Implementation of Real-Time Java using KURT

By

Dinesh Selvarajan

Bachelor of Engineering (Electronics and Communication Engineering),
Bharathiar University, Coimbatore, India, 2000

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

_____

Dr. Jerry James, Chair

_____

Dr. Douglas Niehaus, Member

_____

Dr. John Gauch, Member

_____

Date Project Accepted

To my caring parents and my supportive brother – you are my infrastructure!

To my graduate advisor, who has the remarkable ability to look for the silver lining in any cloud

Acknowledgements

I would like to express my gratitude to Dr. Jerry James, my graduate advisor and committee chair, for giving me this opportunity and guiding me throughout this research project presented here. He not only acted as a project manager promptly overwhelming me with technical inputs, but also supported me in keeping up the morale during the entire course, thereby making me enjoy his guidance to the hilt.

I would also like to thank Dr. Douglas Niehaus and Dr. John Gauch for serving as members on my committee. Special thanks to Dr. Niehaus, for providing us with his fully-grown brainchild, KURT, based on which this implementation has been done. I am also grateful to Hariharan and Mike Frisbie and their co-team members of the KURT project, for their cooperation by providing valuable inputs on the real-time operating system (KURT) that is mainly used for this project.

I would like to extend my thanks to Hans Harmon, Richard Stansbury and Eric Akers, who were involved in a part of this implementation in the spring of 2003.

Finally, my appreciation is extended to Brett Becker and Mike Hulet and their co-system/network administrators of our research center (Information and Telecommunication Technology Center), for their patience and expertise in maintaining the network and smoothly dealing with all our system-level issues every now and then.

Finally, my acknowledgements section would not be complete without acknowledging my parents and my brother, who constantly assist me on the background by showing their love and care toward my growth.

Abstract

Java is best suited for batch computations on objects and not so good for asynchronous, parallel and time-aware systems. There is always a need for a proven predictable behavior in industrial Java applications. The main idea of this project, Real-time Java (RTJ), is to enable Java for real-time systems. In short, real-time behavior guarantees predictability and deterministic outcome of your applications. This is done by implementing the Real-time specification for Java (RTSJ) provided by the Java Expert Group. RTSJ defines the requirements of a library and virtual machine to support Java on a real-time operating system (RTOS). Using the RTSJ as a guide, RTJ library has been created, which would enable the creation, verification, analysis, execution and management of Java threads whose correctness conditions include timeliness constraints (basically known as real-time threads).

Contents

List of Tables

List of Figures

1.0 Introduction

A greater degree of importance and complexity is always present in developing mission-critical applications that show real-time behavior. Real-time systems are those which should produce desired responses within a specified amount of time. Those time-critical systems which cannot afford even a slightest deviation are termed as strict or hard real-time systems, for instance aviation flight control. Soft real-time systems are those that still need to be real-time, but a deviation in the responses will not actually result in any disasters. Some of the practical applications that need control systems exhibiting mission-critical real-time functionality are Military, Aerospace, Satellite Systems, Nuclear Power Plant, Aircraft Control, Submarine Control, Factory Automation, Airport Aviation Flight Control, Energy & Power Systems Supply and commercial process control systems [1].

Java, which is a programmer-friendly language, has a lot of advantages (such as portability, reusability, object-oriented programming, garbage collection, etc.) that make it unique and user-friendly. It would not be an overstatement, if we say that almost any kind of application can be developed by using Java and its supplementary tools/concepts. Having this, it would be a godsend for the application developers if Java were to support certain real-time scenarios.

Understanding the need for the application of Java in real-time, a workshop at the National Institute of Standards and Technology (NIST) was organized to pool all the resources and propose a draft for Real-time Specification for Java (RTSJ). This led to the formation of the Real-time for Java Expert Group (RTJEG) in the Java Community Process (JCP) that has created a specification for incorporating the real-time functionality in Java. Development of real-time Java (RTJ) has always been a challenge for the Java community.

While commercial implementations exist for RTJ (like that of TimeSys and simpleRTJ), there does not appear to be an available implementation free to all users. Using the RTSJ as a guide, we have begun the development of an API, which would enable the creation, verification, analysis, execution and management of Java threads whose correctness conditions include timeliness constraints (basically known as real-time threads). This project aims to develop a real-time library for a real-time operating system (RTOS), which in our case is KURT (KU Real-Time) Linux. KURT is an RTOS developed in the University of Kansas. The Java version that is used is JDK1.4.1.

The rest of the paper is structured in a way that it first discusses the related work in Chapter 2. Then it explains the way this project has been implemented in Chapter 3. It then deals with the testing for the correctness of the results of the application that has been created to demonstrate the determinate scheduling of

events, in Chapter 4. Finally, Chapter 5 conclusively briefs the work and gives the future work in this area.

2.0 Related Work

2.1 TimeSys Java

TimeSys Corp. is a private company that mainly deals with supplying leading-edge Linux products to the developers of embedded systems. They have developed their own Linux real-time operating system called the TimeSys RTOS. This is considered to be the only single-kernel Linux RTOS [3].

TimeSys built the official reference implementation for RTSJ. They developed the world's first RTSJ-compliant JVM. TimeSys RTOS is being used to implement the real-time functionality in the JVM. The RTSJ-compliant JVM that they have developed is called JTime. JTime is a fully integrated customizable Java runtime environment that enables real-time Java development for embedded devices. JTime uses a real-time priority scheduler with greater precision. Eventhough they have pioneered this implementation, neither the RTOS nor the customized JVM along with implementation are made freely available to the users.

2.2 The simpleRTJ

The simple Real-Time Java is a clean room implementation of the Java language. In comparison with the other systems available in the market, this simpleRTJ only requires 18-24kb of memory to run. This being the major

advantage, simpleRTJ can be used for small embedded and consumer devices with only small amount of system memory. [4]

Its implementation also contains native calls to the operating system. Dynamic class loading disrupts the timing behavior. The fact that the simpleRTJ application is made to execute pre-linked Java applications so as to minimize the application start-up times has obviated the dynamic class loading process (of the Java application). The simpleRTJ also comes with a graphical debugger that is helpful in development.

This is known to be the only implementation of JVM that can execute byte codes on devices with bank switching memory model. Mobile phones, electronic toys, smart card readers etc. are some of the applications where simpleRTJ can be used.

3.0 Implementation

3.1 Overview

The total scope of this work (i.e. implementation) can be divided into two major categories viz., the creation of the RTJ library (javax.realtime package) and the development & testing of a driving example, which is a real-time application (to test the determinate scheduling of real-time threads).

3.2 RTJ Library

Real-Time Specification for Java was used as a template for developing RTJ library. That is, a package called javax.realtime has been created. The features that have been implemented can be classified into the following major categories: (The threads and scheduler require a couple of lower-level features like clock and timers)

- Real-time Clocks
- Real-time Timers
- Real-time Threads
- Real-time Scheduling

The current Java Virtual Machines use the native threads. So, instead of making changes to the JVM, the real-time functionality can be can be added by making calls to the native KURT functions. The Java Native Interface (JNI) is employed

for making these native calls. This gives us the convenience of calling the native methods through Java and also of modifying the Java objects from the native methods. The idea is to implement real-time Java with as little dependency on the JVM as possible.

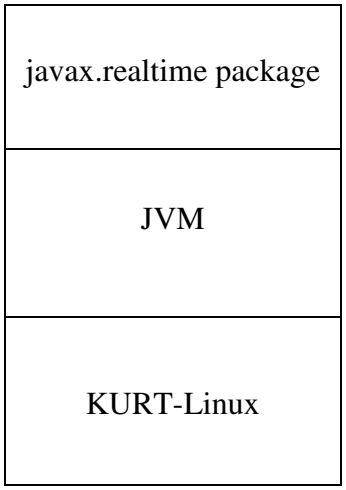

Fig 3.1: Overall view of RTJ library added to JVM & KURT

3.2.1 Clocks and Timers

3.2.1.1 Clocks

As the complex threads and scheduler are built on the Time classes, we have this as the starting point for our implementation. The classes (related to the clock/time) that are implemented are:

Clock.java

The Clock abstract class defines the basic functionality for a clock in Java. Through the Clock.getTime() method, the current time will be available. This basic clock class can be extended for numerous other types of clock such as a real-time clock.

KURTClock.java

The KURTClock represents an implementation of the abstract clock class. This clock utilizes the enhancements of KURT on the system clock. To access the current timeofday, two native methods have been implemented: nativeGetResolution(), obtains the clock resolution, and nativeGetTime(), calls getTimeofDay().

HighResolutionTime.java

The HighResolutionTime abstract class is base class for AbsoluteTime, RelativeTime, and RationalTime. It is used to store a number of milliseconds and nanoseconds. This class also implements the compareTo class that allows comparison between time events to be simplified.

AbsoluteTime.java

The AbsoluteTime class produces a HighResolutionTime object that represents a specific time in milliseconds plus nanoseconds past the epoch (January 1, 1970 00:00:00 GMT). This is used to replace the given standard representation of time for Java with nanosecond resolution. Numerous math functions are available to add or subtract RelativeTime values from an AbsoluteTime, or find the difference between two AbsoluteTime values.

RelativeTime.java

The RelativeTime class represents a period of time of length milliseconds + nanoseconds long. A relative time can be defined relative to the current time from Clock.getTime(). Numerous methods are available to convert, add, or subtract RelativeTime values to create new RelativeTime or AbsoluteTime values.

RationalTime.java

The RationalTime class takes a time RelativeTime object and a frequency value and calculates the interarrival times by dividing the RelativeTime by the frequency.

3.2.1.2 Timers

Timers represent a timed event that fires at a given time relative to a given clock. The following timer classes have been implemented from the RTSJ:

Timer.java

The Timer class is an abstract class that defines the basic functionality for the timer classes. This file extends an AsynchronousEvent which allows handlers to be assigned to a timer. Once the timer expires, all of the handlers are fired. In order to implement the timer, an internal class, which extends RealtimeThread, was created. This class worked for both OneShotTimers and PeriodicTimers. JNI method calls to nanosleep were utilized. Once the sleep terminated, the Timer event would fire.

OneShotTimer.java

The OneShotTimer class behaves as the name suggests. The timer is assigned a start time, AbsoluteTime, and a handler. Once the timer is told to start, it will fire when the given time has been reached. If the time has already passed, the timer will fire immediately.

PeriodicTimer.java

The PeriodicTimer class performs similar to OneShotTimer. However, it also takes a time interval, RelativeTime, which represents the time between successive firings of the timer. Thus, the timer fires once at the given start time. After that firing, the timer will fire after every interval has passed.

3.2.2 Asynchronous Events

Each event can have a set of handlers with it that react when the event is fired. When the event occurs, the handler is placed on the scheduler. In the event of an error or time violation during a real-time thread, an asynchronous event handler is utilized to respond. The classes were implemetned to support asynchronous events are:

AsyncEvent

The AsyncEvent class defines an asynchronous event. One or more handlers are associated with the event. When the event is fired, each handler is scheduled to execute.

AsyncEventHandler

The AsyncEventHandler class defines the basic structure of an asynchronous event handler. It implements Schedulable so that it may be placed on the scheduler. Like threads, the AsyncEventHandler contains many parameters (release, scheduling, etc) that allow the event to be properly scheduled as a real-time task.

3.2.3 Thread parameters

SchedulingParameters.java

The abstract class SchedulingParameters is the parent class for PriorityParameters, KURTSchedulingParameters, and ImportanceParameters. The parameters set by these classes determine the scheduling behavior of the schedulable object they support. Modifications to these parameters will result in a change in these behaviors.

PriorityParameters.java

The class PriorityParameters supports priority-based schedulers by storing an integer value for the Priority of the schedulable object.

KURTSchedulingParameters.java

The class KURTSchedulingParameters extends SchedulingParameters. It was designed to help make our real-time threads interoperable with KURT. This class performs the same basic functions as the rtparam struct found in KURT. This parameter includes real-time process modes and system-level real-time

modes. The constructor for KURTSchedulingParameters takes the values real-time task ID, the period between executions, the execution time, a name for the real-time task, and the KURT real-time mode.

ImportanceParameters.java

The ImporatanceParameters class extends PriorityParameters. In addition to a priority, this class also takes an importance value. During an overload situation, a scheduler (priority or rate-monotonic) could use this value to act as a tiebreaker for threads of the same priority.

ReleaseParameters.java

The abstract class ReleaseParameters describes the release characteristics for real-time threads. Release parameters include two RelativeTime values: cost and deadline. Cost is the threads processing time per interval. The deadline is the latest the thread must complete from when it is released. Two AsyncEventHandler objects are also assigned. First, overrunHandler is fired if the execution of the object exceeds the given cost. The missHandler is fired if the thread is still execution after the deadline.

PeriodicParameters.java

The PeriodicParameters class extends the abstract class ReleaseParameters. In addition to the above parameters, a period is assigned which represents the time between unblocks of the schedulable object.

AperodicParameters.java

The AperiodicParameters class extends the abstract class ReleaseParamters. In actuality, this class provides no unique additions to ReleaseParameters. It simply acts as a placeholder to make ReleaseParameters appear hierarchical.

ProcessingGroupParameters.java

The class ProcessingGroupParameters allows common parameters to be assigned to a group of schedulable objects. Unlike other parameters classes, an instance of ProcessingGroupParameters may apply to multiple objects.

3.2.4 Threads

RealtimeThread.java

The class RealtimeThread extends the java.lang.Thread the standard Java thread class. Unlike tranditional Java threads, these threads also store the various parameter classes discussed above. The parameters are interpreted by the scheduler to ensure that the threads operate in a real-time fashion. Once the thread has been approved by the scheduler (discussed later), a JNI call is used to schedule the thread as a process in KURT.

3.2.5 Scheduling

Schedulable.java

The interface Schedulable defines the requirements of classes that will be placed on the scheduler. Schedulable is implemented by AsynchronousEventHandler and RealtimeThread.

Scheduler.java

The abstract class Scheduler defines the basic functionality that is required for a real-time thread scheduler. This class is used as the parent class for various scheduling policies including priority scheduling and KURT scheduling.

PriorityScheduler.java

The implementation of the PriorityScheduler is required by the RTJ specification. Maximum and minimum priorities are defined to determine the bounds for priority assignments.

KURTScheduler.java

The class KURTScheduler is where Real-time Java becomes real-time. Since we utilize native calls to schedule the thread directly in KURT, this class is only utilized to verify the feasibility of a thread. Based on the various parameters given to the thread, the scheduler determines if the thread may be scheduled or not.

3.3 Application – Bouncing Ball

3.3.1 Bouncing Ball

The RTJ library (javax.realtime package) that has been created by invoking native calls to communicate with KURT provides the real-time functionality. To have it implemented in and check the results for a practical application, a multi-threaded and graphical Bouncing ball application (`Bounce2.java`) has been created. Java Swing and the Abstract Window Toolkit (AWT) are utilized to generate the graphics.

This application demonstrates a multi-threaded application simulating the balls bouncing against the wall. It can be noticed that at low loads, the balls move faster and as the load increases, refreshing takes a longer time and the threads are non-deterministic and their movement is not jerkier. When the real-time functionality is incorporated, the threads follow a deterministic pattern and as a result, the movement of the balls is much smoother.

This application has three features of controlling the motion of the ball viz., add a ball, pause & resume and reset the screen. Whenever a ball is added to the screen, a new thread is spawned and started to run. Each thread is linked to the each of the balls on the screen. This thread will run and cause the ball to move.

When reset, all the balls are removed from the container and all the threads are completely stopped. But, when the Pause button is clicked, all the balls are

stopped i.e. they are still in the container and are being displayed without getting moved from that location. But, all the threads that are linked to the balls are terminated. And, once the balls are made to Resume, then the movement of the each of the balls start from the location where it was left, by making all the threads (linked to the balls) to run.

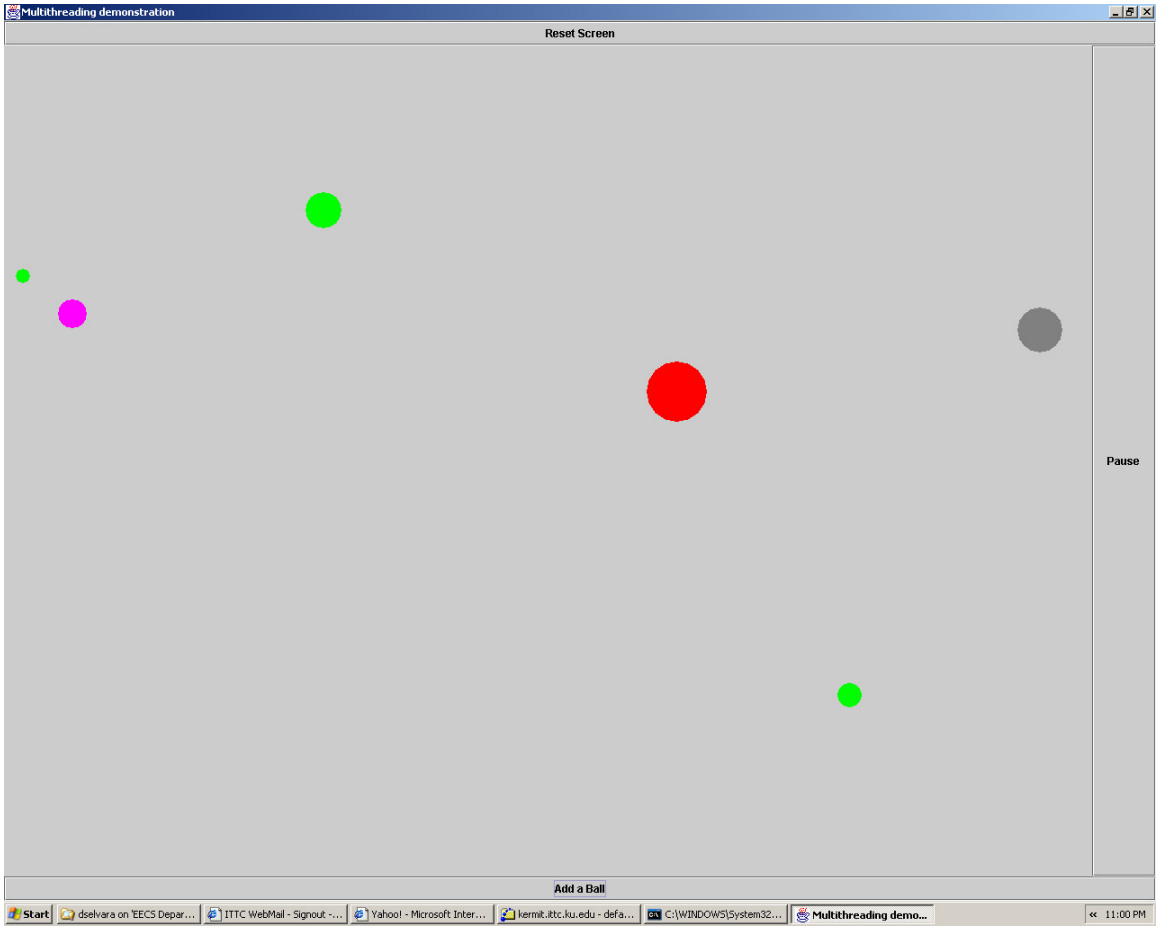

Fig 3.2: Bouncing Ball – multi-threaded application

3.3.2 Real-time threads and Dynamic Scheduling with KURT

From Java, JNI is employed to communicate with the real-time operating system (KURT). Native calls are made from the Java program and they in turn call the core KURT functions written in C code. A Java wrapper class (`DynamicSchedule.java`), containing the prototypes for the native methods (which are defined in a C program – `Java_Dynamic_Schedule.c`), is written to invoke calls from Java.

```
public class DynamicSchedule {
      ...

         public native static int nativeKURT_open();
         public native static int nativeRegister(int kurtdev, int
ballNum);
         public native static int nativeSchedule(int kurtdev, int
total_balls, int[] rt_ids, int speed);
         public native static void nativeSuspend(int kurtdev);
         public native static void nativeDisableSchedule(int kurtdev);
         public native static void nativeKURT_close(int kurtdev);
      ...

}
```

Fig 3.3: Dynamic Schedule – wrapper class

When starting the application, a kurt device is opened. Whenever a new ball i.e. threads, starts to run, first it is registered with KURT as a real-time process. A real-time ID is assigned to it. A name for that particular ball is being assigned and the entire rt_params structure is being internally maintained by KURT.

```
struct rtparams {
 int rt_id;
 unsigned long period;
```

```
 unsigned long exec_time;
 char rt_name[MAX_RT_NAME_LENGTH];
 unsigned int rt_mode;
};
```

Fig 3.4: `rtparams` Data Structure

The total number of balls created in the application (at any point of time), the ball number for each ball, and all the real-time IDs are kept track in the program. The real-time system mode for which each thread is being registered is a combination of the KURT_EXPLICIT and KURT_EPISODIC modes.

Once a single ball, for instance, has been created, it now has to be scheduled. Threads are constantly getting created. Each time a thread is created and started to run, this thread has to be dynamically added to the scheduling of processes. For scheduling with KURT, the concept of explicit time scheduling is employed. The main advantages of this kind of scheduling in KURT are,

1. scheduling can be done by specifying times in micro- and nanosecond resolution and,
2. schedule of events is guaranteed to execute within the specified time

A structure for the schedule is created. A real-time schedule queue (which is nothing but an array) of rt_timer_list structures is being created and populated with the specific times when each thread has to be started, suspended and woken up. The timer flag (for all the threads) has to be set to KURT_WAKEUP.

The structure also contains the real-time ID of each of the threads so as to enable it to run the thread when it is woken up.

```
struct rt_timer_list {
 struct timer_list timer;
 struct rt_timer_list *next;
 int programmed;
 int expired;
};
```

Fig 3.5: `rt_timer_list` Data Structure

Now that the schedule has been created, it has to be submitted to system using submit_dynamic_schedule() function. The scheduling mode needs to be specified as SCHED_KURT_PREFERRED.

After submitting a new schedule to the system, the particular thread is made to suspend (rt_suspend) for the millisecond duration that is given as the command-line parameter while executing the bouncing ball program. This suspend mode is the combination of SUSPEND_IF_NRT and START_SCHED. When this thread gets woken up after the specified duration of time, it is made to move (by invoking the move method). And when another new thread gets created, the same process gets repeated and the bouncing of all the balls goes on smoothly. The point to be noted is that when the thread starts to run, a new schedule (including all the threads that are running so far) is submitted. It has been designed internally that submitting a new schedule overrides the older/existing schedule queue. When the application ends, this dynamic scheduling has to be

disabled (disable_dynamic_schedule function); else the schedule keeps on running without getting terminated. Finally, the kurt device is closed (by the close function).

4.0 Evaluation

This chapter discusses how the bounding ball application is tested and the tools used for tracking the timings of the real-time threads. It explains the following: the list of system level threads that are created when a Java application is created. And subsequently, the removal of GC and Compile threads in the JVM (j2sdk-1.4.1) – for testing (as they need to be gotten rid of for real-time testing and evaluation), instrumentation using DSUI tool and how it is incorporated using JNI, the actual testing of the bouncing ball application, its results (using the .xml file – which is the output of the DSUI instrumentation) and the comparison.

4.1 System Threads and their Problems

When a simple Java application (for instance, a “Hello World!” program) is executed in a Linux box, it spawns a total of nine spooky threads. The Java Debugger (JDB) is used to display six of the threads that get created. The system level threads are Reference Handler, Finalizer, Signal Dispatcher, CompileThread0 and those under the Group Main (i.e. user-level threads) are main thread and Thread-0 (the logger thread). If the Java program creates any thread, then the naming convention of Thread-1, Thread-2 and so on is followed.

```
main[1] threads
Group system:
(java.lang.ref.Reference$ReferenceHandler)0xee Reference Handler cond.
waiting
```

```
(java.lang.ref.Finalizer$FinalizerThread)0xed  Finalizer          cond.
waiting
(java.lang.Thread)0xeb                         Signal Dispatcher  running
(java.lang.Thread)0xec                         CompileThread0     cond.
waiting
Group main:
(java.lang.Thread)0x1                           main              running
(java.util.logging.LogManager$Cleaner)0x123     Thread-0          unknown
(dummyThread)0x124                              Thread-1          cond.
waiting
(dummyThread)0x125                              Thread-2          cond.
waiting
```

Fig 4.1: System Threads - output of JDB

As the system threads have a higher priority and can override any other activities especially the real-time threads that we have created and pose a problem, they need to gotten rid of the JVM. This is because the Memory management and the Garbage collection (GC) features have not been taken care of in our implementation.

One part of the GC is the Finalizer thread and the other is the Reference Handler thread. Removal of the reference handler makes our application not to support weak references. The non-deterministic GC has to be suspended, so the finalizer is also removed. As far as the Signal Dispatcher thread is concerned, this is left untouched. This can be removed only if the timing between the signals can be controlled by our real-time system.

The CompileThread0, is just a best-effort thread, not a real-time thread. This could be a JIT compiler or a HotSpot compiler. It basically runs only if there are

no other real-time process is running. This poses the real-time processes a problem only if it acquires a lock. So, this also has been removed.

The Thread-0 is the one that closes all the open handlers. This is actually spawned only when the program exits. There is one more thread called the Secondary Finalizer Thread that is linked to the Finalizer process. But even this does its job only on exit. There are no problems with these threads and so are left untouched.

The entire Java Development Kit including the HotSpot (for customizing and development purposes) is got from the java.sun.com downloads. The kit that is being customized is the J2SDK-1.4.1, which is the latest version that Sun Microsystems has released. Summarizing all the above, the threads that have been removed in this JVM are Reference Handler, Finalizer and the CompileThread0. The commented locations for the above in the J2SDK are as follows:

- Finalizer thread in: `/projects/kurt/rtj/j2sdk-1.4.0/j2se/src/share/classes/java/lang/ref/Finalizer.java`
- Reference Handler thread in: `/projects/kurt/rtj/j2sdk-1.4.0/j2se/src/share/classes/java/lang/ref/Reference.java` and,
- CompileThread0 thread in: `/projects/kurt/rtj/j2sdk-1.4.0/j2se/src/share/classes/java/lang/Thread.java`

These are the threads that get created (i.e. new Thread …) from the following locations: /java/lang, /java/util, /java/nio – in the J2SDK directory structure. Those, if any, under the /java/awt, swing and /sun/corba are not taken into account.

For these above changes to take effect the Java kit is built again and the resulting executable is used to compile and execute our driving example, the bouncing ball program.

```
make DEV_ONLY=true ALT_BOOTDIR=/tools/java/i586/j2sdk1.4.0/
ALT_OUTPUTDIR=/projects/kurt/rtj/j2sdk-1.4.1/build
ALT_MOZILLA_PATH=/projects/kurt/rtj/j2sdk-1.4.1/devtools
ALT_DEVTOOLS_PATH=/usr/local/bin
ALT_JAVAWS_PATH=/tools/java/i586/j2sdk1.4.2/jre/javaws
ALT_MOTIF_DIR=/projects/kurt/rtj/j2sdk-1.4.1/motif
```

Fig 4.2: Build J2SDK-1.4.1

4.2 Data Streams – Instrumentation & Output

4.2.1 DSUI

The Data Streams User Interface (called DSUI) that comes along with KURT is used to instrument the bouncing ball Java program so as to check the various timings of the threads created.

This is basically a tool that helps to gather the instrumentation data during the execution of programs. The activity of the real-time threads that are used in the application can be measured using high-resolution timers, which are made possible by incorporating the DSUI instrumentation points in the program.

4.2.2 DSUI using JNI

The DSUI calls are to be made from the Java Bouncing ball program by native method calls. This is done using JNI. A wrapper class (`dsui_Bounce2.java`) for the DSUI functions is written in Java. These methods in turn call the JNI-linked C functions (that are defined in `Java_dsui_Bounce2.c`) which make the actual DSUI call.

```
public class dsui_Bounce2 {
       . . .
         public static final int APPLICATION_FAM = 1;
         public static final int EVENT_OPEN = 5;
         public static final int EVENT_START = 4;
         public static final int EVENT_SUBMIT_SCHED = 3;
         public static final int EVENT_SUSPEND = 2;
         public static final int EVENT_WAKEUP = 1;
         public static final int EVENT_EXIT = 0;
         public static final int COUNTER_BOUNCE_SPEED_CONST = 0;

         public static final boolean DSUI_SET = true;

         public native static void native_DSUI_INIT(String identifier,
String enabled);
         public native static void native_DSUI_EVENT_LOG(int family,
int event, int set, int len, Object data);
         public native static void native_DSUI_RESET_COUNTER(int
family, int counter);
         public native static void native_DSUI_ADD_TO_COUNTER(int
family, int counter, int amount);
         public native static void native_DSUI_LOG_COUNTER(int family,
int counter);
}
```

Fig 4.3: DSUI – wrapper class

4.2.3 Instrumentation – family, events and counters

A family with the name APPLICATION is created and is assigned a family ID (which is 1). All the events and a counter come under this family. The counter, named as COUNTER_BOUNCE_SPEED_CONST, is just used to store the value of the speed that is entered as a command-line parameter by the user, (instead of having the values incremented as in a counter). The various events that mark the instrumentation points in the program are (along with their entity IDs):

- EVENT_OPEN 5
- EVENT_START 4
- EVENT_SUBMIT_SCHED 3
- EVENT_SUSPEND 2
- EVENT_WAKEUP 1
- EVENT_EXIT 0

These information pertaining to the events, counters and the family are contained in a namespace file (`namespace_Bounce2.dsui`) which has to be inputted to the DSUI parser.

All these events are logged using the DSUI tool along with the time stamp when it occurs. The timestamp is a 64-bit counter and not an actual time value – this counter value has to be converted to get the actual time when it operates. The

EVENT_OPEN is when a KURT device first gets opened in the application. Whenever a new ball (that is created by adding a ball) starts running EVENT_START is logged. Once started running, a schedule has to be made and submitted, whenever a new ball is started to run. Each time a fresh (dynamic) schedule is submitted, EVENT_SUBMIT_SCHED is logged. After scheduling, a thread gets suspended periodically and then moves then gets woken up and the gets suspended and moves and so on. So, EVENT_SUSPEND is when it gets suspended and EVENT_WAKEUP is when it gets woken up. EVENT_EXIT notes the occurrence of the KURT device getting closed – which is normally towards the end of the application (either smoothly or whenever an exception occurs).

When the application starts, the DSUI has to be initialized by DSUI_INIT() function. Event logging is done through DSUI_EVENT_LOG() function. DSUI_RESET_COUNTER(), DSUI_ADD_TO_COUNTER() and DSUI_LOG_COUNTER() functions are used to reset, add to and log a counter.

4.2.4 Execution and output

Finally, when the entire application (the DSUI-instrumented one) is executed, a binary file which has collected all the necessary information (in a machine-recognizable form) is created in /tmp/ of the machine where the application is executed. This binary file is parsed and the information is extracted in the form of an xml file using the bintoxml parser – which is done in this format: bintoxml

dsui advanced-events.18047 output.xml namespace_Bounce2.dsui; which in our case is,

```
bintoxml dsui /tmp/advanced-events.18047 output.xml
namespace_Bounce2.dsui
```

where, ‘advanced-events.18047’ is the binary file that gets created, ‘output.xml’ is the results file in xml format and ‘namespace_Bounce2.dsui’ is the namespace file which contains the family information.

The results are displayed in terms of a 64-bit counter, whose exact timing and comparison will be explained in the next subsection that deals with the testing – where the initial part of the output.xml is shown.

4.3 Testing and Results

4.3.1 User level

The bouncing ball Java program is made to accept the speed with which the balls should bounce as a command-line parameter while executing java i.e. `java Bounce2 10`. The speed is given in milliseconds – which in this case is 10ms. This speed is stored (as a counter) and is logged in the DSUI.

For running the application finally, one needs to have KURT installed in his machine and the compiling and linking paths must be properly given. The build

for the all files is in a shell script `build-Bounce2.sh` (`/projects/kurt/rtj/application/working/`).

Requirements for executing the application:

- Check if the environment variable LD_LIBRARY_PATH is set to your current working directory or the place where the .so files (libraries of the C code – which are `librtj-kurt.so` and `librtj-kurt-dsui.so`) are located.
- Your login should have the root shell permissions on the machine to execute the KURT commands and making the threads real-time. The machine that I have been working is `testbed43.ittc.ku.edu`.

```
testbed55 [6] # java Bounce2 10
kurtdev is: 20
Real-time ID# (as assigned by KURT) are:
ball1: 255
ball2: 254
ball3: 253
ball4: 252
ball5: 251
ball6: 250
ball7: 249
ball8: 248
ball9: 247
```

Fig 4.4: Output of `Bounce2.java` – real-time ID#

For testing the real-time scheduling of the balls, nine balls (i.e. nine real-time threads) are created. Every time a new ball/thread is created, a new schedule, appending the new thread, is constructed and submitted. All the timers are wakeup timers. The schedule causes 10ms time period between events i.e.

each (wakeup) timer is 10ms apart. Thus, the difference between the times of two consecutive wakeup events gives the duration of sleep of a thread or in other words, shows if the wakeup timers are triggered properly as expected. The basic idea is that, having the speed of ball to be 10ms, each thread should sleep for 10ms and then run and do the process in its assigned quantum of time. Following is the comparison of actual results versus the expected time.

Processor speed = 1399.380 MHz; which means 1399380000 cycles per second.

Timestamp counter of the wakeup events	**Diff. bet. 2 wakeup events**	**Diff. in milliseconds = (diff./cycles per second)* 1000**	**Deviation (in ms) from the expected 10ms**
6838399036229			
6838413029334	13993105	9.9995	0.0004
6838427395794	14366460	10.2663	0.266
6838441011820	13616026	9.7300	0.269
6838455003761	13991941	9.9986	0.0013
6838469000660	13996899	10.0022	0.0022
6838482990727	13990067	9.9973	0.0026
6838496984194	13993467	9.9997	0.0002
6838510979976	13995782	10.0014	0.0014
6838524971172	13991196	9.9981	0.0018

Average error: 0.0605

Standard Deviation: 0.1173

Table 4.1: Results of scheduling of events – timestamp calculation

Based on the timestamp counter results from the DSUI log, the timestamps are converted to milliseconds in the below table. A sample of 10 wakeup consecutive events is taken (from the DSUI log). The difference between two wakeup events is taken and is checked with the input speed value of 10ms. The deviation is found out to be less and bearable. The mean error comes out to be 0.0605 milliseconds and the closest result is for the eighth sample which is deviated only by 0.2 microseconds.

The `kurt_status` command gives the list of currently running real-time processes along with their real-time IDs and their `pid`s. This clearly shows the status of all the processes at any given point of time. It shows how many times a particular thread/process has missed its schedule – which in our case is nil for all the real-time processes proving that everything has fallen in place and there has been no missed schedules.

It also displays the number of times a process,

- got awoken.
- got suspended (rt_suspend).
- got aborted.
- got switched to another i.e. it is being preempted.

The output of the `kurt_status` command is shown in Appendix – A.

4.3.2 Accuracy of Scheduling – Kernel level

The accuracy of the actual scheduling (that takes place in the kernel) is found out using the kernel level instrumentation points through DSKI (Data Streams Kernel Interface). Now, the DSUI is disabled and the application with speed of the ball as the command-line parameter. In our case, the speed of the balls is taken as 10 ms, which means each thread is to sleep for 10ms and then resume its movement.

In these experiments to follow, the application starts with a fixed number of threads/balls running, which in this case is 10. All these are real-time threads that start running when the application is executed.

The same concept of explicit time scheduling is followed by scheduling the balls for 10ms periods i.e. the threads are scheduled to wake up at 0, 10, 20 ,30ms and so on. For checking when actually the RT threads get awoken, the delta value (difference between the actual time a thread gets triggered and the specified wake-up time) of each wake-up event due to this schedule is plotted as a histogram through DSKI. The histogram that is instrumented in the kernel for this purpose is HIST_RT_USER_DELAY (which comes under the KURT_PROFILE family of the namespace file for DSKI). The unit of this delta measurement is in timestamp counter values – based on which the graph is plotted. This is run in a 1399 MHz machine. So, 500 time ticks (that is plotted in the graph) is about 0.36 microsecond and 50000 comes to around 36 microseconds.

When the application is running, `compst` program is used to track the values at the instrumentation points in the kernel. The duration of observation, lower & upper bounds of the histogram and the size of buckets etc. are to be given as command-line parameters for this `compst` program.

Experiments are conducted under the following three conditions (under no load, and with network load & disk activity running in the background). All the experiments are observed for duration of 30 seconds and a histogram is plotted. The upper and lower bounds of the plots are specified based on the number of events occurring in that range, for each case.

4.3.2.1 Unloaded

The Bouncing ball application is the only job that is running on the machine. There are no other external loads.

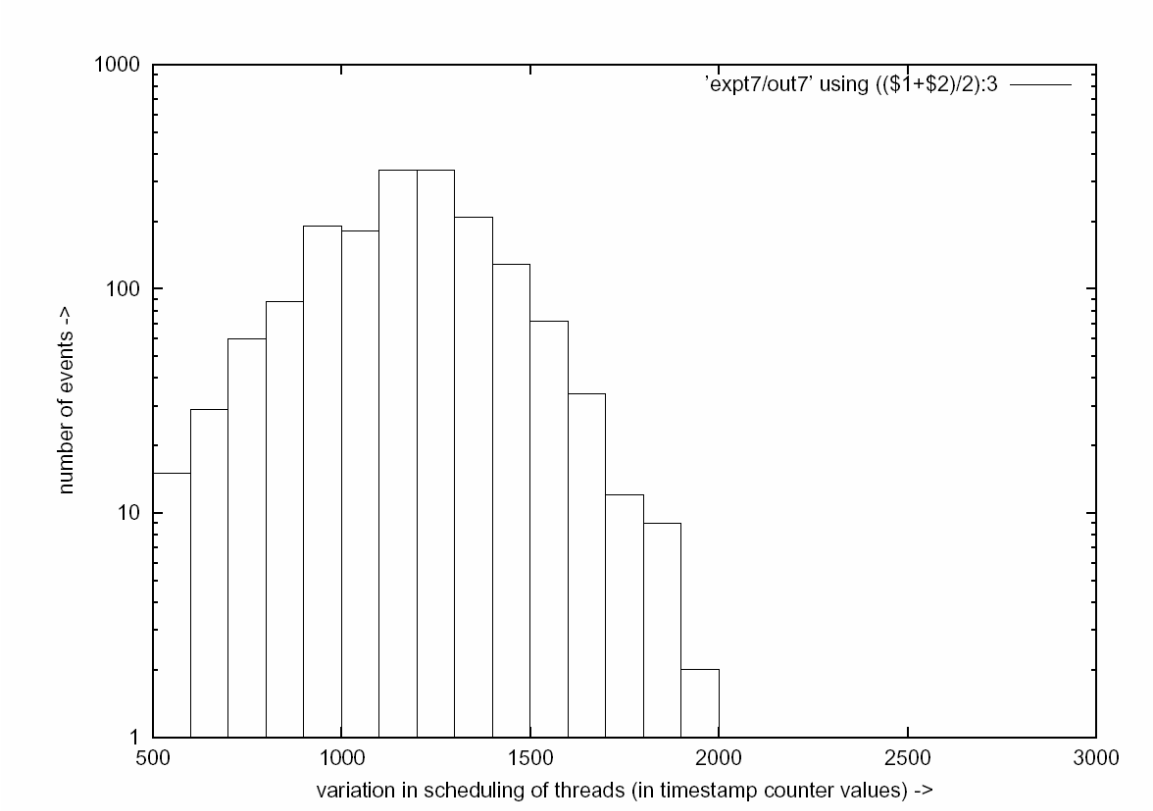

Fig 4.5: Histogram results – no load condition

In this case, the delay of most of the events getting switched is at a peak between 1000 and 1500, which corresponds to 0.72 and 1.07 microseconds. The maximum of the delay value also comes to be around 2000 timer ticks (1.43 microseconds). As the system is not loaded much, this degree of accuracy, in spite of the overload from Java and JNI, is achieved.

4.3.2.2 Disk Load

In an idea to simulate a surge of disk activities, which greatly prove to be a hindrance to the RT processes, a kernel is compiled (in the local disk) in the

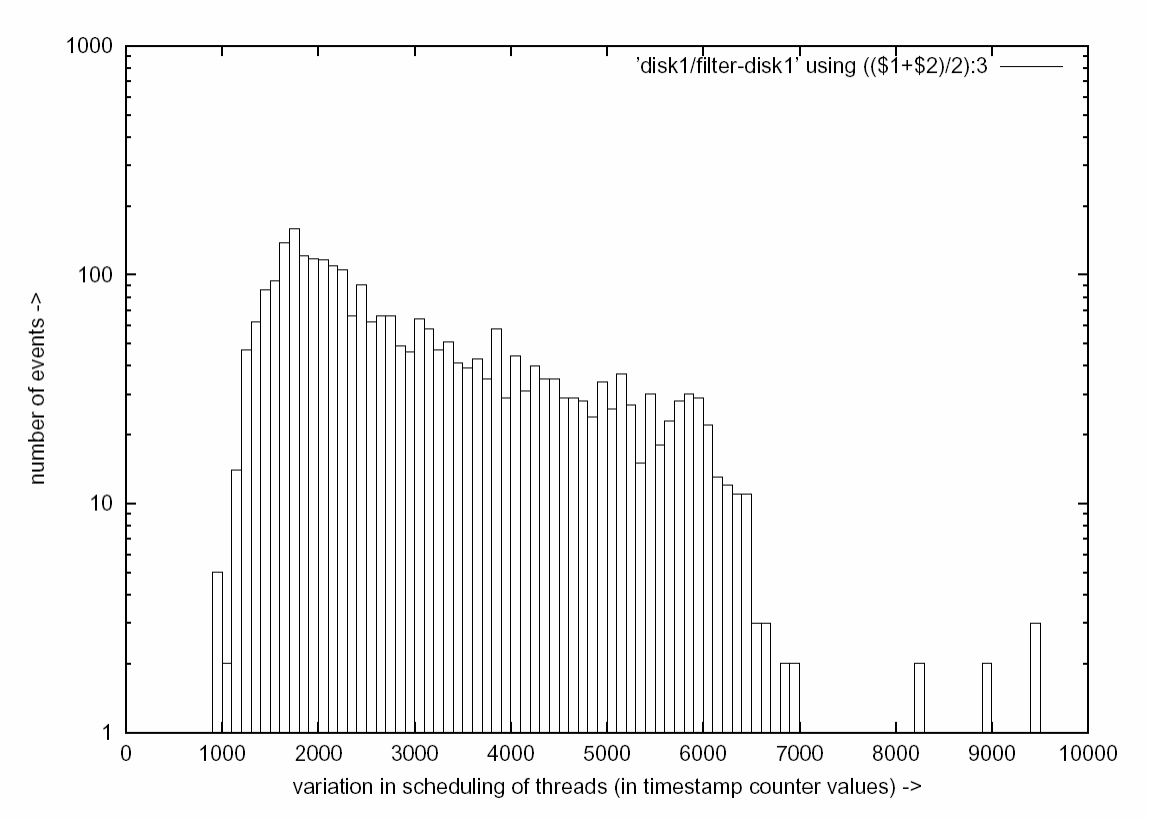

Fig 4.6: Histogram results – with disk load

background. The results are taken from the DSKI (for 30 seconds) simultaneously when the application and the kernel compilation are running in the same machine and the values are plotted as a graph.

Frequent disk accesses make a considerable impact when compared to the unloaded condition. It starts at 1000 and the peak is around 2000 (1.43 microsecond), which was the maximum delay in the unloaded condition. The major cluster of events occurs in the range of 1000 to 7000 (5 microseconds) and a few spikes around 9000 (6.4 microseconds). Moreover, when the xml output of the DSKI is analyzed, it is observed that about 220 events (out of 3000) exceed the delay value of 10000 (7.2 microseconds). The disk activities clearly explain the increased delay which results from the RT processes getting impacted.

4.3.2.3 Network Load

Another way of thrusting load into the system is to interrupt the machine by constantly pinging Ethernet messages from a remote machine. Messages are pinged from a terminal in another network to the machine where the bouncing ball application is running. Now, the deviation is observed through DSKI for 30 seconds.

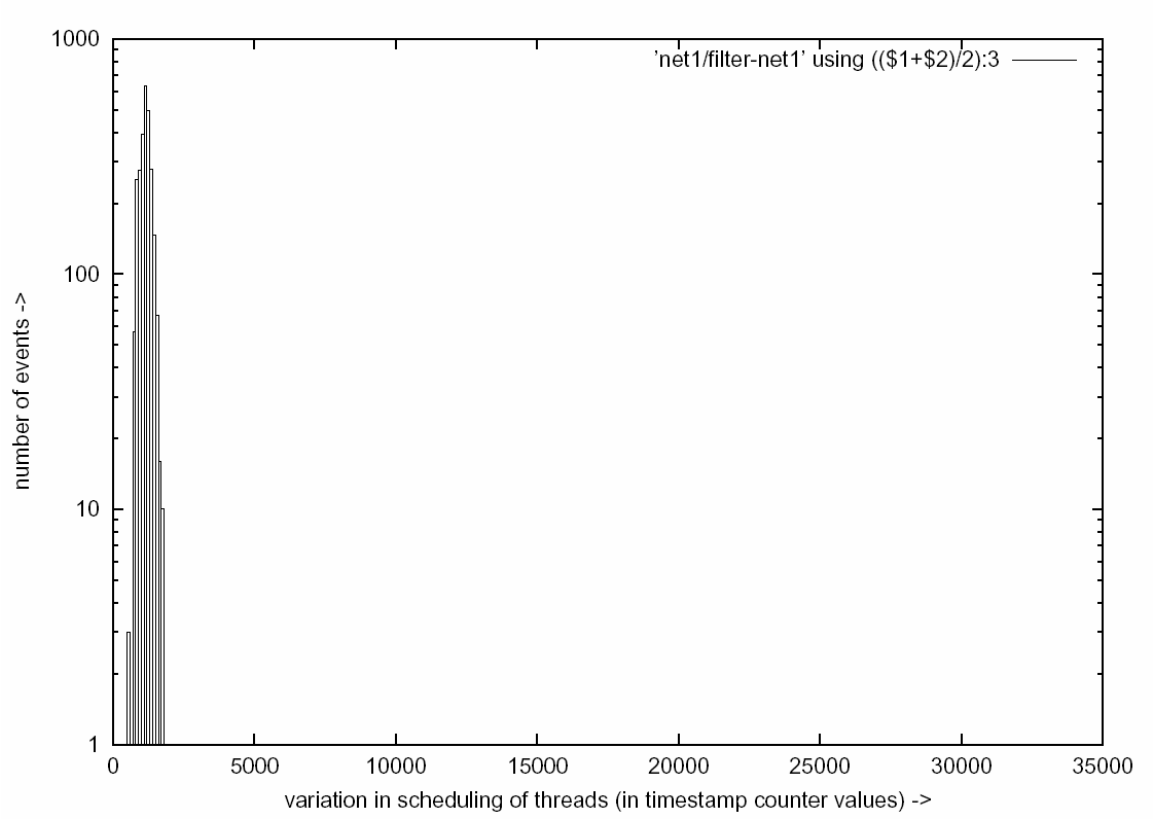

Fig 4.7: Histogram results – with network load

This shows that the majority of the delay values lie within 2500 (1.8 microseconds). The xml file from the DSKI also says that around 360 events (out of 3000) fall outside of 35000 (25 microseconds), which is a relatively considerable variance from the expected behavior.

5.0 Conclusion and Future Work

5.1 Conclusion

This project aims at implementing the RTSJ for JVM using a real-time operating system. The gaps of the Java programming language for Real-Time Systems have been bridged by having developed the API for Java Real-Time, package name "javax.realtime". This work aims at a much simpler and open-to-all implementation. The real-time JVM was a result of small modifications to the Java specification language. We will see Java dominate the other software for the automation and real-time computer control industries over the coming years.

5.2 Future Work – Pointers

The implementation is done by adding a library, javax.realtime package, (as per the RTSJ) to the Java package and then by customizing the JVM. The major areas that were focused in this project are threads, scheduling, clocks and timers. As a future work, the features of memory management and the garbage collection (GC) have to be incorporated. The RTSJ specifies the skeleton classes for the memory management section, using which it can be implemented.

5.2.1 Memory Management

The memory management and the GC also contribute to the unpredictable behavior of the Java application. Memory management in Java introduces unacceptable latencies in embedded and industrial Java applications [3]. The main idea of the existing JVM is that memory is allocated from the heap rather than through a specific allocation policy; so the results could be unpredictable and non-deterministic. For dealing with the memory management, the threads are to be made No Heap Real-Time threads (NHRT) that guarantee deterministic outcome of your applications. By making them no-heap, it can be ensured that those threads can preempt the garbage collector to guarantee that your applications do what you want, when you need them to.

Dynamic checking has to be done so that NHRT threads never access a location containing a reference into garbage-collected heap. At every read, check to make sure result does not point into garbage-collected heap. Similarly, at every write, check to make sure not overwriting reference into GC heap. If checks fail, throw exception [7].

5.2.2 Garbage Collection Algorithm – Non-determinism

As the Garbage Collector (GC) poses the problem acting in a non-deterministic way, the threads need to be able to preempt the GC. One of the problems that this JVM poses in this regard is the under-specification of the GC algorithm. The

basic algorithm and its triggering conditions are not clearly specified and could be non-predictable and non-deterministic.

Sun's Java HotSpot VM, v1.4.1 uses a relatively simpler conservative garbage collection algorithm that invokes periodically and reclaims as much memory as possible. A much closer description of Java's garbage collection algorithm might be 'a compacting, mark-sweep collector with some conservative scanning'.

The Java HotSpot VM employs a state-of-the-art generational copying collector, which provides two major benefits viz., increased allocation speed and overall garbage collection efficiency for most programs when compared to non-generational collectors. This attributes to a proportional decrease in the frequency and duration of user-perceivable garbage collection pauses. To sum it up, the objects (that are to be collected) are mainly classified into nursery objects and older generation objects.

Generational collector is exploited to clear away short-lived objects (nurseries), by allocating all newly created objects contiguously in a stack-like fashion into an object nursery. The allocation becomes fast since it merely involves updating a single pointer and performing a single check for nursery overflow. Moreover, by the time the nursery overflows, most of the objects in the nursery are already dead, allowing the garbage collector to simply move the few surviving objects elsewhere, and avoid doing any reclamation work for dead objects in the nursery.

For older generation objects, a mark-compact collection algorithm is used. The Java HotSpot VM uses a standard mark-compact collection algorithm, which traverses the entire graph of live objects from its roots, then sweeps through the memory, compacting away the gaps left by dead objects. By compacting gaps in the heap, rather than collecting them into a free list, memory fragmentation is eliminated, and old object allocation is streamlined by eliminating free-list searching. [8]

Bibliography

[1] Sione Palu. “Real-time Specification for Java (RTSJ)”, June 18, 2002.

[2] Will Dinkel, Douglas Niehaus, Michael Frisbie, Jacob Woltersdorf. “KURT – Linux User Manual”, University of Kansas March 29, 2002. (http://www.ittc.ku.edu/kurt)

[3] TimeSys Corp. www.timesys.com

[4] The simpleRTJ. http://www.rtjcom.com

[5] Peter Dibble. “Real-Time Java[tm] Platform Programming”

[6] Real-Time Java Expert Group (www.rtj.org)

[7] International Workshop for Embedded Systems, University of California, Berkeley (http://www-cad.eecs.berkeley.edu/~fresco/embedded-software/)

[8] Kim Clohessy. Java runtime components for Embedded Revolution (http://www.microjava.com/articles/techtalk/runtime)

[9] Hans Harmon, Eric Akers, Richard Stansbury. “Real-Time Java for KURT-Linux”, May 14, 2003.

Appendix – A

```
testbed55 [7] % kurt_status -i 2
handled late    dropped invalid wdog    baddog1 baddog2 baddog3
573     225     0       0       0       0       0       0
rt_id   pid      woken     missed    rt_susp     aborts nonrt_susp
switches
handled late    dropped invalid wdog    baddog1 baddog2 baddog3
573     225     0       0       0       0       0       0
rt_id   pid      woken     missed    rt_susp     aborts nonrt_susp
switches
handled late    dropped invalid wdog    baddog1 baddog2 baddog3
573     225     0       0       0       0       0       0
rt_id   pid      woken     missed    rt_susp     aborts nonrt_susp
switches
handled late    dropped invalid wdog    baddog1 baddog2 baddog3
596     225     0       0       0       0       0       0
rt_id   pid      woken     missed    rt_susp     aborts nonrt_susp
switches
  255 10882         23          0         24          0          0
0
handled late    dropped invalid wdog    baddog1 baddog2 baddog3
770     252     0       0       0       0       0       0
rt_id   pid      woken     missed    rt_susp     aborts nonrt_susp
switches
  254 10883          2          0          3          1          0
0
  255 10882        222          0        223          0          0
0
handled late    dropped invalid wdog    baddog1 baddog2 baddog3
875     346     0       0       0       0       0       0
rt_id   pid      woken     missed    rt_susp     aborts nonrt_susp
switches
  253 10884         38          0         39          0          0
0
  254 10883         83          0         84          1          0
0
  255 10882        302          0        303          0          0
0
handled late    dropped invalid wdog    baddog1 baddog2 baddog3
1022    398     0       0       0       0       0       0
rt_id   pid      woken     missed    rt_susp     aborts nonrt_susp
switches
  251 10886          3          0          4          0          0
0
  252 10885         38          0         39          0          0
0
  253 10884         91          0         92          0          0
0
  254 10883        135          0        136          1          0
0
  255 10882        355          0        356          0          0
0
handled late    dropped invalid wdog    baddog1 baddog2 baddog3
1131    489     0       0       0       0       0       0
```

```
rt_id   pid      woken     missed    rt_susp     aborts nonrt_susp
switches
  249 10888          6          0          7          0          0
0
  250 10887         22          0         23          0          0
0
  251 10886         37          0         38          0          0
0
  252 10885         72          0         73          0          0
0
  253 10884        125          0        126          0          0
0
  254 10883        169          0        170          1          0
0
  255 10882        391          0        392          0          0
0
handled late    dropped invalid wdog    baddog1 baddog2 baddog3
1316    504     0       0       0       0       0       0
rt_id   pid      woken     missed    rt_susp     aborts nonrt_susp
switches
  247 10890          3          0          4          0          0
0
  248 10889         19          0         20          0          0
0
  249 10888         30          0         31          0          0
0
  250 10887         47          0         48          0          0
0
  251 10886         63          0         64          0          0
0
  252 10885         98          0         99          0          0
0
  253 10884        151          0        152          0          0
0
  254 10883        195          0        196          1          0
0
  255 10882        416          0        417          0          0
0
```