

Real-Time Garbage Collection Panel

JTRES 2007

Bertrand Delsart, Sun
Sean Foley, IBM
Kelvin Nilsen, Aonix
Sven Robertz, Lund Univ
Fridtjof Siebert, aicas

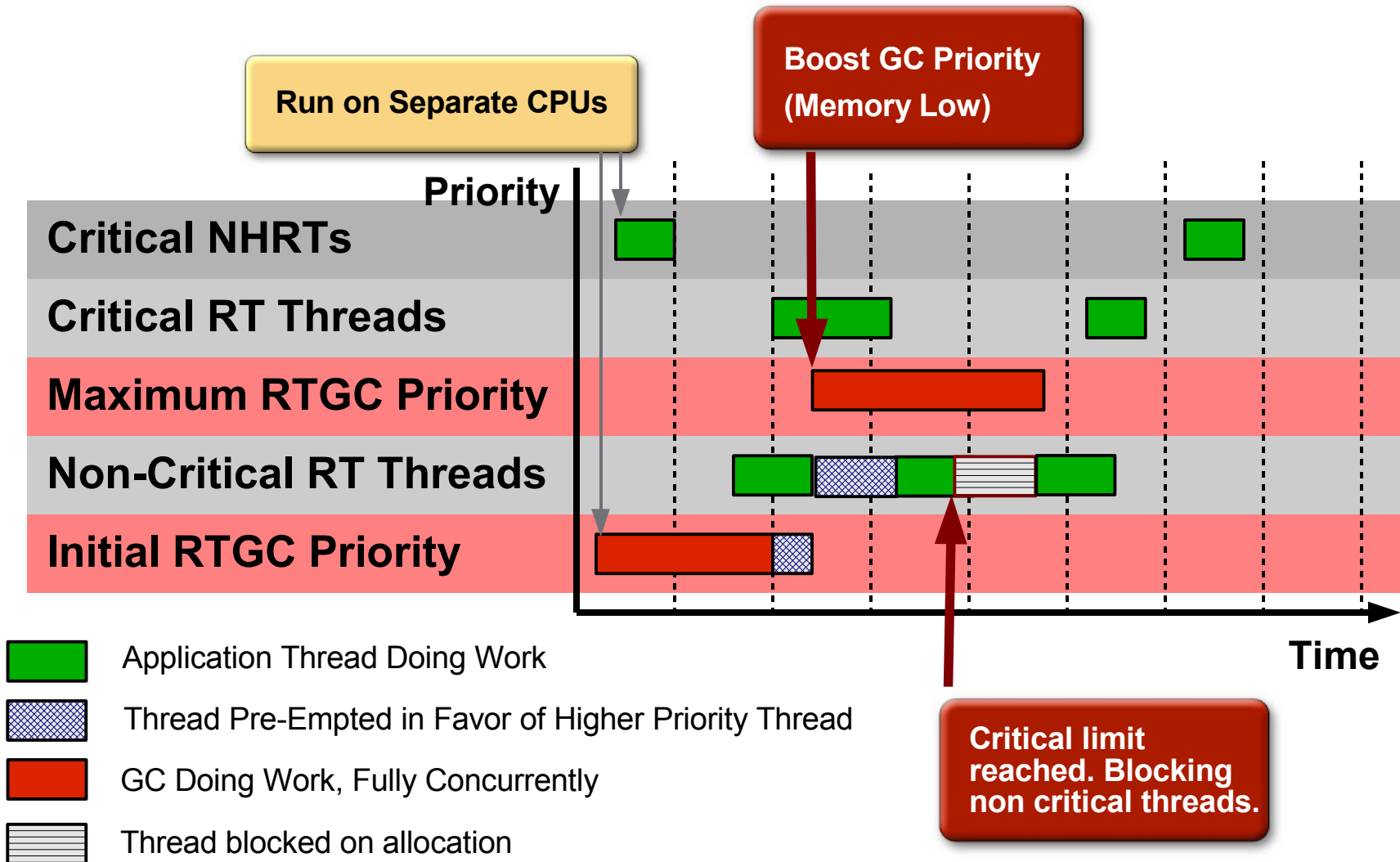
Feedback from our customers

- Is it **fast enough** to do all this work every 50ms ?
- **This particular** task has to be completed in 500us
- What ? I should have **configured** the RTGC ?
- I love your **32-cores** machine. What can you do to reduce the pause times for my Java application ?

Sun's Solutions

- A few threads are **always** more important than GC
- Robustness for a few critical **shielded** threads
 - > Unexpected allocation bursts / memory retention by the other threads do not endanger them
- Overhead required to prove this robustness mainly depends on what these critical threads do... allowing **huge safety margins**
- **Fully concurrent** (no atomic GC steps) to take advantage of multi-processors... and offer very **flexible soft real-time** policies for the other threads

Example: 1CPU GC policy on a bi-pro



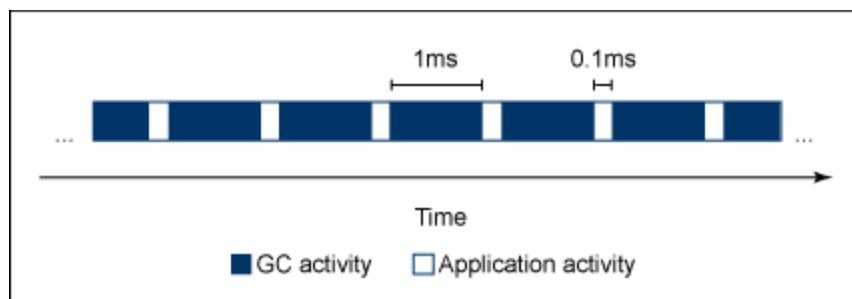
What we usually don't say !

- This is a non moving GC !!!
- We have found ways to fight against fragmentation
- We have additional ideas to fight even better... but it seems we do not need them

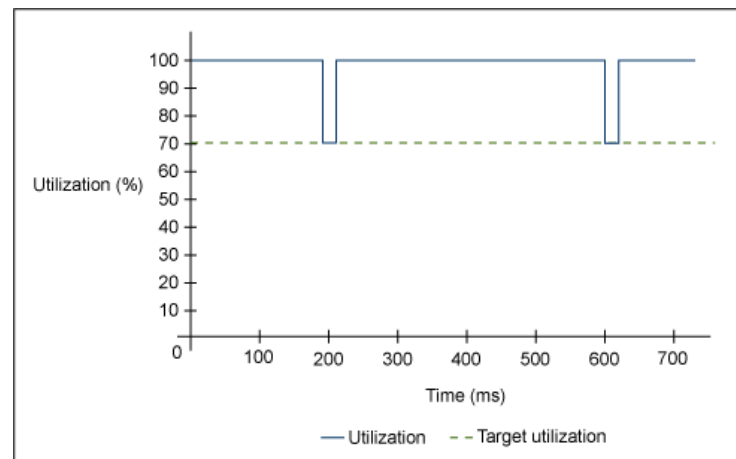
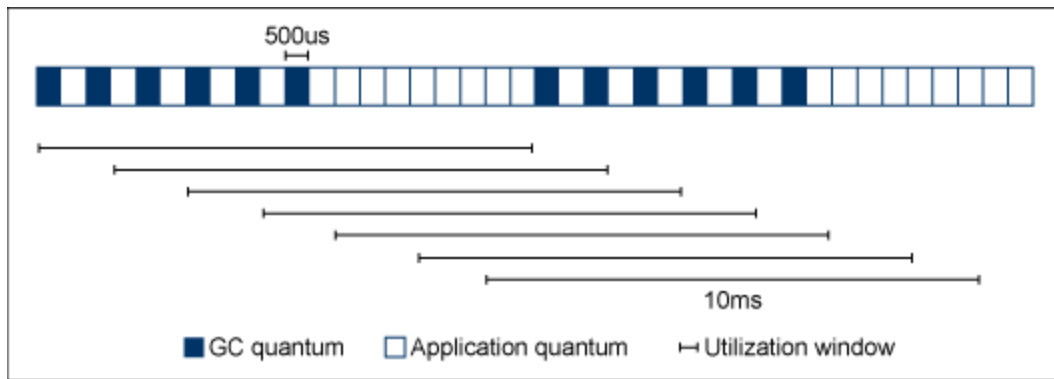
... and in the worst case we can behave like ALCAS :-)

IBM Metronome GC

- **Work-based: Inadequate utilization**

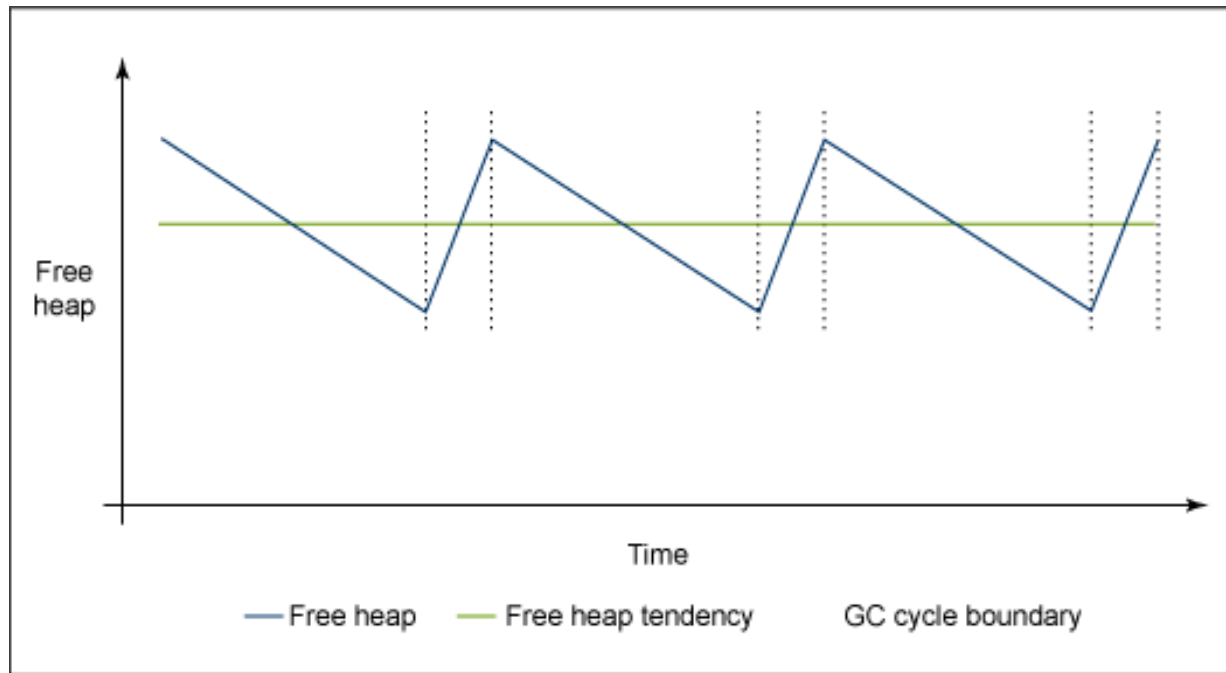


- **Metronome time-based: Sliding window of utilization**



IBM Metronome GC

Stable free heap





Real-Time Garbage Collection Issues

Kelvin Nilsen, CTO



- Greatly simplifies development and maintenance of software
 - Xerox PARC study: “saves 40% of development cost for large, complex systems”
- Facilitates reuse of software (in real-time systems), especially off-the-shelf Java components and “standard libraries”
- Improves software generality and reuse and reliability
 - Protects against dangling pointers
 - Simplifies elimination of memory leaks

- Traditional GC introduces very long delays (ms to tens of seconds)
- Real-time GC (especially) introduces run-time overheads into the execution of Java code
 - Application software and GC contend for access to shared resource (memory)
 - Very efficient low-level “synchronization” is necessary
 - Typical implementation: special instructions associated with every read and/or write access to heap memory
 - Extra meta-data associated with heap objects



- Pacing of garbage collection against allocation behavior works extremely well if application behavior is “perfectly understood”
 - But errors in the pacing parameters can have far-reaching negative impacts on reliable operation and especially on compliance with (hard) real-time constraints
 - In the absence of formal proofs of application behavior, it is essential to study the system’s failure modes (in the presence of pacing parameter errors)
 - Developing and maintaining formal proofs of memory allocation behavior in “arbitrary real-world applications” is extremely difficult (and costly)

Real-Time GC Scheduling

Sven Gestegård Robertz

Department of Computer Science
Lund University
Sweden



Preliminaries



- Application includes mutator and collector
 - This total task set must be schedulable
- Real-time run-time system
 - If a task set is feasible, all deadlines are met
 - Fairness is not desirable
 - Minimize latency and jitter for time-critical tasks
 - Isolation between tasks
 - w.r.t CPU, memory, I/O, ...

RTGC Scheduling



- Mutator + collector schedulable
 - Cannot ensure GC progress at the expense of mutator tasks
- Treat GC as a normal task
 - Deadline
 - Execution time
 - No special scheduling
 - RMS
 - EDF

Isolation



- Scheduler provides temporal isolation
 - High priority threads preempt low priority ones
 - A run-away low priority thread cannot consume all CPU time
- Memory manager should do the same
 - Memory is a global resource
 - Memory allocations in non-critical code must not cause an out-of-memory situation in a critical task

Realtime Garbage Collection is here!

- Pause times in order of single μ secs.
- Proofs that system does not run out of memory
- Proofs that limit GC work

Scoped Memory is obsolete

- not managable for more complex applications
 - will still need analysis of maximum amount of allocation
 - will introduce possible runtime errors, memory leaks, etc.
 - well tested RT GC vs. application-specific manual memory management.
- ➔ Only reason to use Scoped Memory is certification (DO178B etc.).

How do we convince the certification authorities?

RT GC needs guarantees from app

- all need bounds on reachable memory
- Some GCs need bounds on allocation rate
- Some GCs need scheduler input (load of RT threads with higher priority, etc.)

RT GC implementations must be

- Easy to configure
- Currently, knobs include
 - heap size
 - applications bound on reachable memory
 - GC priority
 - max. allocation rate
 - size of GC increments
 - load of higher priority threads
 - etc.

➔ user does not understand GC implementation!

We must

- reduce number of knobs
- realtime GC must run out-of-the-box

The user

- will usually just test it
- he will use it if it works!
- says: „well, we didn't have time to do full WCET analysis, but we don't miss any deadlines“