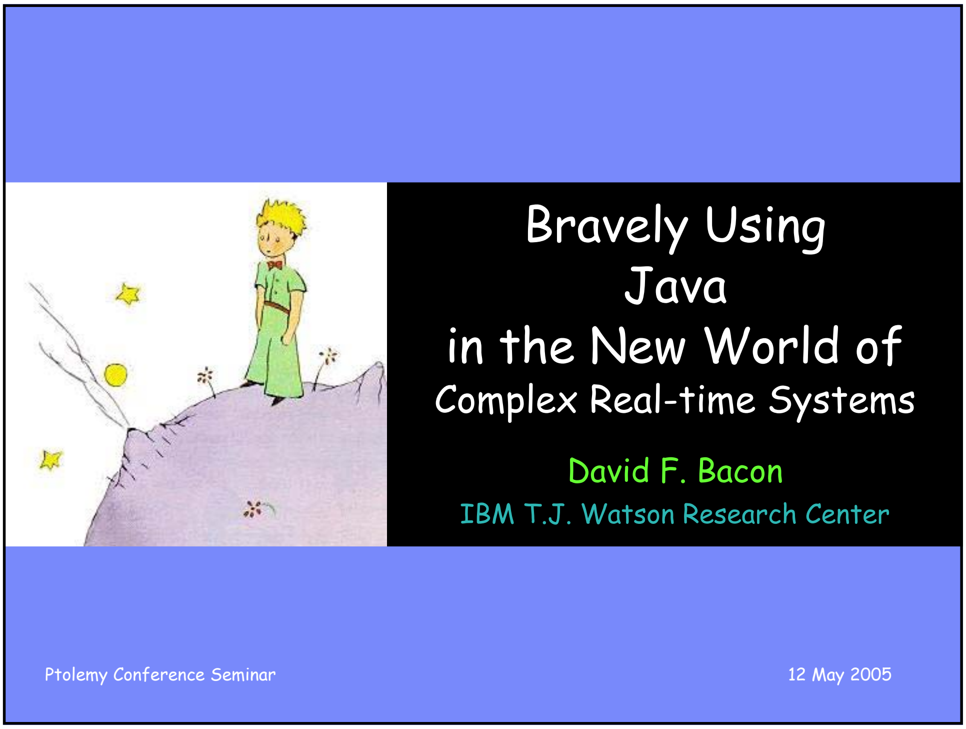


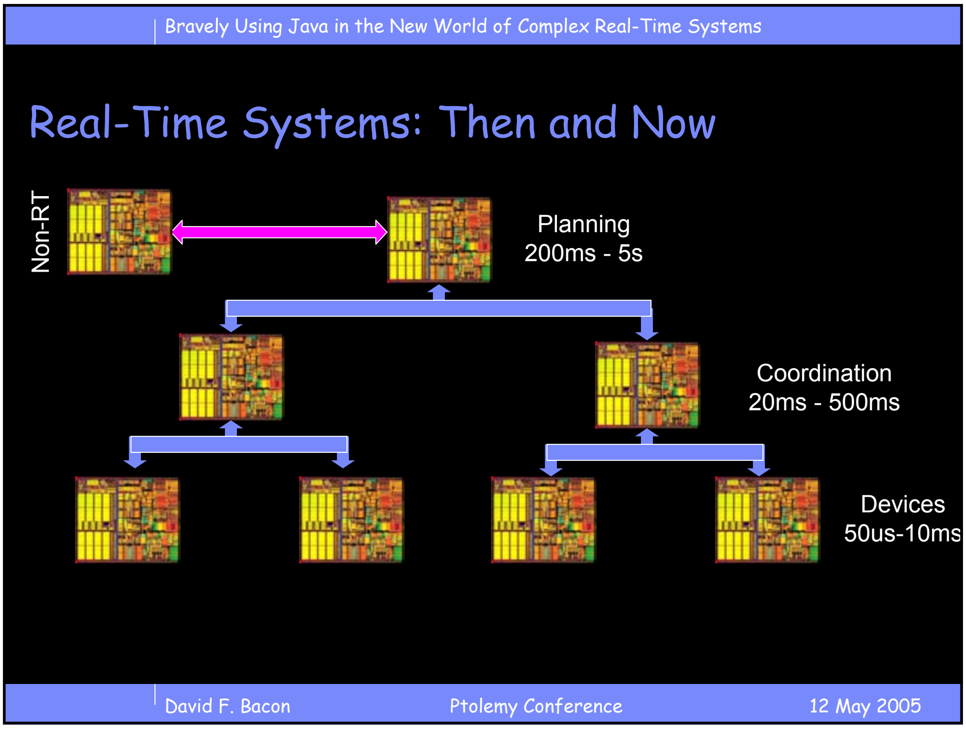
Bravely Using Java in the New World of Complex Real-time Systems

David F. Bacon

IBM T.J. Watson Research Center

Ptolemy Conference Seminar

12 May 2005



Real-Time Systems: Then and Now

Non-RT

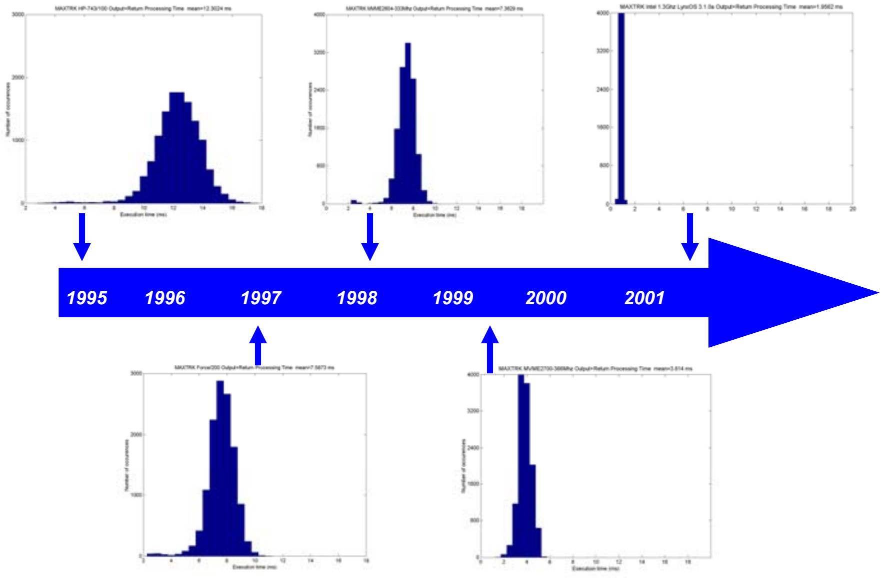
Planning
200ms - 5s

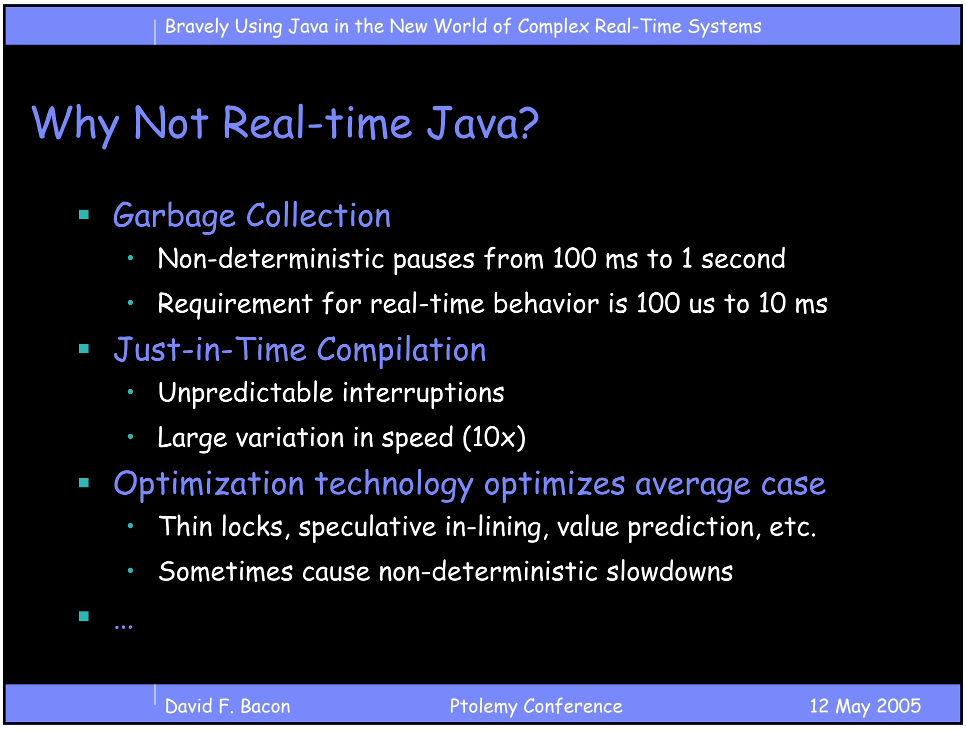
Coordination
20ms - 500ms

Devices
50us-10ms

Why Real-Time Java?

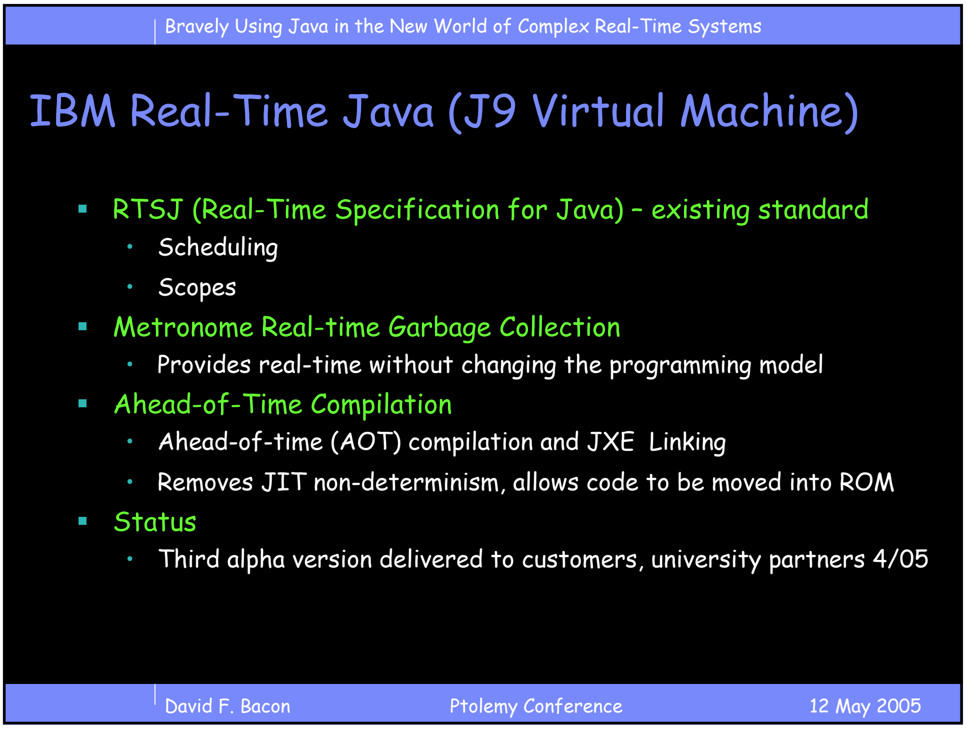
- Traditional methodologies
  - Highly restricted programming models with verifiable properties
  - And/Or low-level languages for explicit control
  - “ad-hoc low-level methods with validation by simulation and prototyping”
- But: these methodologies do not scale
  - Halting problem
  - Low productivity (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
- Just-in-Time Compilation
  - Unpredictable interruptions
  - Large variation in speed (10x)
- Optimization technology optimizes average case
  - Thin locks, speculative in-lining, value prediction, etc.
  - Sometimes cause non-deterministic slowdowns
- …



IBM Real-Time Java (J9 Virtual Machine)

- RTSJ (Real-Time Specification for Java) – existing standard
  - Scheduling
  - Scopes
- Metronome Real-time Garbage Collection
  - Provides real-time without changing the programming model
- Ahead-of-Time Compilation
  - Ahead-of-time (AOT) compilation and JXE Linking
  - Removes JIT non-determinism, allows code to be moved into ROM
- Status
  - Third alpha version delivered to customers, university partners 4/05

Surprise early adopter: defense industry

- Slow, painful death of Ada
  - Lack of programmers
- Mandated use of "commodity off-the-shelf" (COTS)
  - Elimination of duplication of effort by government
  - Sharing of systems, costs, expertise, training
- Elimination of single source of supply
  - More competition
  - Longevity of systems: aircraft carrier, air traffic control
  - Move to open standards and "open source" (!!)
- "Make it so"

- IBM Real-time Java selected by Raytheon for DD(X)
  - First Navy "open architecture" project

Outline

- RTSJ
  - Scheduling
  - Scopes
- Real-time Garbage Collection
- Handlers
- Logical Execution Time
- Ahead-of-Time Compilation
- Conclusions

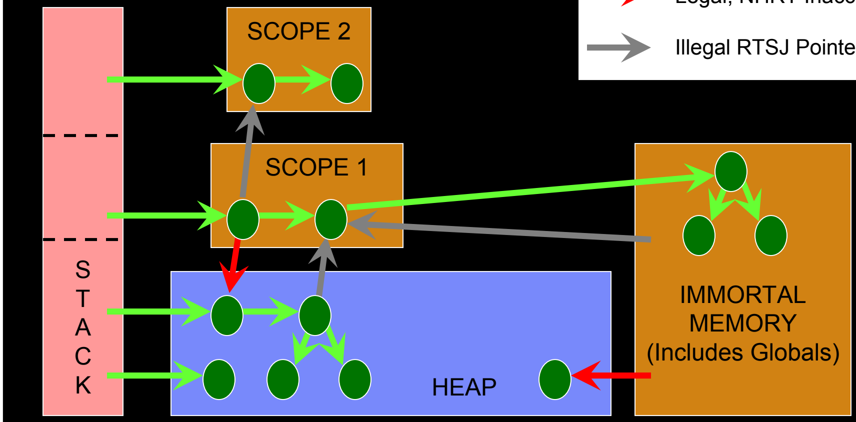
RTSJ Scheduling

- RealTimeThread
  - Can contain general Java code
  - No guaranteed response in the face of garbage collection
- NoHeapRealTimeThread (NHRT)
  - Restricted code; can only access special memory regions
  - Can pre-empt garbage collector at any time
- Scheduling: PriorityScheduler required; others optional
  - Scheduling may behave differently on different platforms
- Thread types: Periodic, Aperiodic, Sporadic (has delta)

RTSJ Scoped Memory

LEGEND

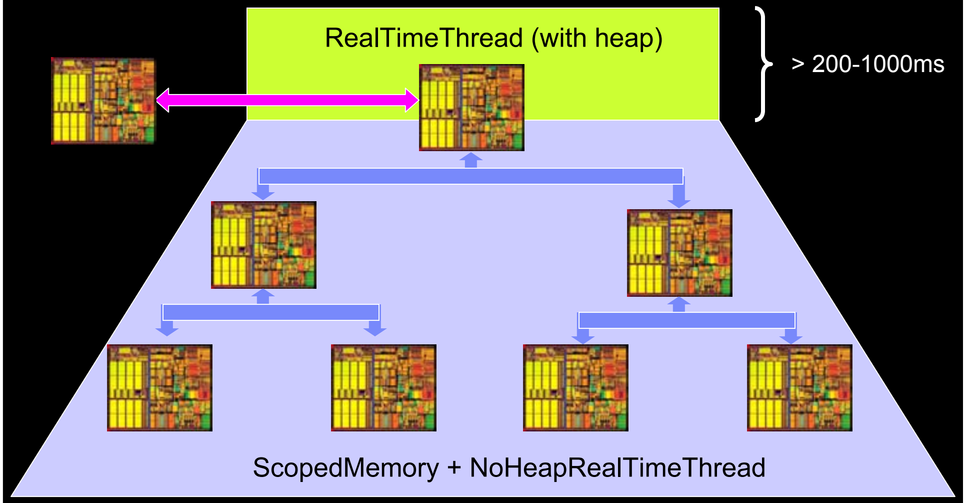
- 🟩 → Legal RTSJ Pointer
- 🟥 → Legal, NHRT Inaccessible
- ⬜ → Illegal RTSJ Pointer



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

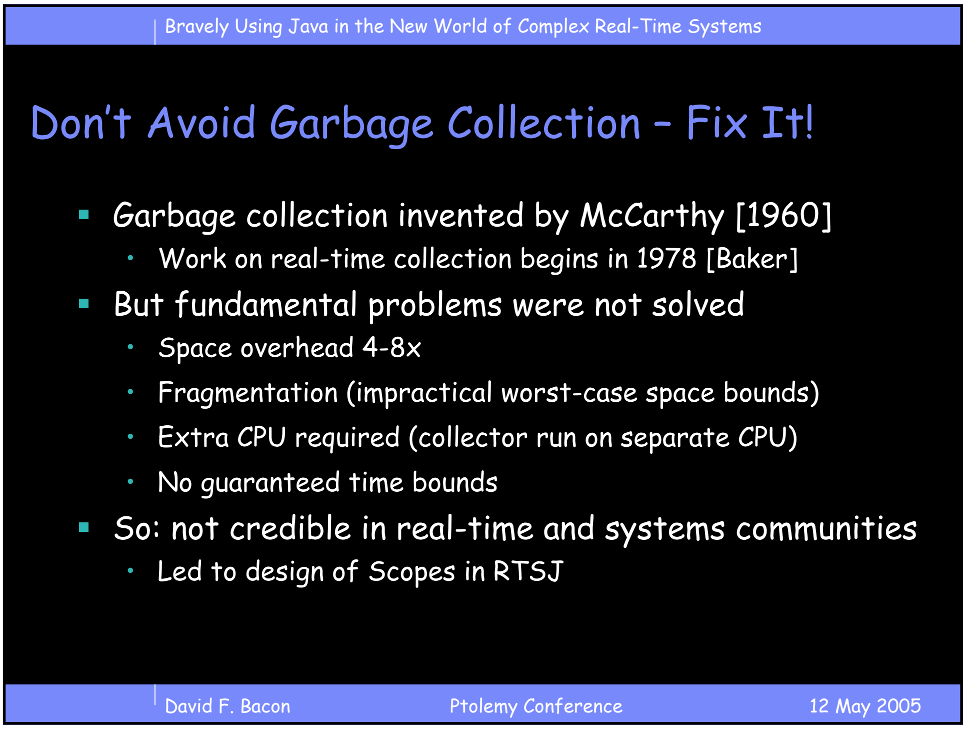
RTSJ Real-time Programming Abstractions



RealTimeThread (with heap)

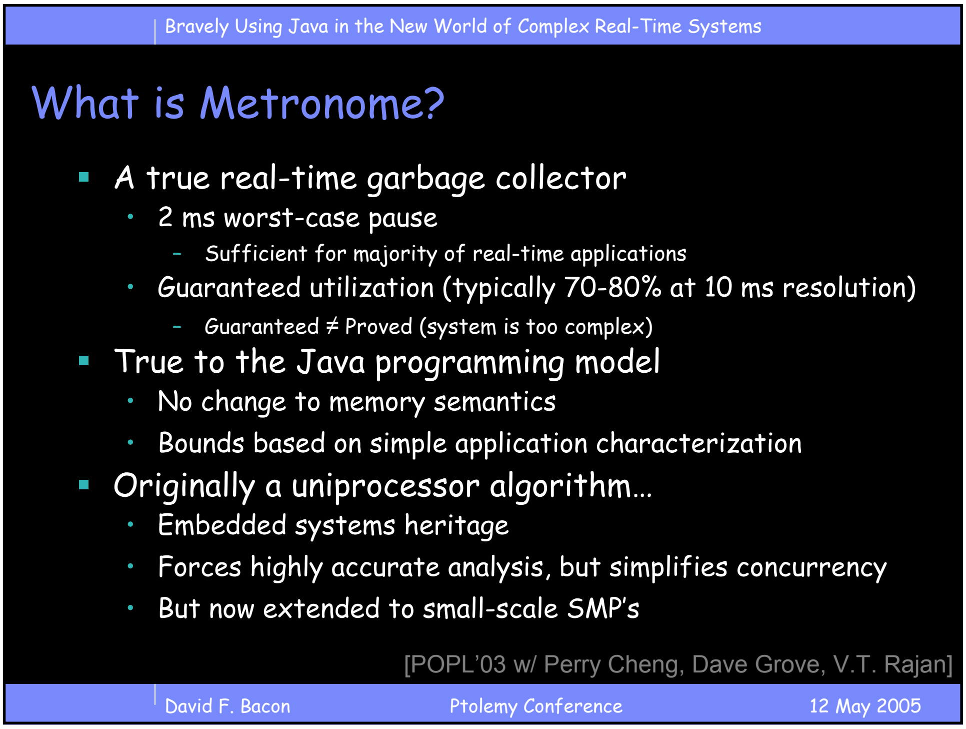
> 200-1000ms

ScopedMemory + NoHeapRealTimeThread



Don't Avoid Garbage Collection – Fix It!

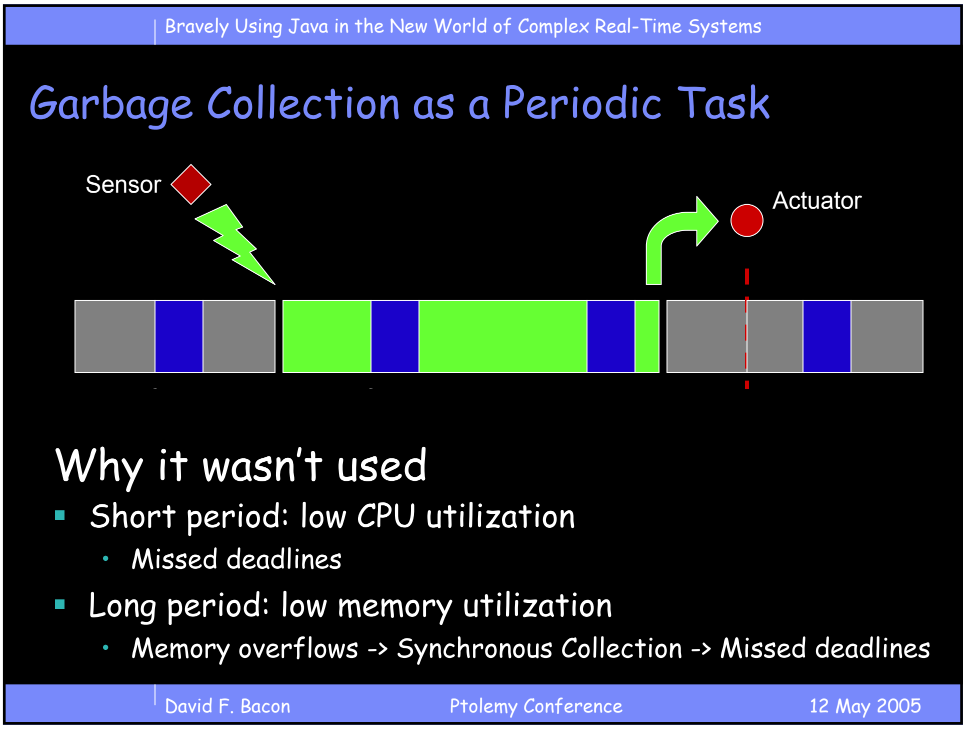
- Garbage collection invented by McCarthy [1960]
  - Work on real-time collection begins in 1978 [Baker]
- But fundamental problems were not solved
  - Space overhead 4-8x
  - Fragmentation (impractical worst-case space bounds)
  - Extra CPU required (collector run on separate CPU)
  - No guaranteed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



What is Metronome?

- A true real-time garbage collector
  - 2 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 small-scale SMP's

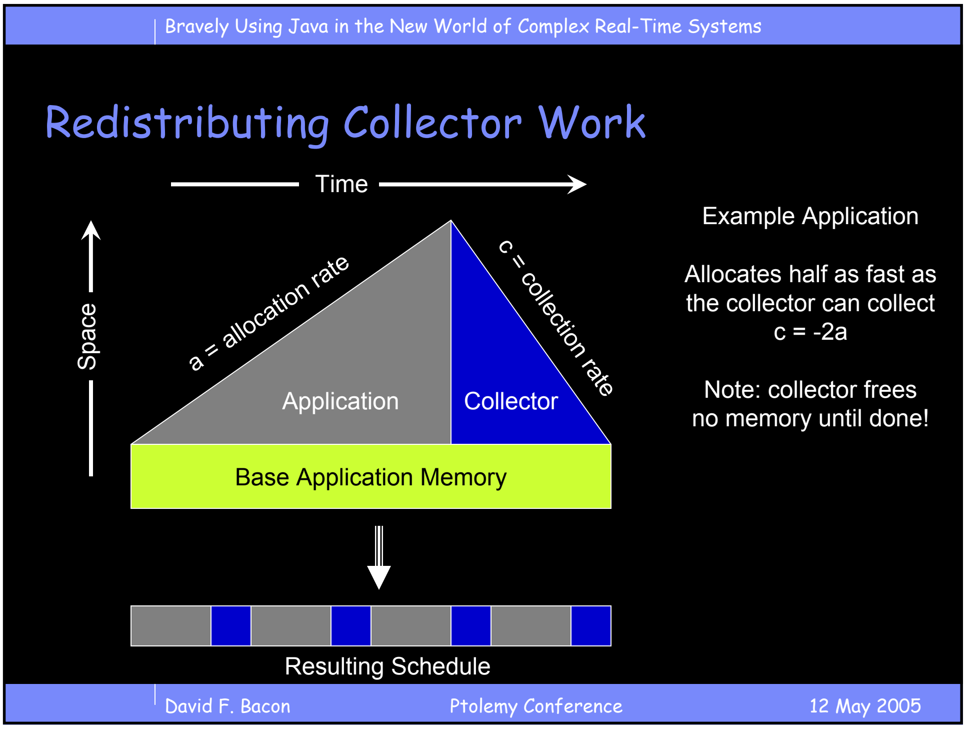
[POPL'03 w/ Perry Cheng, Dave Grove, V.T. Rajan]



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



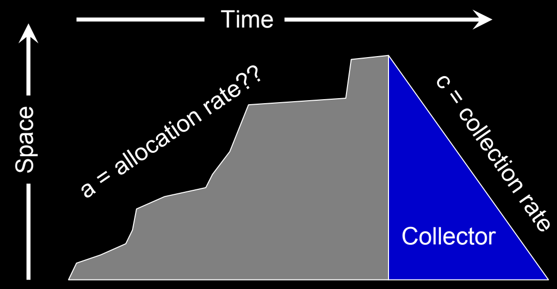
Redistributing Collector Work

Example Application

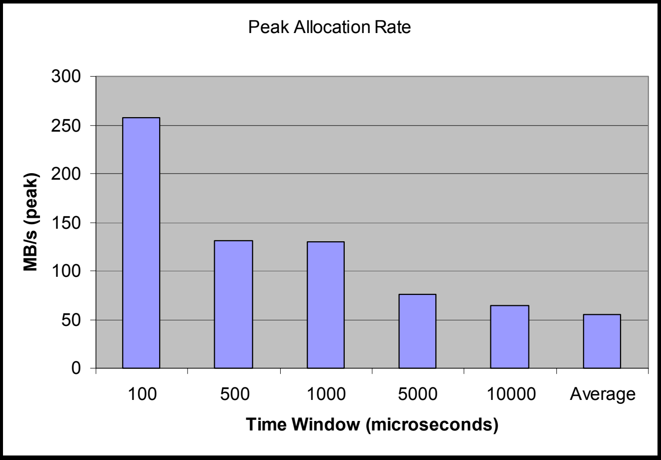
Allocates half as fast as the collector can collect
c = -2a

Note: collector frees no memory until done!

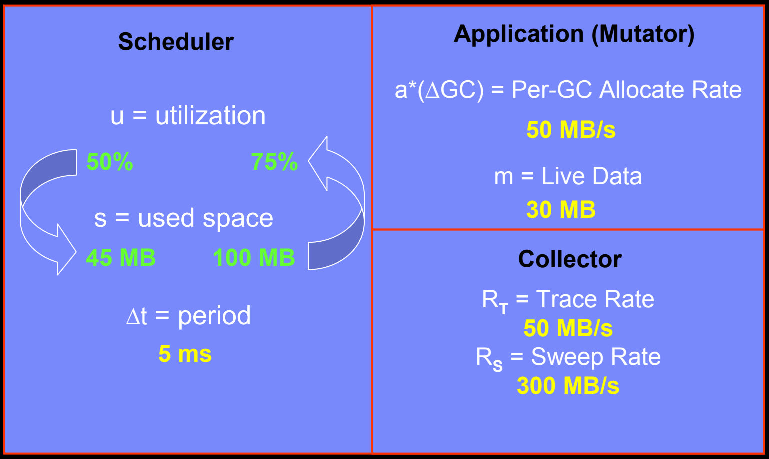
Can Application be Modeled So Simply?



Allocation Stability vs. Time Scale



Space ♑ Time



Scheduler

u = utilization

50% 75%

s = used space

45 MB 100 MB

Δt = period

5 ms

Application (Mutator)

a*(ΔGC) = Per-GC Allocate Rate

50 MB/s

m = Live Data

30 MB

Collector

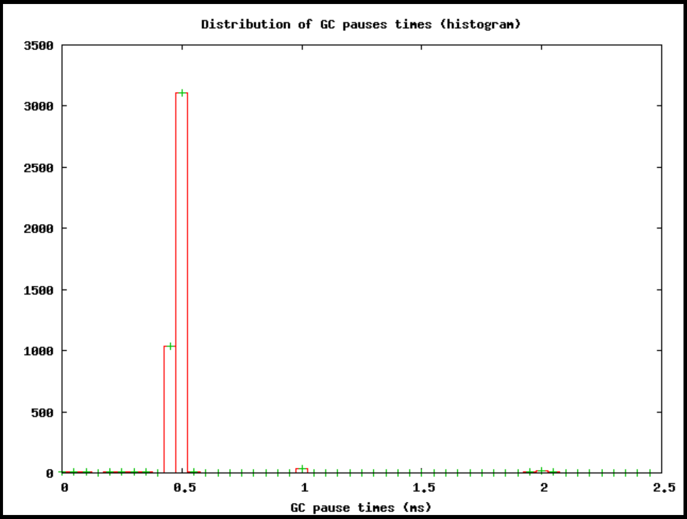
R_T = Trace Rate

50 MB/s

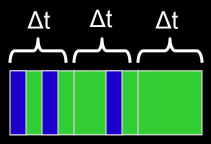
R_S = Sweep Rate

300 MB/s

Metronome: Collector Pause Times



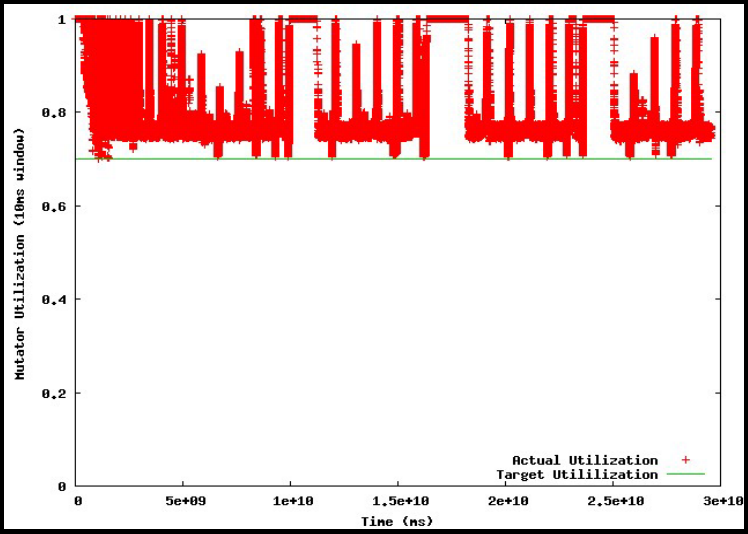
Minimum Mutator Utilization

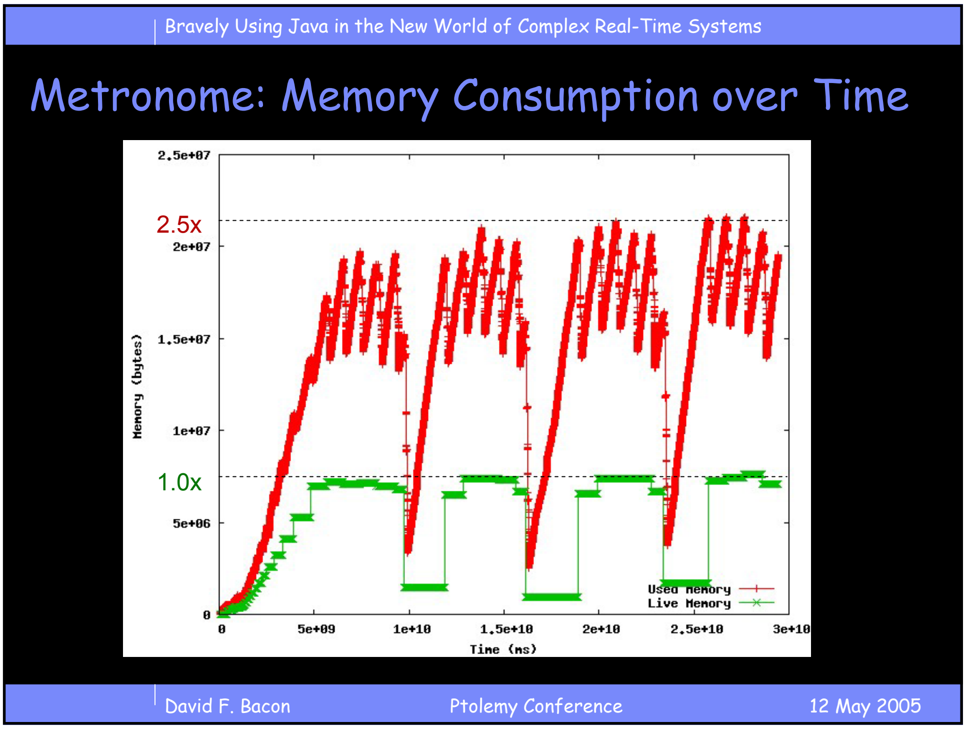


Δt = 4ms
u = 50%
Latency = 1ms
Throughput = 75%

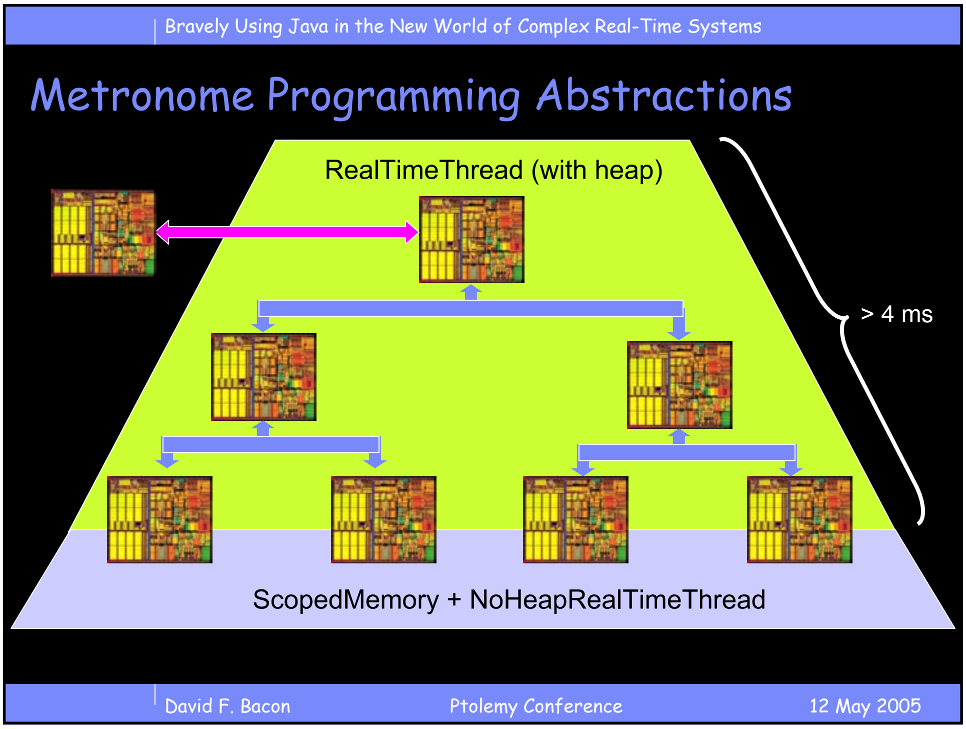
- Metric for "real-timeness"
- What it's not
  - Throughput
  - Latency
- What it is
  - Worst-case utilization over a time interval
  - Interval may contain multiple short interruptions
    - Upper bound on latency (interval x utilization)
  - Other intervals may have higher utilization (100% when GC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

Instantaneous Utilization: 10 ms (100 Hz)





Metronome: Memory Consumption over Time



Metronome Programming Abstractions

RealTimeThread (with heap)

> 4 ms

ScopedMemory + NoHeapRealTimeThread

Limits of Real-time Garbage Collection

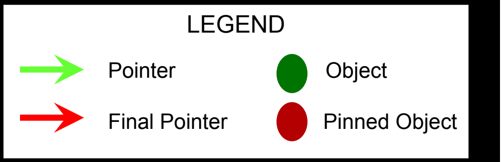
- Changes to heap require synchronization
  - Application modifies pointers and allocates objects
  - Collector moves objects to compact memory
- Synchronization is expensive
  - To keep cost reasonable, done in quanta (Metronome “beats”)
  - Quantization has limit (250-500 us)
- Real-time collection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

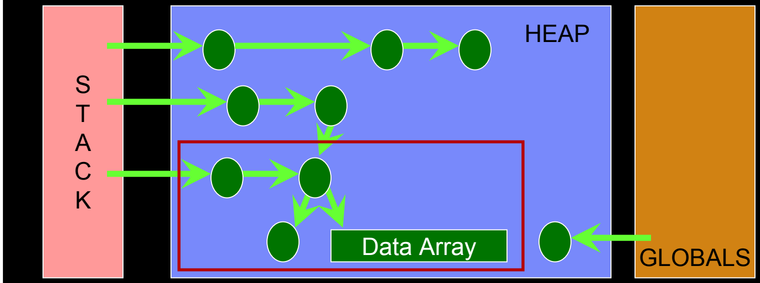
Handlers: Very Low Latency Operations

- 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
- Handlers
  - Data structure must be allocated in advance
    - Usually includes some buffers
  - When Handler is created, code and data are checked
    - If OK, then guaranteed to be free of memory exceptions
  - Handler can pre-empt garbage collector at any time
    - Data structure doesn’t change, so no collector/application synchronization required
  - More complex processing can be done by low-frequency task
    - Builds data structures collected by real-time garbage collector

[w/ Dan Spoonhower]

Handlers

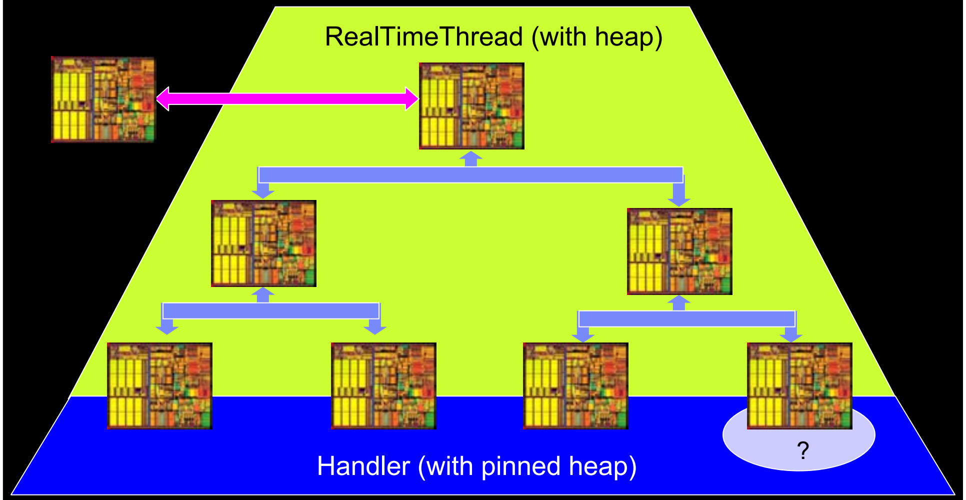




Handlers vs. Scopes

	Handlers	Scopes
Preempt Collector	Yes	Yes
Access to rest of Heap	No	No
Access from Heap	Yes	No
Allocation	No	Yes
Memory Safety	Yes	No

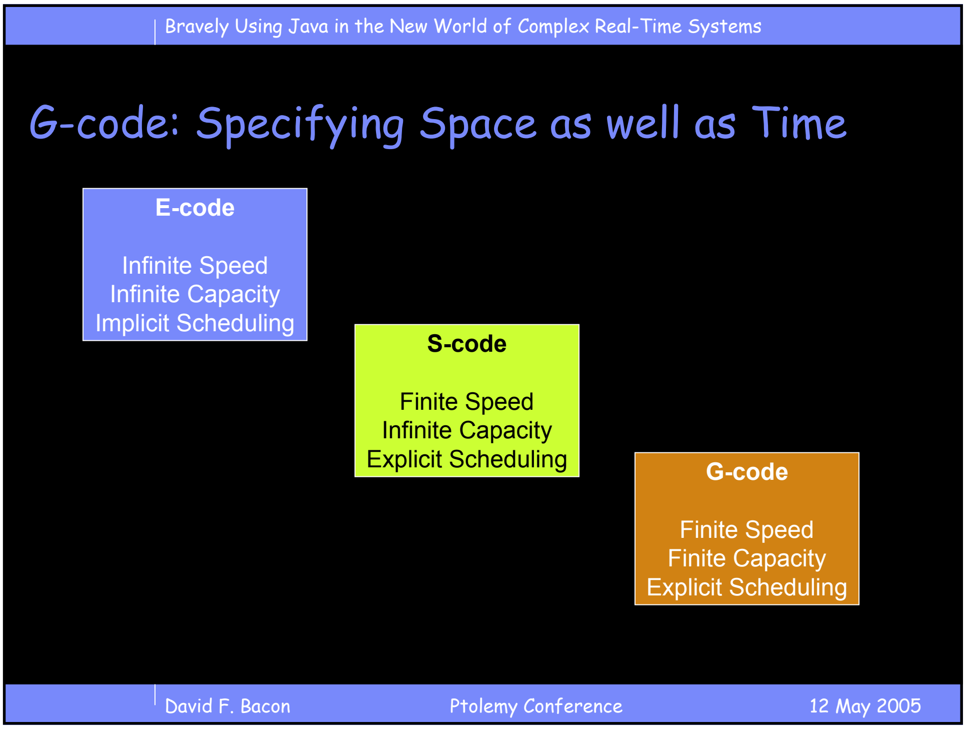
Metronome + Handler Abstractions



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
- Implementing E-machine support directly in the JVM
  - Allows E-machine to be written almost entirely in Java
  - Initial version in April release of IBM Real-Time Java

[w/ Tom Henzinger, Christoph Kirsch]



G-code: Specifying Space as well as Time

E-code

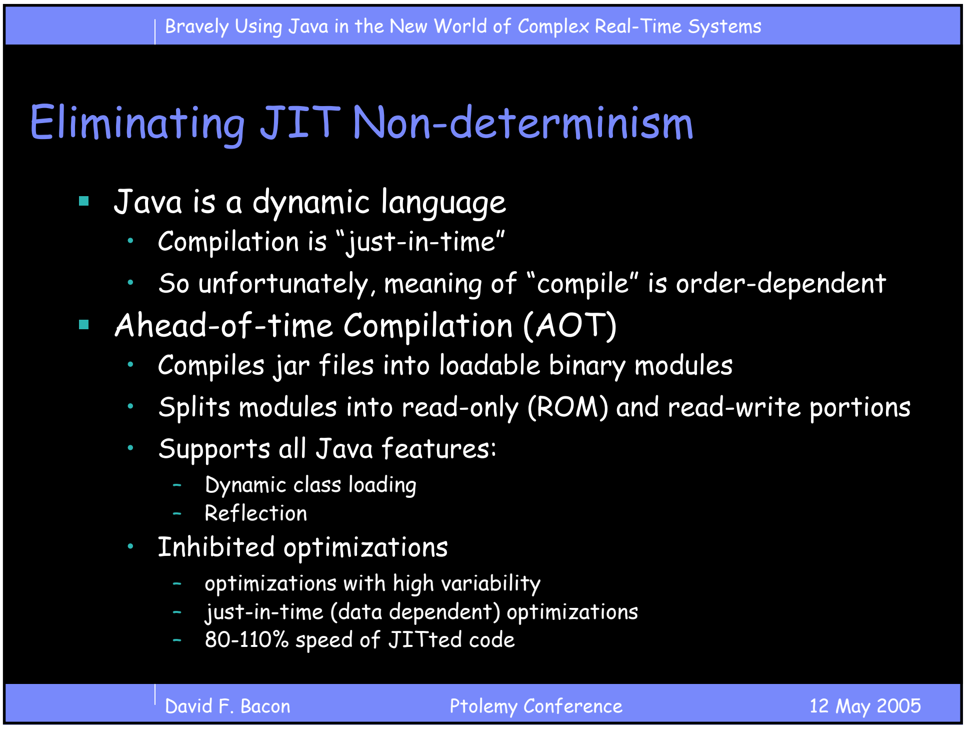
Infinite Speed
Infinite Capacity
Implicit Scheduling

S-code

Finite Speed
Infinite Capacity
Explicit Scheduling

G-code

Finite Speed
Finite Capacity
Explicit Scheduling



Eliminating JIT Non-determinism

- 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

Conclusions

- 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
- IBM Real-Time Java is leading the way
  - Real-time garbage collection, RTSJ, AOT compilation, E-machine hooks
  - Highly integrated research and development effort
    - IBM: Research, Software Group, Federal Systems
    - Industry: Raytheon and others
    - Government: Navy
    - Academia: Berkeley, Salzburg, EPFL, Cambridge, CMU
  - Product-quality JVM with complete suite of libraries
- Seeking collaborators, especially to build real-time Java applications
  - Binary distribution available

Questions?

http://www.research.ibm.com/metronome