Hard real-time Java virtual machine for Space applications

Frédéric Deladerrière

Hard Real-Time Java virtual machine for Space applications

Frédéric Deladerrière, EADS Astrium – Toulouse, France
Email: ***frederic.deladerriere@astrium.eads.net***

ABSTRACT

The AERO (*Architecture for Enhanced Reprogrammability and Operability*) is an ESA project with the objectives to investigate on a real-time Java virtual machine for ERC32 processor. Special attention was put on the garbage collection mechanism and deterministic execution model. The project have first investigate existing virtual machine to choose a potential candidate that will be customized, are then investigates the definition of requirements concerning a real-time interpreter in on-board systems. The second phase of the project was dedicated to the definition of software functions of the real-time Java virtual machine and to their implementation and assessment through validation tests. The resulting application is the AERO-JVM.

1. INTRODUCTION

Space missions require an enhanced level of spacecraft operability and reprogrammability, i.e., an appropriate ability to operate without routine support from ground in order to safeguard the mission objectives. Such requirement can be tackled through the implementation of an embedded interpreter managing high-level operational procedures. Astrium *Avionics Products* Business Unit have propose to the European Space Agency, the AERO (*Architecture for Enhanced Reprogrammability and Operability)* project.

The goal of the AERO project is to provide a software architecture, which improves the reprogrammability and operability of the space systems. Then to propose a standard interface for operability based on state of the art technologies, which allows getting access to commercial low cost products with a good expected lifetime.

2. ON-BOARD INTERPRETER

2.1 Rationale for on-board interpreter

The interpreter provides important capabilities to the space systems. Thanks to the isolation provided by the Interpreter, the Application Programs can be developed and validated separately from the other software components (critical software layers or applications, services). This allows to tremendously reducing the cost of these Application Programs. The interpreter allows specifying, developing and validating Application Programs late in the life cycle and even during operational activity of the system in-orbit (including reprogramming). The execution context of the Application Programs is isolated from the rest of the system as well as the other Application Programs. The execution control of all the Application Programs is kept, so that application software failures, if any, do not disrupt software vital functions (AOCS, FDIR).

The interpreter offers the power of a high level language and supports the implementation of complex algorithms. It also gives the access to the system commands by the mean of a specific interface allowing the control of the entire spacecraft. This permits to implement autonomous operations enabling new complex missions when the ground cannot react in real-time due to the communication delays, including FDIR operations.

In addition, the interpreter technology allows testing on-board a new loaded control processing, running in parallel with the predefined one – a sort of « on-board simulation » which is often very difficult to simulate on-ground, because of real time representativity problems.

All these features show that the Interpreter approach brings benefit to the onboard systems, giving an important flexibility to the developments, operability and maintainability aspects.

2.2 Functional requirements

The interpreter shall offer control structure, data management, subprogram definition and numeric computations. It shall be activated at a cyclic rate or on asynchronous events, with the ability to execute with appropriate determinism, sequences of commands, in the right order and following specified delays, monitoring, and specific algorithms such as thermal controls. The interpreter shall be able to interface with the remaining software layers through different kind of services: mathematical computations, delays, HW commands and acquisitions, system data.

The interfaces to the environment are critical parts of interpreters. They are the interface with the operating system, with the TC, with the data and with the other services:

- *The Operating system interface*: interpreters are either designed to be activate cyclically or executed as a specific task of the system.
- *The TC interface*: interpreter TC interface is standardised and very simple. The interpreted program can be activated either by a TC, by the timeline function or by another interpreted program.
- *The Data interface*: an interpreter shall have access to the system data through an This also allows exchanging messages between the different SW components (TM, AP, AOCS, etc.).
- *The Service Interface*: the interpreter has the capability to use the whole set or only a subset of the general services for basic computation by the means of direct calls or by the means of the standard telecommand interface.
-

2.3 State of the art and limitations

Different application interpreter components have been developed in the scope of other project, (Eureca CLASP Interpreter, Eurostar 2000/2000+ AP Interpreter, E3000 AP Interpreter and Rosetta Interpreter, COF UCL Interpreter). These existing products correspond to four generation of interpreter.

Capabilities
The interpreters suffer from their advantages because they require more and more capabilities. Actually, the development of an interpreted procedure is quicker and cheaper than the development of the equivalent standard application. The current trend consists in the implementation of non-recurring functions under the form of interpreted procedures to minimise the modifications on the rest of the system. Furthermore, the specific functions of a system are defined very late in the life cycle of the software are there is no other possibility than implementing them with interpreted procedures.

Language & Tools
Current embedded interpreters work on non-standard front-end language, and non-standard byte code. This involves many problems during developments and validations. Setting up a development environment for a specific front-end language supposes to develop specifics tools: editor, debugger, analyser etc. Homemade tools have not the level of functionality of commercial tools. The cost associated to a modification of the language and tools is then extremely important. It appears necessary to define a standard for the language that can be reused from a project to another. This standard should support all the features of existing languages.

Performances
The performances of current interpreters are very low, limited by the performance of the CPU on which they are executed. The definition of a high performance interpreter running on a modern processor will permit to execute faster a greater number of procedures

Development strategy
Current interpreters were designed with technologies that could not provide subsequent performances and security management. In the case of the interpreter technology, it's more efficient to take an optimised standard interpreter and to adapt it to the specific space systems constraints. This approach is more robust and secure, than using homemade interpreter, due to the number of specialised developers who worked on it.

Non-standard byte-code and front-end language suppose to develop specific tools and process solutions. The global cost involved is more important than choosing a market-based standard solution. Furthermore, the equivalency of performance, robustness and quality is not guaranteed.

3. ANALYSIS OF POTENTIAL SOLUTION

3.1 Rationale for using the Java Bytecode and Language

They're many interests to have a standard interpreted bytecode. But the selected bytecode to use must have all capabilities to answer future requirements. On market, the standardized available bytecode are: Forth, Java bytecode (≠Java language), Lisp, Perl, SmallTalk,Tcl-Tk (interpreted language during execution).

The bytecode depends on functionalities provided by the front-end language, and must implement all its subtleness, with maximum performances and security. The performance of interpreters depends, in a great part, on the bytecode definition.

Some criteria was defined to compare bytecode: first level functionalities, instructions size, entropy and redundancy, availability/commercial standard etc. The result the trade-off is that the Java bytecode have the most complete features.

The Java bytecode offers many advantages, even if the complex sets of opcode involves having specifically optimized interpreters. But a Java interpreter source code can be easily found, that could be adapted for embedded requirements.

As front-end source code language it's possible to envisage the use of another language than Java, but Java source code presents a new viewpoint in the evolution of programming languages--creation of a small and simple language that's still sufficiently comprehensive to address a wide variety of software application development. Although Java is superficially similar to C and C++, Java gained its simplicity from the systematic removal of features from its predecessors.

Primary design features of Java, namely, it's *simple* (from removing features) and *familiar* (because it looks like C and C++) and the other characteristics (such as object oriented, robust and secure, architecture neutral and portable, interpreted, threaded, and dynamic) finish to decide to choose Java as the source code language.

3.2 Real-time problematic

Unlike most languages designed for real-time programming, interpreted languages are designed more to simplify programming than to enable programmers to write software that complies reliably with real-time constraints. Therefore it is usually says that this is not the purpose of interpreted languages and associated Virtual Machine to include real-time considerations. But, contrary to popular notions, hard real-time problems with extensive hardware interaction can be solved entirely in interpreted languages. The real-time constraints are not tackled by systems that are interpreted or rely on techniques such as Ahead of Time compilation. However, two methods can be proposed to match the real-time requirements. But the determinism of bytecode execution and garbage collection have to be solve regardless of the methods.

The first method consists in the development of a specific Virtual Machine running over a real-time operating system (RTOS). In this case, the real-time considerations are directly managed by the RTOS. The Virtual Machine is simpler but depends on the RTOS.

The second method consists in the implementation of the real-time requirements directly in the Virtual Machine. Several working groups proposed the specification of a real-time Java, like the Real-Time for Java Expert Group, that have finalize the Real-Time Specification for Java (RTSJ). This specification introduces the common problems of the real-time programming: scheduling, memory management, synchronisation, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination, physical memory access, and exception management. RTSJ propose to improve the Java language and/or JVM implementations. Presently, it does not exist any Java Virtual Machine that implements these real-time specifications. Nevertheless, this appears to be the right way to obtain a portable JVM with a standard behaviour whatever the platform used.

3.3 AERO solution

The chosen solution is to involves both of this methods, by implement real-time requirements in a JVM, and use RTOS to ensure low-level hard real-time functions.

A specification that includes all requirements concerning the final interpreter to develop was written; it details real-time comportments and functions required, essential APIs to be supported, and compatibility with existing specifications and designs of space domain or others standards.

A large number of commercial and open source Java Virtual Machines are available on the market. The commercial JVMs are generally not delivered with their sources and their adaptation to the requirements of space systems is not possible. The open source JVM give us the possibility to focus our activity on the development of real-time and safety mechanisms. We already identified numerous virtual machines that should be a good foundation for the development of AERO-JVM.

The selection process has first start with the characterization of methodology, including destructive and selective criteria definition (features, availability, code readability, maintainability etc.) 25 JVM are evaluated, the resulting chosen solution is: Jamaica JVM from Aicas Gmbh.

3.4 Developments and customizations

The base solution have been customized: new functions was developed for real-time support, specific deterministic garbage collection was added, then RTSJ compliance have require a lot of work to add all primitives, memory object and functionalities. A set of APIs required by space domain has been defined; some other features, of the base solution, not required have been removed.

The functionality of the classes of the RTSJ and the determinism of garbage collection are an important requirement for AERO-JVM and the major enhancement to the functionality made during the AERO-JVM project. Priority based scheduling is provided by the AERO-JVM through the RTSJ interface. New real-time thread classes can be used together with the priority interface. This scheduling is enabled by use of the underlying operating system's scheduler.

Implemented garbage collector is based a new specific deterministic algorithm, with a particular implementation. There is no dedicated garbage collector thread, consequently, real-time thread are automatically executing at a priority logically higher than that of the garbage collector. All garbage collection work is performed within the application threads. Even though the AERO-JVM garbage collector does not require special treatment of memory allocations performed within real-time code, the special memory classes defined in the RTSJ are provided by the AERO-JVM to ensure inter-operability with other Java implementations and tools. These classes include scoped memory, immortal memory, physical memory etc.

3.5 Validation and evaluation

Validation of the AERO-JVM has consisted to check the compliance with Java specification, then the RTSJ compliance and the specific space domain requirements compliance. This process is ended, and the application is conform to the full requirements list.

Evaluation is under progress on an ERC32 bench, and consists to run some representative applications, to insure the real-time comportment and performance of the AERO-JVM. This evaluation will finish in March 2003. First results confirm the high potential of the developed solution, but a more complete evaluation is foreseen during next industrialization of the project.

4. AERO-JVM Product

4.1 Definition

The AERO-JVM is a new implementation of the Java Virtual Machine specification that provides hard real-time guarantees for all features of the languages together with high performance runtime efficiency. This enables all of Java's features to be used for on-board hard real-time tasks. This includes features essential to object-oriented software development like dynamic allocation of objects, inheritance, introspection and dynamic binding. Sophisticated automatic class file compaction and dead-code elimination techniques are involved to reduce the code footprint to the bare minimum.

AERO-JVM includes a runtime system for the execution of applications written for the Java API V1.2. It is designed for real-time and embedded systems and offers unparalleled support for this target domain, including deterministic dynamic memory management, which is performed by the AERO GC garbage collector. The use of dynamic class loading is possible with AERO-JVM. This enables the hot swapping of code and the dynamic addition of new features.

4.2 Features

AERO-JVM could interpret Java bytecode directly, but class files and the AERO-JVM may be linked into a standalone binary using a specific Ahead of Time Compiler that gives native performances. The final systems could use Posix standard, VxWorks or RTEMS operating systems.

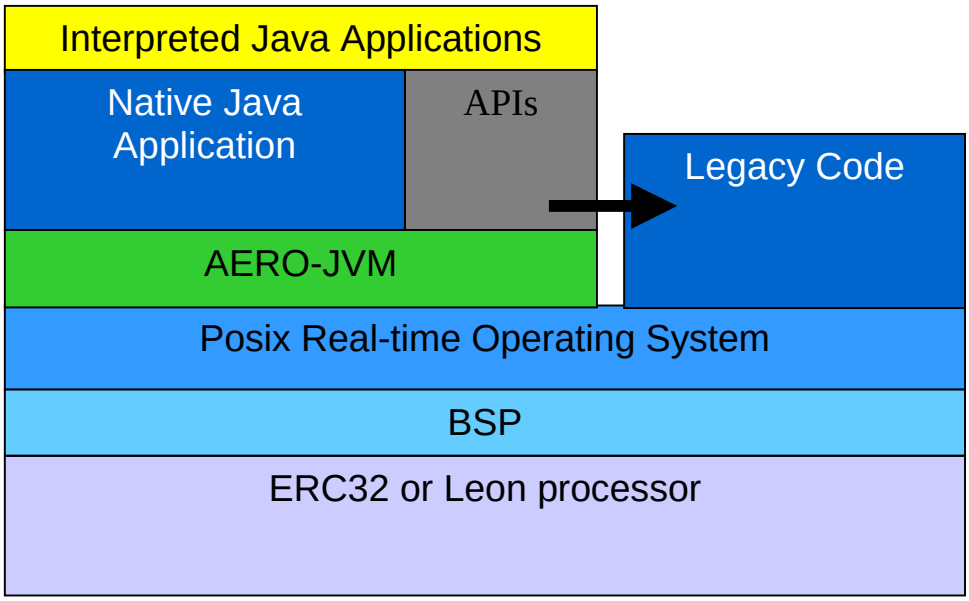
The AERO-JVM provides hard real-time guarantees for all primitive Java operations, but all threads executed by the AERO-JVM are real-time threads too, there is no need to distinguish real-time from non-real-time threads. Any higher priority thread is guaranteed to be able to pre-empt lower priority threads within a fixed worst-case delay.

Among the features of AERO-JVM are:

- Hard real-time execution guarantees
- Deterministic Garbage collection
- Minimal foot print and ROMable code
- Native code support (JNI)
- Dynamic Linking (interpreted mode only)
- Java Portability
- Fast execution (interpreter or AOT compiler)
- RTSJ (Real-Time Specification for Java) compliant

There are no restrictions on the use of the Java language to program real-time code: since the AERO-JVM executed all Java code with hard real-time guarantees, even real-time tasks can use the full Java language, i.e., allocate objects, call library functions, etc. No special care is needed; short worst-case execution delays can be given for any code.

The AERO-JVM is able to run some Java applications at the same time, on a single instance of the JVM. This new capability improves Java execution model efficiency; else a new instance of a JVM is required to execute a new application in classical scheme.



AERO-JVM integration in On Board Architecture

The AERO-JVM run directly on a real-time operating system, but could be also interfaced with a core DHS. Java applications are compiled and/or interpreted, and may access directly native legacy code. APIs are libraries of Java code statically compiled in native. The Java execution model is preserved; AERO-JVM provides, in a real-time deterministic way, segregation between Java applications and the rest of the system.

5. CONCLUSION

Interpreted languages, and Java especially, with their Virtual Machines implement a large number of interesting features for the development of complex applications. The major expected benefits consist in the definition of a homogeneous development environment (object modeling and object development) and a better reusability of the existing components leading to an important reduction of the software cost and development time.

By implementing the Java language, AERO-JVM offer all required support to ensure the operability of complex onboard systems, and provide a standard way for the reprogramming function.

Evolutions and customisations made in the context of the AERO project, allow the use of Java in hard real-time applications with the AERO-JVM. The AERO-JVM is a new implementation of the Java Virtual Machine specification that provides hard real-time guarantees for all features of the languages together with high performance runtime efficiency and dual mode execution (interpretation and Ahead of Time compilation). This enables all of Java's features essential to object-oriented software to be used for on-board hard real-time tasks.

6. REFERENCES

1. A.Walter: Deterministic Execution of Java's Primitive Bytecode Operations, *Java Virtual Machine Research and Technology Symposium (JVM'01)*, Monterey, California, April 2001
2. F.Siebert, Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages, *ETAPS Conference*, Genova, December 2000
3. F.Siebert: Guaranteeing Non-Disruptiveness and Real-Time Deadlines in an Incremental Garbage Collector, *ISMM*, 1998,
4. F.Deladerrière, F.Siebert: Hard real-time JVM for space application, *Galileo Workshop*, Noordwijk, October 2002
5. T.Ritzau, Deterministic Garbage Collection, *ETAPS Conference*, Genova, December 2000
6. F.DeBruin, F.Deladerrière, F.Siebert: Hard real-time JVM for embedded systems, *DASIA 2003*, Prague, June 2003

summary previous next