

JavaOne

# Easily Tuning Your Real-Time Application

Bertrand Delsart
Frederic Parain
Sun Microsystems, Inc.

# Goal of this presentation

Show how to tune and monitor a real-time application using Java Real-Time System tool set

# Outline

- Tuning for Real-Time
- Tuning Compilation
- Tuning Priorities
- Tuning Memory Managers
- The Future

# Outline

- **Tuning for Real-Time**
- Tuning Compilation
- Tuning Priorities
- Tuning Memory Managers
- The Future

# Java Platforms and Real-Time

- Real-Time Specification for Java (RTSJ)
  - Provides an Application Programming Interface that enables the creation, verification, analysis, execution and management of Java threads whose correctness conditions include timeliness constraints
- Java Real-Time System
  - Sun's implementation of the RTSJ
  - Based on JDK 5 platform (32-bit and 64-bit)
  - Real-Time Garbage Collector
  - Initialization-Time Compilation
  - DTrace instrumentation

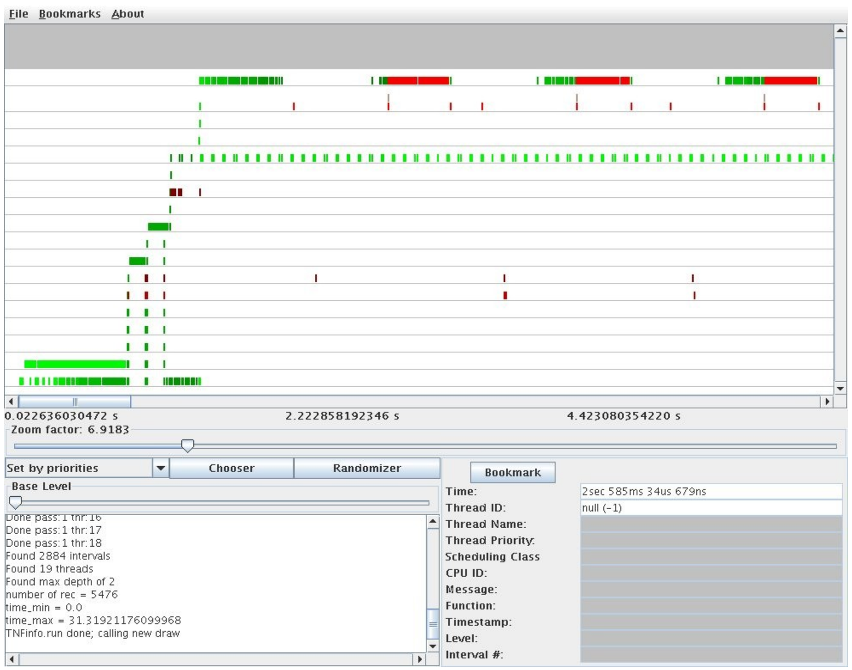
# Why tuning is required

- Tuning for determinism
  - No deadline miss
- A real-time virtual machine is not a crystal ball
  - Application's requirements unknown
- Virtual machine adapts itself
  - Based on current or past requirements of the application
- Dynamic Adaptation
  - Makes developer's life easier
  - Introduces jitter

# Auto-adaptation

> Not a magic bullet
  - Requires a warm-up phase
  - Time consuming
  - Difficult to be accurate
  - Cannot solve all the tuning issues
> Tuning required when auto-adaptation is not enough
> Tuning helps to optimize adaptation

# Tool: Thread Scheduling Visualizer




- Tool to record and analyze thread execution
  - Events are recorded during execution
  - Off-line visualization
- Provide graphical time-line based views

# Outline

- Tuning for Real-Time
- **Tuning Compilation**
- Tuning Priorities
- Tuning Memory Managers
- The Future

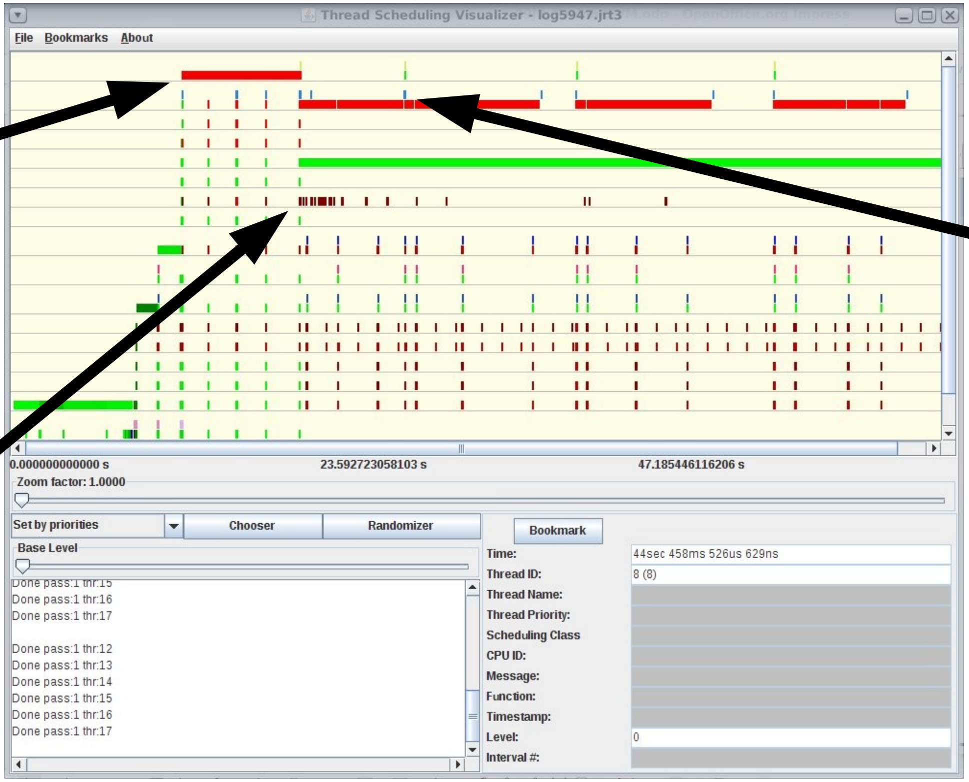
# Class Loading / Compilation

- Default behavior:
  - On-demand class loading
  - Just-In-Time compilation of hot methods
- Neither policy is well suited for RT
  - Jitter late in application's execution
  - Generally happens at worst time: error handling, uncommon situations, ...

# Demo RTImageProcessing

Class initialization performed by a real-time thread

Compilation occurs during first period

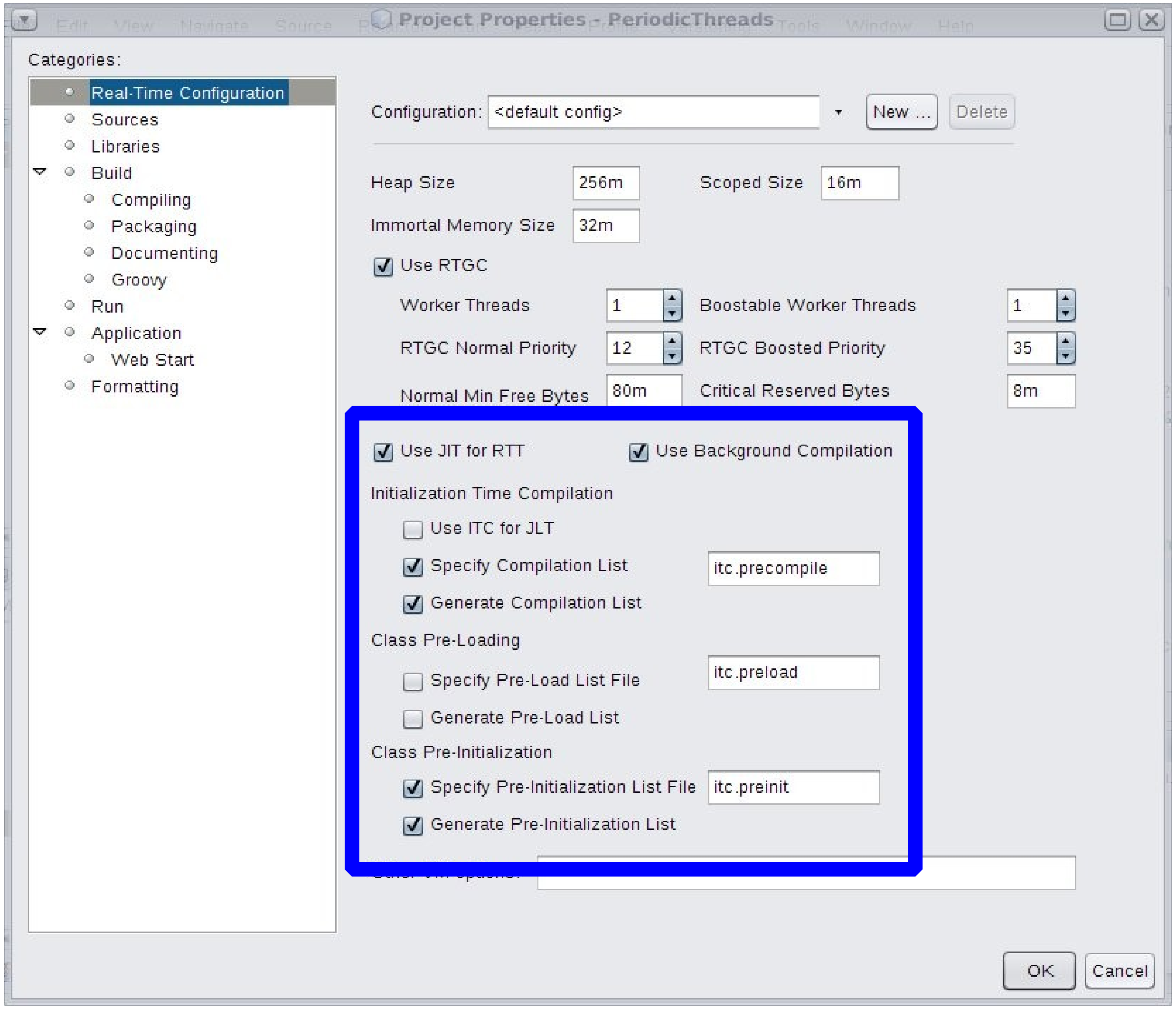



Delays cause first deadline to be missed

# Initialization Time Compilation

- List of pre-loaded classes
- List of pre-initialized classes
- List of methods to be compiled at class initialization time
- Java RTS can generate these lists automatically
- Developers can edit them
  - List format supports wild cards

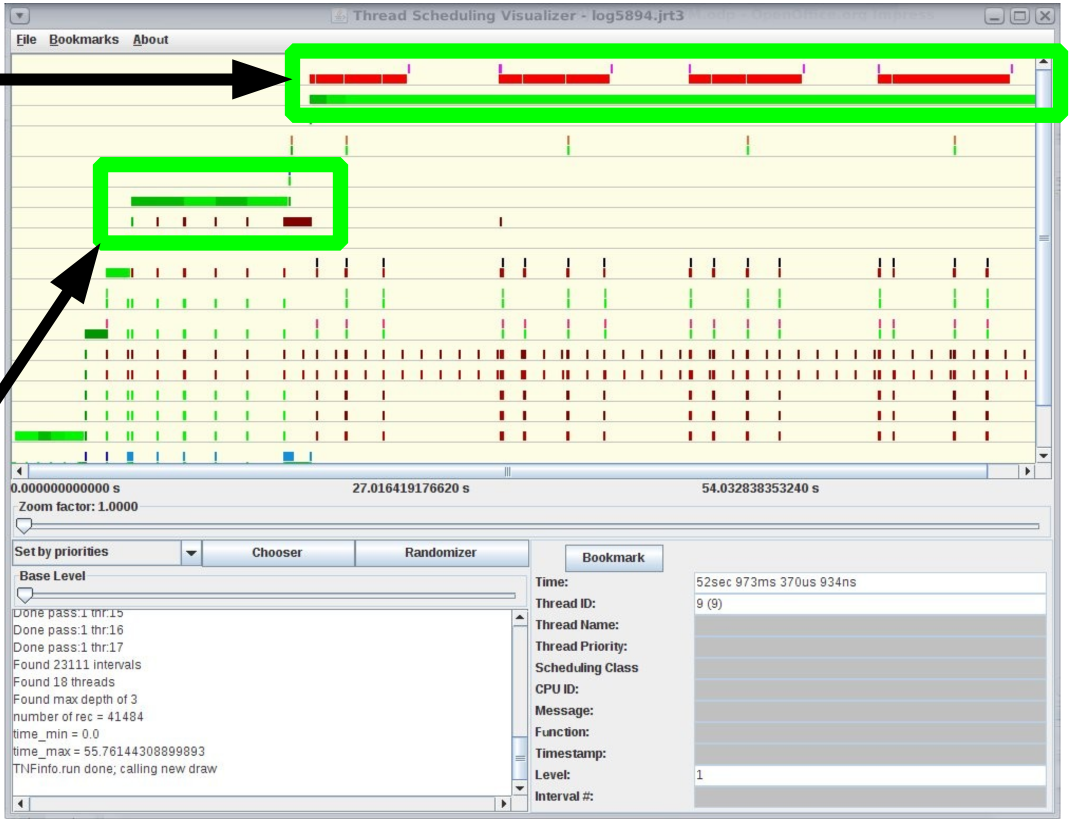
# ITC Configuration



# Demo RTImageProcessing with ITC

The real-time thread has deterministic behavior as of its first execution: no deadline miss

Class initialization and compilation are performed by the VM at startup time



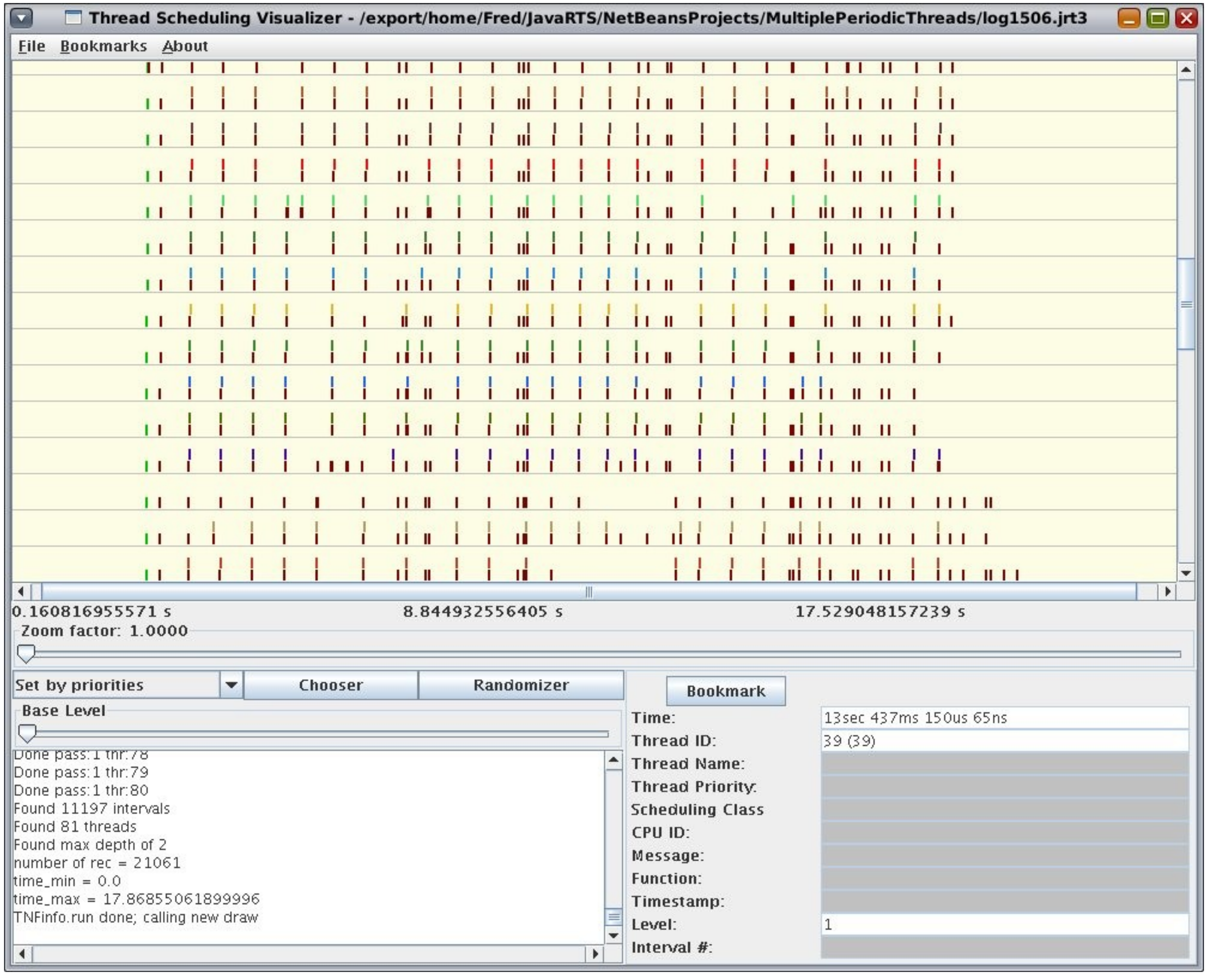
# Outline

> Tuning for Real-Time
> Tuning Compilation
> **Tuning Priorities**
> Tuning Memory Managers
> The Future

# CPU Resources

- Shared among all threads
  - Real-Time Threads / Non Real-Time Threads
  - Virtual Machine Threads
- Importance of the scheduling policy
  - Time-sharing scheduler apportions CPU time out to all threads
  - Real-Time Scheduling always gives CPU to the highest priority threads
    - Can cause starvation, delays, dead-locks
- Priorities control access to CPU

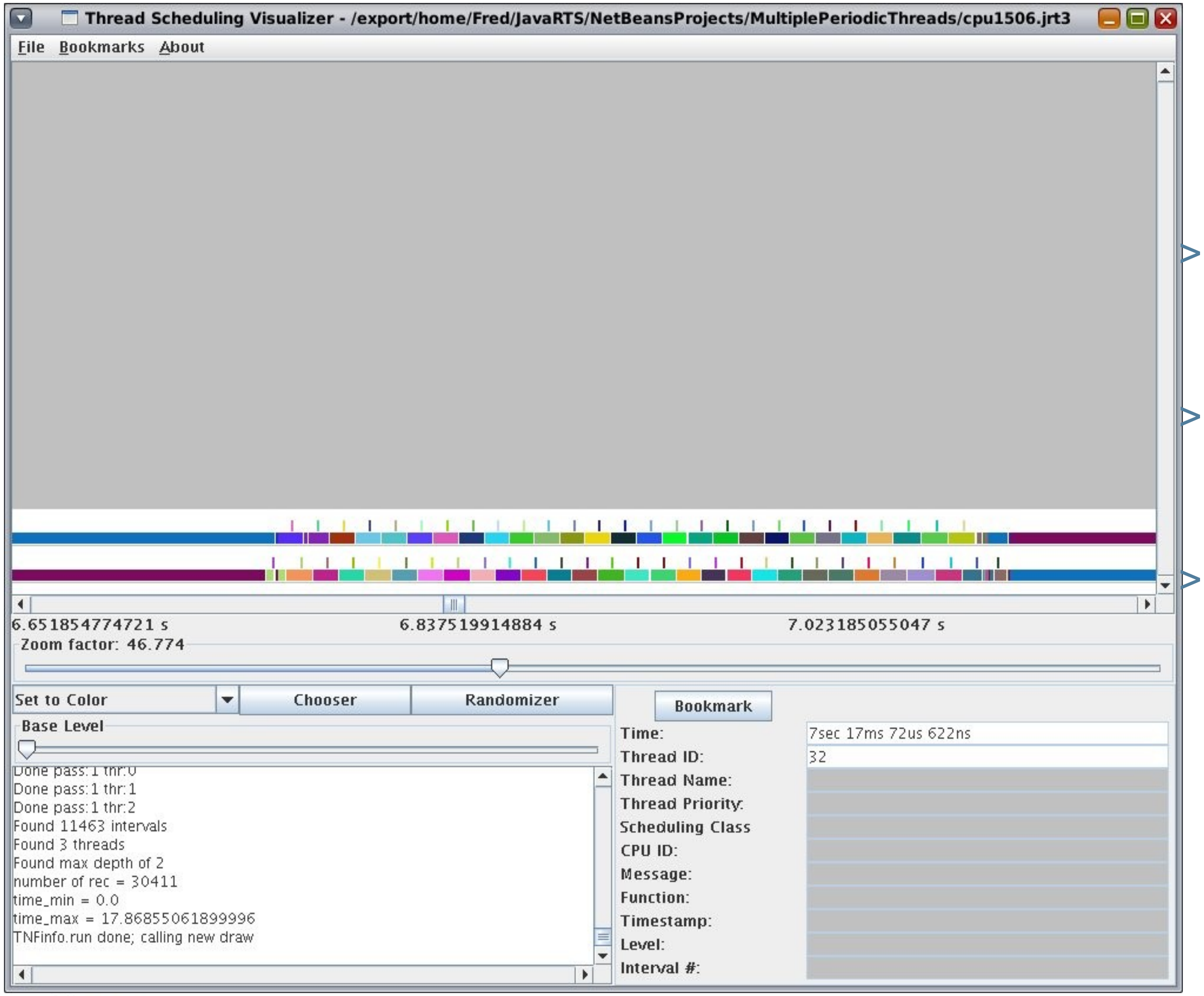
# Application with Multiple Threads




> Difficult to see the relationships among threads

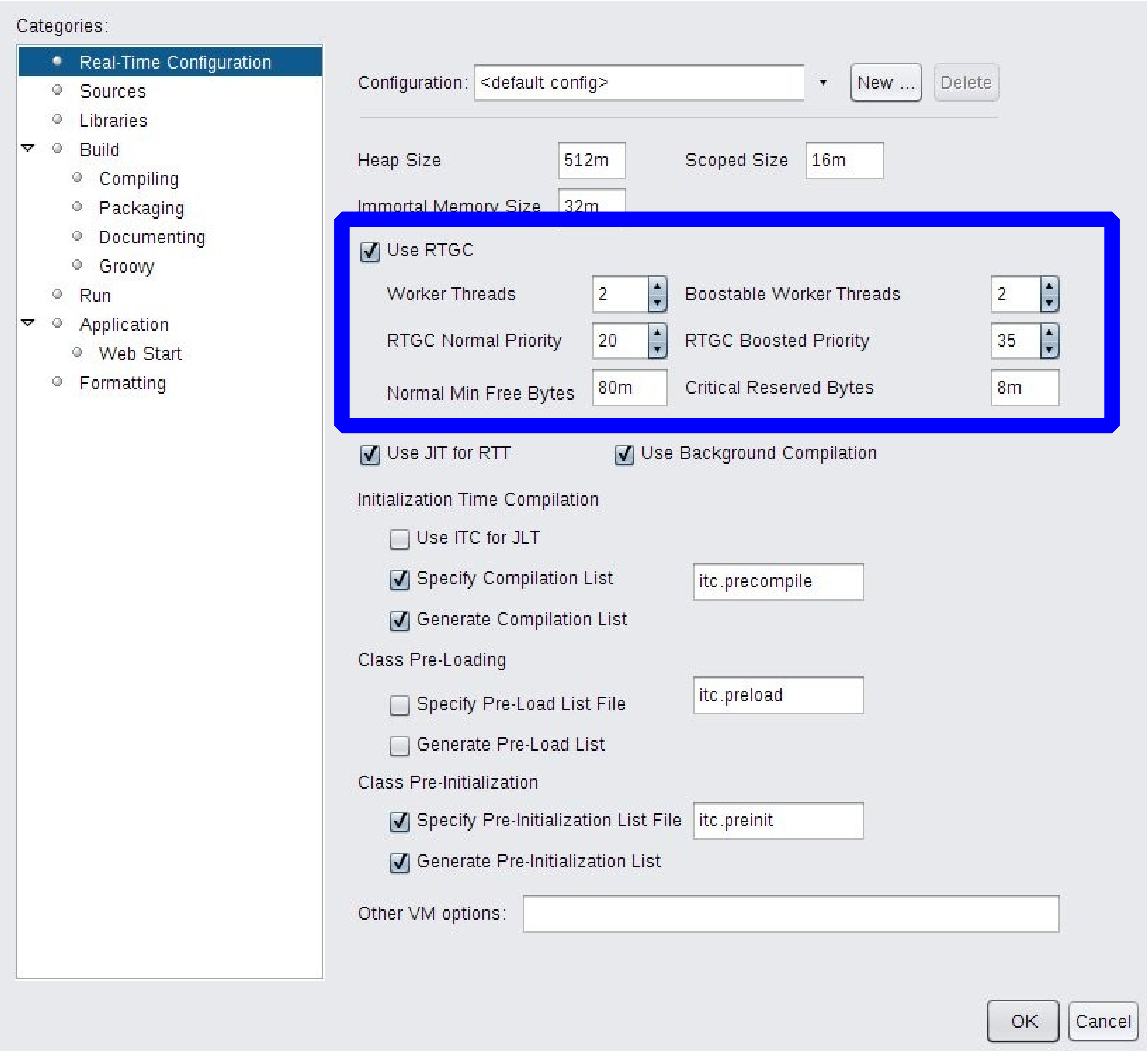
> Very hard to evaluate the CPU load

# Per CPU View

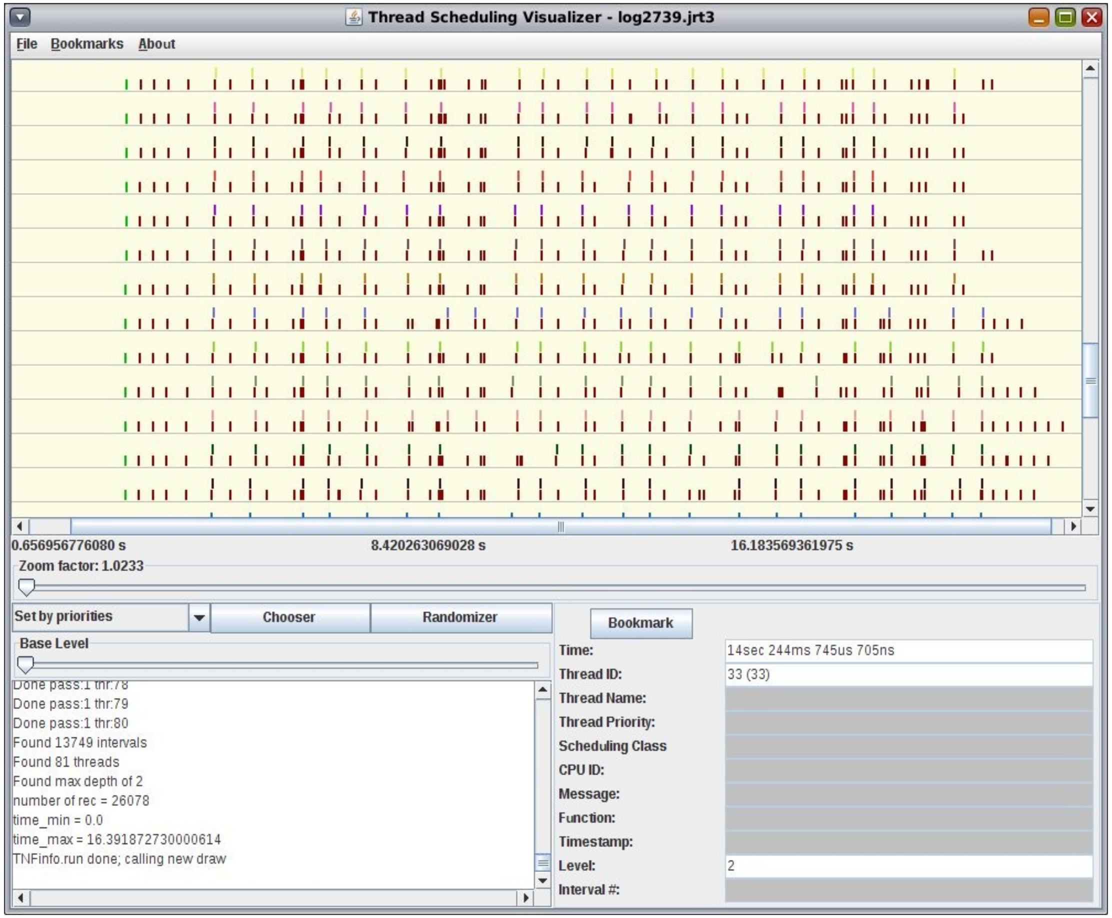



- CPUs are clearly overloaded
- RTTs cannot run during GC cycle
- Thread migrations are easy to see

# Configuring RTGC Threads

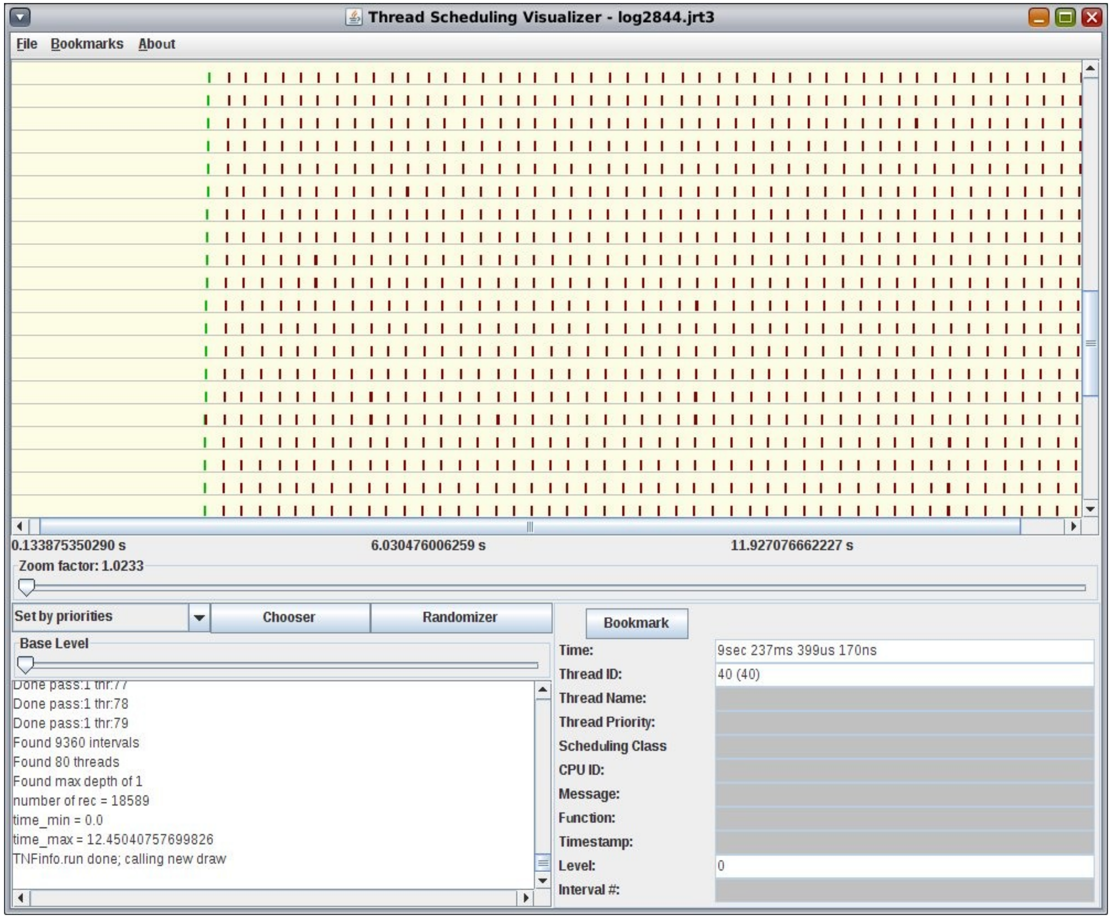


# Giving Priority to the Application




- Application threads run at a higher priority than the RTGC.
- It doesn't solve the problem: deadline misses still occur.
- Threads need memory.
- RTGC cannot run to recycle memory on time.

# Giving Priority to the RTGC

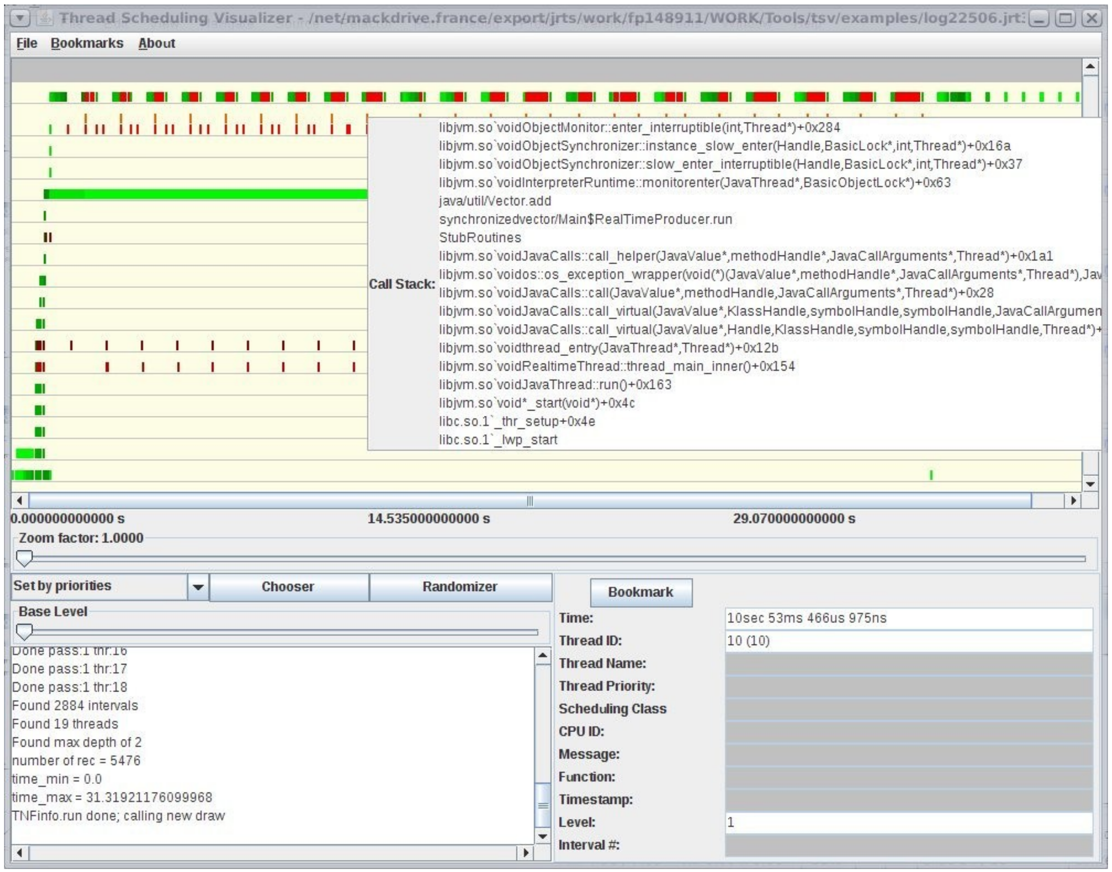



- RTGC runs at a higher priority than the application.
- RTGC runs with a single worker thread.
- Enough CPU time for the application.
- Memory recycled on time.
- No deadline miss.

# Shared locks and real-time scheduling

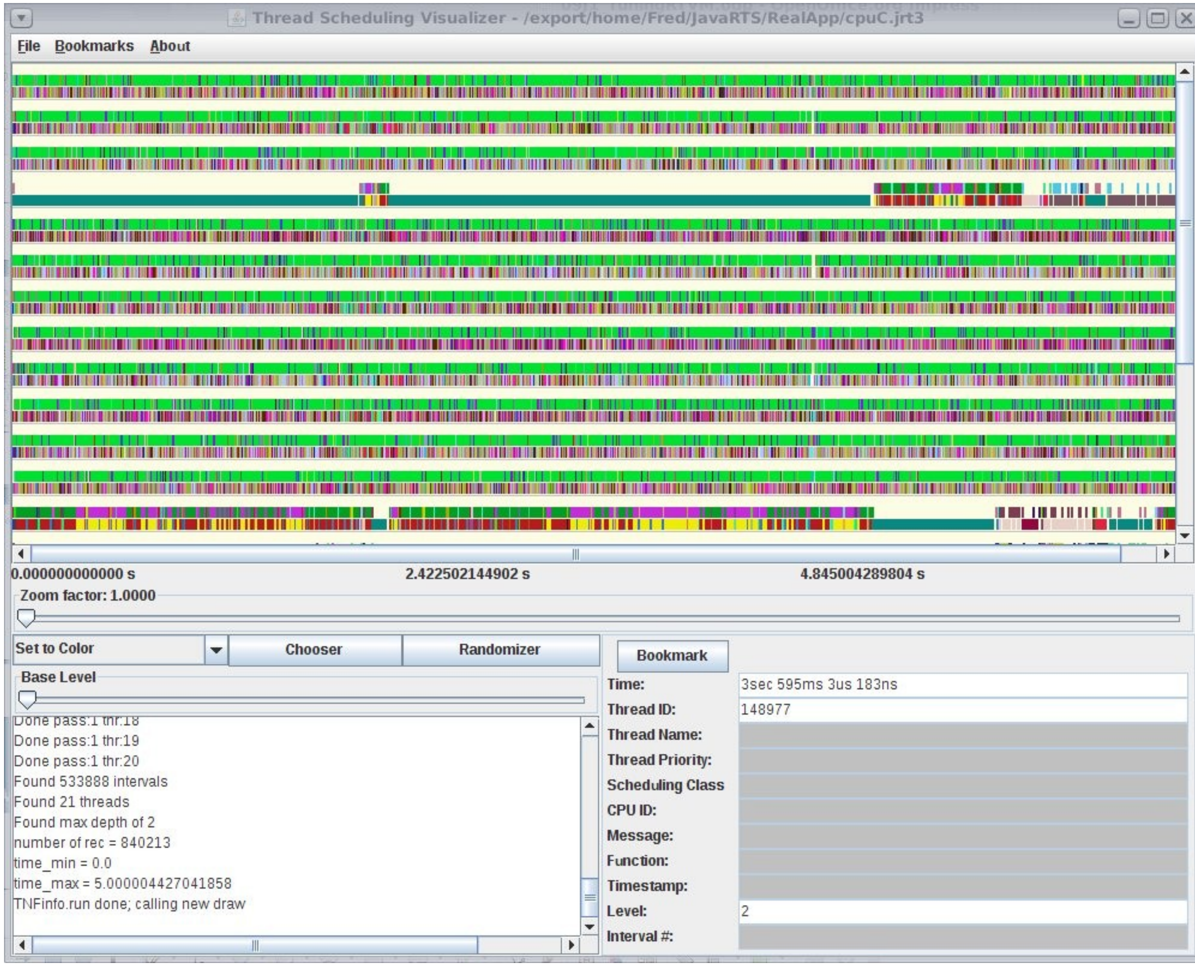
- Priority inversion
  - When a high priority thread A tries to acquire a lock held by a low priority thread B
  - Worse if a medium priority thread C preempts B and prevents it from running: *unbounded priority inversion*
- Priority Inheritance
  - When A blocks while acquiring the lock held by B then B is boosted to A's priority until it releases the lock
  - Solve the issue if both threads execute deterministic code
  - Danger: having real-time threads depending on non deterministic code executed at a real-time priority

# Tracking Application Locking Issues




- A non real-time thread blocks a real-time thread.
- The non real-time thread inherits from the real-time priority.
- Contentions cause deadline misses
- TSV provides the call stack when the contention occurs.

# Real Case Log



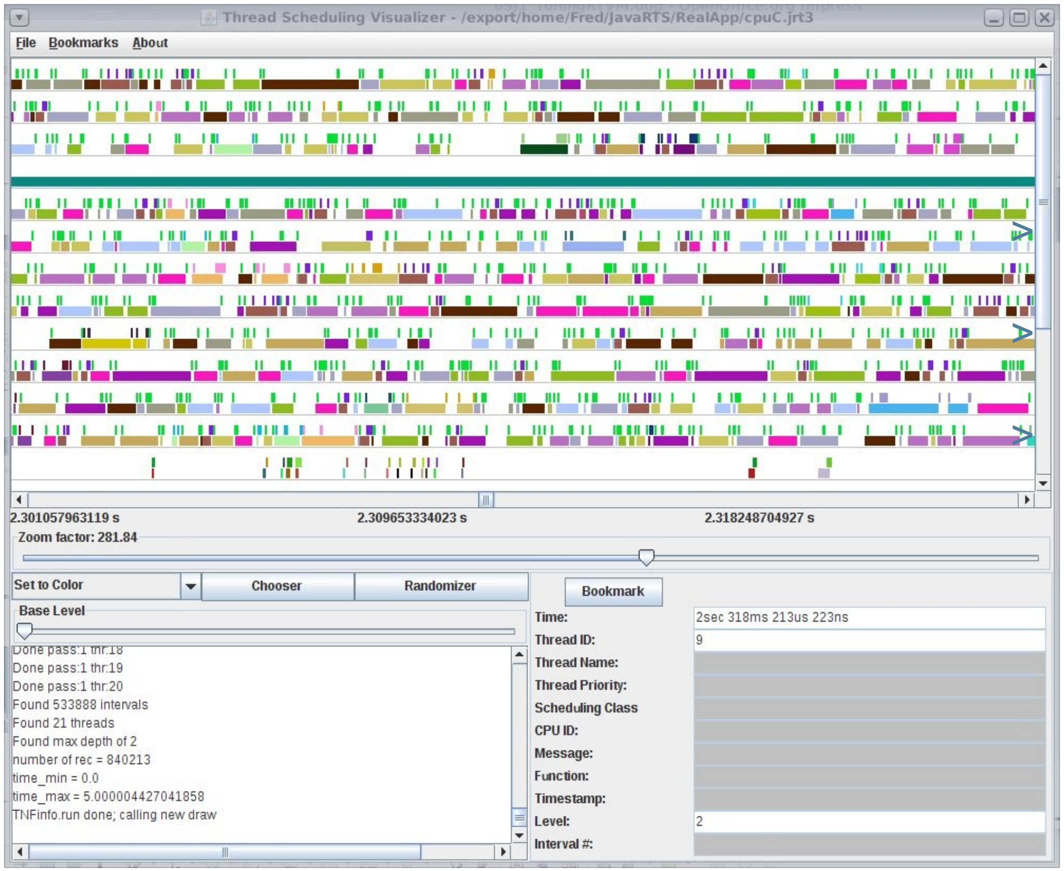

cores

)+ threads

ec of execution

CPU view

# Zoom into the Real Case Log




- Zooming in helps
- Still hard to analyze
- Data synthesis missing

# Log summary

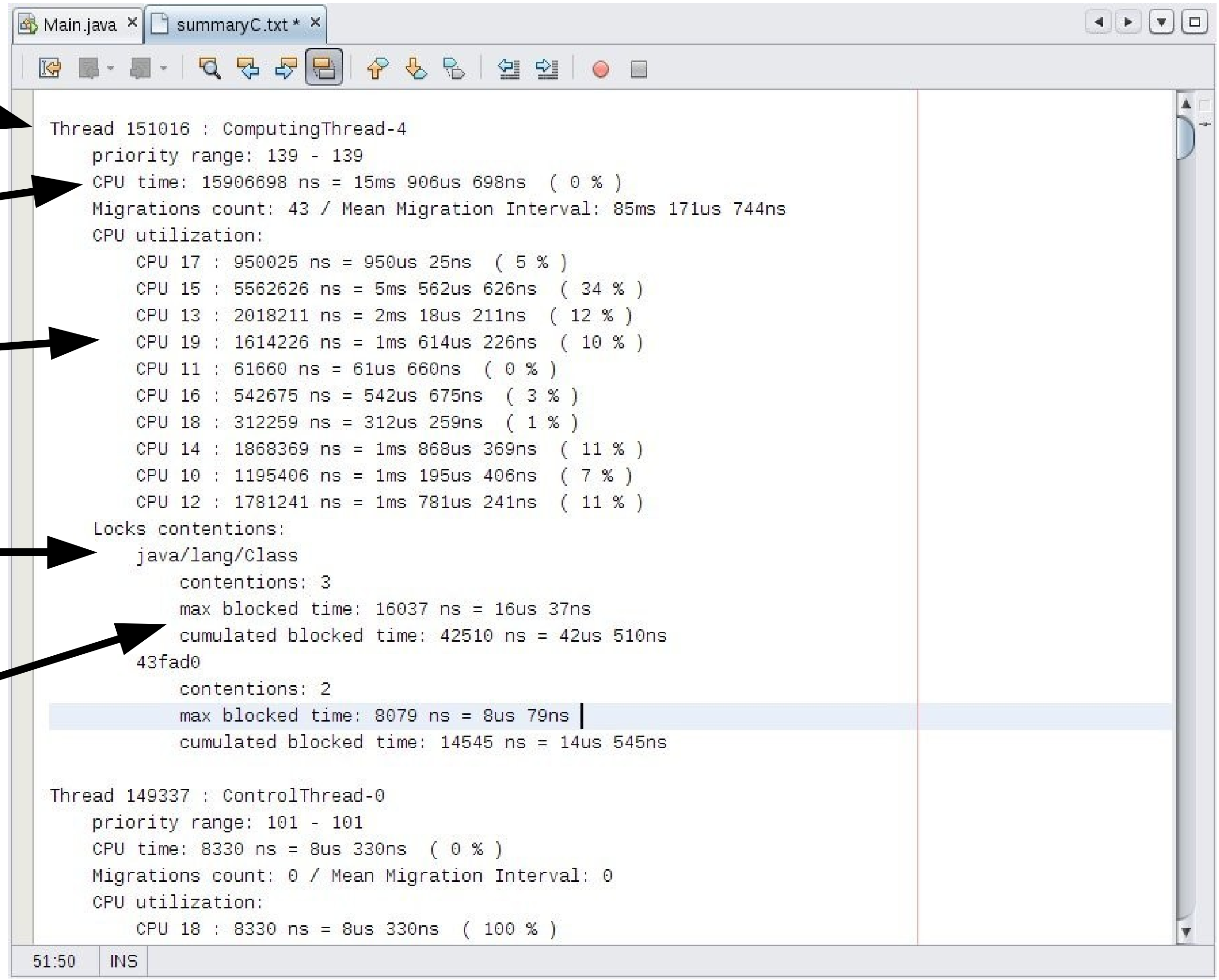
Per-thread summary

CPU time

Migrations

Contentions

Blocked times



```
Thread 151016 : ComputingThread-4
    priority range: 139 - 139
    CPU time: 15906698 ns = 15ms 906us 698ns  ( 0 % )
    Migrations count: 43 / Mean Migration Interval: 85ms 171us 744ns
    CPU utilization:
        CPU 17 : 950025 ns = 950us 25ns  ( 5 % )
        CPU 15 : 5562626 ns = 5ms 562us 626ns  ( 34 % )
        CPU 13 : 2018211 ns = 2ms 18us 211ns  ( 12 % )
        CPU 19 : 1614226 ns = 1ms 614us 226ns  ( 10 % )
        CPU 11 : 61660 ns = 61us 660ns  ( 0 % )
        CPU 16 : 542675 ns = 542us 675ns  ( 3 % )
        CPU 18 : 312259 ns = 312us 259ns  ( 1 % )
        CPU 14 : 1868369 ns = 1ms 868us 369ns  ( 11 % )
        CPU 10 : 1195406 ns = 1ms 195us 406ns  ( 7 % )
        CPU 12 : 1781241 ns = 1ms 781us 241ns  ( 11 % )
    Locks contentions:
        java/lang/Class
            contentions: 3
            max blocked time: 16037 ns = 16us 37ns
            cumulated blocked time: 42510 ns = 42us 510ns
        43fad0
            contentions: 2
            max blocked time: 8079 ns = 8us 79ns
            cumulated blocked time: 14545 ns = 14us 545ns

Thread 149337 : ControlThread-0
    priority range: 101 - 101
    CPU time: 8330 ns = 8us 330ns  ( 0 % )
    Migrations count: 0 / Mean Migration Interval: 0
    CPU utilization:
        CPU 18 : 8330 ns = 8us 330ns  ( 100 % )
```

# Outline

> Tuning for Real-Time
> Tuning Compilation
> Tuning Priorities
> **Tuning Memory Managers**
> The Future

# Memory Areas in JavaRTS

- Garbage-collected Heap
  - Tune the Real-Time Garbage Collector if needed
- Immortal memory consumption
  - Identify and remove Immortal Memory leaks
- Scoped memory recycling
  - Tune the size of each scoped memory area
  - Check when each area is reset

# Demo: Dumping Immortal Consumption on Thread Death

tput

ImmortalMemory (jar) × | **DTrace Monitored Execution: ImmortalMemory.jar [immortal.sh]** ×

```
Running: /home/bd148109/.netbeans/6.5/config/modules/ext/RealtimeTools/DTraceToolBox/immortal.sh /net

*** Error: not enough space in ImmortalMemory to allocate 10006  words (10928 bytes remaining) ***
Immortal allocations for thread   16 :       PeriodicRealtimeThread : 33364912 (last after 4002 ms)
Immortal allocations for thread   17 :               PeriodicThread : none
Immortal allocations for thread   15 :         CyclicCleaningThread : none
Immortal allocations for thread   11 :              CompilerThread0 : none
Immortal allocations for thread    8 :                    Finalizer : none
Immortal allocations for thread   10 :            Signal Dispatcher : none
Immortal allocations for thread    1 :                         main : 218648 (last after 14794 ms)
Immortal allocations for thread    9 :  Surrogate Locker Thread (CMS) : none
Immortal allocations for thread    7 :            Reference Handler : none
Immortal allocations for thread   12 :       LowMemoryDetectorThread : none

RUN SUCCESSFUL (Total time: 16sec 680ms)
```

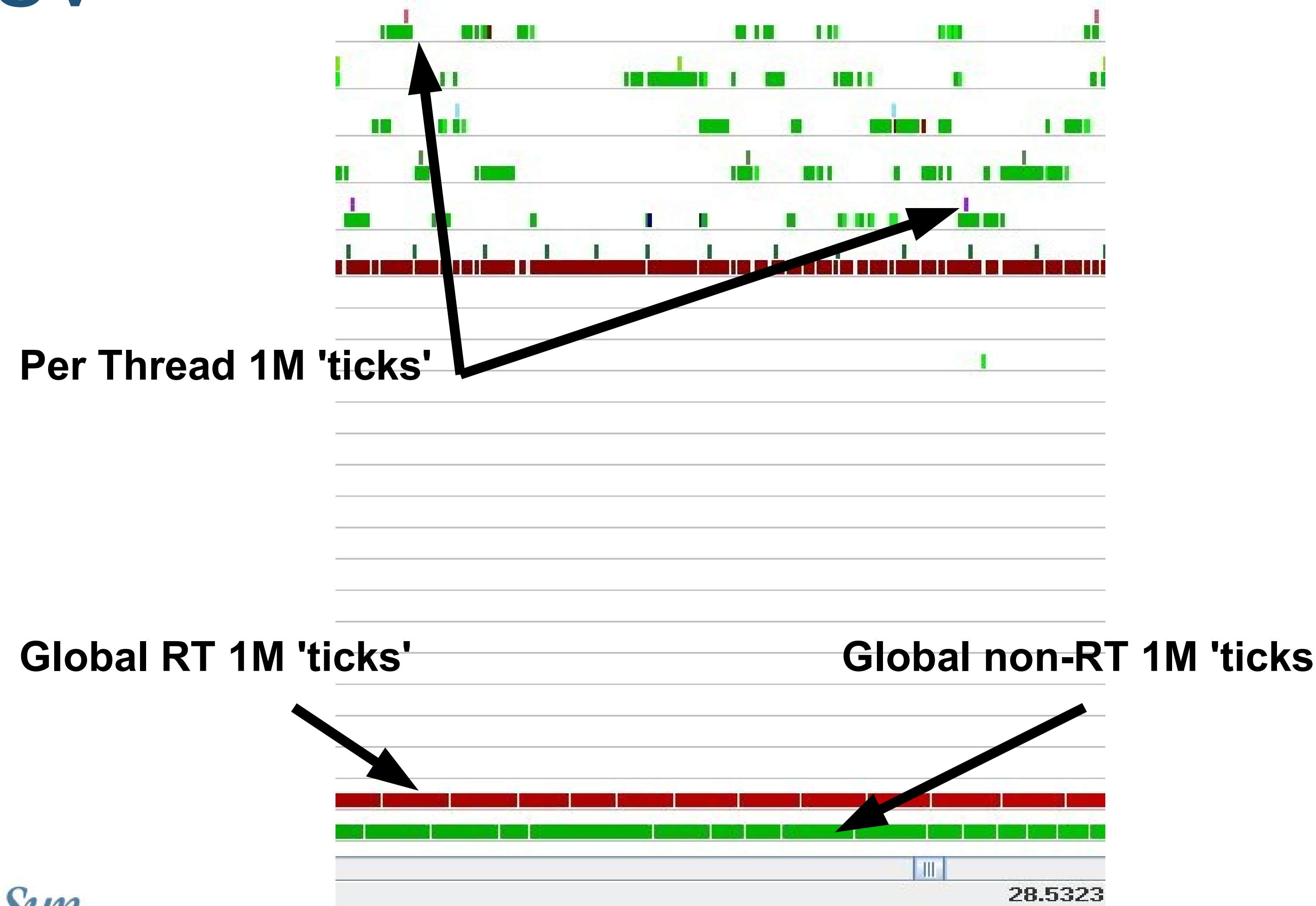
# Real Time Garbage Collector Monitoring

- RTGC threads are visible in TSV
  - Check how they compete with application threads
- Efficient per-thread memory consumption monitoring
- RTGC events available
  - Boosting of RTGC threads when memory falls low
  - Blocking non-critical threads when memory falls very low
  - Information for each GC cycle

# Tuning the RTGC for Soft Real Time

- Tuning the default number of GC threads
- Understanding soft real-time jitter
  - Using GC logs to check the auto-tuning
  - Using GC MBeans to (remotely) monitor the GC
  - Enhancing DTrace scripts or TSV view with GC statistics or memory consumption information
    - Improved GC logs
    - Correlate scheduling events with memory usage in TSV

# Demo: Visualizing Heap Consumption in TSV



# Tuning the RTGC for Hard Real Time...

- Evaluating behavior considering only hard real-time threads:
  - CPU consumption by hard RT threads
  - Time needed to execute two RTGC cycles with the remaining CPU power (two might be needed to guarantee the recycling of dead objects)
  - Memory consumed by hard RT threads during that time
- Deducing the memory limit under which non hard real-time threads must block (critical mode)
  - RTGCCriticalReservedBytes

# … and Tuning the Application

- Even if stable, the critical mode executes only parts of your application
- Tips:
  - Keep RTGCCriticalReservedBytes low to ensure the RTGC will recycle enough memory and unblock the non hard RT threads
  - Minimize the work in hard RT tasks to do only what needs to be done during the critical phase
  - Ensure RTGCCriticalReservedBytes is sufficient for these hard RT tasks

# Demo: Fine Grain Allocation Rate Monitoring

```
1115 K/ms (jlT   464 K/ms, softRT   433 K/ms, hardRT   218 K/ms)
1080 K/ms (jlT   433 K/ms, softRT   428 K/ms, hardRT   219 K/ms)
1085 K/ms (jlT   429 K/ms, softRT   438 K/ms, hardRT   217 K/ms)
1133 K/ms (jlT   485 K/ms, softRT   431 K/ms, hardRT   216 K/ms)
 998 K/ms (jlT   327 K/ms, softRT   447 K/ms, hardRT   222 K/ms) [GCing]
 839 K/ms (jlT   178 K/ms, softRT   438 K/ms, hardRT   222 K/ms) [GCing]
 892 K/ms (jlT   239 K/ms, softRT   431 K/ms, hardRT   221 K/ms) [GCing]
 901 K/ms (jlT   231 K/ms, softRT   447 K/ms, hardRT   222 K/ms) [GCing]
 867 K/ms (jlT   203 K/ms, softRT   441 K/ms, hardRT   222 K/ms) [GCing]
 879 K/ms (jlT   217 K/ms, softRT   438 K/ms, hardRT   222 K/ms) [GCing]
 852 K/ms (jlT   196 K/ms, softRT   432 K/ms, hardRT   222 K/ms) [GCing]
 857 K/ms (jlT   191 K/ms, softRT   440 K/ms, hardRT   224 K/ms) [GCing]
 819 K/ms (jlT   194 K/ms, softRT   400 K/ms, hardRT   223 K/ms) [GCing in critical mode]
 218 K/ms (jlT     0 K/ms, softRT     0 K/ms, hardRT   218 K/ms) [GCing in critical mode]
 810 K/ms (jlT   209 K/ms, softRT   381 K/ms, hardRT   219 K/ms) [GCing in critical mode]
 945 K/ms (jlT   294 K/ms, softRT   431 K/ms, hardRT   218 K/ms) [GCing]
 951 K/ms (jlT   285 K/ms, softRT   446 K/ms, hardRT   219 K/ms) [GCing]
```

# Outline

- Tuning for Real-Time
- Tuning Compilation
- Tuning Priorities
- Tuning Memory Managers
- **The Future**

# Benefiting from DTrace Improvements

> Graphical view of DTrace aggregates with Chime (already available on OpenSolaris)

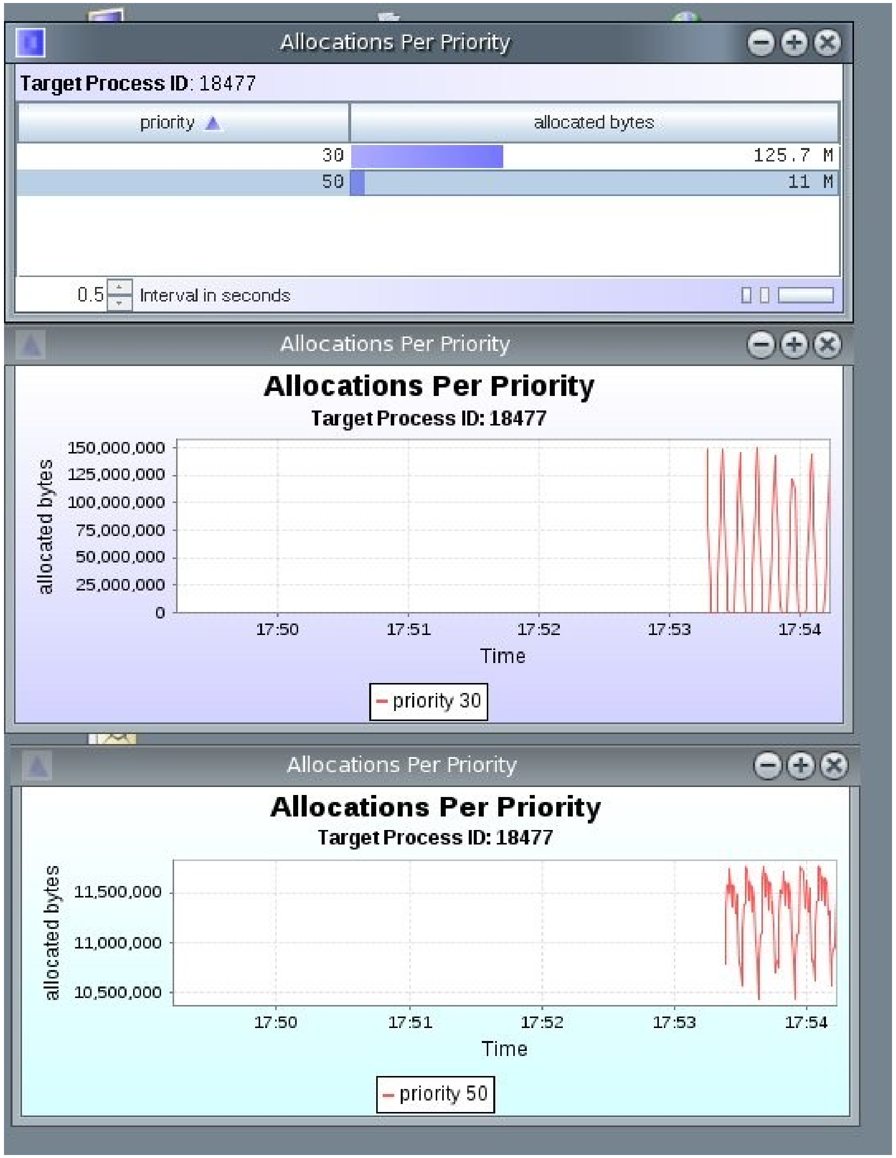
> Example: Tracking memory usage over time
  - Available Memory
  - Per-thread allocation rates
  - Per-priority allocation rate
  - ...

# Demo: Per Priority Memory Usage

**Consumption per priority (interval 0.5s)**

**Plotting for a soft real-time priority (varies from 25MB/0.5s to 150MB/0.5ms)**

**Plotting for a critical priority (varies from 10.5MB/0.5ms to 11.5MB/0.5ms)**



# Building other DTrace based tools

- Prototype of DTrace based sampling profiler
  - Very limited interference on the application
  - Used to successfully identify a performance degradation on a complex user application
- Now leveraging this efficient raw sampler to build a real profiler

# Real-Time Linux Support

- TSV is platform independent
- The issue is to gather the relevant data
  - Need user probes for JVM events
  - Need documented probes for scheduling events
- The Linux community is investigating different probe mechanisms
  - SystemTap, ftrace, utrace, Kprobes, ...

JavaOne[SM]

# Thank You

Bertrand Delsart
Bertrand.Delsart@Sun.COM
Frederic Parain
Frederic.Parain@Sun.COM

Sun Microsystems, Inc.