Real-Time Java
in the
Cornell Electron Storage Ring

Devin Bougie (dab66) *

May 10, 2011

*Advised by Dr. Graeme Bailey and Research Support Specialist Michael Forster
# Contents

1 Introduction 3

2 Java and Real-Time 3
   2.1 Unpredictability in Standard Java 4
   2.2 The Real-Time Specification for Java (RTSJ) 5
   2.3 RTSJ Implementations 9
   2.4 Real-Time Linux Tools 9

3 Java Real-Time System (Java RTS) 10
   3.1 Features 10
   3.2 Testing Environment 13
   3.3 Concerns 14

4 Measuring Determinism 15
   4.1 Programs 15
   4.2 No-Heap Realtime Threads 17
   4.3 Analysis 17
      4.3.1 NonDeterministic 18
      4.3.2 Deterministic 20
      4.3.3 NHRTDeterministic 21
      4.3.4 GCDeterministic 22
      4.3.5 NHRTGCDeterministic 24

5 CESR Control Programs 25
   5.1 Bakeout Control Simulation 26
   5.2 Analysis 28

6 Conclusion 30
   6.1 Java RTS 30
   6.2 Real-Time in CESR 31

7 Resources 31

8 Source Code 32
   8.1 ClockResolution.java 33
   8.2 NHRTGCDeterministic.java 34
   8.3 BakeoutSim.java 41

9 Bibliography 45
1 Introduction

From 1979 to 2008, the Cornell Electron Storage Ring (CESR) collided electron and positron beams, producing a spray of particles observed by the CLEO detector and studied by the CLEO collaboration. With the end of CLEO, CESR was converted to a Test Accelerator (CesrTA) to perform crucial R&D for the International Linear Collider (ILC), specifically investigating damping ring operations and reliability.

In addition to its role in particle physics, synchrotron radiation from CESR has been used as an intense source of X-rays for the Cornell High Energy Synchrotron Source (CHESS). The Energy Recovery Linear accelerator (ERL) would serve as an upgrade to CHESS, and is a proposal by Cornell University physicists to use the CESR ring and a superconducting linac to create a new type of accelerator and advanced light source capable of creating very short and frequent pulses of high coherence and high energy x-rays.

Cornell researchers are in the process of migrating CESR’s control system from the VMS operating system to Linux, which requires porting an immense amount of code from Fortran on VMS. As a part of this effort, CESR is investigating the new technologies and programming languages available on Linux. Of particular interest is the Real-Time Specification for Java, a specification for extending the Java Language and Virtual Machine specifications to allow for real-time programming.

This project will look into the Real-Time Specification for Java, using the Java Real-Time System implementation, as a platform for CESR control system programs.

2 Java and Real-Time

Real-time programming, which guarantees deterministic, predictable response to real-world events, has typically been relegated to embedded systems running on specialized hardware, operating systems and languages. Recent developments to the Java language and Linux kernel (and Solaris operating system), however, have made deterministic real-time behaviour obtainable on standard commodity hardware and operating systems.

Real-time computing is the study of systems that are subject to real-time temporal constraints, such as operational deadlines. Therefore, real-time computing is about predictability - the knowledge that the system will always perform within the required time frame. When creating the Real-Time Specification for Java, the developers defined real-time computing as follows: [14]

"The programming environment must provide abstractions necessary to allow developers to correctly reason about the temporal behaviour of application logic. It is not necessarily fast, small, or exclusively for industrial control. It is all about the predictability of the execution of application logic with respect to time."

There are two main classes of real-time systems. A hard real-time system is one in which all deadlines must be met without fail. In addition, a soft real-time system is one that will still function correctly, according to its specification, if the system occasionally misses a deadline. Soft real-time systems tend to be more flexible and typically specify how often
or what percentage of deadlines can be missed. In soft real-time systems, some latency is acceptable, but overall predictability is still mandatory. [7]

Many hard real-time systems are used to protect humans from injury or danger, and can be further classified as safety-critical systems. Safety-critical java is being studied through Java Specification Request (JSR) 302 and should be considered separate from the RTSJ. [10]

Determinism, the expectation that a cause will produce a desired effect, is the most basic requirement of a real-time system. Determinism can be measured in terms of jitter - the variation in response time for a particular computational scenario. While perfect determinism would be represented by zero jitter, the aim of real-time programming is to reduce jitter to an acceptable, bounded level.

2.1 Unpredictability in Standard Java

Building real-time applications is challenging, as you must guarantee your code meets its deadlines predictably and deterministically. While the performance of Java Standard Edition (Java SE - now at version 6 Update 24) has improved greatly over the years, [21] a number of factors in standard Java can add variability to the timing of execution and insert unbounded latency, breaking determinism. The most common sources of jitter when using Java in a non-real-time context include:

- **Garbage collection.** Garbage collection is the process of automatically freeing objects that are no longer referenced by an executing Java program. While this saves the programmer from having to keep track of when to free allocated memory, it is also the primary source of unpredictability in standard Java.

  The garbage collection algorithms in standard Java Virtual Machines (JVMs) involve a *stop-the-world pause*, allowing the garbage collector to run without interference from application threads. This can cause unacceptable latency as critical application threads are paused at unpredictable intervals and for unpredictable amounts of time during garbage collection.

- **Scheduling.** Although threads are created in Java by the JVM, they are ultimately scheduled by the operating-system scheduler. Therefore, the operating system must provide scheduling latency guarantees. This requires advanced capabilities such as high-resolution timers, program-defined low-level interrupts, and a robust priority-based scheduler.

- **Priority inversion.** Java SE lacks strict priority-based dispatching of threads, thus preventing guaranteed deterministic behaviour. Priority Inversion occurs when the scheduler downgrades the priority of a higher-priority thread to that of a lower-priority thread. For example, this can happen when a high-priority thread shares a resource that is locked by a lower-priority thread. Not only can the higher-priority thread block until the lower-priority thread finishes, but a third medium-priority thread that does not depend on the shared resource could become scheduled before both the high and low-priority threads complete. This further delays the execution of the low-priority thread, and thus, further delays the execution of the high-priority thread.
• **Class loading, initialization, and compilation.** In standard Java, classes are initialized when an application first uses them, which can create a variance in latency the first time that class is used. In addition, the Java specification allows classes to be loaded lazily, which can require disk or network access to locate the class definition. Finally, Java SE uses the just-in-time (JIT) compiler, and can decide when, if ever, to translate a class from byte code into native code. All of this can introduce unexpected delays throughout the execution of an application.

• **The application.** The application itself, including the libraries it uses, can also introduce unpredictability in standard Java. As thread priorities are typically unused in standard Java (and the standard JVM offers such weak guarantees for thread priorities), tasks can be delayed if the available CPU cycles become exhausted.

• **Asynchronous Events.** Finally, other high-priority activities outside of the JVM, such as POSIX signals, hardware interrupts, other real-time applications, etc., can stop the execution of the application and introduce jitter and unpredictability.

2.2 The Real-Time Specification for Java (RTSJ)

The Real-Time Specification for Java (RTSJ) was the first Java Specification Request. While its initial release was in 2001, RTSJ was most recently updated to Version 1.0.2 in 2006. [5]

The RTSJ specifies how Java systems should behave in a real-time context, and introduces several new features to support real-time operations. It is designed to extend any Java family, and requires all implementations to pass both the JSR 1 technology compatibility kit (TCK) and the TCK of the underlying platform (Java SE, Java ME, or Java EE). [9]

Real-time features in the RTSJ include:

• **Real-Time Garbage Collection.** Real-time garbage collection gives the predictability needed in a deterministic environment, and can be considered another complete subsystem of the RTSJ provided by individual RTSJ implementations. Real-time garbage collectors provide means of ensuring critical application threads are never paused during garbage collection.

• **Tasks and Deadlines.** The RTSJ models the real-time components of an application as a set of tasks, each with an optional deadline. These tasks can be classified based on the predictability of their frequency and timing:

  – **Periodic** tasks run on a fixed schedule, such as reading a sensor every one millisecond.

  – **Sporadic** tasks have a maximum frequency, but do not run on a fixed schedule.

  – **Aperiodic** tasks have an unpredictable frequency and timing.

Using these task types, the RTSJ provides several mechanisms for ensuring that critical deadlines are not missed. For example, the RTSJ allows tasks to be associated with a **deadline miss handler** that is called if a task does not complete before its deadline. This can be used to take corrective action or report performance and behavioural
information. Deadline-miss handling can be performed by the thread itself or deferred to a dedicated deadline miss handler thread.

- **Thread Priorities.** While no system can guarantee that all tasks will complete on time, a real-time system can use priorities to guarantee that lower-priority tasks will miss their deadlines before higher-priority tasks. RTSJ provides at least 28 strictly enforced priority levels, all of which are above the maximum priority level available to non-real-time tasks. RTSJ implementations rely on a real-time operating system to properly support multiple priorities and allow higher-priority threads to pre-empt lower-priority threads.

RTSJ requires *priority inheritance* to avoid *priority inversion* by boosting the priority of a thread that is holding a lock to that of the highest-priority thread also waiting for that lock. This both prevents a higher-priority thread from being blocked by a lower-priority thread that isn’t given adequate CPU cycles, and a medium-priority thread that does not depend on the shared resource from pre-empting both the higher and lower-priority thread.

As the RTSJ allows for both non-real-time and real-time activities to coexist within a single application, temporal guarantees are provided depending on the type of thread that is executing:

- **java.lang.Thread (JLT)** threads can use the 10 priority levels specified by the `Thread` class, but provide no guarantees of temporal execution.

- **javax.realtime.RealtimeThread (RTT)** threads can take advantage of the stronger thread priority support offered by RTSJ. The real-time scheduler will only pre-empt an RTT for another RTT of higher priority. By simply replacing `java.lang.Thread` with `javax.realtime.RealtimeThread`, you obtain soft real-time behaviour with a real-time garbage collector, priority inversion avoidance, and strict and precise priority dispatching semantics. To get predictability using a real-time garbage collector, however, you must understand something about your program’s memory usage. While this can be difficult, using command-line arguments you will tell the garbage collector how much garbage your program should produce and how much memory must be reserved for critical RTTs.

  When creating RTT’s, you typically specify the thread’s priority (using a `PriorityParameters` object) and period (using a `PeriodicParameters` object). By calling `RealtimeThread.waitForNextPeriod` instead of `Sleep`, you use a high-resolution timer to ensure the temporal behaviour specified by the `PeriodicParameters` used when creating the RTT.

- **javax.realtime.NoHeapRealtimeThread (NHRT)** threads obtain determinism by using memory outside of the regular Java heap. All-though they are more difficult to program, NHRT’s provide the best determinism out of the three execution contexts provided by the RTSJ and are typically used in hard real-time environments. [11]

  In standard Java, a high-priority thread can become blocked when the garbage collector attempts to free memory allocated by a lower-priority thread. Even when
using RTT’s and the RTSJ, a critical RTT thread can be blocked if the real-time garbage collector is not configured properly and memory becomes exhausted. The RTSJ provides a `NoHeapRealtimeThread` (NHRT) subclass of `RealtimeThread` that is protected from jitter induced by garbage collection. NHRTs maximize predictability and achieve hard real-time by avoiding any use of or references to the garbage-collected heap. Code executing within an NHRT cannot allocate from the heap or reference objects that reside in the heap, and the NHRT itself cannot exist within the heap.

Developers must be careful when using the increased priorities provided by the RTSJ. As critical real-time threads will not be interrupted by any external process, it is possible to saturate a system with un-interruptable threads by creating as many critical real-time threads that execute continuously and never pause as there are CPUs in a system. As experienced several times during initial development, this can frequently require hard resets of a system to recover. If, for example, you forget to call `RealtimeThread.waitForNextPeriod()` in a critical real-time thread, you ensure the thread will continuously run and never be interrupted by any OS process. Because of this, attempting to run as many of these threads as there are CPU’s in a system will effectively cause a “denial of service” condition and hang the system. While the critical RTT threads will continue to execute, no other system processes will gain CPU time and the system will require a hard reboot. To avoid this, the thread’s period needs to be correctly specified or the thread needs to be instructed to pause periodically, giving other threads a chance to execute.

- **Memory Areas.** The RTSJ allows objects to be allocated from various memory areas, each of which have different garbage collection characteristics and allocation limits. In addition to the standard heap, the RTSJ defines two main types of non-heap memory that are outside of the control of the garbage collector, *immortal memory* and *scoped memory*. Objects in these areas and their memory are not reclaimed automatically, so you must manage these areas and associated objects yourself.

The three memory areas provided by the RTSJ include:

- **Standard heap:** Real-time JVMs maintain a garbage-collected heap which can be used by both standard JLT and RTT threads. Although threads allocating from the standard heap are subject to pauses during garbage collection, several different garbage collector technologies provided by RTSJ implementations can balance throughput, scalability, determinism, and memory size.

- **Immortal Memory:** Immortal memory represents a paradigm shift in object allocation and usage, but is crucial for achieving hard real-time behaviour. The RTSJ provides a single immortal memory region that is not subject to garbage collection. Objects allocated from immortal memory will never be reclaimed throughout the lifetime of the JVM. In general, immortal memory is used to avoid dynamic allocation by statically allocating ahead of time and managing all of the required memory. While this provides for a great degree of determinism,
immortal memory is a finite resource, and great care must be taken when working in immortal memory.

To use immortal memory, one can:

* create objects within it from a RTT.
* provide a thread implementing the Runnable interface to enter the immortal memory region.
* explicitly create a RealtimeThread, passing the ImmortalMemory region as a constructor to the RTT.

- **Scoped memory:** Scoped memory is only available to RTT and NHRT threads, and is intended for objects with a known lifetime. It provides a form of memory budgeting. Scoped memory regions are defined and created in your application (specifying the maximum size of the scoped area), and ensure that a rogue task does not over-consume memory and starve other, perhaps higher-priority tasks. Scoped memory is preallocated when a task starts, and is automatically freed when the task completes. Therefore, when properly used allocations will not fail and deallocations are fast and predictable.

  Scoped memory is an object instantiated in your code and used to create other objects. When all threads executing within a scoped area exit, all of the objects within the scoped area are de-allocated immediately. In addition, scoped memory areas can be nested with a parent-child relationship.

  As long as real-time tasks can accurately estimate their memory requirements, scoped memory can eliminate the unpredictability of temporary object allocation. As a result, scoped memory regions are useful when creating temporary objects, when executing time-critical code in response to an event, or for a periodic timer where you clearly understand the allocation needs of the associated code.

- **Advanced Communication Between Threads.** As the RTSJ allows for both real-time and non-real-time activities to coexist within a single VM, there must be methods of communicating between these threads without impacting the determinism of the real-time threads. One obvious method of communicating between threads is to use a queue, where one thread puts data onto the queue while another thread removes data from the queue. The RTSJ supports several types of non-blocking queues for communicating between threads. For example, the WaitFreeReadQueue is designed for a single real-time reader to perform non-blocking reads while multiple writers perform synchronized writes. Likewise, the WaitFreeWriteQueue is designed for a real-time writer to perform non-blocking writes while multiple readers perform synchronized reads.

- **Asynchronous Events.** In typical environments, asynchronous events, such as POSIX signals and hardware interrupts, are handled outside of the application process. The RTSJ avoids these interruptions and prevents the operating system from stealing CPU cycles from real-time applications by bringing them into the visibility of the scheduling and dispatching mechanisms.
2.3 RTSJ Implementations

There are a handful of commercial real-time JVMs that conform to the RTSJ and provide Real Time Garbage Collection. The most prominent offerings include:

- **The Java Real-Time System (Java RTS)** is currently free for academic use, making it the only viable option for CESR at this stage. Commercial pricing starts at $6500 per socket/CPU for development or internal deployment. [19] Java RTS 2.2, released in September 2009, is an implementation of Java SE version 1.5.0_20 with RTSJ extensions, and is compatible with RTSJ 1.0.2. Although there is a Java RTS NetBeans plug-in, the real-time libraries can be added to any project, even when developing on a non-real-time system.

  http://Java.sun.com/javase/technologies/realtime/index.jsp

- IBM’s **WebSphere Real Time** provides full support for Java SE 6.0 and RTSJ 1.0.2. Prices start at $8,075.00 per “Processor Value Unit”. [13]

  http://www-01.ibm.com/software/webservers/realtime/

- **JamaicaVM**, from aicas GmbH, also supports RTSJ 1.0.2 and is intended for use on real-time and embedded systems. While pricing for JamaicaVM is not publicly available, the general sentiment seems to be that organizations using these systems (for example, in avionics and defence) “haven’t been too concerned about licensing costs, to date. They are concerned by much higher program risks.” [4]

  http://www.aicas.com/

2.4 Real-Time Linux Tools

In addition to the real-time RT-PREEMPT additions to the Linux kernel, various tools can be used to ensure determinism on a real-time system.

- **Real-Time Java Virtual Machine**: The real-time JVM will provide various ways for you to develop and debug real-time applications. As will be seen later in this report, the `-XX:+RTGCPrintStatistics` option for the Java RTS VM can be extremely useful in understanding the memory requirements of and ensuring determinism for your application. In addition, the `RealtimeThread.getCurrentMemoryArea()` method can be used to view an object’s current memory area.

- **cpuset**: Cpusets can be used to bind processes to specific sets of processors.

- **taskset**: `taskset` is used to set or retrieve the CPU affinity of a running process or new command. Using `taskset`, processes can be bound to specific processors or sets of processors.

  In addition, you can bind just `RealtimeThread` or `NoHeapRealtimeThread` instances within your application to specific processors using, for example, the `-XX:RTSJBindRTTToProcessors` or `-XX:RTSJBindNHRTToProcessors` options to the real-time Java VM.
• **Tuna**: Tuna is a program used to change attributes of Linux threads (scheduling policy, scheduler priority and processor affinity) and Linux interrupts (processor affinity) on a running system. Tuna can be used (in addition to `taskset`), to assign processes and threads to specific CPUs. In addition, Tuna can be used to prevent operating system interrupts from being scheduled onto specific CPUs.

3 **Java Real-Time System (Java RTS)**

The Java Real-Time System (Java RTS), originally developed by Sun Microsystems, is a Java virtual machine implementation that is compliant with the Real-Time Specification for Java 1.0.2, and can meet both soft and hard real-time requirements. Java RTS provides the runtime, APIs, and tools necessary to reason about and control the temporal behaviour of your Java application and ensure that no system process will interrupt your real-time java threads. Java RTS maintains compatibility with standard Java SE 5, so non-real-time applications and components can execute (albeit slightly slower) alongside and share data with real-time applications.

Tests have shown Java RTS can achieve minimum latencies of 15 microseconds with around 5 microseconds of jitter. [7]

Java RTS runs on Red Hat MRG Linux and Novell’s SUSE Linux Enterprise Real-Time Extension (both POSIX real-time Linux distributions), as well as Solaris 10. In addition to these officially supported operating systems, Java RTS can run on the latest Linux kernel with the `RT-PREEMPT` patches installed.

To benefit from the real-time, deterministic characteristics in Java RTS, threads must execute within a real-time thread. This can be done by creating a class that extends the `Runnable` interface and create a `javax.realtime.RealtimeThread` (RTT) object in place of a `java.lang.Thread` (JLT). Alternatively, your class can extend `RealtimeThread` directly, and set your RTT’s priority and other optional real-time characteristics before starting it like any other thread. This gives you the option of setting the priority and starting the RTT in a constructor, hiding the details of the real-time implementation from the caller.

If your code must perform processing on regular time intervals, through its high-resolution timer facility, Java RTS supports periodic real-time threads that ensure your RTT is released at precise points. This can be invoked by creating a `PeriodicParameters` object, setting the time period, setting it as your RTT’s release parameter, and then calling `RealtimeThread.waitForNextPeriod` instead of `Sleep`.

Finally, Java RTS allows you to set thread priorities, giving precise control over the order of processing within your application. By default, all RTTs execute at a higher priority than the garbage collector, eliminating the unpredictability and most latency associated with Java SE’s collector.

3.1 **Features**

In summary, some of the highlights and benefits of Java RTS include:

• **Java SE 5 compatibility**: Java RTS maintains compatibility with, and executes standard Java applications alongside real-time applications.
• **RealtimeThread thread class.** Java RTS provides the `javax.realtime.RealtimeThread` thread class for soft real-time scheduling and the real-time garbage collector.

• **NoHeaprealtimeThread thread class.** By not using the heap, NHRT’s avoid garbage collection and can be used for hard real-time scheduling and synchronization.

• **Thread Priorities.** Java RTS provides 28 new strictly enforced priority levels, the lowest of which is above all non-real-time thread priorities and the highest of which will never be interrupted under ordinary circumstances.

• **Immortal Memory.** Immortal memory in Java RTS is a pre-defined area of memory that is never destroyed until the JVM exits. While immortal memory provides the highest level of determinism, it is a finite resource and must be used with care. Objects and memory in immortal memory are never reclaimed throughout the lifetime of the virtual machine. Java RTS provides two means of using immortal memory.
  
  – Immortal Memory can be specified as the memory area in a `Schedulable` object’s constructor.
  
  – Alternatively, you can obtain a reference to the `ImmortalMemory` singleton object, call its `enter` method, and provide a `Runnable` object as the parameter.

• **ScopedMemory.** Java RTS allows for the creation of scoped memory areas which are automatically destroyed when threads exit the scope. This allows your application to control precisely when objects are created and destroyed while avoiding garbage collection.

• **Real-time clock.** Java RTS implements the RTSJ Clock API and provides jitter-free time operations and access to a high-resolution clock with nanosecond accuracy. Real-time timer objects represent points in time with the best possible accuracy and precision of the underlying hardware, and can be used to create timers that fire periodically or one-time only.

• **Physical memory access.** Java RTS provides safe direct physical memory access, allowing device driver code to be written in Java.

• **Real-Time Garbage Collector.** Java RTS supports two garbage collectors. The non-real-time serial garbage collector suspends application processes in order to safely free memory, while the fully concurrent, parallel, mark-and-sweep Real-Time Garbage Collector (RTGC) can be configured to execute deterministically without interrupting critical threads running at a higher priority level.

Java RTS doesn’t maintain heap spaces or generations, so the entire heap is one large space where application threads and RTGC threads operate in unison. RTGC threads work at a lower priority than application RTTs, thereby preventing RTTs from interruption by the RTGC or from being dispatched behind RTGC threads when competing for CPU cycles.

Despite the lack of stop-the-world pauses, some per-thread pauses can occur during the RTGC’s marking phase when it scans an application thread’s stack. This amounts
to a maximum of less than 200 microseconds of latency, and overall, the RTGC adjusts its own running characteristics to ensure enough free memory is continually produced and critical application threads are never interfered with. [6]

To guarantee determinism with garbage collection, `RTGCCriticalReservedBytes` needs to be specified to reserve some memory for the critical RTTs that must execute deterministically. This points to the need to balance the number of critical threads and CPU usage so that the RTGC will have sufficient free CPU cycles to work with. The `RTGCCriticalReservedBytes` is used to protect critical threads from allocations by non-critical threads. Only critical threads will be able to continue memory allocation when the critical reserved bytes threshold is reached. Even then, they will only continue to perform deterministically if the RTGC can recycle memory fast enough to keep up with the allocation rate of the critical threads.

- **Static Initialization.** The standard Java VM normally initializes classes then they are first used in a program. Without being able to predict when classes are initialized, or how long that initialization will take, you cannot guarantee determinism. By specifying a list of classes to be initialized before the application starts executing (or asking Java RTS to generate this list for you), you can avoid this common source of jitter.

- **Compilation.** The standard Java VM uses Just-in-Time (JIT) compilation to compile methods when certain internal counters reach specified limits. While this is designed to trigger compilation at optimal times, in a real-time environment this introduces an unacceptable source of jitter. To address this, Java RTS provides Initialization-Time Compilation (ITC) which allows you to specify methods to be compiled at initialization, reducing or eliminating jitter caused by run-time (JIT) compilation. This list of methods can be specified manually or generated automatically by Java RTS.

- **Asynchronous Events.** Asynchronous event handlers handle external events and allow you to schedule your application’s response without disrupting the temporal integrity of the system. Asynchronous transfer of control allows you to quickly terminate a thread or safely transfer control from one thread to another.

The Java RTS provides an asynchronous event handler (AEH) facility designed for system and application specific events that a real-time application may need to handle. Two main classes make up the AEH facility in Java RTS.

- `AsyncEvent` objects represent the event itself, and don’t contain application-specific event data.

- `AsyncEventHandler` is a `Schedulable` object that is executed by the AEH facility within Java RTS when the related event is fired. These objects are populated with `ReleaseParameters`, `SchedulingParameters`, and `MemoryParameters` which indicate how the event handler should be scheduled. When an associated event object is fired, the AEH objects `handleAsyncEvent` method is executed deterministically at a real-time priority.
3.2 Testing Environment

For testing the Java Real-Time system, I used a “cluster” of five systems, three connected at 1Gbps and two connected to a low-latency 10Gbps switch. These systems comprise a variety of generations of commodity desktops, workstations, and servers.

In order of age, the test systems include:

- The two 10Gbps systems are 1U custom servers from Red Barn Computers. They each use two quad-core Intel(R) Xeon(R) X5550 CPUs @ 2.67GHz and 12GB of 800MHz DDR3 RAM.

- One Lenovo e20 Workstation with a single dual-core Intel(R) Core(TM) i5 650 CPU @ 3.20GHz and 4GB of 1066MHz DDR3 RAM.

- One Dell Precision Workstation T3400 with a single dual-core Intel(R) Core(TM)2 Duo E7300 CPU @ 2.66GHz and 1GB of 667MHz DDR2 RAM.

- One 2U custom server from Red Barn Computers with two dual-core Intel(R) Xeon(TM) CPUs @ 3.46GHz and 4GB of 667MHz DDR2 RAM.

As demonstrated by the attached ClockResolution.java, the hardware clock on each of these systems has a resolution of one nanosecond. For example, here is the output of running ClockResolution.java on the 1U Red Barn server that will be used to demonstrate determinism in our next section.

```
[dab66@ilc326 ~] % /opt/java/jrts/bin/java -jar /opt/java/dist/ClockResolution.jar
Real-time clock resolution = (0 ms, 1 ns)
RTT executing with a period of 2ms ...

(1 ms, 997522 ns)
(1 ms, 997612 ns)
(1 ms, 997946 ns)
(1 ms, 997671 ns)
(1 ms, 997917 ns)
(1 ms, 997574 ns)
(1 ms, 997886 ns)
(1 ms, 997933 ns)
(1 ms, 997533 ns)
(1 ms, 998420 ns)
(1 ms, 997321 ns)
(1 ms, 997809 ns)
(1 ms, 997579 ns)
(1 ms, 997867 ns)
(1 ms, 997843 ns)
(1 ms, 997874 ns)
(1 ms, 997689 ns)
```

Each system runs fully-updated Scientific Linux 5.5 (based on RedHat Enterprise Linux 5U5) with the MRG Realtime packages from the Scientific Linux CERN (SLC) 5 distribution,
giving the 2.6.33.7rt29.47.el5rt kernel. Each system has a local copy of the latest Java RTS 2.2 (based on Java SE 1.5.0_20) and the latest Java SE 1.6.0_24 provided by SL (RHEL).

For improved determinism, the /tmp directory, where Java RTS keeps some of its temporary files, is mounted with the tmpfs in-memory file system. Finally, each system grants a realtime user-group the limits necessary for running real-time applications.

While the installation of the MRG Realtime packages and Java RTS 2.2 were relatively straight-forward, there are some remaining issues that would need to be resolved (or better understood), before relying on this setup in a production system. For example, the real-time kernel appears to break the am-utils amd automount daemon which is currently used to mount many NFS filesystems. NFS mounts do work, however, when mounted statically or when mounted using the autosfs autmounter. In addition, while based on the enterprise-class open-source MRG Realtime components available from Red Hat, the packages from CERN are explicitly labeled as “for TEST PURPOSES ONLY.” [20]

3.3 Concerns

Most of the concerns with Java RTS revolve around the recent acquisition of Sun by Oracle. As with many of Sun’s projects, there is a tangible fear among the Java and RTSJ community as to the future of Oracle’s commitment to the RTSJ and Java RTS. Without a strong corporate backing, real-time Java is generally considered by the community to be too much of a niche market requiring too much engineering and certification to be adopted or used as an open source project. [3]

- The following footnote, currently on the Java RTS home page, illustrates the current state of uncertainty surrounding the future of Java RTS. [18]

Oracle is reviewing the Sun product roadmap and will provide guidance to customers in accordance with Oracle’s standard product communication policies. Any resulting features and timing of release of such features as determined by Oracle’s review of roadmaps, are at the sole discretion of Oracle. All product roadmap information, whether communicated by Sun Microsystems or by Oracle, does not represent a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. It is intended for information purposes only, and may not be incorporated into any contract.

- When first starting this project, no reply was received after submitting a message to the Java RTS engineers at Oracle informing them of our project and asking for clarification on the Academic License.

- When beginning this project, the last “Java RTS” blog posting was over two years old. [16] When now visiting the Java RTS site, a message states ”Feed Not Available at this time”. [18]

- The only community forum available appears to be a general “RTSJ (Real-Time Specification for Java)” forum hosted at Oracle. This forum seems to be decreasing in
activity with only 10 postings in the first quarter of 2011, compared with approximately 23 in the first quarter of 2010 and 60 in the first quarter of 2009. Moreover, many of the postings to the forum appear to go unanswered. [1]

- The last official release of Java RTS was in September 2009. While there is documentation available online for a Java RTS 2.2u1, [2] the only available downloads are for 2.2. Another query to the Java RTS developers and a posting to the forum received no reply. According to an older posting on the RTSJ forum, as of October 2010 there were no plans for Java 6 support in Java RTS. [12]

- While great progress has been made and more can be expected, there are currently very useful tools available for debugging real-time applications in Solaris that are not available for Linux. For example, using Dtrace, the Sun Thread Scheduling Visualizer (TSV) records and displays the scheduling of all threads within the Java RTS VM. In addition, the JRTS VM contains a Dtrace provider which allows you to monitor the internal activities of the VM (thread scheduling, memory-related operations, compiler operations, the execution of the RTGC, etc.). [8]

4 Measuring Determinism

To measure the level of determinism, we calculate the elapsed time for each iteration of a sequence of calls that perform calculation and memory allocation. The jitter is then defined as the difference between the best and worst execution time throughout the test.

In addition to example programs provided by Java RTS 2.2 that demonstrate non-deterministic and deterministic behaviour using JLTs and RTTs, I created two more programs for comparison using NHRT’s. We run these programs through a series of scenarios that show the effect of configuration changes (such as thread priorities, length of time a scheduled thread pauses, load on CPU, memory allocation, and garbage collection, etc.) on the determinism of the programs. [15]

4.1 Programs

To demonstrate and compare the level of determinism in various scenarios, we begin with a set of programs that use system resources in a controllable way and measure execution time. While a wealth of information is created by these programs, the two most important values when measuring determinism are jitter and standard deviation on execution time.

For our tests, each program runs a set of threads that generate Fibonacci numbers, sleep for a specified time or wait for the next period, and optionally allocate memory to put a controlled load on the heap and stress the garbage collector. One of these threads, the bench thread calculates and records the time spent executing each iteration. These are instances of java.lang.Thread in the NonDeterminstic program, javax.realtime.RealtimeThread in the Deterministic and GCDeterministic programs, and javax.realtime.NoHeapRealtimeThread in the NHRTDeterministic and NHRTGCDeterministic programs. In addition, at most two other stress threads (one JLT and on RTT) execute simultaneously to place a controlled load on system resources.
Each program accepts the following parameters.

- **nb_outer_iterations**: an integer representing the number of outer iterations. For bench threads, we end up with `nb_outer_iterations` time measurements.

- **nb_inner_iterations**: an integer representing the number of inner iterations. Within each outer iteration, we execute the stress class or garbage producing class `nb_inner_iterations` times.

- **nb_stress_class_iterations**: an integer representing the number of iterations inside the stress class. For a thread calculating Fibonacci numbers, this represents the recursion depth, or in other words, the number of Fibonacci numbers to calculate. For a garbage producing thread, this represents the first dimension of the two-dimensional `int` and `float` arrays allocated inside the `GarbageProducer` classes.

- **allocation_array_size**: an integer representing the allocation array size. This is the second dimension of the two-dimensional `int` and `float` arrays allocated inside the `GarbageProducer` classes.

- **pause_time**: an integer representing the pause time in milliseconds inside of the outer loop. This defines the length of time each thread pauses between each time measurement in order to allow other threads (garbage collection, JIT compilation, OS processes, etc.) to execute.

- **priority_level**: the thread priority level, “max”, “norm”, or “min”. For RTT and NHRT threads, this is a real-time priority (starting from 11) which is obtained through the `getMaxPriority()`, `getNormPriority()`, and `getMinPriority()` methods. For JLT threads, the value is obtained from the three `MAX_PRIORITY`, `NORM_PRIORITY`, and `MIN_PRIORITY` constants from the `java.lang.Thread` class.

Finally, we use `taskset` to enable processor affinity and create cpusets, restricting the set of processors used by each test.

Our TestDeterminism package consists of five main programs:

- The first program provided by Java RTS, `NonDeterministic`, demonstrates the lack of determinism in non-real-time programs using regular JLT threads, even in situations where jitter from garbage collection is removed (by not allocating memory) and class initialization is avoided (by executing preliminary stress with the `Fibonacci` and `GarbageProducer` classes in the static initialization code).

- In addition, Java RTS provides a `Deterministic` program to demonstrate a real-time Java RTS program that does not consume memory in its steady state, thus avoiding the effects of garbage collection. `Deterministic` scenarios run free of garbage collection activity to show that in many cases determinism can be achieved by simply changing JLTs to RTTs, while the best determinism can be achieved using the ITC complication mode.
• Finally, a third `GCDeterministic` program can be used to show how to combine real-
time thread priorities with a properly configured Java RTS Real-Time Garbage Col-
lector to achieve determinism.

• In addition to these provided programs, I developed two additional programs to demon-
strate the use of `NoHeapRealtimeThread`’s to achieve hard real-time. While switching
from JLTs to RTTs is generally a simple change, it is not trivial to switch from RTTs
to NHRTs. Great care must be taken when working in immortal and scoped memory,
as the NHRT must not access the heap or exhaust its allocated memory region. Doing
so will result in a run-time error which can be very difficult to track down and fix.
The main way to find where your code attempts to access the heap from a NHRT is
through calls to `RealtimeThread.getCurrentMemoryArea()` on the objects used and
accessed by your NHRT.

`NHRTDeterministic` is meant to provide a comparison with the `Deterministic`
program, and demonstrates the use of NHRT threads in situations where programs do not
allocate memory in their steady state.
Finally, `NHRTGCDeterministic` can be used to show how to properly allocate objects in
and work with immortal memory, providing a comparison with the `GCDeterministic`
program described above.

4.2 No-Heap Realtime Threads
While working with No-Heap Realtime Threads can be extremely difficult and involved, there
are multiple ways to create NHRTs and work in immortal memory. As with all No-Heap
Realtime Threads, the challenge is to make sure you understand your memory requirements
and that the NHRT does not access any object in the heap in any way. None of the parameters
passed to the NHRT can reside in the heap, during execution the NHRT cannot access any
object in the heap, and the NHRT itself cannot be allocated in the heap.

`NHRTGCDeterministic.java`, attached to the end of this report, shows one successful
approach for creating NHRT threads for my `TestDeterminism` package. With this setup,
the bench thread runs in a NHRT while the stress threads continue to run either in a regular
JLT or RTT.

To begin, I create a regular `FibonacciLoops` class that implements `Runnable`. I then use
a `RealtimeThread` (allocated in immortal memory) in `main()` to run `FibonacciLoops`.

For this to work, a few class variables need to be made static and some of the local vari-
ables in `main()` need to be made final. For example, `wallclock_times_begin_loops` needs
to be static and defined in the constructor as: `wallclock_times_begin_loops=(long[]) ImmortalMemory.instance().newArray(long.class,nb_loops);`

4.3 Analysis
To demonstrate the advantages of Java RTS and the RTSJ, we run the `NonDeterministic`,
`Deterministic`, `NHRTDeterministic`, `GCDeterministic`, and `NHRTGCDeterministic` pro-
grams through various scenarios. The thread parameters used in each scenario are standard-
ized and only vary where specified.
In each of the following thread parameters, we use 1000 outer loop iterations and 10 inner loop iterations, with 10 stress class iterations. Therefore, with the Fibonacci calculations, we calculate 10 Fibonacci numbers 10 times, and measure this execution 1000 times. With the garbage producing threads, we allocate \( \text{int}[10][200] \) and \( \text{float}[10][200] \) two-dimensional arrays. We do not pause between any of the 1000 iterations. Finally all of the bench threads run at max priority and all of the stress threads run at min.

# Bench thread args – no allocations
BENCH_NOALLOC_ARGS='1000 10 10 0 0 max'

# Bench thread args – medium garbage allocations
BENCH_ALLOC_ARGS='1000 10 10 200 0 max'

# Non-existent Stress thread args
NO_STRESS_ARGS='0 0 0 0 0 min'

# Stress thread args with no allocations
STRESS_NOALLOC_ARGS='1000 10 10 0 0 min'

# Stress thread args with medium allocations
STRESS_ALLOC_ARGS='1000 10 10 200 0 min'

### 4.3.1 NonDeterministic

- **Run 1** - Only one bench JLT thread running at max priority with no pauses between iterations.

  Although not a realistic real-time application, this represents the ideal situation for obtaining determinism in a regular JLT. The bench thread is the only thread executing, and runs at max priority performing basic calculations without producing garbage. In addition, class initialization is avoided by executing preliminary stress with the Fibonacci classes in the static initialization code. However, the application is still subject to interruptions from external events. While there are no deviations over 500 microseconds and we achieve very good jitter and standard deviation, they are still not quite as low as we would like with an actual real-time application.

---

#### Summary of results:

Mean execution time: 81 microseconds
Best execution time: 80 microseconds
Worst execution time: 401 microseconds
Most frequent execution time: 79 microseconds, 446 occurrences
Execution time jitter: 321 microseconds
Standard deviation: 13.27 microseconds

- **Run 2** - Only one bench JLT thread producing garbage and running at max priority, with no pauses between iterations. This run is identical to **Run 1**, except that the bench thread also produces garbage that must be collected.
The dramatic decrease in determinism (increase in jitter and standard deviation) shows the effect of garbage collection pre-empting the bench thread, even though it is running at max priority. This results in a non-deterministic scenario with three outliers where the execution time deviates by more than 500 microseconds.

Summary of results:

Mean execution time: 148 microseconds
Best execution time: 129 microseconds
Worst execution time: 1748 microseconds
Most frequent execution time: 130 microseconds, 343 occurrences
Execution time jitter: 1619 microseconds
Standard deviation: 70.52 microseconds

• Run 3 - One bench JLT thread running at max priority, and one stress JLT thread running at min priority. Neither threads produce garbage to be collected. This is identical to Run 1 except for the addition of the stress JLT thread.

Again, this scenario shows that even without memory consumption, determinism cannot be guaranteed with such a simple Java program running in a JLT. While no garbage is collected in this run, we are unable to achieve determinism and see 150 iterations that deviate over 500 microseconds.

Summary of results:

Mean execution time: 158 microseconds
Best execution time: 80 microseconds
Worst execution time: 2087 microseconds
Most frequent execution time: 79 microseconds, 404 occurrences
Execution time jitter: 2007 microseconds
Standard deviation: 271.3 microseconds

• Run 4 - Using the Java RTS 2.2 VM, one bench JLT thread running at max priority, and one stress JLT thread running at min priority. Neither threads produce garbage to be collected.

This scenario, which is identical to Run 3 except that it use the JRTS VM, demonstrates two main points. First of all, non-real-time programs run without modification in the Java RTS VM. However, just running a non-real-time program on a real-time VM is not sufficient to obtain determinism. In fact, we receive very comparable runs when comparing the same non-deterministic scenario on the standard Java and on the Java RTS VM. While the execution time is slightly slower with the Java RTS VM, the overall determinism has improved as represented by the decrease in jitter and increase in number of occurrences of the most frequent execution time.
4.3.2 Deterministic

No garbage is produced in any of the Deterministic scenarios.

- **Run 1** - One bench RTT running at max priority, with one stress JLT running at min priority.

  This scenario demonstrates that in cases where there is no memory consumption (and thus no garbage collection), determinism can be obtained simply by replacing instances of `java.lang.Thread` with `javax.realtime.RealtimeThread`. This scenario is identical to NonDeterministic Run 4, except that the bench thread runs in an RTT instead of an JLT. From this simple change, we achieve excellent determinism with a jitter of only 43 and a standard deviation of 3.14. While the best execution time is unchanged when compared to NonDeterministic Run 4, the mean execution time is about twice as fast when using RTTs with Java RTS and the worst execution time is approximately 10 times as fast.

---

**Summary of results:**

Mean execution time: 109 microseconds  
Best execution time: 101 microseconds  
Worst execution time: 144 microseconds  
Most frequent execution time: 110 microseconds, 397 occurrences  
Execution time jitter: 43 microseconds  
Standard deviation: 3.14 microseconds

---

**Run 2** - One bench RTT running at max priority, with one stress RTT running at min priority.

This scenario is identical to Deterministic Run 1, except that the stress thread runs in a RTT instead of a JLT. As expected from the scheduling model of RTSJ, the addition of the stress RTT has no effect on the determinism of the bench thread. This is because the stress RTT (at min priority) will only execute when the bench RTT
(at max priority) is not. The output of this execution is virtually identical to that of Deterministic Run 1.

---

**Summary of results:**

Mean execution time: 109 microseconds  
Best execution time: 100 microseconds  
Worst execution time: 153 microseconds  
Most frequent execution time: 110 microseconds, 411 occurrences  
Execution time jitter: 53 microseconds  
Standard deviation: 3.32 microseconds

---

**Run 3** - Using ITC compilation mode with one bench RTT running at max priority and one stress RTT thread at min priority.

This scenario is identical to Deterministic Run 2 except that we use the ITC compilation mode instead of the standard JIT compilation mode. As we eliminate the main remaining source of jitter by using ITC compilation, we are sure to achieve the best determinism possible.

---

**Summary of results:**

Mean execution time: 108 microseconds  
Best execution time: 100 microseconds  
Worst execution time: 152 microseconds  
Most frequent execution time: 110 microseconds, 369 occurrences  
Execution time jitter: 52 microseconds  
Standard deviation: 3.35 microseconds

---

### 4.3.3 NHRDT
deterministic

- **Run 1** - Using ITC Compilation mode with one bench NHRT running at max priority and one stress RTT running at min priority.

This scenario is identical to Deterministic Run 3, except that the bench thread runs in a NHRT instead of a RTT. As no garbage is produced in this scenario, there are no tangible benefits to using a NHRT in ImmortalMemory over a critical RTT. However, the program does demonstrate the use of NHRT’s and ImmortalMemory, and shows that (at the sacrifice of increased complexity and decreased performance) we can achieve determinism using NHRTs just as we can with RTTs.

While our execution time is about 10 times slower than when using regular RTTs, the determinism is increased with jitter decreasing to 23 and standard deviation to 2.38.
4.3.4 GCDeterministic

- **Run 1** - Run 1 is identical to **Deterministic Run 3**, except that the stress RTT allocates memory and produces garbage that must be collected by the Real-Time Garbage Collector (RTGC).

  When analysing the output of this scenario, we see the garbage collector running during execution of the bench thread. Most importantly, the value of the “worst” amount of non-fragmented memory reaches 0. This indicates that we have exhausted the amount of memory available to our application. Moreover, since the “grand total” for “Bytes allocated by critical threads” is not zero, we see that there were, in fact, some critical threads that were blocked due to the memory saturation. Because of this, we do not achieve a deterministic run with two iterations that deviate more than 500 microseconds and an unacceptable amount of jitter with a high standard deviation.

- **Run 2** - This scenario is identical to **GCDeterministic Run 1**, except the bench RTT also allocates memory.

  Again this is not deterministic, because the RTGC is not given enough time to recycle memory from the bench RTT. Therefore, the bench RTT must be blocked for RTGC to recycle sufficient memory. In fact, this run is much worse than **GCDeterministic Run 1**. From the output of the RTGC, we see that garbage collection ran twice as frequently during the execution of the bench thread and the available memory was exhausted more quickly. In addition to a much higher jitter and standard deviation, we see a total of 18 iterations that deviated by more than 500 microseconds.
Summary of results:

Mean execution time: 367 microseconds
Best execution time: 197 microseconds
Worst execution time: 13256 microseconds
Most frequent execution time: 222 microseconds, 28 occurrences
Execution time jitter: 13059 microseconds
Standard deviation: 1185.54 microseconds

• Run 3 - Identical to Run 2, except that we use the Java RTS -XX:+RTGCriticalReservedBytes option to reserve 2MB of heap memory for critical RTTs.

As exemplified by the previous two runs, we must configure the RTGCriticalReservedBytes parameter in order to guarantee determinism for hard real-time. The default value for this parameter is zero, which reserves no memory at all for the critical threads when the RTGC is boosted to its higher priority. The information provided by the -XX:+RTGCPrintStatistics option can be extremely useful in understanding the memory requirements of your application and determining the necessary number of reserved bytes to ensure your critical threads are never interrupted.

With RTGCriticalReservedBytes set to 2 MB, the “worst” non-fragmented memory level is 222 KB. This shows that 2MB was the correct value for RTGCriticalReservedBytes to ensure that memory was never exhausted and determinism has been guaranteed for the critical threads. However, since “worst” is lower than the value of RTGCriticalReservedBytes, some non-critical threads might have blocked on allocation.

Summary of results:

Mean execution time: 254 microseconds
Best execution time: 198 microseconds
Worst execution time: 332 microseconds
Most frequent execution time: 226 microseconds, 35 occurrences
Execution time jitter: 134 microseconds
Standard deviation: 35.18 microseconds

• Run 4 - Identical to Run 2 and 3, except that we add a pause time to the bench thread in addition to properly setting RTGCriticalReservedBytes.

As we are properly setting RTGCriticalReservedBytes, we again achieve a deterministic run. In addition, by adding a pause time to the critical RTT bench thread, we give the RTGC more time to recycle memory while the bench thread is paused. The effects of this are seen in the output of the RTGC, as the amount of available non-fragmented memory in this scenario never falls below 1649 KB. From this value we know that with
a 10 millisecond pause, 1MB is the minimal RTGCCriticalReservedBytes value to ensure determinism.

### Summary of results:

- **Mean execution time**: 254 microseconds
- **Best execution time**: 200 microseconds
- **Worst execution time**: 347 microseconds
- **Most frequent execution time**: 229 microseconds (28 occurrences)
- **Execution time jitter**: 147 microseconds
- **Standard deviation**: 36.77 microseconds

### 4.3.5 NHRTGCDeterministic

- **Run 1** - This scenario is identical to GCDeterministic Run 1, except that the bench thread runs in an NHRT instead of a RTT. The stress RTT allocates memory at min priority, while the bench NHRT simply calculates Fibonacci numbers.

  As with **GCDeterministic Run 1**, the RTGC still runs to recycle memory and we still exhaust the available heap memory during the execution of the bench thread. However, the NHRT is completely isolated from the heap and protected from any pauses due to garbage collection. Because of this basic property of NHRT threads, this results in a deterministic run with excellent amounts of jitter and a very low standard deviation. As we saw when comparing Deterministic with NHRTDeterminstic, our best execution time is approximately 10 times slower when using NHRTs instead of RTTs.

  Summary of results:
  - **Mean execution time**: 1165 microseconds
  - **Best execution time**: 1090 microseconds
  - **Worst execution time**: 1214 microseconds
  - **Most frequent execution time**: 1166 microseconds (381 occurrences)
  - **Execution time jitter**: 124 microseconds
  - **Standard deviation**: 8.18 microseconds

- **Run 2** - This scenario is identical to GCDeterminisite Run 2, except that the bench thread runs in an NHRT instead of a RTT. In other words, this is also identical to NHRTGCDeterministic Run 1, except that the bench thread also allocates memory that must be recycled.

  This scenario quickly demonstrates the dangers of working with NHRT threads in immortal memory. With the bench NHRT thread performing allocations, we quickly
exhaust the default amount of available immortal memory (64 MB). This results in two errors during the execution of this run:

∗∗∗ Error: not enough space in ImmortalMemory to allocate 105 words (224 bytes remaining) ∗∗∗

∗∗∗ Error: not enough space in ImmortalMemory to allocate 14 words (104 bytes remaining) ∗∗∗

And an obviously non-deterministic run.

Summary of results:

Mean execution time: -1314759652206 microseconds
Best execution time: -1302926815571043 microseconds
Worst execution time: 1267 microseconds
Most frequent execution time: 0 microseconds, 804 occurrences
Execution time jitter: 1241690454 microseconds
Standard deviation: 41367925840202.95 microseconds

• Run 3 - This scenario is identical to NHRTGCDeterministic Run 2, except that we use -XX:ImmortalSize=512M to specify the size of the ImmortalMemory region to meet the requirements of the NHRT.

By allocating 512 MB of ImmortalMemory, we ensure the bench NHRT has enough memory to run without error. We again achieve a deterministic run, with no deviations over 500 microseconds and excellent values for jitter and standard deviation. Again, however, we witness an order of magnitude decrease in best execution time when using NHRTs instead of RTTs.

Summary of results:

Mean execution time: 1261 microseconds
Best execution time: 1188 microseconds
Worst execution time: 1300 microseconds
Most frequent execution time: 1262 microseconds, 252 occurrences
Execution time jitter: 112 microseconds
Standard deviation: 10.66 microseconds

5 CESR Control Programs

Physical device control generally comprises three main functions: sensing, control, and actuation. Sensing is the process of measuring and reporting physical properties, and control takes the values from the sensors and produces commands to send to an actuator. In most real-time environments, the period between sensing and control needs to be deterministic and predictable.

CESR Control Programs can be broken down into the following categories.
- **Single monitor, single control** (i.e. bakeout control). Bakeout control involves increasing and decreasing voltage to keep the temperature within a specified range. Therefore, the period between sensing a change in temperature and controlling the voltage (heating device) must be deterministic. A failure in determinism will result in alarm states where the maximum or minimum temperatures are violated.

- **Single monitor, distributed control** (i.e. magnet ramp). In this example, we are controlling the current through a series of magnets throughout the storage ring. We would need to be able to deterministically sense a change in condition (for example, the machine starting up, an alarm condition, or an operator manually changing the desired current) and send the appropriate command to each magnet.

- **Distributed monitor, no control** (i.e. logging software). As implied, this involves logging data from a series of devices throughout the storage ring and control room. Without any device control, in general logging does not have real-time requirements. We can tolerate delays between when a condition is sensed and when that event is logged to the appropriate filesystem or database.

- **Distributed monitor, single control** (i.e. monitor beam position signals, check for dangerous change, trigger beam dump OR monitor gun current and vacuum while changing voltage for processing). As with **single monitor, single control**, the period between sensing conditions and acting on that condition must be deterministic.

- **Distributed monitor, distributed control** (i.e. Change states of the machine while verifying that all elements step together). A combination of the previous categories, this must also deterministically control many devices while sensing conditions from multiple sources.

CESR currently makes limited use of real-time programming throughout its control system. VxWorks, an embedded real-time operating system, has been in use since 1992 primarily on Motorola 68040 microprocessors. As several critical systems still run on these platforms, we expect to continue needing support for VxWorks for the foreseeable future. Currently, maintenance for VxWorks costs $5,000 per year.

In addition to VxWorks, CESR has also used RTEMS on VME and ColdFire boards since 2003. While it is a complete open-source real-time operating system for embedded systems, the current use of RTEMS is primarily limited to providing a network stack and network connectivity on these devices.

### 5.1 Bakeout Control Simulation

As a sample Java RTS control program, I implemented a bakeout control simulation that can be run in either real-time or non-real-time mode. The main **BakeoutSim.java** class, including the creation of the real-time monitor thread, is attached to the end of this document.

The application consists of a user interface (a simple temperature gauge implemented using **Swing** in a regular Java thread), a temperature sensor and a temperature monitor. The
temperature sensor is also a regular JLT thread that increases or decreases the temperature every two milliseconds, depending on whether the heating system is on or off. Per *Newton's Law of Cooling*, the temperature exponentially decays toward either room temperature or increases toward the temperature of the heating system.

For example, when $T(\text{room})$ is the ambient temperature of the room, $T(\text{heater})$ is the temperature of the heating element, and $\Delta t$ is the change in time from the last sensor reading (two milliseconds, as stated above) the temperature at time $n + 1$, $T(n + 1)$, can be represented as follows. [17]

- Heater off: $T(n + 1) = T(\text{room}) + (T(n) - T(\text{room})) \times e^{-\text{decayRate}\times\Delta t}$
- Heater on: $T(n + 1) = T(\text{heater}) + (T(n) - T(\text{heater})) \times e^{-\text{heatRate}\times\Delta t}$

$\text{decayRate}$ and $\text{heatRate}$ are arbitrary values in our simulation, so we approximate $\text{heatRate}$ to be five times the $\text{decayRate}$.

When run in real-time mode, the temperature monitor is a RTT with a one-millisecond period. In non-real-time mode, the temperature monitor is a regular JLT that sleeps for one millisecond before checking the temperature. Based on the latest temperature reported by the sensor and a prediction of the next value, the monitor turns the heating system on or off to keep the temperature within a specified temperature range. In addition to checking the temperature sensor and controlling the heating element, the monitor thread keeps track of the highest and lowest temperature values reported by the temperature sensor.

To facilitate communication between the RTT monitor thread and the standard JLT sensor thread, without risk of the JLT affecting the determinism of the RTT, we use the *WaitFreeReadQueue* provided by Java RTS. The sensor thread runs every two milliseconds, and places the simulated temperature change onto the queue. In turn, the monitor thread runs every one millisecond and, if one is available, takes the first temperature from the queue. After projecting the next temperature, the monitor thread controls the heating device to keep the temperature within allowable limits.

To model an alarm response, we set up one additional listener thread. To ensure the JLT alarm thread does not interfere with the execution of the RTT monitor thread, the alarm thread reads values from a *WaitFreeWriteQueue* that is populated from the monitor thread.

Finally, my bakeout simulation demonstrates the use of a deadline miss handler. This is done by creating a *DeadlineMissHandler* class that extends *AsynchEventHandler* and is given a reference to the RTT it’s associated with. When a deadline miss occurs, the associated *DeadlineMissHandler* is fired. As the bakeout simulation represents a soft real-time application, we would normally choose to simply report and track any deadline misses rather than increase the period of the monitor thread or terminate the simulation. For demonstration purposes, however, in our simulation we do choose to increase the period of both the monitor and sensor threads when a deadline miss occurs. While deadlines should not be missed during normal operation of our simulation, it is trivial to trigger a deadline miss by suspending and then resuming the simulation.
5.2 Analysis

For this analysis, we run multiple real-time and non-real-time instances of the bakeout simulation with the acceptable temperature range set to between 125° and 150°.

The benefits of running in a RealtimeThread are readily apparent when stressing a system with multiple instances of the Bakeout Control Simulation. When enough non-real-time instances of the bakeout simulation execute simultaneously, alarms will be raised as the simulations continually exceed the allowable high and low temperatures. Even before this emergency situation is reached, we see repeated situations where the monitor thread is delayed and not able to keep up with the sensor thread. With the specified periods of the sensor and monitor threads, the monitor thread should run roughly twice as frequently as the sensor thread. Therefore, the monitor thread should be able to keep up with all of the values placed on the shared queue, and there should never be more than one value on the queue. When the monitor thread runs in a regular JLT, however, we regularly see the size of the temperature queue grow as the monitor thread falls out of step with the sensor.

Conversely, the monitor thread in any real-time instances of the Bakeout Simulation are guaranteed to run uninterrupted. Therefore, we never see a situation where the real-time threads exceed the allowable high and low temperatures. Before we see a real-time instance trigger an alarm, we will exhaust the systems memory and the operating system will start killing instances to reclaim memory and stay alive. Note that even this behaviour occurs because we properly set the period of the real-time monitor thread. If we set a period that was lower than the execution time of the monitor thread (causing a miss on every call to RealtimeThread.waitForNextPeriod()), or we neglected to call RealtimeThread.waitForNextPeriod() entirely, we would instead produce a “denial of service” situation where the critical real-time threads saturate the available CPU power and leave no cycles for other operations to complete.

This comparison between JLT and RTT monitor threads is illustrated in the following screen-shot showing six real-time Bakeout Simulations running simultaneously with one non-real-time simulation. The real-time instances are stable, meet their deadlines, and never raise any alarms. Conversely, we see the single non-real-time scenario (in the upper-right corner) quickly raise warnings as the temperature queue begins to fill and trigger emergency alerts as the temperature increases above the upper threshold and falls below the lower threshold.
6 Conclusion

The Real-Time Specification for Java is an established and stable specification for using Java in a real-time context, with many successful implementations and examples of use throughout academia and industry. Java RTS, developed by Sun Microsystems as the first conformant implementation of the RTSJ, enables developers of real-time applications to take full advantage of Java while maintaining the predictability of current real-time development platforms.

While the process of ensuring determinism does increase complexity and requires a greater understanding of your system than when developing non-real-time systems, the tools exist to appropriately manage the CPU and memory usage of your application.

Some of the advantages to using Java in both a real-time and non-real-time environment are the strongest arguments for using Java itself; it is an established language with a large user community and an abundance of resources and tools. These are strong arguments for continuing to integrate Java into the CESR Control System in some capacity. This would provide the flexibility to move into Java RTS when its future becomes more stable, or another commercial RTSJ implementation when the need arises and funds are available.

6.1 Java RTS

Obtaining true determinism is difficult, and as we have seen there is typically a trade-off between complexity, average throughput, and predictability. With the RTSJ and Java RTS, however, the process of adapting standard Java programs into a real-time environment is greatly simplified and made available to the standard Java developer.

In section 4, we demonstrated the inability to ensure determinism in a standard Java environment. By moving to a real-time Java Virtual Machine and simply switching from `java.lang.Thread` to `javax.realtime.RealtimeThread`, we obtain soft real-time behaviour with the Real-Time Garbage Collector, priority inversion avoidance, and strict and precise priority dispatching semantics. In order to ensure determinism with the real-time garbage collector, however, we must understand the memory usage of our critical real-time threads and configure the RTGC accordingly. While we saw a slight decrease in performance when switching from the standard Java virtual machine to the Java RTS VM, we saw no change in fastest execution time, and in fact a dramatic decrease in average execution time, when switching from JLT threads to RTT threads. Finally, at the cost of further increasing the complexity of your application and the average execution time, the best determinism can be achieved with No-Heap Realtime Threads.

Likewise, our bakeout simulation demonstrated the benefits of using Realtime Threads in an application that combines a graphical user interface and simulated alarm system with both real-time and non-real-time components.

Unfortunately, however, the future of the Java RTS implementation has become steadily less certain with the acquisition of Sun by Oracle, and the subsequent decrease in visible activity around Java RTS. While it is still a viable and stable product, the lack of support for Java SE 6 (not to mention the next-generation Java 7), is increasingly becoming a critical limitation of Java RTS. Until its future becomes more certain, we currently cannot recommend the adoption of Java RTS for mission-critical applications in the CESR Control
6.2 Real-Time in CESR

While CESR’s current use of real-time programming is limited, especially with the ERL expansion to CESR, there could certainly be many situations where real-time processes with ensured determinism would be beneficial and, in some cases, critical. Any process that functions as a control for other processes or devices could benefit from the predictable deadlines provided by a real-time application. For example, the vacuum systems that we currently monitor from VMS could certainly benefit from the ability to ensure deadlines are met deterministically. In addition, 100MHz FPGA feedback control systems are currently being run on MicroTCA crates that come with the Ubuntu operating system. Especially as we start using these for higher-rate orbit feedback or lower periods, tangible benefits could be gained by the ability to run an enterprise-class real-time operating system with real-time programming languages, such as the RTSJ.

Where CESR currently uses a combination of VxWorks and RTEMS, the availability of an RTSJ implementation of Java ME (Micro Edition - for running on embedded systems), could be useful for ensuring predictable, deterministic behaviour. Likewise, the managed interrupts and digital signal processors run on Low Level Radio Frequency (LLRF) boards throughout CESR and the ERL could benefit from the deterministic behaviour provided by a real-time software specification such as the RTSJ.

The question may be more broadly posed as, what programming language(s) will comprise CESR’s environment in a fully Linux based operating system. The Real-Time Specification for Java, and the availability of an affordable implementation of the RTSJ, would be valuable assets in the CESR Control System. This would also provide a path for increasing our toolkit to include, for example, Safety-critical Java.

The ability to run real-time applications on Scientific Linux 5 using commodity hardware could be of great use to CESR. There could be tangible benefits, increased efficiency, and cost-savings from using the same operating system and programming languages in both real-time and non-real-time environments. As real-time operations become more critical throughout CESR, the availability of the open-source enterprise MRG Realtime packages and the Real-Time Specification for Java add real-time capabilities to the operating systems and programming languages already supported and used throughout the lab.

7 Resources

- [http://download.oracle.com/javase/realtime/rts_productdoc_2.2.html](http://download.oracle.com/javase/realtime/rts_productdoc_2.2.html): Sun Java Real-Time System 2.2 Technical Documentation
- [http://linux.web.cern.ch/linux/mrg.shtml](http://linux.web.cern.ch/linux/mrg.shtml): MRG (Messaging Realtime Grid) @ CERN
8 Source Code
package clockresolution;

import javax.realtime.*;

public class ClockResolution {
    static RealtimeThread rt;

    static {
        // period of rtt
        final int period = 2;
        rt = new RealtimeThread(new PriorityParameters(
                PriorityScheduler.instance().getMaxPriority(),
                new PeriodicParameters(new RelativeTime(period, 0)))) {
            public void run() {
                // create real-time clock
                Clock clock = Clock.getRealtimeClock();
                // get resolution of clock
                RelativeTime t = clock.getResolution();
                System.out.println(
                        "Real-time clock resolution = " + t.toString());

                // demonstrate simple iteration every 2 milliseconds
                System.out.println(
                        "RTT executing with a period of " + period + " ms ...");
                AbsoluteTime start = new AbsoluteTime();
                AbsoluteTime end = new AbsoluteTime();
                final int ITERATIONS = 20;
                RelativeTime tab[] = new RelativeTime[ITERATIONS];
                
                int i;

                for (i = 0; i < ITERATIONS; i++) {
                    tab[i] = new RelativeTime();
                }

                for (i = 0; i < ITERATIONS; i++) {
                    start = clock.getTime();
                    waitForNextPeriod();
                    end = clock.getTime();
                    end.subtract(start, tab[i]);
                }

                for (i = 3; i < ITERATIONS; i++) {
                    System.out.println(tab[i]);
                }
            }
        }; 
    }

    public static void main(String[] args) {
        ClockResolution.rt.start();
        try {
            ClockResolution.rt.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public class NHRTGCDeterministic implements Runnable {

    static {
        // Forces a load and initialization of Fibonacci classes
        Fibonacci f1 = new Fibonacci1(10);  
        Fibonacci2 f2 = new Fibonacci2(10); 
        f1.computeFib(); 
        f2.computeFib(); 

        // Forces a load and initialization of GarbageProducer classes
        GarbageProducer1 gp1 = new GarbageProducer1(10); 
        GarbageProducer2 gp2 = new GarbageProducer2(10); 
        gp1.produceGarbage(10); 
        gp2.produceGarbage(10); 

        // Uncomment the next 3 lines, and replace Fibonacci by the name 
        // of your own StressingObject class 
        // to preload and preinitialize your own stressing code.
        // GCDeterministic tmpGCDeterministic = new GCDeterministic(1); 
        // tmpGCDeterministic.setVerbosity(false); 
        // tmpGCDeterministic.stress("Fibonacci", 10, 10); 
        System.out.println("Static initialization finished");
        System.out.println(" === Stressing Fibonacci and GarbageProducer classes initialized and loaded");
    }

    // stressing iteration data (captured as program parameters)
    int nb_outer_iterations = 0;
        int nb_inner_iterations = 0;
        int nb_stress_class_iterations = 0;
        int pause_time = 0;

    // stressing Allocation data (captured as program parameters)
    int allocation_array_size = 0;
        boolean produce_garbage = false;

    // verbosity flag (captured as program parameter)
        boolean verbosity = false;

    // stress java.lang.Thread priority
        int thread_priority;

    // timestamps array for bench thread
    static long[] wallclock_times_begin_loops = null;
        static long[] wallclock_times_end_loops = null;

    // array of number of missed deadline for a given loop
    static int[] missed_deadlines = null;

    // measurements instance
        Measurements measurements = null;

    // bench thread priority
        static int rt_bench_thread_priority = 0;

    // bench thread period
        static RelativeTime benchPeriod = null;

    // delegates result printing to measurement
    private void outputResults() {
        measurements.computeAndPrintStatResults(wallclock_times_begin_loops, wallclock_times_end_loops, missed_deadlines);
    }

    ** Instantiate a StressingObject from stressingObjectClassName and 
        and stresses nb_inner_iterations times with this StressingObject 
        which is supposed to iterate nb_stress_class_iterations times by itself */
    void stress(String stressingObjectClassName, int nb_inner_iterations, int nb_stress_class_iterations) {
        StressingObject so = null;
        try {
            so = (StressingObject) (Class.forName(stressingObjectClassName).newInstance());
            so.setStressIterations(nb_stress_class_iterations);
            for (int i = 1; i <= nb_inner_iterations; i++) {
                so.stress();
            }
        } catch (Exception e) { System.out.println(e); }
    }

    // Fibonacci computation stress method
        // computes 2 fibonacci sequences "nb_inner_iterations" times 
        // - nb_stress_class_iterations : quantity of fib numbers to compute 
    static void computeFibs(int nb_inner_iterations, int nb_stress_class_iterations) {
        Fibonacci1 f1 = new Fibonacci1(nb_stress_class_iterations);
        Fibonacci2 f2 = new Fibonacci2(nb_stress_class_iterations);
        for (int i = 1; i <= nb_inner_iterations; i++) {
            f1.computeFib();
            f2.computeFib();
        }
    }
}
// Memory allocation stress method
// 2 garbage production sequences "nb_inner_iterations" times
// - nb_stress_class_iterations : number of int arrays to allocate
// - allocation_array_size: size of the arrays to allocate
static void produceGarbage(int nb_outer_iterations, int nb_stress_class_iterations, int allocation_array_size) {
    GarbageProducer1 gp1 = new GarbageProducer1(allocation_array_size);
    GarbageProducer2 gp2 = new GarbageProducer2(allocation_array_size);
    for (int i = 1; i <= nb_outer_iterations; i++) {
        gp1.produceGarbage(nb_stress_class_iterations);
        gp2.produceGarbage(nb_stress_class_iterations);
    }
}

/** Creates a new instance of Deterministic */
NHRTGCDeterministic(int nb_loops) {
    wallclock_times_begin_loops = {long[]} ImmortalMemory.instance().newArray(long, nb_loops);
    wallclock_times_end_loops = {long[]} ImmortalMemory.instance().newArray(long, nb_loops);
    missed_deadlines = {int[]} ImmortalMemory.instance().newArray(int, nb_loops);
    measurements = new Measurements(nb_loops);
}

/** Changes verbosity level */
private void setVerbosity(boolean verbose_flag) {
    verbosity = verbose_flag;
    measurements.setVerosity(verbosity);
}

For additional time consumption from a time-sharing thread
public void run() {
    try {
        for (int i = 0; i < this.nb_outer_iterations; i++) {
            // Fibonacci calculation
            computeFibes(this.nb_inner_iterations, this.nb_stress_class_iterations);
            // Stressing with your own code.
            // Replace Fibonacci1 by your StressingObject class name
            // and uncomment the following line to stress with your own code.
            // stress("Fibonacci1", nb_inner_iterations, nb_stress_class_iterations);
            // parameterized pause time
            if (this.pause_time != 0) Thread.currentThread().sleep((long) this.pause_time);
            // parameterized garbage memory production
            if (this.produce_garbage) produceGarbage(this.nb_inner_iterations, this.nb_stress_class_iterations, this.allocation_array_size);
        }
        catch (InterruptedException ie) {
            System.out.println(ie);
        }
    }
}

// Dedicated inner class to evaluate to execution time needed for each loop
class CheckMeanCalculationForGarbageAndFibonacciLoops extends RealtimeThread {
    long meantime_to_compute = 0;
    int nb_inner_iterations = 0;
    int nb_stress_class_iterations = 0;
    int check_allocation_array_size = 0;
    boolean garbage_producer = false;

    CheckMeanCalculationForGarbageAndFibonacciLoops(
        int inner_iterations,
        int stress_class_iterations,
        int alloc_array_size,
        SchedulingParameters schedulingParams)
        super(schedulingParams);
        nb_inner_iterations = inner_iterations;
        nb_stress_class_iterations = stress_class_iterations;
        check_allocation_array_size = alloc_array_size;
        if (check_allocation_array_size != 0) garbage_producer = true;
}

// Returns the mean time needed to perform the computation given the input parameters
// This must be used after the thread has been run once
public RelativeTime getComputingMeanTime() {
    return new RelativeTime((meanTimeToCompute / 1000000L)
        (int) (meanTimeToCompute % 1000000L));
}

public void run() {
    final Clock rtclock = Clock.getRealtimeClock();
    int no_loop = 0;
    // the AbsoluteTime from which to measure
    AbsoluteTime rtWallClockTimeBefore = rtclock.getTime();
    AbsoluteTime rtWallClockTimeAfter = rtclock.getTime();

    // recording time before computation and allocation
    rtclock.getTime(rtWallClockTimeBefore);
    long wallclock_times_begin_loops =
        (rtWallClockTimeBefore.getMilliseconds() * 1000000L) +
        rtWallClockTimeBefore.getNanoseconds();
    while (no_loop < Nb_CHECK_OUTER_ITERATIONS) {
        // Fibonacci computation sample
        NHRTGCDeterministic.this.computeFibes(nb_inner_iterations, nb_stress_class_iterations);
        if (garbage_producer) {
            NHRTGCDeterministic.this.produceGarbage(nb_inner_iterations, nb_stress_class_iterations, check_allocation_array_size);
        }
        no_loop++;
    }
}
class GarbageLoops implements Runnable {
    boolean records_time_measures = false;
    int nb_outer_iterations = 0;
    int nb_inner_iterations = 0;
    int nb_stress_class_iterations = 0;
    int allocation_array_size = 0;
    boolean garbage_producer = false;
    int pause_time = 0;
    GarbageLoops(boolean time_recording_state, int outer_iterations, int inner_iterations, int stress_class_iterations, int alloc_array_size, int time_to_pause) {
        records_time_measures = time_recording_state;
        nb_outer_iterations = outer_iterations;
        nb_inner_iterations = inner_iterations;
        nb_stress_class_iterations = stress_class_iterations;
        allocation_array_size = alloc_array_size;
        if (allocation_array_size <= 0) garbage_producer = true;
        pause_time = time_to_pause;
    }
    public void run() {
        final Clock rtClock = Clock.getRealtimeClock();
        final RelativeTime time_to_sleep = new RelativeTime(pause_time, 0);
        // the AbsoluteTime from which to measure
        AbsoluteTime rtWallClockTimeBefore = rtClock.getTime();
        AbsoluteTime rtWallClockTimeAfter = rtClock.getTime();
        int no_loop = 0;
        try {
            while (no_loop < nb_outer_iterations) {
                if (records_time_measures) {
                    // benchmark thread -> recording time before computation and allocation
                    rtClock.getTime(rtWallClockTimeBefore);
                    wallclock_times_end_loops = (rtWallClockTimeAfter.getMillisecs() * 1000000L) +
                        rtWallClockTimeAfter.getNanosecs();
                    meantime_to_compute =
                        (wallclock_times_end_loops - wallclock_times_begin_loops) / NB_CHECK OUTER ITERATIONS;
                }
                // fibonacci computation sample
                computeFibs(nb_inner_iterations, nb_stress_class_iterations);
                // allocation sample
                if (garbage_producer) {
                    produceGarbage(nb_inner_iterations, nb_stress_class_iterations, allocation_array_size);
                }
                if (records_time_measures) {
                    // this should be benchmark thread -> recording time after computation and allocation
                    rtClock.getTime(rtWallClockTimeAfter);
                    wallclock_times_end_loops[no_loop] = (rtWallClockTimeAfter.getMillisecs() * 1000000L) +
                        rtWallClockTimeAfter.getNanosecs();
                    while (RealtimeThread.currentThread().waitNextPeriod() == false) {
                        missed_deadlines[no_loop]++;
                    }
                } else {
                    // stress thread -> parameterized names if asked for
                }
                no_loop++;
            }
        } catch (InterruptedException ie) {
            System.out.println(ie);
        }
    }
}
if (pause_time != 0) Thread.currentThread().sleep(time_to_sleep);
} 

no_loop++;
}

} 

} 

catch (Exception ie) { 
    System.out.println("java.lang.Thread interrupted"); 
}

}

} 

class RealTimeGarbageLoops extends RealtimeThread {

    boolean records_time_measures = false;
    int nb_outer_iterations = 0;
    int nb_inner_iterations = 0;
    int nb_stress_class_iterations = 0;
    int allocation_array_size = 0;
    boolean garbage_producer = false;
    int pause_time = 0;

    RealTimeGarbageLoops(boolean time_recording_state,
                          int outer_iterations,
                          int inner_iterations,
                          int stress_class_iterations,
                          int alloc_array_size,
                          int_to_pause,
                          SchedulingParameters schedulingParams,
                          ReleaseParameters releaseParams) {
        super(schedulingParams,releaseParams);
        records_time_measures = time_recording_state;
        nb_outer_iterations = outer_iterations;
        nb_inner_iterations = inner_iterations;
        nb_stress_class_iterations = stress_class_iterations;
        allocation_array_size = alloc_array_size;
        
        if (allocation_array_size <= 0) garbage_producer = true;
        pause_time = time_to_pause;
    }

    public void run() {

        final Clock rtClock = Clock.getRealtimeClock();
        final RelativeTime time_to_sleep = new RelativeTime(pause_time, 0);

        // the AbsoluteTime from which to measure
        AbsoluteTime rtWallClockTimeBefore = rtClock.getTime();
        AbsoluteTime rtWallClockTimeAfter = rtClock.getTime();

        int no_loop = 0;
        try {
            while (no_loop < nb_outer_iterations) {

                if (records_time_measures) {
                    // bench thread -> recording time before computation and allocation
                    rtClock.getTime(rtWallClockTimeBefore);
                    wallclock_times_begin_loops[no_loop] = 
                        (rtWallClockTimeBefore.getMillisseconds() * 1000000L) +
                        rtWallClockTimeBefore.getNanoseconds();
                }

                // fibonacci computation sample
                NHRTGCDeterministic.this.computeFibs(nb_inner_iterations, nb_stress_class_iterations);
                // stressing with your own code.
                // Replace Fibonacci1 by your StressingObject class name
                // and uncomment the following line to stress with your own code.
                // GCDeterministic.this.stress("Fibonacci1", nb_inner_iterations, nb_stress_class_iterations);
                // allocation sample

                if (garbage_producer) {
                    NHRTGCDeterministic.this.produceGarbage(nb_inner_iterations, nb_stress_class_iterations, allocation_array_size);
                }

                if (records_time_measures) {
                    // this should be bench thread -> recording time after computation and allocation
                    rtClock.getTime(rtWallClockTimeAfter);
                    wallclock_times_end_loops[no_loop] = 
                        (rtWallClockTimeAfter.getMillisseconds() * 1000000L) +
                        rtWallClockTimeAfter.getNanoseconds();

                    while (RealtimeThread.currentThread().waitForNextPeriod() == false) {
                        missed_deadlines[no_loop]++;
                    }
                } else {
                    // stress thread -> parameterized pause if asked for
                    if (pause_time > 0) sleep(time_to_sleep);
                }

                no_loop++;
            }

            catch (InterruptedException ie) { 
                System.out.println("java.lang.Thread interrupted");
            }
        }
    }
}

/*
 *
 *    NHRTGCDeterministic
 *      Test Determinism using No-Heap Realtime Threads
 *      and the Realtime ... 
 
        }
        catch (InterruptedException ie) {
            System.out.println(ie);
        }
 
    }
}
[18x-2013]}
[18x-1981]catch (InterruptedException ie) {
    gcd.outputResults();
    jlt_stress_thread.join();
    if (jrt_stress_thread_present) {
        System.out.println("   Bench javax.RealtimeThread terminated ");
        System.out.println("Total amount of VM memory : " + Runtime.getRuntime().totalMemory());
        System.out.println("   Priority level: " + jlt_stress_priority_level);
        System.out.println("   Number of inner iterations: " + gcd.nb_inner_iterations);
        System.out.println("  ==========================================");
    }
    System.out.println("   Size of elementary array to allocate: " + jrt_stress_allocation_array_size);
    System.out.println("  Realtime stress thread parameters:");
    System.out.println("   Priority level: " + bench_priority_level);
    System.out.println("   Entered pause time: " + bench_pause_time + " milliseconds");
    System.out.println("   Number of outer loops: " + bench_nb_outer_iterations);
    System.out.println("  =================================");
    System.out.println("  Realtime bench thread parameters: ");
    System.out.println("============================================");
    gcd.thread_priority = jlt_stress_thread_priority;
    gcd.pause_time = jlt_stress_pause_time;
    gcd.nb_inner_iterations = jlt_stress_nb_inner_iterations;
    gcd.nb_outer_iterations = jlt_stress_nb_outer_iterations;
    jlt_stress_thread = new Thread(gcd);
    else if (jlt_stress_priority_level.equalsIgnoreCase("norm")
        // Any iteration value to zero means NO java.lang.Thread stress thread
        // java.lang.Thread stress thread instanciation
        jrt_stress_nb_stress_class_iterations = - jrt_stress_nb_stress_class_iterations;
        jrt_stress_nb_outer_iterations = - jrt_stress_nb_outer_iterations;
        benchPeriod, null, null,
        bench_nb_outer_iterations,
        new PriorityParameters(rt_bench_thread_priority),
        benchPeriod = benchThreadComputationTime.add(new RelativeTime(bench_pause_time,0));
        catch (InterruptedException ie) {
            while (no_loop < nb_outer_iterations) {
                if (records_time_measures) {
                    // bench thread -> recording time before computation and allocation
                    rtClock.getTime(rtWallClockTimeBefore);
                    wallclock_times_begin_loops[no_loop] = 
                        (rtWallClockTimeBefore.getMillisseconds() * 1000000L) +
                        rtWallClockTimeBefore.getNanoseconds();
                }

                // fibonacci computation sample
                NHRTGCDeterministic.this.computeFibs(nb_inner_iterations, nb_stress_class_iterations);
                // stressing with your own code.
                // Replace Fibonacci1 by your StressingObject class name
                // and uncomment the following line to stress with your own code.
                // GCDeterministic.this.stress("Fibonacci1", nb_inner_iterations, nb_stress_class_iterations);
                // allocation sample

                if (garbage_producer) {
                    NHRTGCDeterministic.this.produceGarbage(nb_inner_iterations, nb_stress_class_iterations, allocation_array_size);
                }

                if (records_time_measures) {
                    // this should be bench thread -> recording time after computation and allocation
                    rtClock.getTime(rtWallClockTimeAfter);
                    wallclock_times_end_loops[no_loop] = 
                        (rtWallClockTimeAfter.getMillisseconds() * 1000000L) +
                        rtWallClockTimeAfter.getNanoseconds();
                }

                // stress thread -> recording time after computation and allocation
                rtClock.getTime(rtWallClockTimeBefore);
                wallclock_times_begin_loops[no_loop] = 
                    (rtWallClockTimeBefore.getMillisseconds() * 1000000L) +
                    rtWallClockTimeBefore.getNanoseconds();

                while (no_loop < nb_outer_iterations) {

                    if (records_time_measures) {
                        // bench thread -> recording time before computation and allocation
                        rtClock.getTime(rtWallClockTimeBefore);
                        wallclock_times_begin_loops[no_loop] = 
                            (rtWallClockTimeBefore.getMillisseconds() * 1000000L) +
                            rtWallClockTimeBefore.getNanoseconds();
                    }

                    // fibonacci computation sample
                    NHRTGCDeterministic.this.computeFibs(nb_inner_iterations, nb_stress_class_iterations);
                    // stressing with your own code.
                    // Replace Fibonacci1 by your StressingObject class name
                    // and uncomment the following line to stress with your own code.
                    // GCDeterministic.this.stress("Fibonacci1", nb_inner_iterations, nb_stress_class_iterations);
                    // allocation sample

                    if (garbage_producer) {
                        NHRTGCDeterministic.this.produceGarbage(nb_inner_iterations, nb_stress_class_iterations, allocation_array_size);
                    }

                    if (records_time_measures) {
                        // this should be bench thread -> recording time after computation and allocation
                        rtClock.getTime(rtWallClockTimeAfter);
                        wallclock_times_end_loops[no_loop] = 
                            (rtWallClockTimeAfter.getMillisseconds() * 1000000L) +
                            rtWallClockTimeAfter.getNanoseconds();
                    }
                }
            }
        catch (InterruptedException ie) {
            while (no_loop < nb_outer_iterations) {
                if (records_time_measures) {
                    // bench thread -> recording time before computation and allocation
                    rtClock.getTime(rtWallClockTimeBefore);
                    wallclock_times_begin_loops[no_loop] = 
                        (rtWallClockTimeBefore.getMillisseconds() * 1000000L) +
                        rtWallClockTimeBefore.getNanoseconds();
                }

                // fibonacci computation sample
                NHRTGCDeterministic.this.computeFibs(nb_inner_iterations, nb_stress_class_iterations);
                // stressing with your own code.
                // Replace Fibonacci1 by your StressingObject class name
                // and uncomment the following line to stress with your own code.
                // GCDeterministic.this.stress("Fibonacci1", nb_inner_iterations, nb_stress_class_iterations);
                // allocation sample

                if (garbage_producer) {
                    NHRTGCDeterministic.this.produceGarbage(nb_inner_iterations, nb_stress_class_iterations, allocation_array_size);
                }

                if (records_time_measures) {
                    // this should be bench thread -> recording time after computation and allocation
                    rtClock.getTime(rtWallClockTimeAfter);
                    wallclock_times_end_loops[no_loop] = 
                        (rtWallClockTimeAfter.getMillisseconds() * 1000000L) +
                        rtWallClockTimeAfter.getNanoseconds();
                }
            }
        }
    } 
*/

/**
 * #param args the command line arguments
 */
public static void main(String[] args) {
    // local thread variables
    Thread jlt_stress_thread = null;
    // RealTimeGarbageLoops rt_bench_thread = null;
    RealtimeThread rt_bench_thread = null;
    RealTimeGarbageLoops rt_stress_thread = null;
    int jlt_stress_thread_priority = 0;
    int rt_stress_thread_priority = 0;
    // local run variables
    boolean jlt_stress_thread_present = false;
    boolean jlt_stress_thread_present = false;
    // extract program command-line parameters
    // The java.RealtimeThread bench thread parameters (all mandatory - no value check)
    final int bench_nb_outer_iterations = new Integer(args[0]).intValue();
    final int bench_nb_inner_iterations = new Integer(args[1]).intValue();
    final int bench_allocation_array_size = new Integer(args[3]).intValue();
    final int bench_pause_time = new Integer(args[4]).intValue();
    String bench_priority_level = args[5];
    // The java.RealtimeThread stress thread parameters (all mandatory - no value check)
    final int jlt_stress_nb_outer_iterations = new Integer(args[6]).intValue();
    final int jlt_stress_nb_inner_iterations = new Integer(args[7]).intValue();
    final int jlt_stress_allocation_array_size = new Integer(args[9]).intValue();
    final int jlt_stress_pause_time = new Integer(args[10]).intValue();
    String jlt_stress_priority_level = args[11];
    // The java.lang.Thread stress thread parameters (all mandatory - no value check)
    final int jlt_stress_nb_outer_iterations = new Integer(args[12]).intValue();
    final int jlt_stress_nb_inner_iterations = new Integer(args[13]).intValue();
    final int jlt_stress_allocation_array_size = new Integer(args[15]).intValue();
    final int jlt_stress_pause_time = new Integer(args[16]).intValue();
    String jlt_stress_priority_level = args[17];
    // Initialization
    PriorityScheduler ps = (PriorityScheduler)Scheduler.getDefaultScheduler();
    final NHRTGCDeterministic gcd = new NHRTGCDeterministic(bench_nb_outer_iterations);
    // verbosity level (optional)
    if (args.length >= 16) if (args[15].equals("verbose")) gcd.setVerbosity(true);
    // check if stress thread will be present
    if ((jlt_stress_nb_outer_iterations == 0) &&
        (jlt_stress_nb_inner_iterations == 0) &&
        (jlt_stress_nb_class_iterations == 0))
    jlt_stress_thread_present = true;
    else jlt_stress_thread_present = false;
    // compute average computation time for bench thread
    CheckMeanCalculationForGarbageAndFibonacciLoops computeBenchThreadTime =
        gcd.new CheckMeanCalculationForGarbageAndFibonacciLoops(
            bench_nb_inner_iterations,
            bench_nb_class_iterations,
            bench_allocation_array_size,
            new PriorityParameters(ps.getMaxPriority()));
    System.out.println("\nComputing stress cost for bench thread");
    computeBenchThreadTime.start();
    try {
        computeBenchThreadTime.join();
    } catch (InterruptedException ie) {
        System.out.println("Stress cost thread was interrupted\n" + ie );
        System.exit(1);
    }
    RelativeTime benchThreadComputationTime = computeBenchThreadTime.getComputingMeanTime();
    // period and deadline for bench thread
    System.out.println("Measured computation time for bench: " + benchThreadComputationTime);
    benchmarkPeriod = benchThreadComputationTime.add(new RelativeTime(bench_pause_time, 0));
    if (bench_pause_time == 0) benchmarkPeriod = benchmarkPeriod.add(new RelativeTime(0, 100000));
    System.out.println("Computed period for bench: " + benchmarkPeriod);
    System.out.println("Stress cost time for bench computed\n");
    // Bench realtime thread instanciation
    // Iteration values must all be set to strictly positive for bench threadR
    if (bench_nb_outer_iterations <= 0) {
        if (bench_nb_inner_iterations <= 0) {
            System.out.println("Invalid number of iterations specified for bench thread\n");
            System.exit(1);
        } else if (bench_nb_priority_level.equalsIgnoreCase("max"))
            rt_bench_thread_priority = ps.getMaxPriority();
        else if (bench_nb_priority_level.equalsIgnoreCase("norm"))
            rt_bench_thread_priority = ps.getNormPriority();
        else rt_bench_thread_priority = ps.getMinPriority();
    } else {
        rt_bench_thread =
            gcd.new RealTimeGarbageLoops(
                true,
                bench_nb_outer_iterations,
                bench_nb_inner_iterations,
                bench_nb_class_iterations,
                bench_allocation_array_size,
                bench_pause_time,
                new PriorityParameters(rt_bench_thread_priority),
                new PeriodicParameters(null, benchmarkPeriod, null, null));
    }
}
public void run() {
    // System.out.println("rt_bench_thread_priority in " + MemoryArea.getMemoryArea(rt_bench_thread_priority));
    // System.out.println("wallclock_times_begin_loops in " + MemoryArea.getMemoryArea(wallclock_times_begin_loops));
    ImmutableMemory.instance().enter();
    new Runnable() {
        public void run() {
            GarbageLoops garbage_loops = gcd.new GarbageLoops(true,
            bench_nb_outer_iterations,
            bench_nb_inner_iterations,
            bench_nb_stress_class_iterations,
            bench_allocation_array_size,
            bench_pause_time);
            NoHeapRealtimeThread nht = new NoHeapRealtimeThread;
            new PriorityParameters(rt_bench_thread_priority),
            new PeriodicParameters(null,
            benchPeriod, null, null,
            null, null, null, ImmediateMemory.instance(), null, garbage_loops);
            nht.start();
        }
    };
    }
}

// Realtime stress thread instanciation
if (jrt_stress_thread_present) {
    // negative iteration values are ignored and changed to positive
    System.out.println("negative iteration values changed to positive for java.realtime.RealtimeThread stress thread");
    if (jrt_stress_nb_outer_iterations < 0) {
        jrt_stress_nb_outer_iterations = - jrt_stress_nb_outer_iterations;
    }
    if (jrt_stress_nb_inner_iterations < 0) {
        jrt_stress_nb_inner_iterations = - jrt_stress_nb_inner_iterations;
    }
    if (jrt_stress_nb_stress_class_iterations < 0) {
        jrt_stress_nb_stress_class_iterations = - jrt_stress_nb_stress_class_iterations;
    }
    System.out.println("negative iteration values changed to positive for java.realtime.RealtimeThread stress thread");
    }
}
    // setting requested priority for stress realtime thread
    // local run variables
    jrt_stress_thread_present = false;
    // java.lang.Thread stress thread instanciation
    if (jrt_stress_thread_present)
    System.out.println("Setting priority for stress java.lang.Thread");
    }
    }
    // Iteration data ( captured as program parameters)
    gcd_nb_outer_iterations = jlt_stress_nb_outer_iterations;
    gcd_nb_inner_iterations = jlt_stress_nb_inner_iterations;
    gcd_nb_stress_class_iterations = jlt_stress_nb_stress_class_iterations;
    gcd.pause_time = jlt_stress_pause_time;
    gcd_allocation_array_size = jlt_stress_allocation_array_size;
    if (gcd_allocation_array_size != 0) gcd.produce_garbage = true;
    gcd.thread_priority = jlt_stress_thread_priority;
    // verbosity level (optional)
    System.out.println("Initialization and configuration of program finished ");
    }
    System.out.println("CMMVODeterministic program parameters");
}
System.out.println(" Calculated period (deadline + pause): " + benchPeriod);
System.out.println(" Priority level: " + bench_priority_level);
System.out.println(" Effective priority assigned: " + rt_bench_thread_priority);
System.out.println();

if (jrt_stress_thread_present) {
    System.out.println(" Realtime stress thread parameters:");
    System.out.println(" Number of outer loops: " + jrt_stress_nb_outer_iterations);
    System.out.println(" Number of inner iterations: " + jrt_stress_nb_inner_iterations);
    System.out.println(" Size of elementary array to allocate: " + jrt_stress_allocation_array_size);
    System.out.println(" Entered pause time: " + jrt_stress_pause_time + " milliseconds");
    System.out.println(" Priority level: " + jrt_stress_priority_level);
    System.out.println(" Effective priority assigned: " + rt_stress_thread_priority);
    System.out.println();
}

if (jlt_stress_thread_present) {
    System.out.println(" java.lang.Thread stress thread parameters:");
    System.out.println(" Number of outer loops: " + gcd_nb_outer_iterations);
    System.out.println(" Number of inner iterations: " + gcd_nb_inner_iterations);
    System.out.println(" Size of elementary array to allocate: " + gcd_allocation_array_size);
    System.out.println(" Entered pause time: " + gcd_pause_time);
    System.out.println(" Priority level: " + jlt_stress_priority_level);
    System.out.println(" Effective priority assigned: " + gcd_thread_priority);
    System.out.println();
}

System.out.println(" Output detail level: " + (gcd.verbosity ? "detailed" : "summary");
System.out.println("" + nb_loops);   
System.out.println(" \n ================ Starting " + bench_nb_outer_iterations + " iterations ================ \n ");

// stabilise memory and output state before allocating objects
System.gc();
System.out.println("Total amount of VM memory : " + Runtime.getRuntime().totalMemory());
System.gc();
System.out.println("Amount of free VM memory : " + Runtime.getRuntime().freeMemory());
System.gc();
System.out.println("Max amount of VM memory : " + Runtime.getRuntime().maxMemory());

// starts the java.lang.Thread stress thread
if (jlt_stress_thread_present) {   
    System.out.println(" Starting java.lang.Thread stress thread");
    jlt_stress_thread.start();
}

// starts the java.RealtimeThread bench thread
rt_bench_thread.start();

// starts the java.RealtimeThread stress thread
if (jrt_stress_thread_present) {   
    System.out.println(" Starting java.RealtimeThread stress thread");
    rt_stress_thread.start();
}

try {
    rt_bench_thread.join();
    System.out.println(" Bench java.RealtimeThread terminated ");
    if (jlt_stress_thread_present) {   
        jlt_stress_thread.join();
        System.out.println(" Stress java.RealtimeThread terminated");
    }
    if (jrt_stress_thread_present) {   
        jrt_stress_thread.join();
        System.out.println(" Stress java.RealtimeThread terminated");
    }
    System.out.println(" \n ================ Requested " + bench_nb_outer_iterations + " iterations finished ================ \n ");
    gcd.outputResults();
    System.out.println(" \n ================ Finished results output ================ \n ");
} catch (InterruptedException ie) {   
    System.out.println(ie);
}
package bakeoutcontrol;

import bakeoutcontrol.util.*;
import javax.realtime.*;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;

/**
 * BakeoutSim class for Bakeout Control Simulation.
 * @author Devin Bougie
 */
public class BakeoutSim extends GUI {
    // simulated temperature sensor
    private static Sensor tempSensor = null;
    // amount of time sensor sleeps between readings
    private static long sensorTimer = 2;
    // monitory (control) sched
    private static Monitor tempMonitor = null;
    // period or amount of time monitor sched sleeps
    private static long monitorTimer = 1;
    // RTT of JLT?
    private boolean realtime = false;
    // Alarm
    private static Alarm alarm = null;
    // upper range of allowable temperatures
    private static long highTemp = 150;
    // lower range of allowable temps
    private static long lowTemp = 125;
    // debug messages or not?
    private boolean verbose = false;
    // heaterTemp
    private static long heaterTemp = 200;
    // roomTemp
    private static long roomTemp = 70;

    /**
     * Deadline miss handler
     */
    static class DeadlineMissHandler extends AsyncEventHandler {
        // schedulable object
        private Schedulable sched = null;
        // alarm queue
        WaitFreeWriteQueue alarmQueue = null;
        /**
         * Create a new Deadline Miss Handler
         */
        public DeadlineMissHandler(Schedulable thread,
                                     WaitFreeWriteQueue alarmQueue) {
            super(new PriorityParameters(
                PriorityScheduler.instance().getMaxPriority(),
                null, null, null, null, null));
            this.sched = thread;
            this.alarmQueue = alarmQueue;
        }
        /**
         * async event handler
         */
        public void handleAsyncEvent() {
            // handle the deadline miss here.
            alarmQueue.write("DEADLINE MISS!");
            // as an example, could also increase the period
            monitorTimer = 2;
            PeriodicParameters release =
                new PeriodicParameters(new RelativeTime(monitorTimer,0));
            sched.setReleaseParameters(release);
            //And then re-schedule the sched
            if (sched instanceof RealtimeThread)
                ((RealtimeThread)sched).schedulePeriodic();
            // and, increase sensorTimer
        }
    }
}
sensorTimer = 3;
tempSensor.setTimer(sensorTimer);
}

/**
 * Create a new Bakeout Control Simulation
 * @param realtime
 */
public BakeoutSim(boolean realtime) {
    this.realtime = realtime;
    if (realtime) {
        setTitle("REALTIME BAKEOUT");
    } else {
        setTitle("NON-RT BAKEOUT");
    }
}

/**
 * Is this a realtime simulation?
 * @return realtime
 */
public boolean isRealtime() {
    return realtime;
}

/**
 * Return sleep time for sensor
 * @return sensorTimer
 */
public long getSensorTimer() {
    return sensorTimer;
}

/**
 * Return monitor period
 * @return monitorTimer
 */
public long getMonitorTimer() {
    return monitorTimer;
}

/**
 * Set monitor timer
 * @param monitor timer
 */
public static void setMonitorTimer(long t) {
    monitorTimer = t;
}

/**
 * Return heater temp
 * @return heaterTemp
 */
public long getHeaterTemp() {
    return heaterTemp;
}

/**
 * Return room temp
 * @return roomTemp
 */
public long getRoomTemp() {
    return roomTemp;
}

/**
 * Return max goal temp
 * @return highTemp
 */
public long getHighTemp() {
    return highTemp;
}

/**
 * Return goal low temp
 * @return lowTemp
 */
```java
package bakeoutcontrol;

import bakeoutcontrol.util.*;
import javax.realtime.*;
import java.awt.Graphics;

public void paintComponent(Graphics graphics) {
    Font f = null;
    super.paintComponent(graphics);
    paintTheBackground(graphics, lowTemp, highTemp, heaterTemp);
    if (tempSensor == null) {
        return;
    }
    graphics.setColor(Color.BLACK);
    double temp = tempSensor.getCurrentTemp();
    double min = tempMonitor.getMin();
    double max = tempMonitor.getMax();
    f = new Font("Arial", Font.BOLD, 14);
    graphics.setFont(f);
    graphics.drawString("Current temp: " + temp, 20, 30);
    f = new Font("Arial", Font.PLAIN, 14);
    graphics.drawString("min: " + min, 52, 240);
    graphics.drawString("max: " + max, 50, 260);
    graphics.drawLine(getCenterX(), getCenterY(),
                             nextXCoord((long) temp, heaterTemp),
                             nextYCoord((long) temp, heaterTemp));
}
```

RealtimeThread monitorThread =
    new RealtimeThread(
        sched, null, null, null, null, tempMonitor);
    // Create a deadline miss handler
    DeadlineMissHandler dmh =
        new DeadlineMissHandler(monitorThread, alarmQueue);
    // create periodic parameters using deadline miss handler
    PeriodicParameters period = new PeriodicParameters(null,
        new RelativeTime(sim.getMonitorTimer(), 0),
        null, null, null, dmh);
    // set periodic parameters
    monitorThread.setReleaseParameters(period);
    // start sched
    monitorThread.start();
} else {
    // Create a normal (non-realtime) sched
    Thread monitorThread = new Thread(tempMonitor);
    // start sched
    monitorThread.start();
}

    // start the sensor sched
tempSensor.start();
    // start the alarm sched
alarmThread.start();

    try {
        // Monitor and display temperature
        while (true) {
            sim.repaint();
            Thread.sleep(250);
        }
    } catch (Exception e) {
        System.out.println("Exception caught: "+e.toString());
    }
9 Bibliography

References


[4] andersoj@andersoj.org. Which real-time (RTSJ) JVM is most preferred?, 2010.


45


