



JavaOne™

[java.sun.com/javaone](http://java.sun.com/javaone)

# D-I-Y (Diagnose-It-Yourself): Adaptive Monitoring for Sun Java™ Real-Time System

**Frédéric Parain, Olivier Lagneau, Carlos Lucasius**

Java Real-Time Group

Sun Microsystems, Inc.

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

TS-5716



# Agenda

- Need for a Real-Time Monitoring Tool
- Sun Java™ Real-Time System Solaris™ Operating System (Solaris OS) Dynamic Tracing Provider
- Test Cases
- Troubleshooting Tools
- Conclusion

# Need for a Real-Time Monitoring Tool

- Complexity of real-time scheduling
  - Run-to-block scheduling policy
  - Priority Inheritance
  - Real-Time Garbage Collection (RTGC)
- Tracking of fleeting events
  - Polling is unadapted
  - “Exact” monitoring
- Computation-oriented rather than method-oriented
  - CPU time / non-running time per period
  - Behavior variations
  - Priority

# The Observer Effect

## ➤ Definition

- Changes that the act of observing will make to the phenomenon being observed

## ➤ Inevitable in a real system

- Monitoring code needs:
  - Memory
  - CPU
  - Synchronizations

## ➤ Example with dynamic bytecode instrumentation

- **redefineClass**
  - New class loaded
  - Compilation of modified methods
  - De-optimization of running threads

# Observer Effect and Real-Time

- Acceptable in many non-real-time cases
  - Throughput measurement: pessimistic values
  - Memory consumption: over-estimate memory requirement
- Impact on real-time application behavior
  - Additional CPU consumption can cause deadline misses
  - Additional memory consumption can require RTGC reconfiguration
  - Code modification will impact compilation scheme (ITC)
- Consequence
  - Real-time application switching to error management mode

# WANTED



- Real-Time Monitoring tool
- Low overhead
- No synchronization
- Supported in production
- No Java technology-based heap allocations
- No Java code modification

# Agenda

- Need for a Real-Time Monitoring Tool
- Sun Java Real-Time System Solaris OS Dynamic Tracing Provider
- Test Cases
- Troubleshooting Tools
- Conclusion

# Solaris OS Dynamic Tracing (DTrace)

- DTrace components
  - Probes
  - Activation mechanism
  - D language
- Dynamic Tracing
  - No overhead if probes not enabled
  - Safe in production mode
  - Dynamic activation
  - Low overhead
- D language
  - Predicates
  - Aggregates
  - Speculative tracing



# DTrace Script Example

```
#!/usr/bin/dtrace -q
```

```
jrts$target:::thread-wfnp-exit
```

```
#!/arg0/
```

```
{
```

```
    deadlinemiss[tid]++;
```

```
}
```

```
jrts$target:::thread-end
```

```
/deadlinemiss[tid]>0/
```

```
{
```

```
    printf("Thread %d missed %d deadlines\n",
           tid, deadlinemiss[tid]);
```

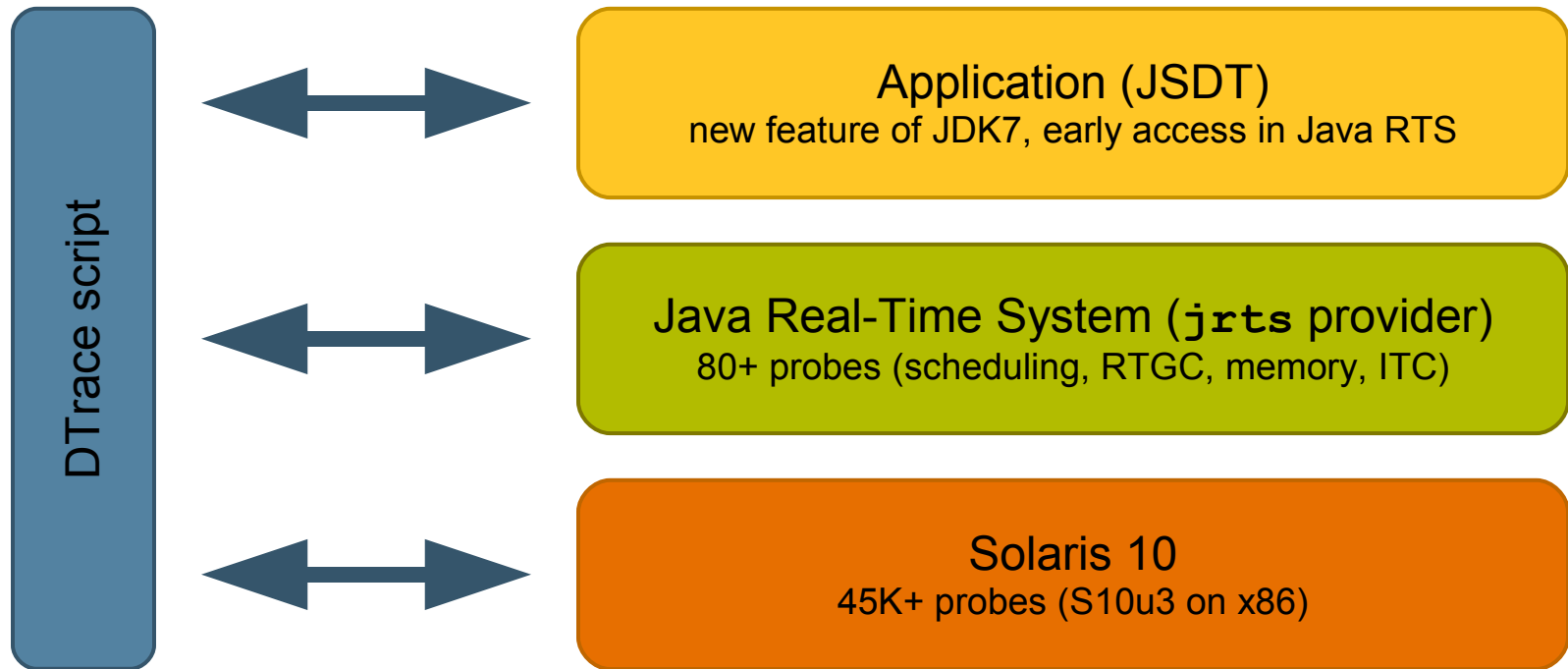
```
}
```

probe

predicate

clause

# DTrace with the Java Real-Time System (Java technology-based RTS)/ Operating System (Solaris OS) Stack

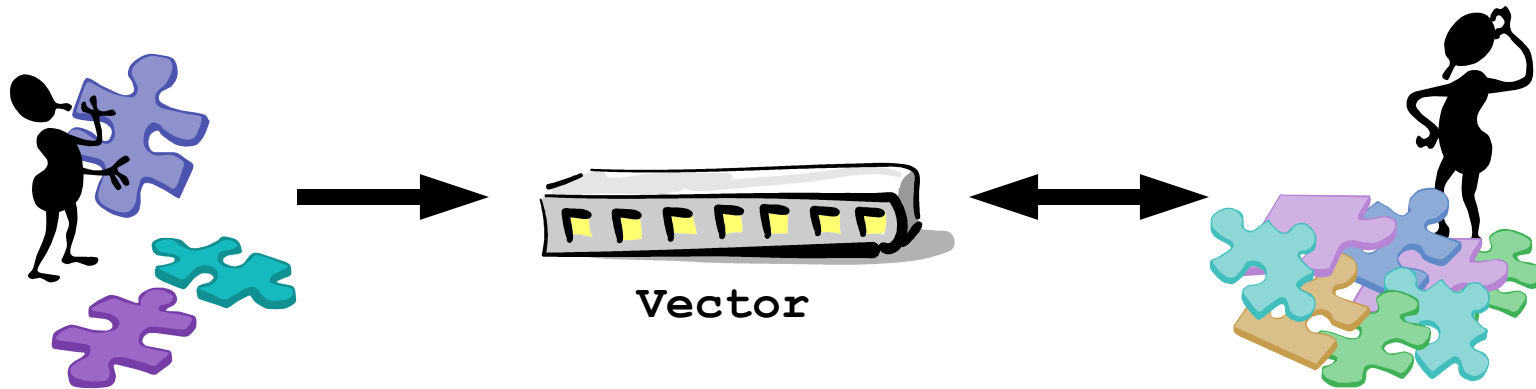


One script, one language, to monitor everything from the application to the OS

# Agenda

- Need for a Real-Time Monitoring Tool
- Sun Java™ Real-Time System Solaris OS Dynamic Tracing Provider
- Test Cases
  - Deadline Miss Analysis
  - Profiling
- Troubleshooting Tools
- Conclusion

# First Test Case: Deadline Miss Analysis



RealTimeProducer  
RealtimeThread  
Period: 500ms  
Adds 10 values to the  
Vector at each period

ConsumerThread  
java.lang.Thread  
Loop:  
Sort Vector's data  
Sleep ~500ms

Problem: The RealTimeProducer thread misses some of its deadlines.

# First Step: Scheduling Recording

## ➤ Objective

- Find a hint about the cause of the deadline misses

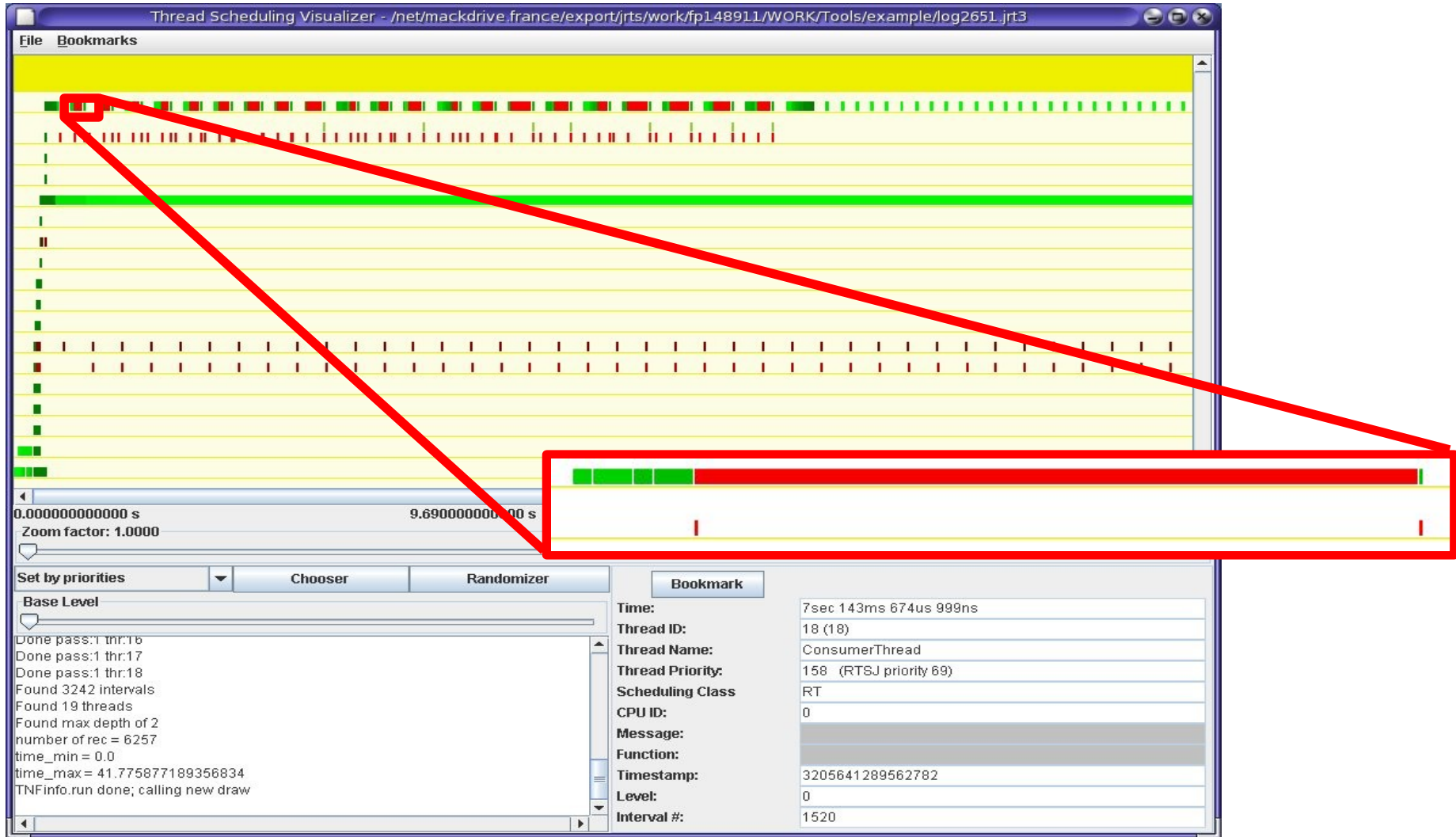
## ➤ DTrace monitoring script

- Scheduling events (threads getting and leaving a CPU)
- Priority changes
- Log of events generated for off-line analysis

## ➤ Scheduling visualization

- External tool
- Post-mortem analysis

# Second Step: Scheduling Visualization



# Third Step: Lock Contention Investigation

## ➤ Objective

- Confirm the hypothesis of the lock contention issue
- Identify the problematic lock

## ➤ New DTrace script

- Focused on the RealtimeProducer thread
  - `jrts$target:::thread-start`
  - Predicate: `/thr_id == tid/`
- Tracking lock contention
  - `jrts$target:::monitor-contended-enter`
  - `jrts$target:::monitor-contended-entered`
- Tracking priority boosting
  - `sched:::change-pri`
- Deadline miss notification
  - `jrts$target:::user-event`

# Too many events!

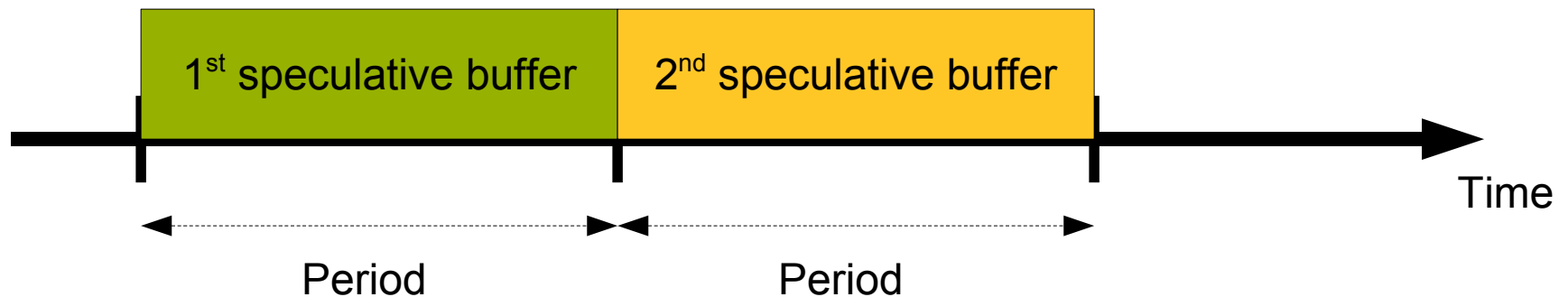
```
3642445575887862 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c10d8
3642445575932205 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
3642445796606949 : [JRTS] RealTimeProducer enters a contended monitor (after 220719 microseconds)
3642446575892966 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c10d8
3642446575943118 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
3642446767605273 : [JRTS] RealTimeProducer enters a contended monitor (after 191712 microseconds)
3642447575882614 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c10d8
3642447575924705 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
3642447753314551 : [JRTS] RealTimeProducer enters a contended monitor (after 177431 microseconds)
3642448575929562 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c10d8
3642448575986579 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
3642448779819289 : [JRTS] RealTimeProducer enters a contended monitor (after 203889 microseconds)
3642449575885886 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c10d8
3642449575936830 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
3642449821475798 : [JRTS] RealTimeProducer enters a contended monitor (after 245589 microseconds)
3642450575887460 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c10d8
3642450575939881 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
3642450875563014 : [JRTS] RealTimeProducer enters a contended monitor (after 299675 microseconds)
3642451575890636 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c10d8
3642451575944081 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
3642451924810593 : [JRTS] RealTimeProducer enters a contended monitor (after 348919 microseconds)
3642452575889523 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c10d8
3642452575940020 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
```



# Fourth Step: Speculative Tracing

## ➤ Objective:

- Reduce the output to the faulty periods



- Beginning of the period: creation of a speculative buffer
- During the period: all events are logged into this buffer
- At the end of the period:
  - If no deadline miss occurred, buffer is discarded.
  - If a deadline miss is detected, buffer is flushed to the output.

# Focused Output from Speculative Tracing

## > Output from faulty periods

Deadline miss at iteration 14

```
3216035761543004 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c2a78
3216035761604293 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
3216036277599464 : [JRTS] RealTimeProducer enters a contended monitor (after 516056 microseconds)
3216036278548215 : [USER] RealTimeProducer is notifying: Deadline miss at iteration 14
```

Deadline miss at iteration 23

```
3216040261484198 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c2a78
3216040261532375 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
3216040791348997 : [JRTS] RealTimeProducer enters a contended monitor (after 529864 microseconds)
3216040791528654 : [USER] RealTimeProducer is notifying: Deadline miss at iteration 23
```

- > The class of the lock is known.
- > Is it possible to get the method name?
  - Call stack inspection: `jstack()`

# Final Step: Call Stack Inspection

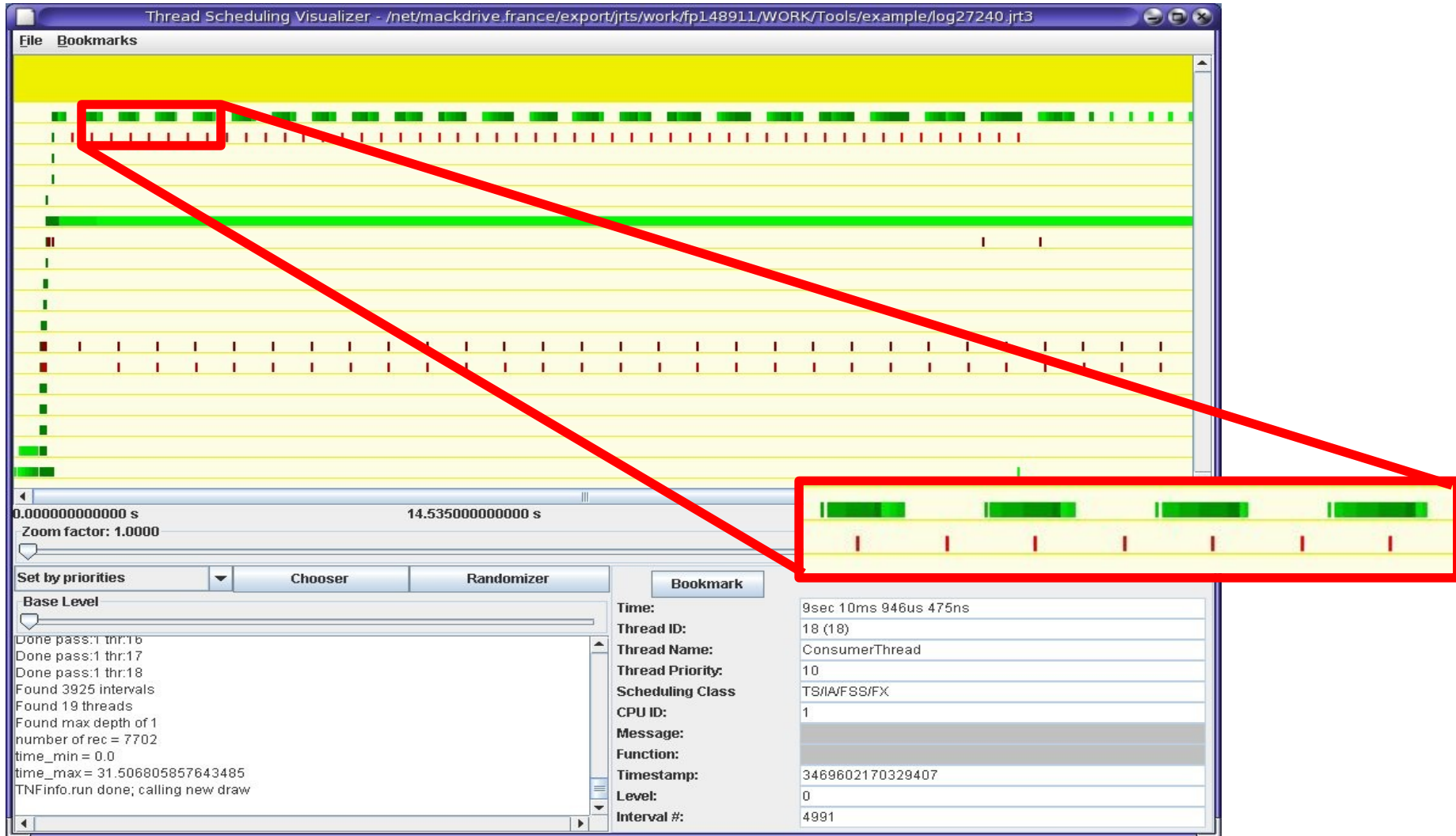
Deadline miss at iteration 16

```

3554713806699954 : [JRTS] RealTimeProducer tries to enter contended monitor java/util/Vector@81c2a78
    libjvm.so`void ObjectMonitor::enter_interruptible(int,Thread*)+0x284
    libjvm.so`void ObjectSynchronizer::instance_slow_enter(Handle,BasicLock*,int,Thread*)+0x16a
    libjvm.so`void ObjectSynchronizer::slow_enter_interruptible(Handle,BasicLock*,int,Thread*)+0x37
    libjvm.so`void InterpreterRuntime::monitorenter(JavaThread*,BasicObjectLock*)+0x63
    java/util/Vector.add
    synchronizedvector/Main$RealTimeProducer.run
    StubRoutines (1)
    libjvm.so`void JavaCalls::call_helper(JavaValue*,methodHandle*,JavaCallArguments*,Thread*)+0x1a1
    libjvm.so`void os::os_exception_wrapper(void(*) (JavaValue*,methodHandle*,JavaCallArguments*,Thre
    libjvm.so`void JavaCalls::call(JavaValue*,methodHandle,JavaCallArguments*,Thread*)+0x28
    libjvm.so`void JavaCalls::call_virtual(JavaValue*,KlassHandle,symbolHandle,symbolHandle,JavaCall
    libjvm.so`void JavaCalls::call_virtual(JavaValue*,Handle,KlassHandle,symbolHandle,symbolHandle,T
    libjvm.so`void thread_entry(JavaThread*,Thread*)+0x12b
    libjvm.so`void RealtimeThread::thread_main_inner()+0x154
    libjvm.so`void JavaThread::run()+0x163
    libjvm.so`void*_start(void*)+0x4c
    libc.so.1`_thr_setup+0x4e
    libc.so.1`_lwp_start

3554713807556194 : [SOLARIS] RealTimeProducer is changing ConsumerThread's priority to 158
3554714315214592 : [JRTS] RealTimeProducer enters a contended monitor (after 508514 microseconds)
3554714316098828 : [USER] RealTimeProducer is notifying: Deadline miss at iteration 16
  
```

# Scheduling of the New Code



# First Test Case Solved

- Cause of deadline misses found
  - Synchronized method of class `Vector`
  - Replacement of the `Vector` instance by a `WaitFreeWriteQueue` instance solves the issue
- What has been achieved with DTrace
  - Overview of the application scheduling
  - Tracking of Java application environment events: lock contention on Java technology-based object
  - Focused tracing on faulty behavior using speculative tracing
  - Inspection of call stacks including Java code

## Second Test Case: Profiling

- Real-Time measurements are often “computation-based” and not “method-based”
- Execution time has two components:
  - CPU time
  - Non-running time (waiting, blocked, preempted)
- DTrace script to profile periodic execution
  - Measure execution for each periodic execution (whatever methods are called)
  - Measure elapsed time and CPU time
    - `timestamp`, `vtimestamp`
  - Display results with distribution graphs
    - `@exectime[tid] = quantize(end[tid]-begin[tid]);`

# Profiling : Test Case Description

- Same computation code executed in three different contexts:
  - High Priority **RealtimeThread**
    - High Real-Time priority
    - using `waitForNextPeriod()`
  - Low Priority **RealtimeThread**
    - Low Real-Time priority (same priority as RTGC)
    - using `waitForNextPeriod()`
  - **java.lang.Thread**
    - Time-Sharing priority
    - Emulating `waitForNextPeriod()` with `sleep()`
    - Using user-events to emulate `waitForNextPeriod()` DTrace probes

# Execution Times per Period

Per thread, expressed in microseconds

## HighPriorityRealtimeThread

value	Distribution	count
22000		0
23000	@	200
24000		0

## LowPriorityRealtimeThread

value	Distribution	count
22000		0
23000	@	195
24000		1
25000		0
26000		0
27000		1
28000		1
29000		0
30000		2
31000		0

## RegularJavaThread

value	Distribution	count
22000		0
23000	@	178
24000		1
25000		0
26000		1
27000		2
28000	@	6
29000		0
30000		0
31000		1
32000		0
33000		0
34000		0
35000		0
36000		0
37000		0
38000		0
39000		0
40000		0
41000		0
42000		0
43000		0
44000		0
45000		0
46000	@	6
47000	@	5
48000		0

➤ Each thread shows a different behavior





# Blocked+Preempted Times per Period

Per thread, expressed in microseconds

## HighPriorityRealtimeThread

value	Distribution	count
< 0		0
0	@@@	200
1000		0

## LowPriorityRealtimeThread

value	Distribution	count
< 0		0
0	@@@	196
1000		0
2000		0
3000		1
4000		1
5000		0
6000		2
7000		0

## RegularJavaThread

value	Distribution	count
< 0		0
0	@@@	179
1000		0
2000		1
3000	@ @	8
4000		0
5000		0
6000		1
7000		0
8000		0
9000		0
10000		0
11000		0
12000		0
13000		0
14000		0
15000		0
16000		0
17000		0
18000		0
19000		0
20000		0
21000		0
22000		0
23000	@ @	11
24000		0

➤ Differences come from the scheduling

# Agenda

- Need for a Real-Time monitoring tool
- Sun Java™ Real-Time System Solaris OS Dynamic Tracing Provider
- Test cases
- Troubleshooting Tools
- Conclusion

# Java RTS Troubleshooting Tools

- Java 2 Platform, Standard Edition (J2SE™) 5.0 Serviceability Agent tools have been modified:
  - Need for memory-related statistics (**jmap**) and stack traces (**jstack**)
  - They suspend the live process and real-time behavior is unavailable.  
To be used on core files (generated using **gcore**).
  - **jmap**: memory information on heap, immortal, scoped areas.
  - **jstack**: stack trace and nature of all type of threads provided.  
(Thread, RealtimeThread, NoHeapRealtimeThread)

# Memory Areas Dump

- Jmap dumps memory objects in Hprof binary format
  - “`jmap -heap:format=b`” generates `heap.bin` file that follows Hprof binary format.
  - All memory areas are dumped : Heap, Immortal memory and Scoped memory.
  - `heap.bin` can be browsed using `jhat` or `hat` ([hat.dev.java.net](http://hat.dev.java.net)), or even VisualVM.

# jstack Sample Output

**javax.realtime.RealtimeThread** t@18: (state = IN\_JAVA)

- Fibonacci2.computeFib() @bci=17, line=35  
  (Interpreted frame)
- Deterministic.computeFibs(int, int) @bci=35, line=99  
  (Interpreted frame)
- Deterministic\$RealTimeFibonacciLoops.run() @bci=90,  
  line=184 (Interpreted frame)

**java.lang.Thread** t@1: (state = BLOCKED)

- java.lang.Object.wait(long) @bci=-977304266  
  (Interpreted frame)
- java.lang.Object.wait(long) @bci=0 (Interpreted frame)
- java.lang.Thread.join(long) @bci=38, line=1302  
  (Interpreted frame)
- java.lang.Thread.join() @bci=2, line=1355  
  (Interpreted frame)
- Deterministic.main(java.lang.String[]) @bci=1428,  
line=409           (Interpreted frame)

# jmap -heap Sample Output

## Immortal memory block:

```
capacity = 33554432 (32.0MB)
used      = 33554424 (31.99MB)
free      = 8 (8B)
99.99997615814209% used
```

## RT Collected Heap:

```
capacity = 67108864 (64.0MB)
used      = 67101344 (63.99MB)
free      = 7520 (7.34375KB)
99.98879432678223% used
```

## Scoped memory block 1:

```
capacity = 16777216 (16.0MB)
used      = 7838344 (7.47MB)
free      = 8938872 (8.52MB)
46.720170974731445% used
```

## Scoped memory block 2:

```
capacity = 16777216 (16.0MB)
used      = 7788872 (7.43MB)
free      = 8988344 (8.57MB)
46.42529487609863% used
```

## Unallocated scoped memory chunk:

```
size = 33554432 (32.0MB)
```

## Total unallocated scoped memory:

```
size = 33554432 (32.0MB)
```



Immortal Memory

RT Heap

Scoped  
Memory 1

Scoped  
Memory 2

# VisualVM Heap Dump Screenshot

core.13203 x

Overview threaddump-1205430939323.tdump x heapdump-1205431066244.hprof x

## Core dump core.13203 (pid 0)

Heap dump

Summary Classes Instances

### Classes

Class Name	Instances [%]	Instances	Size
<b>Fibonacci1</b>	3537 (22%)	56592 (5%)	
<b>Fibonacci2</b>	3537 (22%)	56592 (5%)	
<b>char[]</b>	1825 (11%)	192150 (17%)	
java.lang. <b>String</b>	1751 (11%)	42024 (4%)	
<b>int[]</b>	858 (5%)	187868 (18%)	
<b>short[]</b>	824 (5%)	38284 (3%)	
<b>byte[]</b>	574 (4%)	145543 (13%)	
java.lang. <b>Object[]</b>	333 (2%)	12884 (1%)	
java.lang. <b>Integer</b>	276 (2%)	3312 (0%)	
java.util. <b>HashMap\$Entry</b>	218 (1%)	5232 (0%)	
java.util. <b>Hashtable\$Entry</b>	153 (1%)	3672 (0%)	
java.lang. <b>String[]</b>	132 (1%)	3044 (0%)	
java.lang. <b>StringBuilder</b>	115 (1%)	1840 (0%)	
java.util.regex. <b>Pattern\$Ctype</b>	105 (1%)	1680 (0%)	

[Class Name Filter]



# Conclusion

- DTrace is a fantastic tool to monitor the Solaris OS/Java RTS stack
  - Low impact on real-time behavior
  - Generic scheduling overview
  - Focused tracing
  - Precise measurement
- It's up to you to write THE script that will solve your problem
  - Your imagination is the limit
- But sometimes DTrace is not enough
  - Troubleshooting tools updated for Java technology-based RTS
    - **jstack**
    - **jmap**
    - Visual VM

## Other Sessions of Interest

- LAB-7420: (Tue. 10:50am)
  - The Real-Time Programming Challenge on the Java? Platform: How to Build Real-Time Solutions for Real-World Devices
- TS-5767: (Tue. 4:40pm)
  - Real-Time Specification for Java (JSR 1): The Revolution Continues
- TS-5609: (Wed. 2:50pm)
  - Real Time: Understanding the Trade-offs Between Determinism and Throughput
- BOF-6353: (Wed. 7:30pm)
  - Meet the Sun Java RTS and SE Embedded Team
- BOF-5223: (Thu. 7:30pm)
  - VisualVM: Integrated and Extensible Troubleshooting Tool for the Java Platform

# THANK YOU



## D-I-Y (Diagnose-It-Yourself): Adaptive Monitoring for Sun Java Real-Time System

Frederic Parain, Olivier Lagneau, Carlos Lucasius

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

TS-5716

