Performance Evaluation of Java Architectures in Embedded Real-Time Systems

Carlos Eduardo Pereira, Fernando Henrique Ataide, Guilherme Oliveira Kunz
Federal University of Rio Grande do Sul
Electrical Engineering Department
90035-190 Porto Alegre, RS, Brazil
{cpeireira, fhtaide, guilhermekunz}@eletro.ufrgs.br

Edison Pignaton Freitas, Elias Teodoro Silva Jr,
Fabiano Costa Carvalho
Federal University of Rio Grande do Sul
Computer Science Institute
90501-970 Porto Alegre, RS, Brazil
{epfreitas, etsilvajr, fccarvalho}@inf.ufrgs.br

Abstract

The embedded real-time development community is investigating different approaches in order to provide modularity and reuse features for system design in this area, as well as a more appropriate mapping technique between requirements and implementation. The Java technology is very promising for this community, mainly after the research efforts on its real-time extension RTSJ - Real-time Specification for Java. However, there are still some critic factors related to the adoption of Java for real-time applications, and some of them deserve special attention. This paper reports a study of some of these decisive factors, such as the choice of the underlying Operating System, the use of a middleware (virtual machine) or a native code, and the use of the Java real-time API (RTSJ).

1. Introduction

Embedded real-time systems are of utmost importance nowadays due to its applicability in a great variety of safety-critical applications. This family of systems may deliver reliable services with predictable time behavior even in the presence of faults. Among many design facilities the use of Java technology has been emphasized due its desirable characteristics such as reusability, modularity, easy maintenance, and so on. However, the use of Java for real-time system design is not straightforward yet. The resulting time behavior of the application depends not only on the code but also on many other aspects like the underlying OS platform, the use of language extensions and the adoption of a virtual machine. It is important to perform a careful analysis of different design alternatives in order to take the best implementation decision.

The main components related to design of Java applications could be assembled in a generic architecture consisting of three layers that represents the application programming interface (API), the virtual machine and the operating system.

The complete knowledge of the entire architecture presented in Figure 1 is desirable, since it can easily orientate developers to take more cost-effective design decisions. It is of fundamental importance to know which layer has more positive or negative impact in the final performance.

This paper describes some experiment realizations on a set of different architecture configurations. The main goal is to show with practical results how distinct design decisions can influence the resulting temporal behavior of the system.

Figure 1. Generic Architecture

The remaining of this paper is organized as follows: section 2 presents some related works. In section 3, a brief description of the runtime architecture is presented; section 4 provides the test configurations used; section 5 describes the periodic thread test bench while section 6 describes the asynchronous event handling one. Section 7 presents the results of the periodic thread test, and section 8 the results with the asynchronous RTSJ. Section 9 provides some comments about RTSJ, and finally, section 10 offers some final conclusions.

2. Related Works

Some previous works have analyzed performance of different RTSJ implementations.

Corsaro [1] presented results of bench tests with jRate in relation to the TimeSys reference model. In his work an implementation of RTSJ without JVM was explored, which are going to be explained in the next sections.
In the other hand, Fridtjof [2], working with Jamaica set tools, presented a performance study of real-time application over an architecture using JVM. In his work no comparison was made with other implementation approach.

This work establishes a link between the two studies related above, extending the comparison to different implementations in different system architecture configurations.

3. Architecture Components Description

Several options were analyzed in order to compose the set of API, Middleware and Operating System to the proposed architecture. The following sub-sections present those configurations that were evaluated.

3.1. API and Middleware

3.1.1. JDK

The Java Development Kit (JDK) is a package that contains the Java Runtime Environment (JRE) [3]. It is composed of a small set of executables files that provides the standard Java platform.

The Java Runtime Environment is composed of libraries, the Java Virtual Machine (JVM) and other components that are necessary for the execution of Java applications. The JRE does not have development utilities such as compilers and debuggers. These utilities are delivered together with the JDK.

The Java Virtual Machine (JVM) is the middleware responsible for the interpretation of the bytecodes. The concept of virtual machine allows Java applications to be multi-platform. One of the main features delivered by the JVM is thread synchronization that provides concurrent programming mechanisms, automatic memory allocation for new objects and garbage collector.

3.1.2. GCJ

The GCJ (The GNU Compiler for the JavaTM) [4] is a front end for the GCC (GNU Compiler Collection) that can translate a Java code into native system code. Both GCJ and GCC are utilities distributed under GPL (GNU General Public License) and can be ported to many architectures and operating systems.

Applications generated with the GCJ are linked to the libgcj library, which provides all Java classes, garbage collector and bytecode interpreter. As a result, the final solution does not need the virtual machine, there is no need of middleware and it does not offer support for language extensions.

Implementations without middleware (JVM) has the advantage of increasing performance but lacking portability, that is, it is not possible to run the application over different operating systems without recompiling.

3.1.3. jRate

The jRate (Java Real-Time Extension) [5] is an extended utility that provides support for the Real-Time Specification for Java (RTSJ) [6].

jRate was included in this evaluation because it does not use Virtual Machine. Like GCJ, Java applications compiled with jRate are native executable code and platform specific.

The RTSJ API, the garbage collector and the real-time threads are accessed through the GCJ and jRate libraries. Unlike the GCJ, the jRate has only support for Linux OS and some of its real-time variants.

3.1.4. Jamaica

The Jamaica package [7] consists of an ensemble of utilities to develop Java applications for real-time embedded systems, delivered by Aicas [8]. The JamaicaVM is the core of the package, which is a virtual machine implementation compatible with the RTSJ extension and with the Java standard. Jamaica was first designed for real-time embedded applications.

This package has also other development utilities. One of them is the Builder Tool, which is responsible for building Java applications running directly on the native operating system not needing any kind of middleware. Actually, the application layer is tightly coupled with the OS.

One of the most undesirable shortcomings of Java application with time constraints is the lack of determinism caused by the garbage collector control, which is a primary component of any virtual machine. The JamaicaVM implements the garbage collector in a different way, where it is executed in a predictable manner like any other system thread. This design choice was taken as an alternative of a final solution that runs on a virtual machine and explores the RTSJ extension programming facilities.

Jamaica was chosen because it is a Java implementation that uses a virtual machine and supports the RTSJ specification.

3.2. Operating System

3.2.1. Linux

Linux is an operational system based on Unix, which supports multiple tasks, virtual memory, shared libraries, memory management and IPv4 and IPv6 support [9].

This OS was chosen because it is compatible with all Java implementations that will be analyzed in this paper.

3.2.2. RTAI

RTAI (Real-Time Application Interface) [10][11] is a patch applied to Linux, which gives the determinism required by real-time applications. With this patch, the conventional Linux tasks can only be executed when
there is no real-time task waiting for the CPU. All Linux tasks can be pre-empted by a real-time task any time.

With RTAI patch, Linux has an extension of its standard environment defined by the RTHAL (Real-Time Hardware Abstraction Layer), and thus, the RTOS controls the interruptions, treating and dispatching them.

RTAI supports everything necessary to the real-time processing related to task management and scheduling, inter process communication, periodic and oneshot timers, and interruptions.

RTAI was included in the experiment because it provides real-time support on the Operating System level to compare GCJ, jRate and JDK evaluations.

3.2.3. QNX

The QNX Neutrino RTOS [12] is based on POSIX specification with micro-kernel architecture. The applications are treaded as group of process (with several threads) that communicate by messages. This method of changing messages is the primary manner of inter-process communication (IPC) within this RTOS.

All services, except those provided by the micro-kernel or by the task manager, are manipulated by standard process. Alike the method used by execution of threads, the micro-kernel is never scheduled. The operation system is completely pre-emptive, even when process are sending messages, it will return in the message that was stopped in the moment of pre-emption.

The Jamaica support for QNX RTOS was the main reason to use it for the tests in this paper.

4. Environment Settings

Four different base arrangements (called architectures) were tested, according to figure 2. Each one of the architectures combines different implementations of the layers shown in figure 1.

![Figure 2: Tests Configuration](image)

When the analysis of a layer is made, it is necessary to keep the other two layers unchanged, and vary the implementation used by the layer under analysis.

The table 1 presents the comparative tests needed to analyze each layer. The first line indicates tests to evaluate the effects of the use of RTSJ, or not. The second line considers the presence of a Virtual machine. Finally, the third line evaluates the support provided by RTOS.

The Java implementations used in the tests are presented in the table 2 grouped by the operating system used.

Test benches were executed on a Pentium III 933MHz with 256MB of RAM. No extra loading was included during the tests.

![Comparative Layer Table (configurations)](image)

<table>
<thead>
<tr>
<th>Virtual Machine</th>
<th>RTSJ</th>
<th>jRate</th>
<th>JRE</th>
<th>Jamaica</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCJ X JRE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JRE X Jamaica</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>jRate (Linux) X</td>
<td>jRate (RTAI)</td>
<td>Jamaica (Linux) X</td>
<td>Jamaica (QNX)</td>
</tr>
</tbody>
</table>

Table 1: Test Configurations

<table>
<thead>
<tr>
<th>Operating Systems</th>
<th>RTSJ</th>
<th>jRate</th>
<th>JRE</th>
<th>Jamaica</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU Linux 2.6.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTAI 3.1 (GNU Linux 2.6.8.1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNX 6.2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a) JRE</td>
<td>Sun JDK 1.4.2</td>
<td>Sun JDK 1.4.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b) GCJ</td>
<td>GCJ 3.3.5</td>
<td>GCJ 3.3.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c) jRate</td>
<td>JRate 0.3.6-3.3.3</td>
<td>JRate 0.3.6-3.3.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d) Jamaica</td>
<td>Jamaica 2.4.10</td>
<td>-</td>
<td>Jamaica 2.4.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Evaluated Versions

5. Periodic Threads Test Bench

Considering timing as a critical feature in real-time systems, the behavior of periodic thread was traced to analyze the differences between architectures and configurations. All tests performed 500 samples.

public class ThreadSUN {
   // class initialization
   static Timer timerTask01 = new Timer();
   public static void main(String[] args) {
      TimerTask task01 = new TimerTask() {
         public void run() {
            actTime[activation++] =
            java.lang.Math.abs(timers.microTimer.getmicroseconds());
         }
      }
      timerTask01.scheduleAtFixedRate(task01, 0, 100);
   }
}

Figure 3. Source Code: Standard Java Thread Implementation

In the standard Java tests the periodic thread was an instance of the class TimerTask (Figure 3), as defined in standard Java API. The Thread is activated by the method scheduledAtFixedRate with a 100 milliseconds
period and release time zero. To measure the time with the necessary precision it was made a native method, using JNI, that captures time in microseconds. It appears in the `timers.microTimer.getmicroseconds()` call.

In the RTSJ implementation it a `RealtimeThread` object was used. All timing parameters were defined as a `RelativeTime` instance. The period was established 100 milliseconds like in the standard java test. `Costtime` and `Deadtime` parameters were equal to period because they were not under evaluation, neither missing deadline nor thread cost feature. All timing parameters are grouped in the `releaseParams` object, which is an instance of the `PeriodicParameters` class. They are passed to the `RealtimeThread` object, `rtThread`, with the `Runnable` object `testapp`, which consists of the logical behavior of the thread. This logic is only a time capture and a call for the `waitForNextPeriod` method. The `rtThread` is initialized by calling the `start` method, as can be seeing in figure number 4.

```java
public class ThreadRT {
    // class initialization
    public static void main(String[] args) {
        final Clock clock = Clock.getRealtimeClock();
        RelativeTime starttime = new RelativeTime(0, 0);
        RelativeTime periodtime = new RelativeTime(100, 0);
        RelativeTime costtime = new RelativeTime(100, 0);
        RelativeTime deadtime = new RelativeTime(100, 0);
        PeriodicParameters releaseParams =
                new PeriodicParameters(starttime, periodtime, costtime, deadtime, null, null);
        Runnable testapp = new Runnable() {
            public void run() {
                RealtimeThread rtThread =
                        RealtimeThread.currentRealtimeThread();
                for (int i=0; i < count; i++) {
                    rtThread.waitForNextPeriod();
                    time[i] = clock.getTime();
                }
            }
        };
        RealtimeThread rtThread =
                new RealtimeThread(null,
                        releaseParams,
                        null, null, null,
                        testapp);
        rtThread.start();
        //...
    }
}
```

Figure 4. Source Code: RTSJ Thread Implementation

### 6. Asynchronous Event Test Bench

Another critical feature in real-time systems is the asynchronous event handling. Once an event occurs, a system with real-time requirements has to be able to handle this kind of event as soon as possible.

In standard Java there is no class to handle explicitly asynchronous events, so it was used the AWT `InvocationEvent` class with the `dispatch` method call, as shown in figure 5. The logic executed when an event occurs is contained in the `Runnable` object called `handlerLogic`. At the end of execution, the `Start` array contains the instants of the events occurrences and the `Stop` array, the instants of the handler activation. The measured time shown in table 4 is the difference between the `Stop` and `Start` time.

```java
public class AsyncJava{
    // class initialization
    public static void main(String[] args){
        Runnable handlerLogic = new Runnable() {
            public void run() {
                Stop[counterStop++] =
                        java.lang.Math.abs(timers.microTimer.getmicroseconds());
            }
        };
        Object obj = new Object();
        InvocationEvent event =
                new InvocationEvent(obj,handlerLogic);
        for (int i=0; i<500; i++) {
            Start[counterStart++] =
                    java.lang.Math.abs(timers.microTimer.getmicroseconds());
        }
    }
}
```

Figure 5. Source Code: Standard Java Event Handling

```java
public class AsyncRTSJ{
    public static void main(String[] args){
        final Clock clock = Clock.getRealtimeClock();
        Runnable handlerLogic = new Runnable() {
            public synchronized void run() {
                timerStop[counterStop++] = clock.getTime();
            }
        };
        AsyncEventHandler handler = null;
        handler = new BoundAsyncEventHandler(null, null, null, null, null, false,
                handlerLogic);
        AsyncEvent event = new AsyncEvent();
        event.addHandler(handler);
        for (int i=0; i<count; i++) {
            timerStart[counterStart++] = clock.getTime();
            event.fire();
        }
    }
}
```

Figure 6. Source Code: RTSJ Asynchronous Event Handling
In the real-time implementation it was used the AsyncEvent class. The RTSJ has the notion of an event handler, that is materialized in the AsyncEventHandler class, which has some specializations like BoundAsyncEventHandler. In code shown in figure 6, there is an AsyncEvent object, event, that add a class instance of BoundAsyncEventHandler as its handler. The execution logic of this handler is within the run method of the handlerLogic Runnable object. To invoke the event, the method fire is called as shown in figure 6. The time measurement was just explained for standard Java.

7. Periodic Thread Test Bench Results

7.1. Result Presentation

The results extracted from the execution of the test bench can be resumed and divided in two sets, one showing what happened over the GNU Linux operation system, and other over an RTOS. The table 3 shows numerical results (in milliseconds), divided according to the configurations and architectures described in Section 3.

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Standard Deviation</th>
<th>MTBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCJ</td>
<td>Linux</td>
<td>100.000</td>
<td>3.009</td>
</tr>
<tr>
<td></td>
<td>RTAI</td>
<td>100.001</td>
<td>1.742</td>
</tr>
<tr>
<td>Java SUN</td>
<td>Linux</td>
<td>99.985</td>
<td>3.008</td>
</tr>
<tr>
<td></td>
<td>RTAI</td>
<td>99.984</td>
<td>2.073</td>
</tr>
<tr>
<td>Jamaica</td>
<td>Linux</td>
<td>99.999</td>
<td>3.314</td>
</tr>
<tr>
<td></td>
<td>QNX</td>
<td>99.999</td>
<td>0.121</td>
</tr>
<tr>
<td>jRate</td>
<td>Linux</td>
<td>100.984</td>
<td>0.056</td>
</tr>
<tr>
<td></td>
<td>RTAI</td>
<td>100.984</td>
<td>0.057</td>
</tr>
</tbody>
</table>

Table 3: Periodic Thread Results

The data presented in table 3, Mean, Standard deviation and Maximum Time Between Activation (MTBA), provide the following information:
- How predictable and stable is the behavior of each architecture;
- How much variation can occur in a period of a thread, and
- How the maximum activation time is far from the target value (100 milliseconds).

7.2. Result Analysis

Figures 7 and 8 show that the GCJ solution, operating without virtual machine nor RTSJ support, has not too much satisfactory results, since its behavior is not stable. But it is important to mention that over an RTOS the results were better than over Linux.

The solution using the Sun JVM and standard Java API and virtual machine, without RTSJ support, has also results that are not satisfactory for real-time applications. It has an unstable behavior and hence unpredictable, what can be observed in figures 9 an 10.

Comparing these results with those obtained with GCJ, it is possible to conclude that the GCJ solution has a slight advantage when running over an RTOS, although when both are running over Linux, no significant difference is noticed, as one could suppose.
The evaluation of jRate, which does not use virtual machine and supports RTSJ, presented good stability over both platforms (Linux and RTOS). It is noteworthy the fact that the mean was about one millisecond over the target value. This strange, yet consistent behavior, is reported by Corsaro [1] in his experiments, but with a 9 times greater difference. Figure 13 and 14 give a better comprehension of the analysis made. There is almost no difference between the execution over RTOS and Linux.

The results show inter-dependency between the virtual machine and the Operating system used. This is clear when comparing results obtained by Jamaica running over Linux and QNX. They have respectively the worst and the best results of all set. It can sound strange at a first moment, but it reinforces the idea of this paper, the importance of the analysis of the three-layer architecture (API, Virtual Machine and Operating system) and their components.

8. Asynchronous Event Test Bench Results

8.1. Result Presentation

The results of this test bench follow the same layout used in the section 6. They were divided in two sets, one showing what happened over the GNU Linux operating system, and other over an RTOS, according to the configurations and architectures described in Section 3. The table 4 shows numerical results (in milliseconds).

The data presented in table 4 show the delay between the instant of an event occurrence and the instant in which its associated run method really starts. In the first column is the Mean time while in the second the Standard Deviation. In the last column is the Maximum Delay Time (MDT).
<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Standard Deviation</th>
<th>MDT</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCJ</td>
<td>0.001</td>
<td>0.001</td>
<td>0.009</td>
</tr>
<tr>
<td></td>
<td>0.001</td>
<td>0.001</td>
<td>0.012</td>
</tr>
<tr>
<td>Java SUN</td>
<td>0.003</td>
<td>0.044</td>
<td>0.995</td>
</tr>
<tr>
<td></td>
<td>0.004</td>
<td>0.047</td>
<td>1.061</td>
</tr>
<tr>
<td>Jamaica</td>
<td>0.407</td>
<td>0.072</td>
<td>1.500</td>
</tr>
<tr>
<td></td>
<td>0.194</td>
<td>0.395</td>
<td>1.000</td>
</tr>
<tr>
<td>jRate</td>
<td>0.029</td>
<td>0.055</td>
<td>1.250</td>
</tr>
<tr>
<td></td>
<td>0.042</td>
<td>0.069</td>
<td>1.572</td>
</tr>
</tbody>
</table>

Table 4: Asynchronous Event Tests Results

8.2. Result Analysis

Analyzing the results of the Asynchronous Event test bench presented in table 4, one can have a bed surprise. Every execution over a real-time operating system was worst than those over a non real-time operating system. Something that can explain such bad results is the lack of a direct interface between Sun JVM and GCJ with RTAI OS, besides the fact that the asynchronous event handling in real-time Java solutions (jRate and Jamaica) requires from CPU a lot of processing effort, so the system scheduler could have bad influence in the general result for a process that occupies the CPU for a long time. This analysis is directly influenced by the schedule policy that is being used, i.e. FIFO, Round Robin, between others.

As one can observe in figure 16, the real-time implementation for asynchronous event handling has a great overhead. Even thought it does not justify the bad results obtained, it is something to be remarked.

Another very important fact to notice is that the Java Sun does not have a rigorous asynchronous event handling such as the one found in the RTSJ API. The test bench implemented to evaluate standard Java, shown in figure 5, was developed intending to be as closer as possible to the one deployed on the RTSJ API. At this point, the overhead question is also important to be considered, because the standard Java does not have the same features provided by RTSJ.

Aicas was contacted in order to look for an explanation for the strange results extracted from the Jamaica evaluation, shown in figures 15 and 16. Aicas related that better results could be observed if the JamaicaVM Builder (that builds a native code for the host target) were used, instead of the JamaicaVM. However, it was not the goal of this article to analyze this feature, because it would eliminate the Virtual Machine layer. Another point related by Aicas was the fact that the QNX version of Jamaica is under continuous development, so they are working to improve some features in this version.

Analyzing figures 15 and 16, it is possible to understand better the values presented in table 4 for Jamaica tests. Even though the Mean and the MDT are higher in the Linux execution, it is noticeable the instability when running over QNX in comparison to the test over Linux. The Standard Deviation tries to represent what is easy to see observing figures 15 and 16.

![Figure 16. Asynchronous Event Handling Behavior: Jamaica over QNX](image1)

![Figure 17. Asynchronous Event Handling Behavior: jRate over RTAI](image2)

The jRate solution did not present a very good performance (it has the higher MDT), maybe due the same reasons presented above to explain the performance obtained in the other tests. Something that
is important to notice is the fact that the tests with jRate for asynchronous events over RTOS and Linux were very similar. For this reason it is only presented the graph of one of the tests in figure 17.

Graphs for GCJ and Sun JVM tests do not have to much information, all relevant information can be find in table 4. The graphs are quite similar and have only a horizontal line close to zero (except for the respective MDT point of each test), so they are not presented in the paper.

9. Comments on RTSJ Powerful

There are two other implicit factors not shown in the results above. One is the expressiveness of the source code when RTSJ is used. It provides a gain in the programmer’s productivity once he/she does not need to have specific knowledge about the underling RTOS. The other is the code legibility provided by RTSJ, once the use of this API provides a clear mapping of the real-time requirements over the application source code. Figure 4 shows the source code used in the test with RTSJ and, analyzing this code it is possible to understand the relation of each parameter with a precise timing requirement. Using the standard Java API it is not possible to express so precisely the same set of real-time constraints as it is possible while RTSJ is being used [13].

It is clear that as far as there are more symbols in a language, more difficult it is to understand it, but in the other hand, you have a richest set to express what you mean. In fact there is an optimal point in which it is possible to have a rich language (and so very expressive), and at the same time clear to be understood, it means legible. Using the RTSJ API the programmer can find something very close to the optimal point that meet expressiveness and legibility.

10. Concluding Remarks And Future Work

This paper presented different alternatives to the use of Java Technology for real-time system implementation, with a spot over the performance obtained by each layer of a proposed architecture under analysis. It was possible to see that during the project specification phase, the three-layer analysis (API, Virtual Machine and Operating system) is very important to achieve the desired performance. The results show that when a three-layer architecture is adopted, if two of them have real-time support it does not assure real-time requirements. In other words, during a project, if it is not possible to provide real-time feature in all three layers, it is safer to adopt a two-layer architecture (without VM), as shown in the tests that used jRate and GCJ.

It can be observed in the result analysis that the RTSJ API is a powerful tool for the development of real-time systems with Java Technology. Besides the clearly mapping between real-time requirements and the application source code, the offered timing conformity shows that the RTSJ is very efficient. There are much to be done to make the Java Technology completely able to be used in real-time system, but the present study and others, like those described in [14] and [2], indicate that Java will be a technology completely adequate to implement real-time requirements in the near future.

The authors intend to perform some other complementary tests in order to evaluate other real-time features, like the memory allocation time, timeliness, and event handling, as well as making a profound study - together with Aicas - of the reasons for those results presented for asynchronous event tests.

References