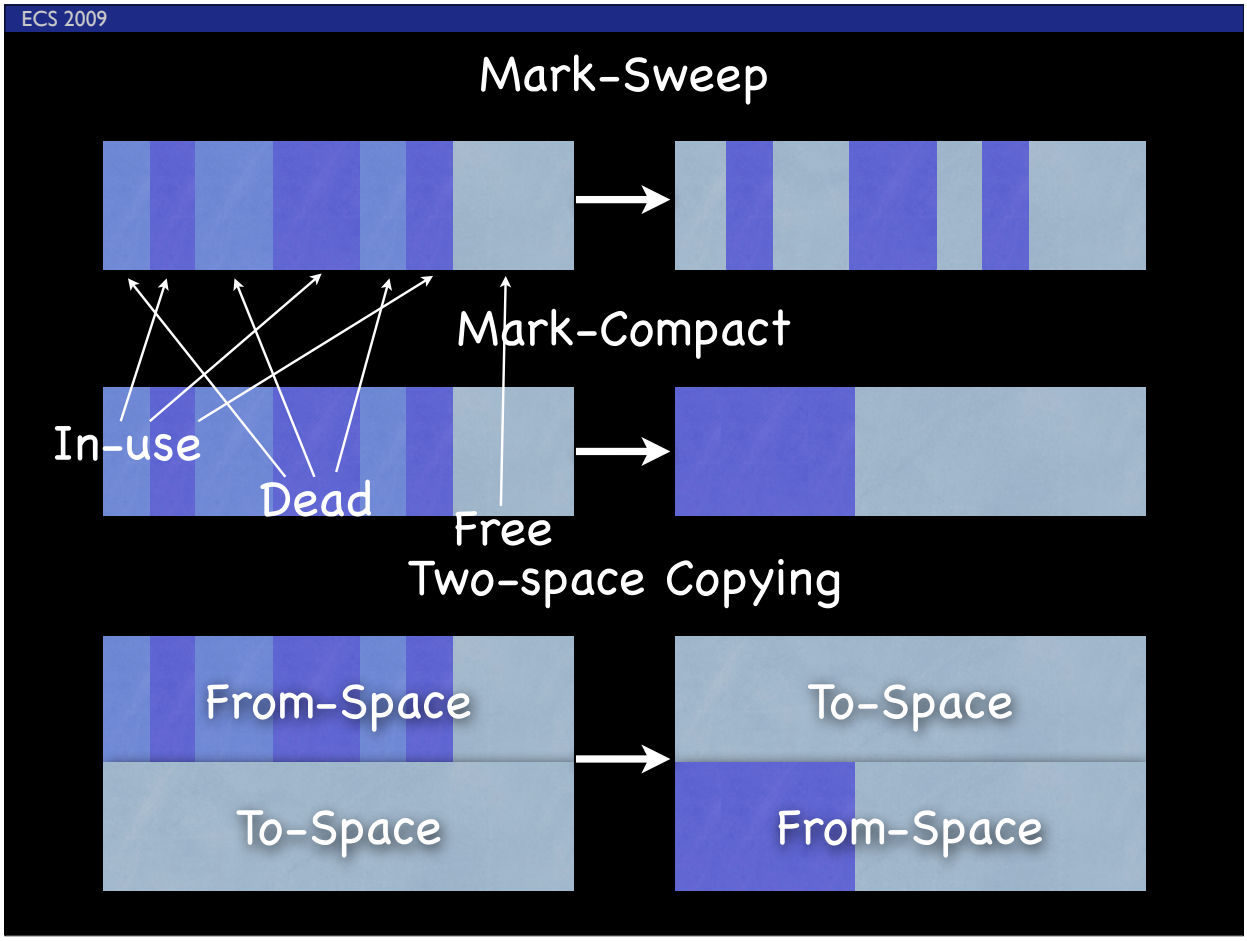
Real-time Garbage Collection

Real-time GC

- A Real-time GC must provide time and space predictability
  - provide a performance model that can be used to guarantee that programs do not run out of memory or experience pauses that violate their timing constraints
- A Real-time-GC must support defragmentation of the heap if it is to be used with long-lived applications
- Multi-processor support is unavoidable
- Throughput should not degrade overly

Main Collector Types

- Reference counting
  - keep count of incoming references. Requires auxiliary GC to reclaim cycles.
- Mark-Sweep
  - one phase to mark used objects, another to sweep unused space. No copying is performed.
- Mark-Compact
  - one phase to mark used objects, another to copy all used objects to one side of the heap.
- Copying
  - one phase to simultaneously mark and evacuate all used objects; the space that previously held all objects is then completely free.



RTGC: state of the art

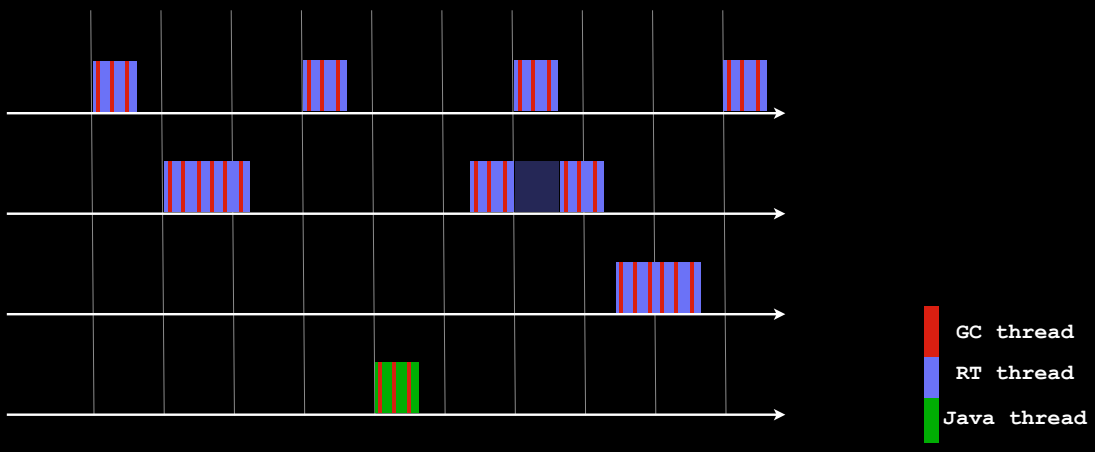
- Sort-of Real-time GCs:
  - Too many to list, Baker (1978) was first
- Really Real-time GCs:
  - JamaicaVM (1999) - *work-based*
  - Henriksson (1998) - *slack-based*
  - Metronome (2003) - *time-based*

Work-based RTGC

Baker'78, Siebert'99

(ς^3) PURDUE UNIVERSITY Fiji Systems LLC IBM

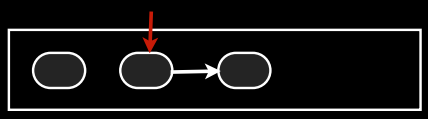
Work-based scheduling

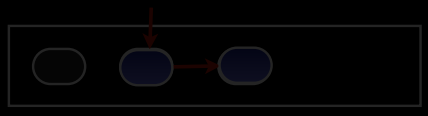


Baker

- First attempt at RTGC by Baker '78
- For LISP where all objects are the same, small, size
- Copying collector that instruments the mutator to maintain:
  - To-space invariant:
    - mutator uses to-space, never sees from-space, objects are copied on first read
  - Steady state:
    - collector keeps up with mutator allocations, collection work performed on each allocation

Incrementalizing marking





Collector marks object

Application updates reference field

Compiler inserted write barrier marks object

Dissecting Baker

- The chief problem with Baker is the to-space invariant
  - Reads are frequent, a heavy read barrier leads to poor performance
  - Variable sized objects in Java make costs harder to bound
- What RT programmers fear:
  - a burst of copying reads, causing substantial slow-downs

Siebert

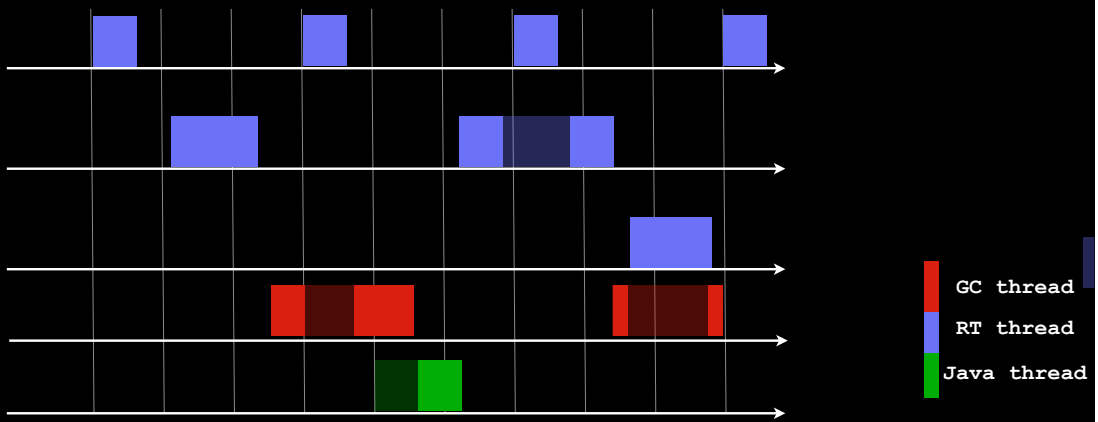
- Modern work-based collector in the JamaicaVM:
  - Mark-Sweep, no copying
  - All objects are same size, 32 bytes
    (large objects⇒ lists, arrays⇒ tries) Stack is logically an object
  - Root scanning is fully incremental
  - Write barrier to mark objects (including the stack)
  - Allocation triggers a bounded amount of collector work

Slack-based RTGC

Henriksson'98

(ς³) PURDUE UNIVERSITY Fiji Systems LLC IBM

Slack-based GC Scheduling

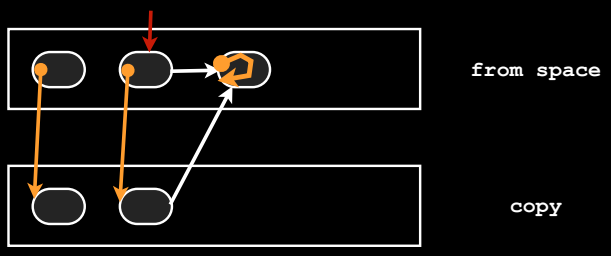


Slack-based GC

- Henriksson takes a different approach to real-time guarantees:
  - tries to guarantee real-time tasks never experience GC interference
- How is this done?
  - RT tasks can always preempt GC
  - Objects have normal representation.
  - Read & write barriers are used to ensure marking/copying soundness.
  - Copying collector, with roll backs when the mutator preempts GC

Incrementalizing copying/compaction

- Forwarding pointers are used to refer to the current version of the object.
- Every access must start with a derefence



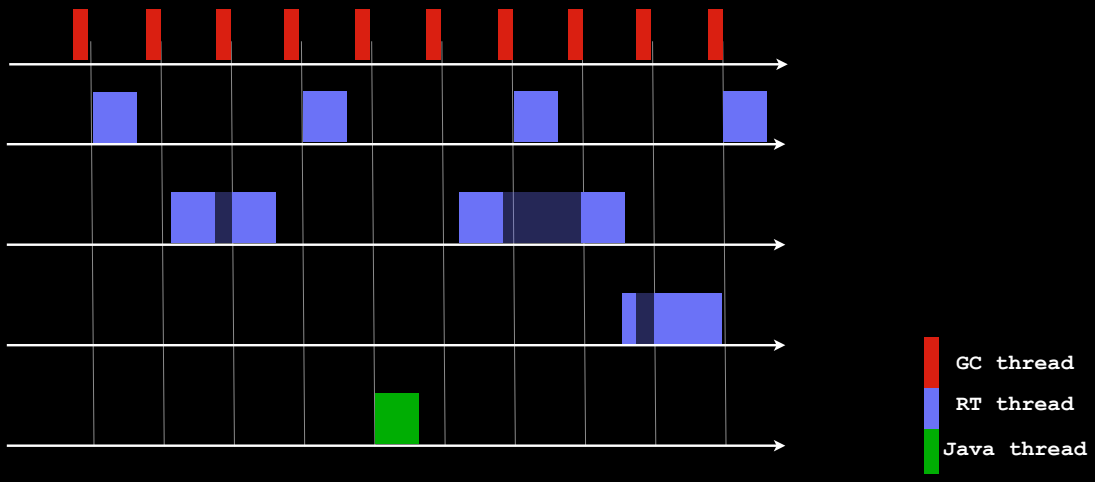
Dissecting Henriksson

- Pros:
  - Gives the programmer full control over the GC schedule
  - SUN uses a variant of this collector in their RTSJVM
- Cons:
  - Read barrier are expensive
  - Mutator induced copy aborts make collector progress hard to ensure
  - Programmer must know allocation rate, and to budget for GC work
  - Priority inversion can lead to GC interference

Time-based RTGC

Metronome'03

Time-based GC Scheduling



Time-based GCs

- Metronome runs periodically for a bounded amount of time
  - Amount of collector interference is easy to understand
  - Progress is easier to guarantee than in Henriksson
- A Mark-Sweep-and-sometimes-Copy collector:
  - Write barrier to ensure marking soundness
  - Light read barrier to ensure copying soundness
  - Almost-empty pages evacuated in response to fragmentation
  - Arrays are split into tries with page-sized leaves
  - Collector increments lower-bounded by time to copy a page-size objects

Suming up

Siebert

Local ref. assignment	mark(b) a = b
Primitive heap load	a = b->f
Reference heap load	a = b->f mark(a)
Primitive heap store	a->f = b
Reference heap store	mark(b) a->f = b

Henriksson

Local ref. assignment	a = b
Primitive heap load	a = b->forward->f
Reference heap load	a = b->forward->f mark(a)
Primitive heap store	a->forward->f = b
Reference heap store	mark(b) a->forward->f = b->forward

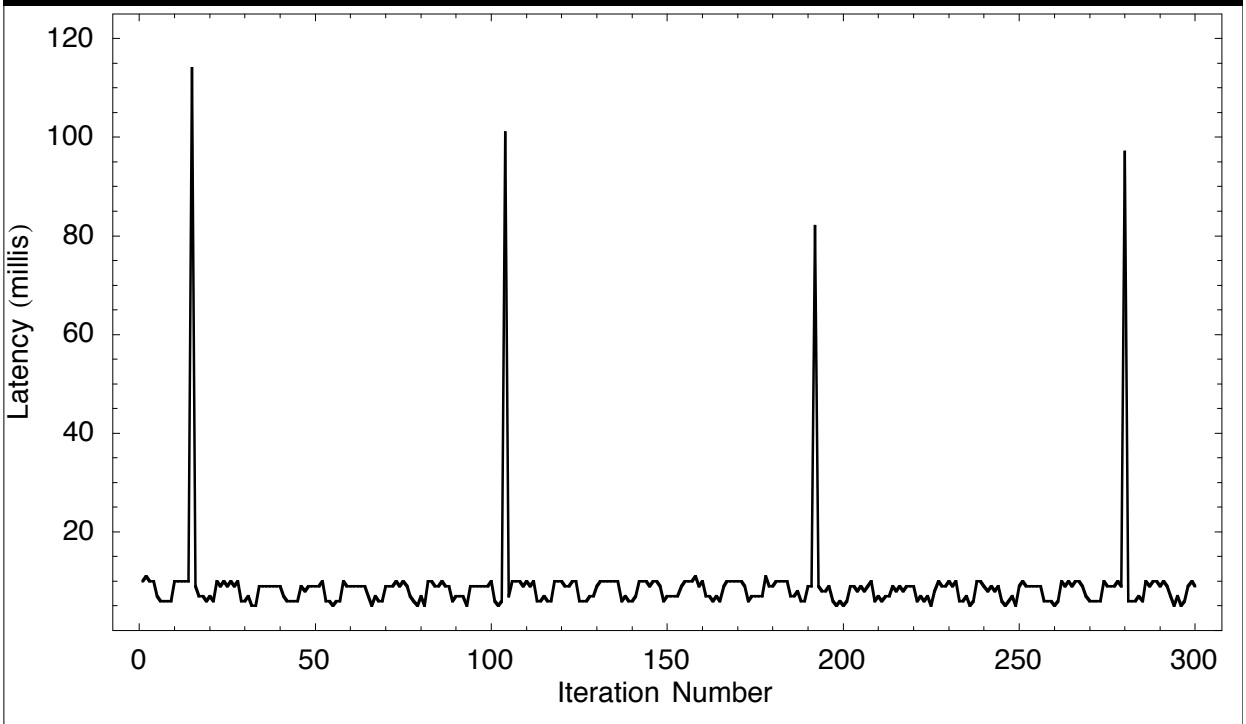
Metronome

Local ref. assignment	a = b
Primitive heap load	a = b->forward->f
Reference heap load	a = b->forward->f
Primitive heap store	a->forward->f = b
Reference heap store	mark(a->forward->f) a ->forward->f = b->forward

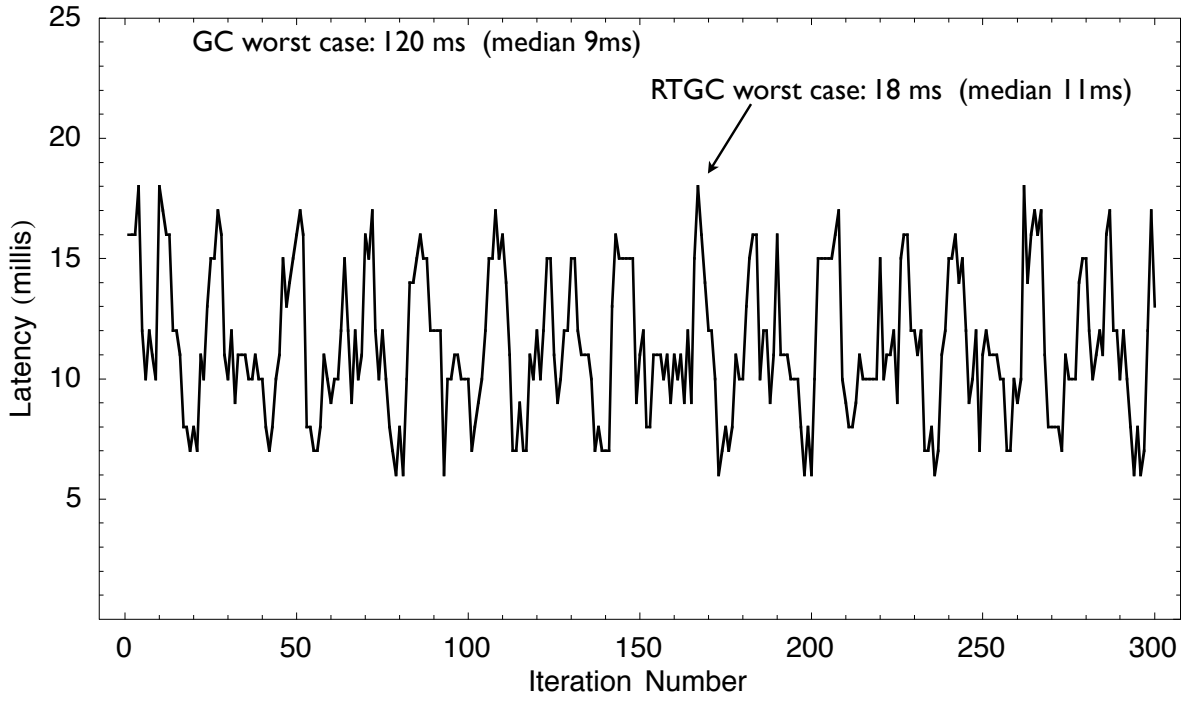
Collision Detector

- Experiment:
  - Pentium IV 1600 MHz, 512 MB RAM, Linux 2.6.14, GCC 3.4.4
  - Application: Real-time Java collision detector (20Hz)
  - Virtual machine: Ovm

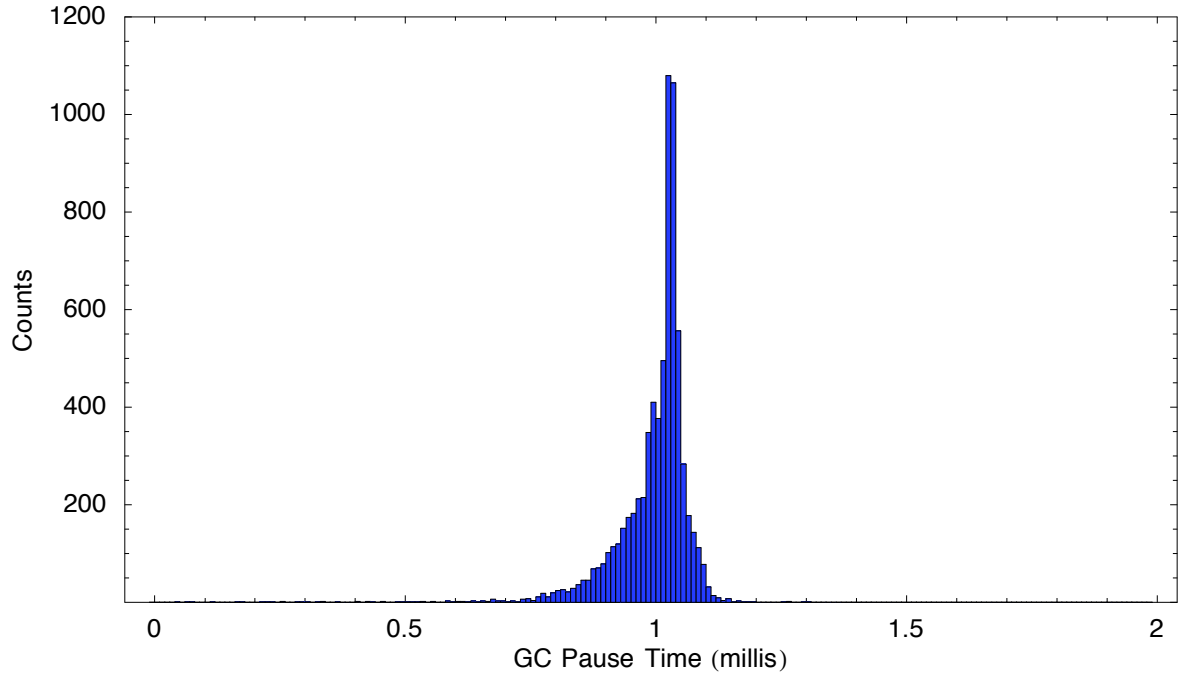
CD with GC



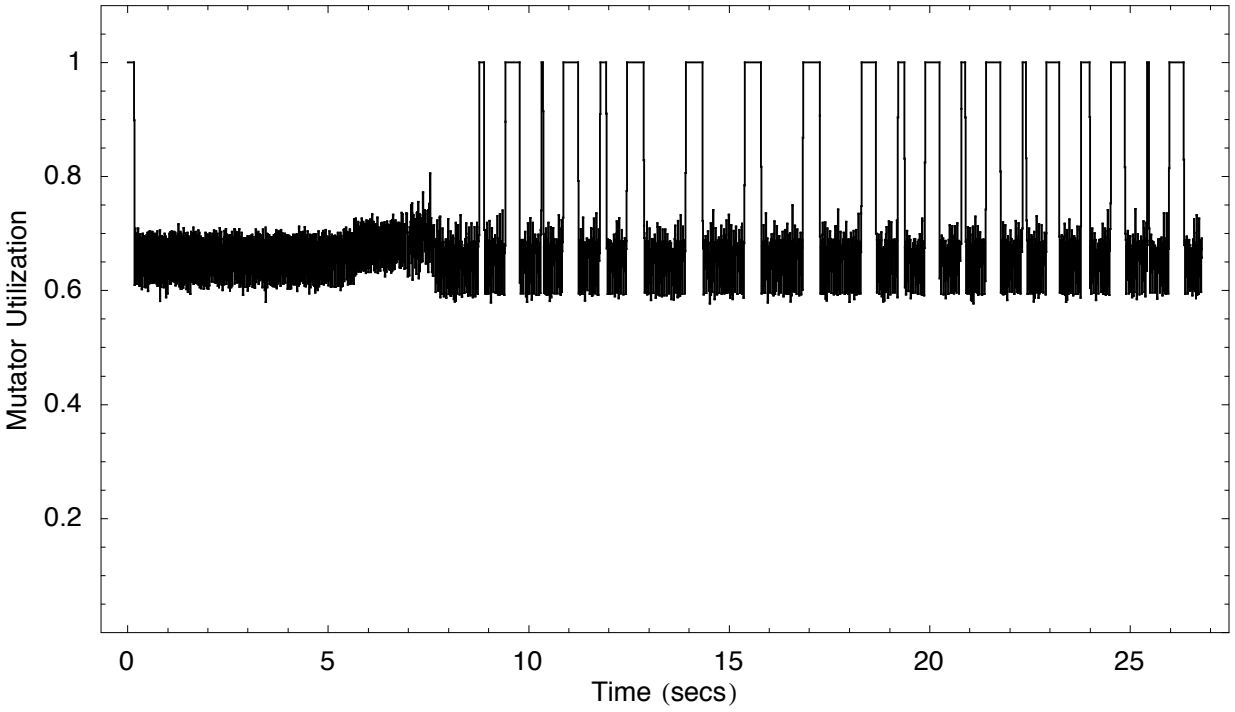
CD with RTGC



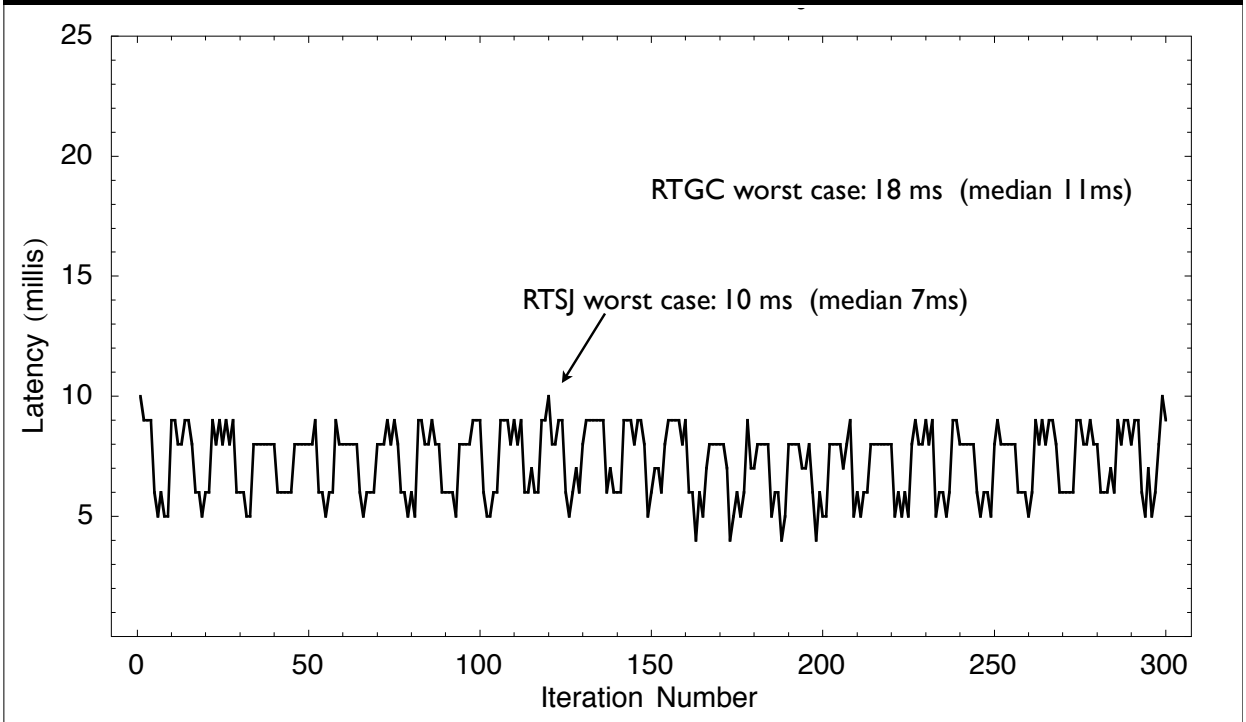
RTGC Pause time distribution



Utilization

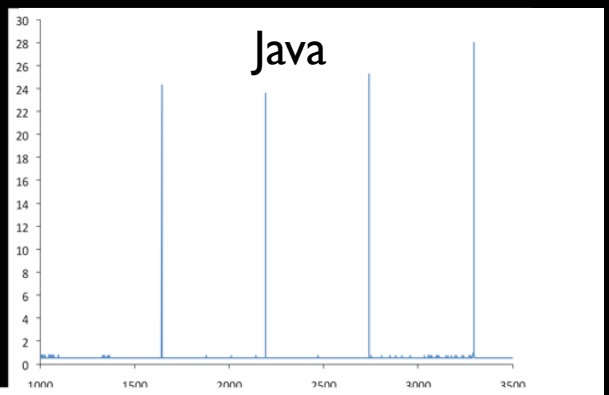


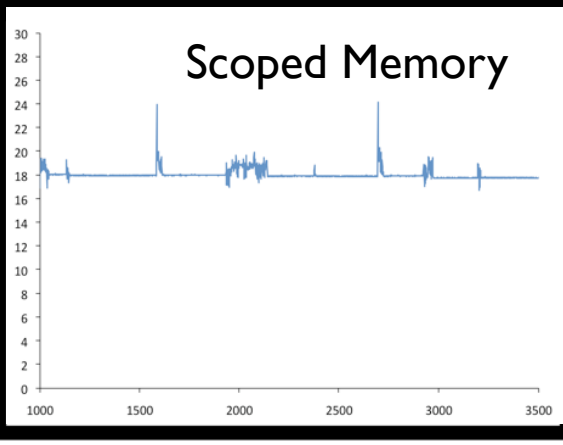
CD with RTSJ

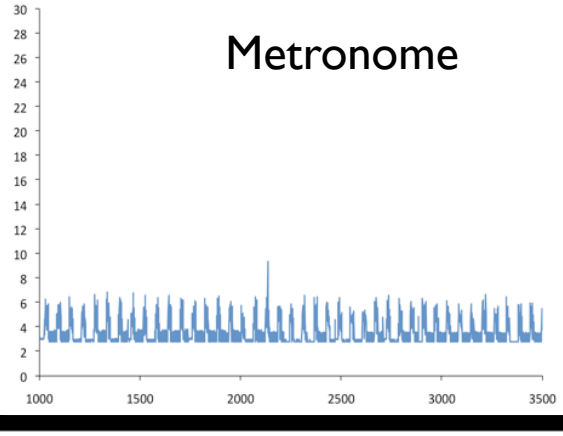


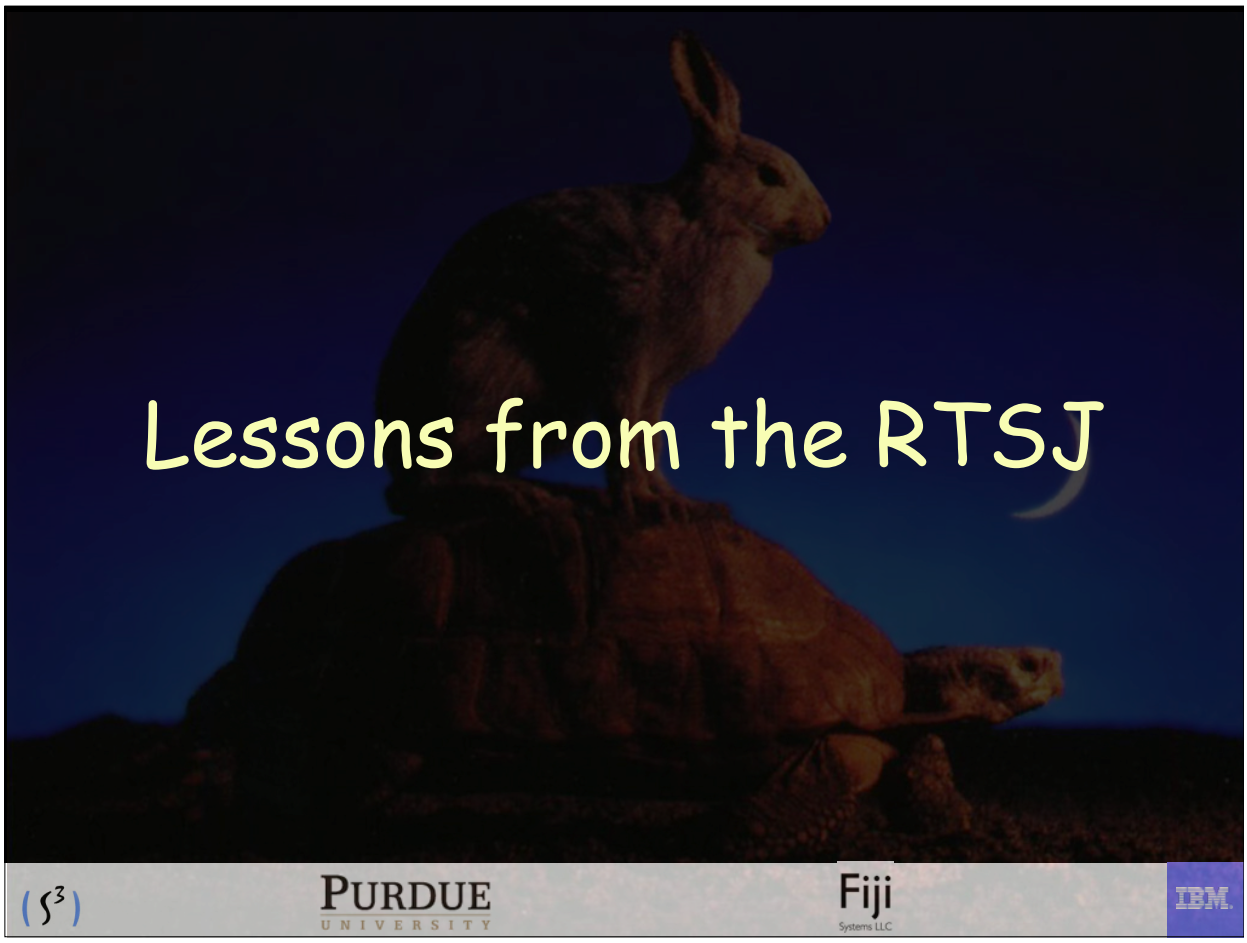
CD on J9

- The implementation seems to favor GC over scopes









1/9/90 Rule

- The main reason of RTSJ's success is that it allows mixing freely real-time and timing-oblivious code in the same platform
- Rule of thumb: 1% hard real-time, 9% soft real-time, 90% non-RT
- RT Programming Model should:
  - Allow non-intrusive injection of real-time in a timing-oblivious system
  - Avoid paradigm switches---single semantic framework for RT/non-RT

Small Footprint

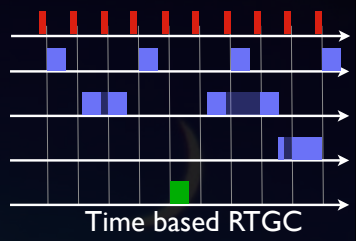
- The RTSJ requires sweeping changes in the Virtual Machine and has complex interaction with Java features.
  - The RTSJ changes the semantics of "normal" Java code:
    - any access to a reference variable may throw an exception,
    - meaning of "throws" changed to support asynchronous exception,
    - finalization interacts with memory management.
- Avoid complex APIs with ill-defined feature interactions
- Don't change the semantics of non-RT code

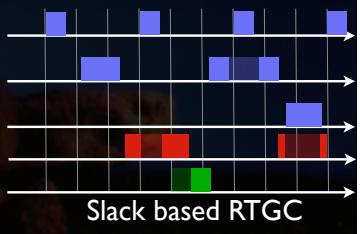
Schedule Flexibly

- Support for real-time scheduling is crucial with a clean integration in the language concurrency model
  - Priority Preemptive Scheduling for periodic tasks
  - Support PIP and PCE locks for priority inversion avoidance
- Provide facilities for writing non-blocking algorithms
- Allow for user-defined schedulers.

Don't Fear Garbage

- Real-time Garbage Collectors good enough for most RT apps
  - Time-based collectors (Metronome) can easily bound pauses to one millisecond.
  - Automatic defragmentation for long lived apps.
  - Slack-based collectors (SUN's) ensure RT code never preempted by GC.
  - Ensuring that the GC can keep up requires some understanding of whole-program allocation behavior.

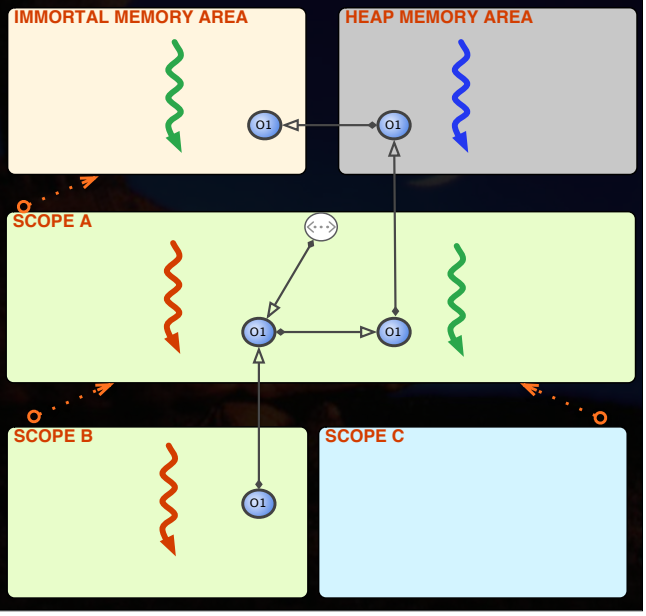




- Avoid manual memory management unless absolutely necessary.

Type-check Region-based Allocation

- Regions are the worst mistake of the RTSJ
  - Dynamic safety checks slow down programs and cause runtime errors that are hard to prevent
  - Distinction between heap-using and non-heap thread is tricky and error prone.
- Use static type system for assurance and efficiency



Shared-memory Concurrency is Hard

- Real-time presents a number of concurrency related challenges.
  - Critical sections must be short --- avoid over-synchronization
  - Avoid priority inversion between real-time and non-RT code
  - Avoid blocking operations in RT code
- Limit/localize communication between RT and non-RT.
- Support flexible isolation to encourage determinism.