Early Experience with an Implementation of Java on RTLinux

Miguel Masmano, Ismael Ripoll, Jorge Real, and Alfons Crespo
Department of Computer Engineering, Polytechnic University of Valencia
Camino de Vera 1, Valencia, Spain
{mmasmano, iripoll, jorge, alfons}@disca.upv.es

Abstract

A deterministic response time is the most important and necessary feature in Hard Real-Time Systems. This is the main reason why some years ago it was unthinkable to implement Hard Real-Time Systems with the Java language, an object-oriented, interpreted language, mainly because Java uses garbage collection to implicitly deallocate dynamic memory; This is widely known as"the garbage collection problem" in the Real-Time Systems community, due to its temporal unpredictability. Nevertheless, few years ago a new standard, known as RTJava, has been specified to transform the Java language into a Real-Time compliant one, making it possible to implement Real-Time applications in Java. This new Real-Time standard tries to benefit from the experience gained in the design of other Real-Time languages, taking advantage of their real-time properties like scheduling control. On the other hand, RTLinux is a Hard Real-Time Operating System which allows to implement Hard Real-Time applications while preserving all of the Linux functionalities available for the Soft Real-Time part of the application. This paper will describe how we have ported the JamVM, a free Java Virtual Machine implementation to run Java bytecode directly on RTLinux, which allows to benefit from both the capabilities of the Java language and the RTLinux OS. This work represents our first milestone in the more ambitious goal to provide an Real-Time RTJava platform based on RTLinux.

1 Introduction

The most important aim of a hard real-time operating system is to achieve a deterministic response time as well as a correct logical behaviour of the applications, being that the main reason because the use of the Java language has been avoided till now in this kind of systems.

Nevertheless, with the recent apparition of the new specifications of Java for Real-Time [3], which defines, among other things, new classes to work in a real-time environment as well as removes from the language all non necessary stuffs (avoiding the garbage collection problem), the hard real-time applications are able to be written completely in Java.

In the RTLinux case the existing approach to use Java, before the creation of these new standards, was to program the hard real-time part of the application in C/C++/Ada, which was executed by RTLinux with a guaranteed bounded response time, and the soft real-time part was written in Java, which was executed by Linux without real-time guarantees.

This paper presents all the work done to accomplish our first milestone: to port a Java Virtual Machine to the RTLinux environment, providing RTLinux the capability to execute application compiled as byte code. With this new approach we expect to gain Java properties in the Hard Real-Time applications.

2 RTLinux-GPL

Linux was initially designed as a general purpose OS, mainly for use as a desktop OS. Once Linux became more popular, it started to be used in server machines. Kernel developers added multiprocessor support, management of large amounts of memory, efficient handling of large numbers of running processes, etc. Nowadays, Linux is a full-featured, efficient (good throughput) OS, but with little or no real-time performance.

There are two different approaches to provide real-time performance in a Linux system:

1. To modify the Linux kernel to make it a kernel with real-time characteristics. For this purpose, the kernel needs to be:

Predictable: This requires to improve the internal design to use efficient and time-bounded algorithms. For example, a fast scheduler.

Responsive: The kernel needs to be preemptable in order to interrupt (or postpone) the servicing of a low priority process when a high priority process becomes active and requires to use the processor.

Real-time API: Real-time applications have special requirements not provided by the classic UNIX API. For example, POSIX real-time timers, message queues, special scheduling algorithms, etc.

2. To add a new operating system (or executive) that takes the control of some hardware subsystems (those that can compromise the real-time performance) and that gives Linux a virtualised view of the controlled devices.

RTLinux was the first project to provide real-time in Linux using two separate OSs: Linux as the general purpose OS and a newly developed executive running beneath Linux. The RTLinux internal structure can be divided into two separate parts: 1) a hardware abstraction layer, and 2) the executive operating system.

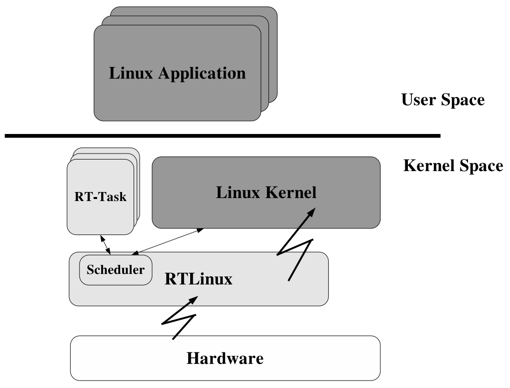

FIGURE 1: *The RTLinux-GPL architecture*

The RTLinux HAL [1] is in charge of the hardware devices that conflict with Linux (interrupt controller and clock hardware). This layer provides access to these hardware devices to both the RTLinux executive and Linux in a prioritised way. Hardware virtualisation is achieved by directly modifying the Linux kernel sources applying a patch file. These changes do not replace the hardware drivers by virtual drivers but prepare the kernel, adding hooks, to dynamically replace the drivers. The code that actually virtualises the interrupt and clock devices are two separate kernel modules [2]: `rtl.o` and `rtl_timer.o`.

Currently, there are several Linux projects that use this (or a similar) approach: Real-Time Application Interface (RTAI), Adaptive Domain Environment for Operating Systems (ADEOS) and RTLinux-GPL. The work presented in this paper has been based on the code of the RTLinux-GPL project. In what follows, RTLinux-GPL will be referred to as RTLinux for short.

The RTLinux executive implements a partial Minimal Real-Time POSIX.13 interface. The executive is highly customisable. It has been implemented in a modular way that permits to dynamically load into the system the parts of the API that are used by the application. In addition, several parts of the code are surrounded by conditional preprocessor directives that allow to tailor the final RTLinux executive even more, by using a graphical interface.

3 Executing Java in RTLinux

There exist two possible ways to run a Java application in RTLinux:

1. Compiling the application as a binary object code file (an RTLinux module) and executing it directly, with no differences with any other RTLinux application. The application is executed as a native application with all the benefits that it provides. Loosing same Java good properties as the mobility of code, however.

2. Compiling the application as a byte code file and inserting an interpret inside of RTLinux to execute it. The main drawback of this approach is the lost of throughput of the application, since it is executed with less speed than binary code.

We have decided to use the second method, that is, to insert a interpreter inside RTLinux as defined by Real Time Standard for Java.

3.1 Porting a JVM

The first step before porting a JVM to the RTLinux environment is to choose the more suitable one. Our choice has been the JamVM virtual machine.

[1]The mechanism to intercept hardware access off a general purpose OS to implement a real-time layer is covered by U.S. Patent No. 5,995,745, titled: "ADDING REAL-TIME SUPPORT TO GENERAL PURPOSE OPERATING SYSTEMS". The owner of the patent permits the use of the patented method if the software is released under the General Public License (GPL). For more details read "The RTLinux Open Patent License, version 2.0" [6]

[2]A Linux kernel module is a piece of code that can be loaded and unloaded into the kernel upon demand.

3.1.1 JamVM: a small but complete Java Virtual Machine

Currently there exist a great quantity of implementations of the Java Virtual Machine (JVM for short), examples of available JVM are the Sun Java Development Kit (JDK), Kaffe, JamVM, IBM Jikes, and a large etc.

Nevertheless, choosing a suitable one to be ported to RTLinux is not an easy task, mainly because among other reasons, not all existing JVM's source code is freely available, or are written in C (there is a great quantity of JVMs written in Java itself or C++).

Our preferences to choose a JVM between all existing ones have been:

1. The foot-print: the selected JVM must be inserted into the kernel space as a rtlinux module, hence the smaller, the better.
2. The compatibility of implemented features: It is extremely important that the selected JVM was fully compatible with the Java specification.
3. The language used to implement the JVM [C, assembly]: The use of a different language in RTLinux provoke the necessity of the use of extra support, that is the case of C++ or Ada. Therefore it will be convenient to easy the port and the maintenance that the JVM is written in C.
4. The Java classes must be fully compatible with the Java standard: The Java classes are the heart of all JVMs, since any JVM implementation itself uses them as the Runtime System. Therefore the classes used by the selected JVM must be fully compatible with the Java standard and free, nonetheless it is not necessary that they are fully implemented since not all Java functionalities are useful inside the RTLinux environment.

At the end we have opted to use JamVM because it accomplishes all previously enumerated requirements.

JamVM [1] is a Java Virtual Machine which conforms to the JVM specification version 2 (blue book). This virtual machine in comparison to most other VM's (free and commercial) it is extremely small. However, it has been designed to support the full specification, and includes support for object finalisation, the Java Native Interface (JNI) and the Reflection API.

JamVM includes a number of optimisations to improve speed and reduce foot-print. A list of JamVM's features is:

- The utilisation of the native threading (Posix threads, despite Java threads), as well as a full thread implementation, including the Thread.interrupt() method.
- Object references are direct implemented as pointers (i.e. not handlers).
- It supports class loaders.
- Efficient thin locks for fast locking in uncontended cases (the majority of locking) without using spin-locking.
- Two words object header to minimise heap overhead (lock word and class pointer).
- Execution engine supports basic switched interpreter and threaded interpreter, to minimise dispatch overhead (requires gcc value labels).
- Stop-the-world mark and sweep garbage collector.
- Thread suspension uses signals to reduce suspend latency and improve performance (no suspension checks during normal execution).
- Full object finalisation support within the garbage collector (with finaliser thread).
- Garbage collector can run synchronously or asynchronously within its own thread.
- String constants within class files are stored in hash table to minimise class data overhead (string constants shared between all classes).
- Supports JNI and dynamic loading for use with standard libraries.
- Uses its own lightweight native interface for internal native methods without overhead of JNI.
- JamVM is written in C, with a small amount of platform dependent assembler, and is easily portable to other architectures.

Modifications suffer by the JamVM In order to accomplish our first milestone, that is, executing java byte-code directly in RTLinux, the modifications performed in the JamVM have been minimum, basically they have consisted in adding a small library (the RTJL, explained in the section 3.3) which among other things implements some library functions, another performed modification has been the division of the JamVM into two different parts:

1. The Java interpreter, which is compiled as an RTLinux module and executed inside this environment. In order to accomplish our first milestone we have been focused only in executint the JamVM in RTLinux without taken into account its real-time behaviour. Therefore the modifications carried out in this part have been focused to remove GLIBC function replacing them with our own RTLinux Java Layer, which is described in the 3.3 section, as explained above.

2. The user command interpreter, which is executed as a Linux user program and which loads/unloads into memory all the classes which JamVM requires. This second part is better described in the section 3.2.

Regarding the garbage collection problem for this first release we have decided to disable the JamVM garbage collector mechanism, therefore all allocated memory is never released.

3.1.2 GNU Classpath: A free implementation of the essential libraries for Java

To interpret a java application, two different parts are required, a Java Virtual Machine which executes the byte code and the Java Runtime, which implements necessary support to manage the application. In the JamVM, the Java runtime is implemented through the GNU Classpath [2] libraries.

The GNU Classpath is a project to provide a free implementation of essential Java Classes, released under the terms of the GPL License version 2. It is still under development but it implements all necessary classes to run a basic or not so basic Java Virtual Machine.

These must be the reasons because the developer of JamVM chose GNU Classpath implementation.

Modifications suffer by the GNU Classpath At the current moment of the porting has not still been necessary to modify the GNU Classpath at all.

For the second milestone is planned to remove all unnecessary stuffs from the GNU Classpath like all graphic classes and to add the most important classes defined in the RTSJ.

3.2 The Linux class loader

Once compiled, any java program is compiled as a byte-code file, this file is open and executed by the interpreted. However, at least currently, RTLinux do not implement any way to access directly to the Linux filesystem, therefore it is necessary to implement some methods to load/unload classes to/from memory. The Linux class loader is a Linux user application which loads/unloads classes under demand. This application implements all necessary operations to manage Linux files and is communicated via an RTL-Fifo and shared memory with the RTL-JVM which has been porting by us.

The working of this application is as follow, when the RTL-JVM needs some classes, it sends a message via the opened RTL-Fifo requesting the opening and the reading of the requested class, once the class is opened and read, it is sent through a shared memory buffer to the RTLinux context.

Besides the Linux class loader also interacts with the user to manage the working of the RTL-JVM.

Another functionality of this part is to insert the JVM in RTLinux as well as to send it the user commands, that is, which class has to be executed, etc.

3.3 RTLinux Java Layer (RTJL)

In order to substitute the GLIBC functionality, we have implemented the RTLinux Java Layer (RTJL for short) which implements a minimum C library. It is important to recall that all file management is carried out by the linux class loader, hence they have not been included in the RTJL, RTJL just implements math functions, printf, and string functions.

RTJL also includes a dynamic memory allocator: TLSF dynamic memory allocator [4, 5], which is designed to achieve Real-Time capabilities.

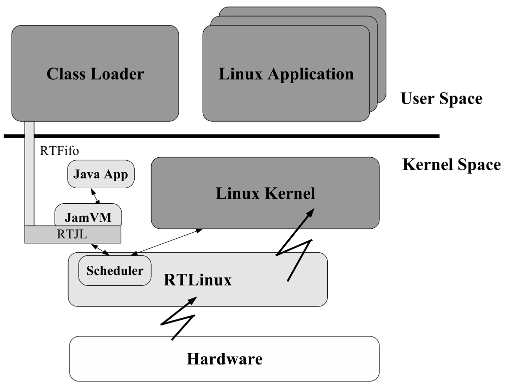

FIGURE 2: *RTLinux-GPL running Java*

Figure 2 shows the resulting architecture, the java virtual machine is communicated via an RTL-Fifo and shared memory with the class loader which loads/unloads classes to/from memory. All threads, which are created by the java virtual machine, are directly mapped as RTLinux Posix threads.

4 Conclusions and Future Work

This paper presents the work done to accomplish our first milestone, the port of a JVM to be executed as RTLinux module. Allowing us to execute directly Java byte code as any other RTLinux application.

However there are still much work to be done. Our second milestone contains between other things:

1. To study the behaviour of real java applications in the RTLinux environment.
2. To study current existing solutions for the garbage collector problem and implement some of them in the JamVM, removing the current JamVM mechanism. In the case that this study would be unsuccessful, the adopted solution will be the same that now, to avoid the release of dynamic memory, using only the allocation operation.
3. To strip GNU Classpath removing all non-RTJava compliance stuffs: GNU Classpath has been implemented keeping in mind the fully compatibility with Sun Java Classes. However, a lot of implemented stuffs are useless in the RTLinux environment (i.e. RMI features, file handling classes, etc).
4. To implement some RTJava new real-time classes.

References

[1] Robert Lougher, *JamVM*, available at http://jamvm.sourceforge.net

[2] *GNU Classpath*, available at http://www.gnu.org/software/classpath/-classpath.html

[3] *RTSJ: The Real-Time Specification for Java 1.0*, available at http://rtj.org

[4] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real, *TLSF: a New Dynamic Memory Allocator for Real-Time Systems*, Proc. of the 16th Euromicro Conference on Real-Time Systems.

[5] Miguel Masmano, Ismael Ripoll, and Alfons Crespo, *Dynamic storage allocation for real-time embedded systems*, Proc. of Real-Time System Simposium WIP.

[6] FSMLABS, *The RTLinux Open Patent License, version 2.0*, available at http://fsmlabs.com/-products/rtlinuxpro/rtlinux_patent.html