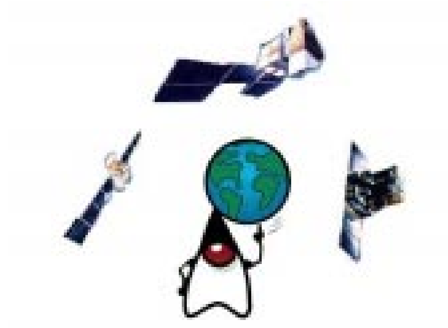
AERO:Architecture for Enhanced Reprogrammability and Operability

Contract ESTEC 15750/02/NL/LVH



Evaluation of AERO-VM on ERC32SC

Validation and performances results

(Extract of original document)

F. Deladerrière, M.Julien - Astrium SaS

F.Siebert, T.Ruppe, R.Schnider, N.Scharnberg - Aicas Gmbh

T.Ritzau, Linköping Universitet

Reference: AERO/TN4

Issue: 0.3

Date: **2003-04-09**

Abstract:

This document is the evaluation report of the AERO Real-time Java Virtual Machine project, output of the task 1.b.2.

This document provides evaluation results of the AERO JVM, who is a Java Virtual Machine with real-time capacities, for ERC32 processor. This document contains an overview of tests, and results.

Written by:	**Name**	**Company**	**Signature**	**Internal reference**
	F. Deladerrière	Astrium		

AERO (Architecture for Enhanced Reprogrammability and Operability) is an ESA project (Contract ESTEC 15750/02/NL/LVH) conducted by a consortium led by ASTRIUM-SAS with Aicas Gmbh and Linköping Universitet. For more information please contact:

Frank J. de Bruin
ESTEC. Keplerlaan 1. PO Box 299
2200 AG Noordwijk ZH – The Netherlands
Tel: +31 (0) 71 565 4951. Fax: +31 (0) 71 565 5420
e-mail fdebruin@estec.esa.nl

Frédéric Deladerrière ASTRIUM 31, avenue des cosmonautes F-31 402 Toulouse Cedex 4, France Tel: +33 5 62 19 56 49. Fax: +33 5 62 19 78 97 e-mail: frederic.deladerriere@astrium-space.com	Fridtjof Siebert AICAS Gmbh Hallesche Allee 11 D-76139 KarlsRuhe, Germany Tel: +49 721 680 6722 Fax: +49 721 968 6073 e-mail: siebert@aicas.com	Tobias Ritzau Linköping Universitet Dep. Of Computer and Information Science SE-58183 Linköping, Sweden Tel: +46 13 28 4494. Fax: +46 13 28 5899 e-mail: tobri@ida.liu.se

Revision History

Version	Date	Paragraphs modified	Comments
0.1	2003-03-07		First issue
0.2	2003-03-12	2.2.3, 2.3.3, 5.1, 5.2.4	Pictures updated, Test added, 1553 data rate added, ERC32 measures adjusted, Leon results added
0.3	2003-04-09	3.2.3, 6, 7, 8	QR updates

Table of Contents

1. INTRODUCTION ... **6**

1.1 SCOPE ... 6

1.1.1 Scope of the Project ... *6*

1.1.2 Scope of the Document ... *6*

1.2 RELATED DOCUMENTATION ... 6

1.3 APPLICABLE DOCUMENTATION ... 6

1.4 TEST CASES APPLICABILITY ... 7

1.5 DEFINITION OF TERMS AND ACRONYMS ... 7

1.5.1 Definition of Terms ... *7*

1.5.2 Acronyms and Abbreviations ... *7*

2. OVERVIEW ... **8**

2.1 AERO-VM DEFINITION ... 8

2.2 TEST ENVIRONMENT ... 9

2.2.1 First level of tests ... *9*

2.2.2 Representative tests using Software Validation Facilities ... *10*

2.2.2.1 Tornado ... 11

2.2.2.2 Others functions ... 12

2.2.3 ERC32 board ... *15*

2.3 SPACE CONTEXT TESTS PRESENTATION ... 17

2.3.1 Classical interpreted procedure functional tests ... *17*

2.3.2 Basic computation algorithms ... *19*

2.3.3 Complex functions algorithm ... *20*

2.3.4 Low level functions test ... *22*

3. VALIDATION RESULTS ... **23**

3.1 INTRODUCTION ... 23

3.2 TEST ENVIRONMENT ... 23

3.2.1 The Mauve Project ... *23*

3.2.2 Tests for the RTSJ ... *24*

3.2.3 Jamaica/AERO-VM test framework ... *24*

3.2.4 Test framework ... *30*

3.3 TEST RESULTS FORMAT ... 33

3.4 TEST RESULTS ... 34

4. COMPILER TEST, CYCLOMATIC COMPLEXITY & CODE COVERAGE ... **35**

4.1 JACKS TEST ... 35

4.2 CYCLOMATIC COMPLEXITY ... 36

4.3 CODE COVERAGE ... 37

5. EVALUATION RESULTS ... **40**

5.1 FUNCTIONAL EVALUATION SUMMARY ... 40

5.2 PERFORMANCE EVALUATION ... 41

5.2.1 SpecJVM98 ... 41
5.2.2 CaffeineMark ... 41
5.2.3 Specifics tests ... 41
5.2.4 Results ... 42
5.3 PERFORMANCES TRADE OFF (OBCP/ASTRIUM ERC32-VM/AERO-VM/NATIVE C) ... 45
5.3.1 Bytecodes ... 45
5.3.2 Performances of execution ... 47

6. SUMMARY OF SUPPORTED AND VALIDATED API ... 51

7. CONCLUSION ... 52

8. ANNEXES ... 53

8.1 ANNEXE : OBJA SYSTEM ... **ERREUR ! SIGNET NON DEFINI.**
8.2 ANNEXE : VALIDATION TESTS ... **ERREUR ! SIGNET NON DEFINI.**
8.3 ANNEXE : CYCLOMATIC COMPLEXITY DETAILS ... **ERREUR ! SIGNET NON DEFINI.**
8.4 ANNEXE : CODE COVERAGE DETAILS ... **ERREUR ! SIGNET NON DEFINI.**
8.5 ANNEXE : VALIDATED APIS SUPPORTED BY AERO-VM ... **ERREUR ! SIGNET NON DEFINI.**
*8.5.1 Standard classes provided by AERO/JVM ... **Erreur ! Signet non défini.***
*8.5.2 Validation using test suite on Host system ... **Erreur ! Signet non défini.***
8.6 NON VALIDATED REQUIREMENTS ... **ERREUR ! SIGNET NON DEFINI.**
8.7 ANNEXE: METHODS AND FIELDS PROVIDED IN THE STANDARD APIS AND TESTED... **ERREUR ! SIGNET NON DEFINI.**
*8.7.1 package java.lang ... **Erreur ! Signet non défini.***
*8.7.2 package java.lang.ref; ... **Erreur ! Signet non défini.***
*8.7.3 package java.lang.reflect ... **Erreur ! Signet non défini.***
*8.7.4 package java.io ... **Erreur ! Signet non défini.***
*8.7.5 package java.math ... **Erreur ! Signet non défini.***
*8.7.6 package java.net ... **Erreur ! Signet non défini.***
*8.7.7 pacakge java.security ... **Erreur ! Signet non défini.***
*8.7.8 pacakge java.sql ... **Erreur ! Signet non défini.***
*8.7.9 package java.text: ... **Erreur ! Signet non défini.***
*8.7.10 pacakge java.util ... **Erreur ! Signet non défini.***
*8.7.11 package javax.realtime ... **Erreur ! Signet non défini.***

1. Introduction

1.1 Scope

1.1.1 Scope of the Project

AERO (Architecture for Enhanced Reprogrammability and Operability) is an ESA project (contract ESTEC 15750/02/NL/LVH). The objectives of the project are to investigate on a real-time Java virtual machine for ERC32. Special attention is put on the garbage collection mechanism and deterministic execution model.

The project is split in two phases. The first phase investigates existing virtual machine to choose a potential candidate that will be customized, are then investigated the definition of requirements concerning a real-time interpreter in on-board systems. An implementation plan is proposed for the second phase. This second phase is dedicated to the definition of software functions of the real-time Java virtual machine and to their implementation and assessment through validation tests.

1.1.2 Scope of the Document

This document is an output of the task 1.b.2 “Evaluation of the software JVM on ERC32SC”.

The objective of this document is to provide results of the relevant tests cases allowing to perform the validation of the AERO JVM functions. All requirements of AERO JVM specification (SP1) have to be verified.

This document includes the following parts:

- Generalities including : summary of AERO JVM validation test objectives and strategy,
- Detailed description of space context validation tests
- Results of validation

1.2 Related Documentation

[AERO]	Architecture for Enhanced Reprogrammability and Operability, ESTEC Contract n°15750/02/NL/LVH.
[Prop]	Architecture for Enhanced Reprogrammability and Operability, Proposal for ESA ITT AO/1-3959/01/NL/PB. Astrium EEA.PR.FD.3682269.01.
[MNM]	Minutes of AERO Project Negotiation Meeting, Noordwijk, NL, January 31, 2002
[MP]	Management Plan of AERO Project
[BOOK1]	Inside Java 2 Virtual Machine, B.Veners, Mac Graw Hill, 1999 2d Edition ISBN 0-07-135093-4
[VP]	Validation Plan of AERO Project.
[DDD]	AERO Project detailed design document

1.3 Applicable Documentation

[RTSJ]	Real-Time Specification for Java (RT for Java Expert Group) final release, December 2001.

1.4 Test cases applicability

The test cases results in this document are derived from, and consistent with the specification issue defined in VP document..

1.5 Definition of Terms and Acronyms

1.5.1 Definition of Terms

None

1.5.2 Acronyms and Abbreviations

Acronyms and abbreviations used in this text are defined as follows :

AERO Architecture for Enhanced Reprogrammability and Operability

AIE AsynchronouslyInterruptedException

ATC Asynchronous Transfer of Control

ESA European Space Agency

ESTEC European Space Technological Centre

GC Garbage Collector

ICD Interface Control Document

ICR Individual Control Register

JVM Java Virtual Machine

OBS On Board Software

RTSJ Real-Time Specification for Java

TBC To Be Confirmed

TBD To Be Defined

TN Technical Note

VM Virtual Machine

WP Work Package

2. Overview

2.1 AERO-VM Definition

AERO-VM is a standard Java bytecode interpreter for real-time embedded systems. The AERO-VM must provides hard real-time guarantees for all primitive Java operations. 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, and dynamic binding. Sophisticated automatic class file compaction and dead-code elimination techniques must be involved to reduce the code footprint to the bare minimum.

Class files and the AERO-VM may be linked into a standalone binary for execution out of ROM. A file-system is not necessary for running Java code even if the final systems will use VxWorks and RTEMS operating systems.

The use of dynamic class loading must be possible with AERO-VM. This enables the hot swapping of code and the dynamic addition of new features.

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. It is 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. This includes dynamic memory management, which is performed by the AERO GC garbage collector.

Among the features of AERO JVM shall be :

- Hard real-time execution guarantees
- Minimal footprint
- ROMable code
- Native code support
- Dynamic Linking
- Portability
- Fast execution
- Powerful Tools

All threads executed by the AERO Java Virtual Machine will be real-time threads, there is no need to distinJVMsh 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.

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.

2.2 Test environment

2.2.1 First level of tests

First tests are made with AERO JVM software, with no specific cradle environment. A “black box” approach is used, each test executes AERO JVM software with specific java application, test results are checked in output files : trace file , or branch coverage result file.

Static test are made under MS-Windows or Linux operating system depending of tools available plate-forms.

Dynamic test are made under Linux when using specific tools and VxWorks (release 5.4) operating system when checking direct JVM execution.

External free testing tools used will be :

- MAUVE Test suite (basic VM and API functionalities) : http://sources.redhat.com/mauve/
- Jacks : http://www-124.ibm.com/developerworks/oss/cvs/jikes/~checkout~/jacks/jacks.html
- SPECjvm98 : http://www.spec.org/osg/jvm98/
- JavaGrande : http://www.epcc.ed.ac.uk/javagrande/index_1.html
- Cafeine microbenchmark : http://www.pendragon-software.com/pendragon/cm2/
- Javancss :http://www.kclee.com/clemens/java/javancss/
- gcov test suite : http://gcc.gnu.org

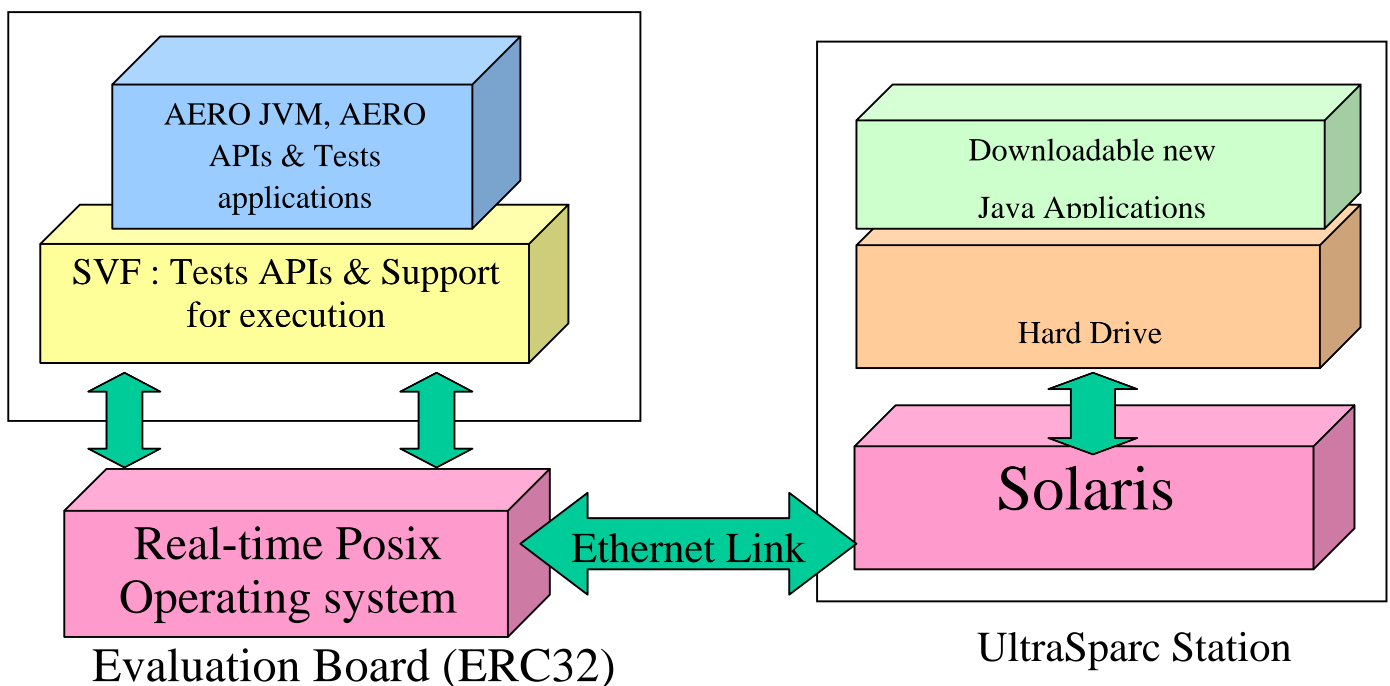
2.2.2 Representative tests using Software Validation Facilities

Software Validation Facilities (SVF) will be developed to provide representative context to execute onboard applications on AERO-VM.

SVF will consists in a set of java APIs and tools that provides stubs of onboard functions :

- support for evaluation on validation bench
- capabilities to be extended
- Space context representation
- Communication and base modules
- Monitoring
- I/O simulated services (loading, monitoring)

Detailed SVF design is provided in DDF document, high level principle with real hardware is :



In the same way, thanks to tests APIs and support for execution are writing in Java, it is also possible to run validation on target simulator under Linux operating system.

SVF results objectives consists too :

- checks performance, determinism, security and robustness, system interfaces …
- verify what appends in Real-Time loop when an external call is made in Java applications
- Control out of context Java bytecode execution security

SVF will solve questions concerning difference between Java bytecode, Java Native Binary, and Non Java native on :

- Performances of executions
- System interface capabilities,
- Security and robustness,
- Local determinism of execution
- Real-time loop scheduling determinism

2.2.2.1 Tornado

Main SVF features are implemanted under VxWorks using **Tornado** environement, that include a target agent, target server and tools.

The target agent is a scalable software agent that can be inserted into the target processor. This target agent connects all Tornado host-resident tools to the target run-time system, giving the target an unprecedented level of independence from the host system. Both the agent and the agent's driver interface are independent of the run-time system. As a result, the target agent can execute before the run-time kernel is running, thus simplifying the bring-up of the operating system on custom hardware. The agent can execute in either task-specific or system-wide breakpoints, which greatly simplifies debugging.

The Target server: the host-based target server allows development tools to be independent of the target system. There is one server per target; all host tools access the target through this server, which functions to satisfy tool requests by breaking each request into the necessary transactions with the target agent. The target server includes features that improve the performance of the cross-development structure: a target memory cache, host-based target memory management, and a streamlined host-target protocol to reduce communication traffic. A target server need not reside on the same host as Tornado tools, as long as the server and tools hosts are connected. This provides great flexibility in setting up a development network.

Loading
Tornado permits developers to incrementally load object modules into a target system. This ability to dynamically link and load object modules is central to the Tornado architecture: developers do not need to link the application to the kernel on the host, nor download the entire executable as one static environment. As a result, each edittest-debug cycle is dramatically shortened. All application modules can be shared among development team members, but do not need to be re-linked on the host. This makes it possible for developers to add object modules to a “live” VxWORKS target environment when debugging or reconfiguring applications.

Several application program interfaces (APIs) are available and published for reference, from integrated development environment (IDE) interfaces down to the connection implementation. At the IDE level, the API provides front-end tool extensibility and customization. Many aspects of the user interface are under the user's control, including menu items and extensions to the underlying Tcl code. The next level of API provides an interface to all the target information from the host. At the operating system level, there is an API to the VxWorks kernel itself, allowing new configurations of the run-time system and additional driver development to occur independently.

Following page gives an overview of Tornado display, when running AERO-VM.

2.2.2.2 Others functions

The remaining SVF features are manually coded :

- the ClassLoader
- JNI Interface and 1553 bus driver

ClassLoader

Java class files are not loaded into memory all at once, but rather are loaded on demand, as needed by the program. The ClassLoader is the part of the JVM that loads classes into memory.

The Java ClassLoader, furthermore, is written in the Java language itself. This means that it's possible to create a specific ClassLoader without having to understand the finer details of the JVM. If the JVM has a ClassLoader, then why would you want to write another one? The default ClassLoader only knows how to load class files from the local filesystem. This is fine for regular situations, with Java program fully compiled and waiting on local computer.

But one of the most innovative things about the Java language is that it makes it easy for the JVM to get classes from places other than the local hard drive or network. In the case of AERO-VM custom ClassLoader is required to load executable (bytcode) content from the plateform. For evaluation purposes, a ClassLoader have been written to allow OBJA loading, using network ERC32 bench features.

Besides simply loading files a custom ClassLoader could by written to :

- Automatically verify a digital signature before executing untrusted code
- Transparently decrypt code with a user-supplied password
- Create dynamically built classes customized to the user's specific needs

Anything conceivable to write that can generate Java bytecode can be integrated into an application.

JNI and driver

The Java Native Interface (JNI) is the native programming interface for Java that is part of the JDK. By writing programs using the JNI, it is possible to ensure that the code is completely portable across all platforms.

The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly. In addition, the *Invocation API* allows you to embed the Java Virtual Machine into your native applications.

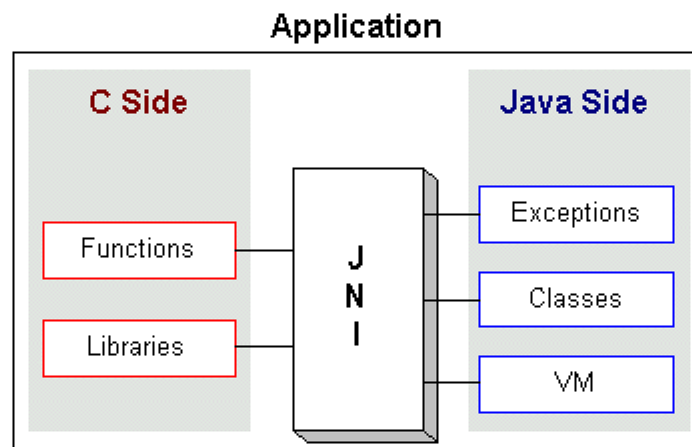
Programmers use the JNI to write native methods to handle those situations when an application cannot be written entirely in the Java programming language. For example, use native methods and the JNI could be required in the following situations:

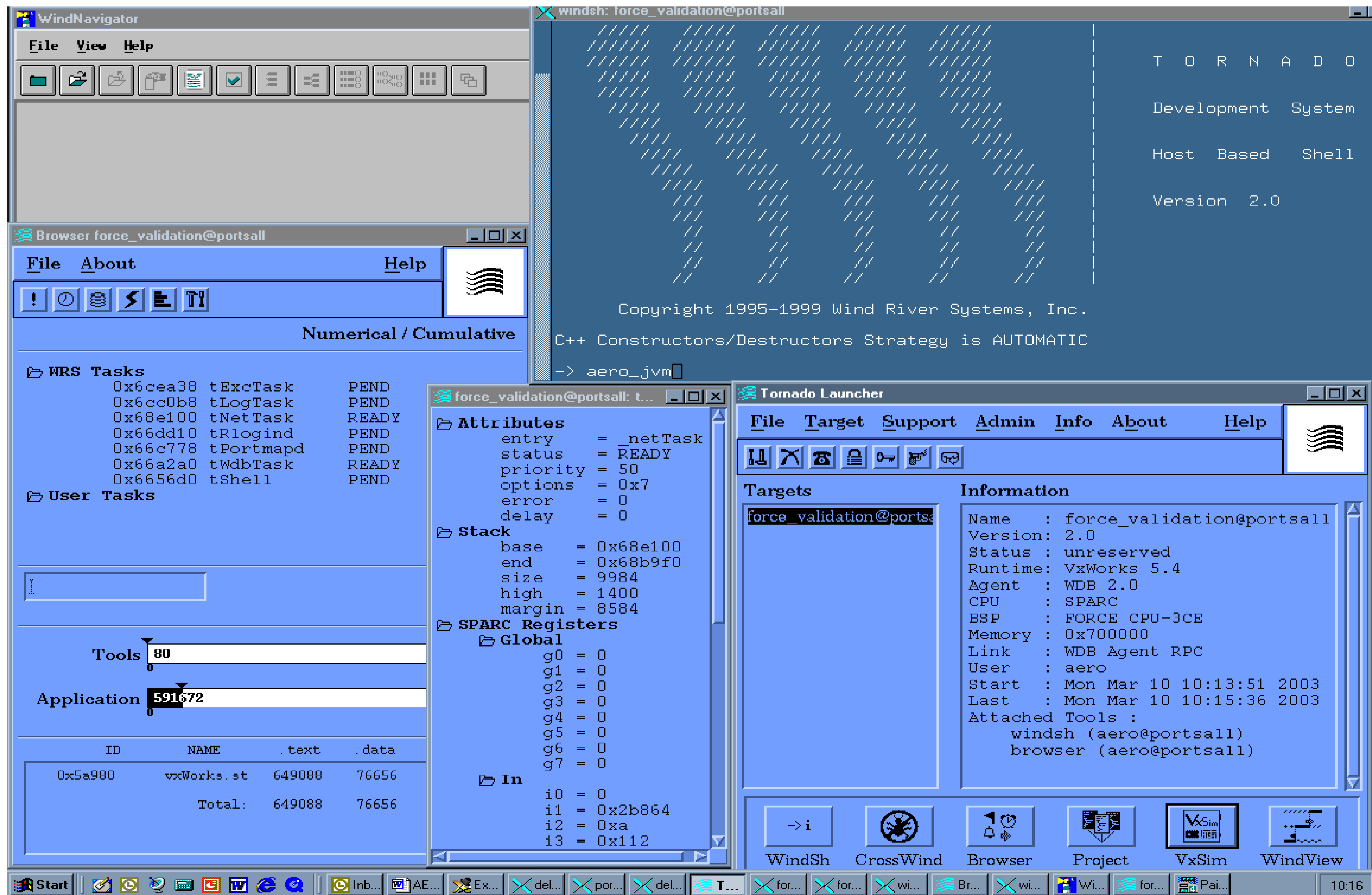
- The standard Java class library may not support the platform-dependent features needed by your application.
- It is possible to use an existing library or application written in another programming language and make it accessible to Java applications.
- You may want to implement a small portion of time-critical code in a lower-level programming language, such as assembly, and then have your Java application call these functions.

The 1553 Bus driver is in the first category, it’s objective is to provide access to plateform dependant feature.

Programming through the JNI framework lets use native methods to do many operations. Native methods may represent legacy applications or they may be written explicitly to solve a problem that is best handled outside of the Java programming environment.

It is easy to see that the JNI serves as the glue between Java and native applications. The following diagram shows how the JNI ties the C side of an application to the Java side.



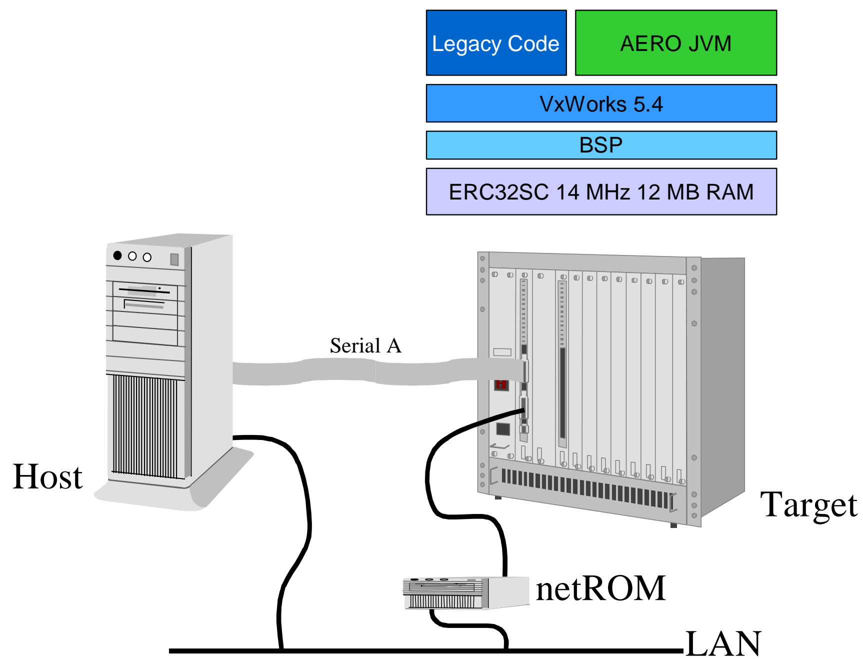

WindNavigator
File View Help
windsh: force_validation@portsall
T O R N A D O
Development System
Host Based Shell
Version 2.0
Copyright 1995-1999 Wind River Systems, Inc.
C++ Constructors/Destructors Strategy is AUTOMATIC
-> aero_jvm
Browser force_validation@portsall
File About Help
Numerical / Cumulative
WRS Tasks
0x6cea38 tExcTask PEND
0x6cc0b8 tLogTask PEND
0x68e100 tNetTask READY
0x66dd10 tRlogind PEND
0x66c778 tPortmapd PEND
0x66a2a0 tWdbTask READY
0x6656d0 tShell PEND
User Tasks
Tools 80
Application 591672
ID NAME .text .data
0x5a980 vxWorks.st 649088 76656
Total: 649088 76656
force_validation@portsall: t...
Attributes
entry = _netTask
status = READY
priority = 50
options = 0x7
error = 0
delay = 0
Stack
base = 0x68e100
end = 0x68b9f0
size = 9984
high = 1400
margin = 8584
SPARC Registers
Global
g0 = 0
g1 = 0
g2 = 0
g3 = 0
g4 = 0
g5 = 0
g6 = 0
g7 = 0
In
i0 = 0
i1 = 0x2b864
i2 = 0xa
i3 = 0x112
Tornado Launcher
File Target Support Admin Info About Help
Targets
force_validation@ports
Information
Name : force_validation@portsall
Version: 2.0
Status : unreserved
Runtime: VxWorks 5.4
Agent : WDB 2.0
CPU : SPARC
BSP : FORCE CPU-3CE
Memory : 0x700000
Link : WDB Agent RPC
User : aero
Start : Mon Mar 10 10:13:51 2003
Last : Mon Mar 10 10:15:36 2003
Attached Tools :
windsh (aero@portsall)
browser (aero@portsall)
WindSh CrossWind Browser Project VxSim WindView
Start
10:16

2.2.3 ERC32 board

Validation at Astrium Toulouse, is made using a real ERC32 processor on an evaluation board.

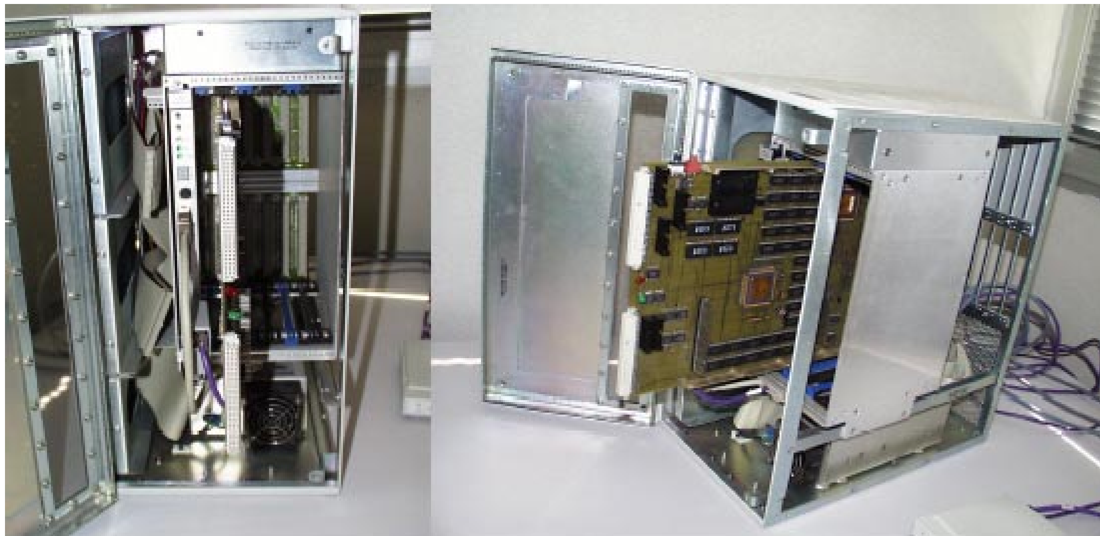
This board implements following features :

- ERC32 Single Chip, 14 Mhz, 12Mo Ram
- VxWorks 5.4 real-time operating system
- Netrom for Ethernet capabilities and control from a remote UltraSparc Workstation
- Enhanced debug and validation environment through Tornado tools



An ERC32 SC Board

This board was used to validate the AERO-VM in a representative environment. The processor and configuration (speed, RAM etc.) is the same than used in satellite.



An ERC32 Board in VME Rack

The board inserted in VME Rack, with serial link and Ethernet connection.

2.3 Space context tests presentation

Different category of tests, involve large specter of AERO-VM functions

2.3.1 Classical interpreted procedure functional tests

OBCP/IP/OBJA functional level

This class of application consist in current used interpreted procedure used on Rosetta and E3000 family of telecom satellites.

Function implanted correspond to mode manager, monitoring, reprogrammable mission functions etc.

A first Java portage was made to test the ERC32-VM, a standard java interpreter (ie non real-time) developed by Astrium. This set of testing application are run on the new AERO-VM.

OBJA Object

Java is a fourth generation language, all component are object. The system is based on inheritance mechanism, and instead of define new type of component for each new OBJA, the idea is to define a generic OBJA base object, from which all new OBJA will inherit. OBJA are under control of an OBJA Manager.

To provide simple and robust control on OBJA, the solution is to use a Java thread for each OBJA. The novelty is that in AERO-VM all thread are real-time thread. Using thread provide direct control methods to implement the start, stop, suspend and resume functions of the OBJA Manager. The CPU slot allocation could be made directly on the AERO-VM by using the priority mechanism of the Java thread.

To provide a simple used of all OBJA manager functionalities, the OBJABase provide a set of methods. These methods are all finals overrides of the different ones implements in the distinct class of the system, except for:

- waitEvent, waitAsynchronousEvent and processAsynchronousEvents (see Events Managers)
- “go”: the implementation of “go” permit the definition of the OBJA comportment.

Update was made to integrate new capabilities provided by real-time AERO-VM APIs

OBJA Manager

The OBJA manager is a set of class in charge to manage information about: OBJA and their threads, messages from/to the system and events. In addition, the manager offers a set of methods to control OBJA. To perform this job the OBJA manager is split in two parts: an active part (running threads) and a passive part.

Composition of the active part of the manager

The manager is implemented through three threads:

- **The message manager**, who manages messages from the OBSW (read and execution), action from an OBJA to other used directly the methods of OBJABase class.
- **The waiting events manager** who restarts OBJA waiting an event occurrence.
- **The asynchronous events manager** who warn OBJA when precise events occur, like in the other event manager an OBJA specify this events.

Passive part of the manager

The passive part of the manager is compound of a set of methods that permits to:

- Create a new OBJA
- Create a new OBJA child (Thread creation inside OBJA)
- Suspend an OBJA
- Resume an OBJA
- Set the priority of an OBJA
- Gets OBJA's informations
- Stop an OBJA.
- Wait for specific events
- React to specific events
- Choose the action to do when a bug is encountered by an OBJA.

Capacity tests

This tests involve to run in the same time a large number of application (more than 64)

- native mode
- fully interpreted mode

2.3.2 Basic computation algorithms

- Attitude algorithm

  Standard attitude algorithm with a set of entry data that correspond to star position and catalogue.

- Barycenter algorithm (centroïd)

  Algorithm that calculate barycenter on 5 stars in a small noisy image.

- Matrix calculation

  Library to manipulate matrix, even if APIs could exists, this tests allow to evaluate capacity of calculation.

- FFT algorithm

  Dynamic FFT algorithm on image size from 2x2 up to 512x512

- Stellar sensor detection algorithm

  Another implemantation of centroid algorithm that calculate barycenter on 5 stars in a small noisy image.

- BigPi

  An algorithm that compute the first 250 digit of PI value

- Fibonnaci

  Classic Fibonnaci series calculation (recursive algorithm)

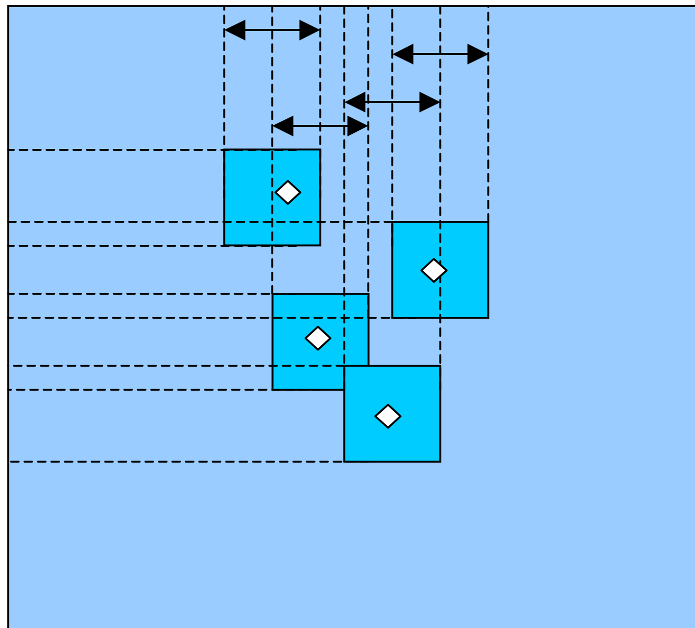
- Life

  Genetic algorithm, that simulate cells growth.

2.3.3 Complex functions algorithm

- APS algorithms

This algorithm consist to manage a new image device : the APS captor. During stars tracking, the APS captor is read only on defined windows positioned on stars. This windows are regularly read, and window positions are adjusted depending of star drift in the field of view. The problem consist to read windows in the optimal order to reduce cycle of process.



This figure gives an idea of the problem : stars are located inside Windows, and the main software implements a tracking of this stars. To get image of them, images are taken with APS, but only windows are reads. The APS captor return lines corresponding to Windows, even if windows are mixed. To gives high performances, the windows have to be read in optimized order. The algorithm return the ordered list of windows.

- Attitude Control System algorithms

These algorithms include orbitography calculations, ephemerids conversion, velocity and quaternion calculation, cartesian to lagrangian conversion, runge kutta algorithme etc. They are the base calcul of orbit propagator and autonomous navigation functions.

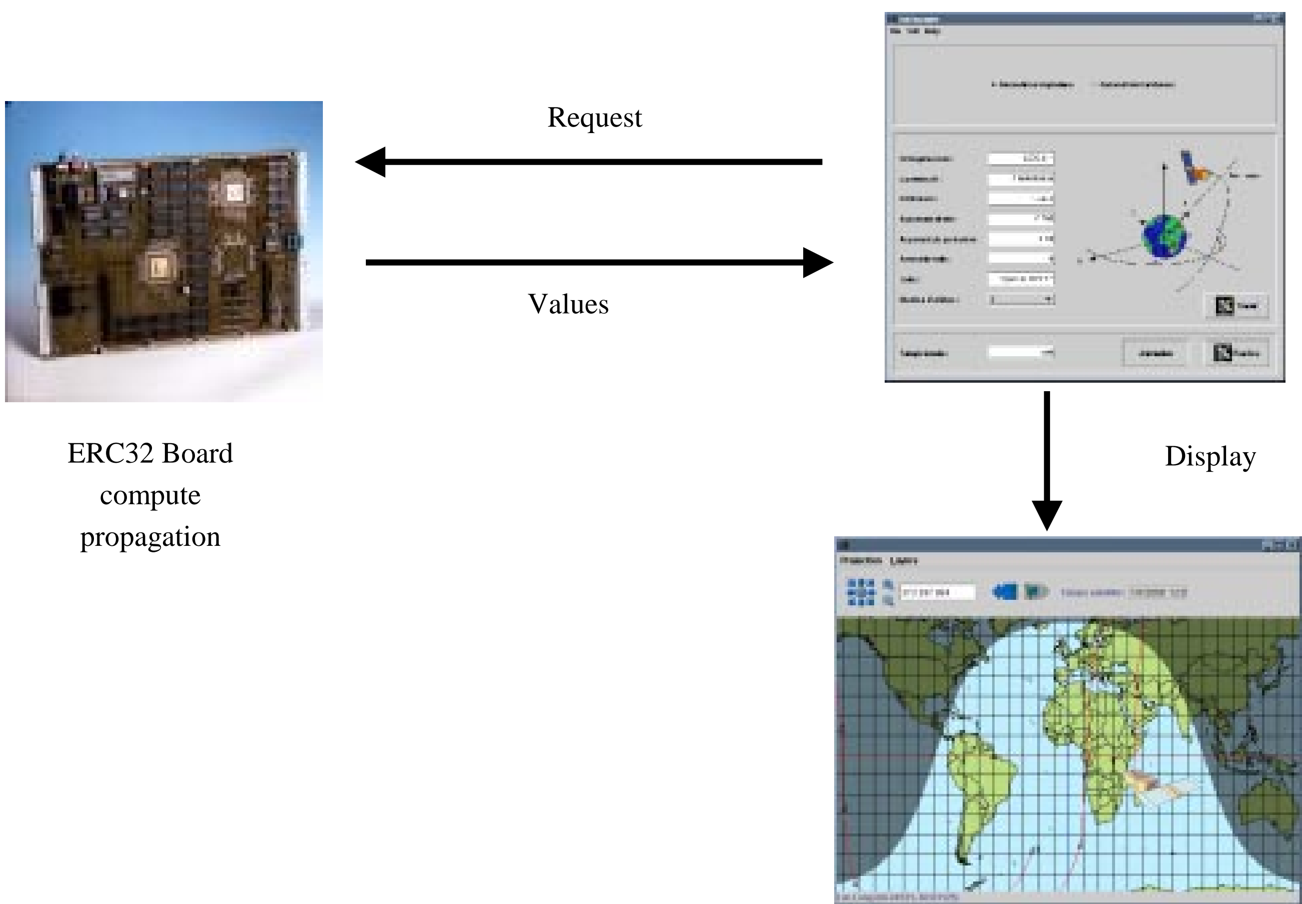
- Orbit propagator

  This is a Java version of onboard propagator used on OBMM. This software was runned on both Astrium ERC32-VM and AED-Java processor, then on AERO-VM.

  In entry the propagator take Kepler's parameters, number of orbits etc. and returns positions with a user defined resolution.

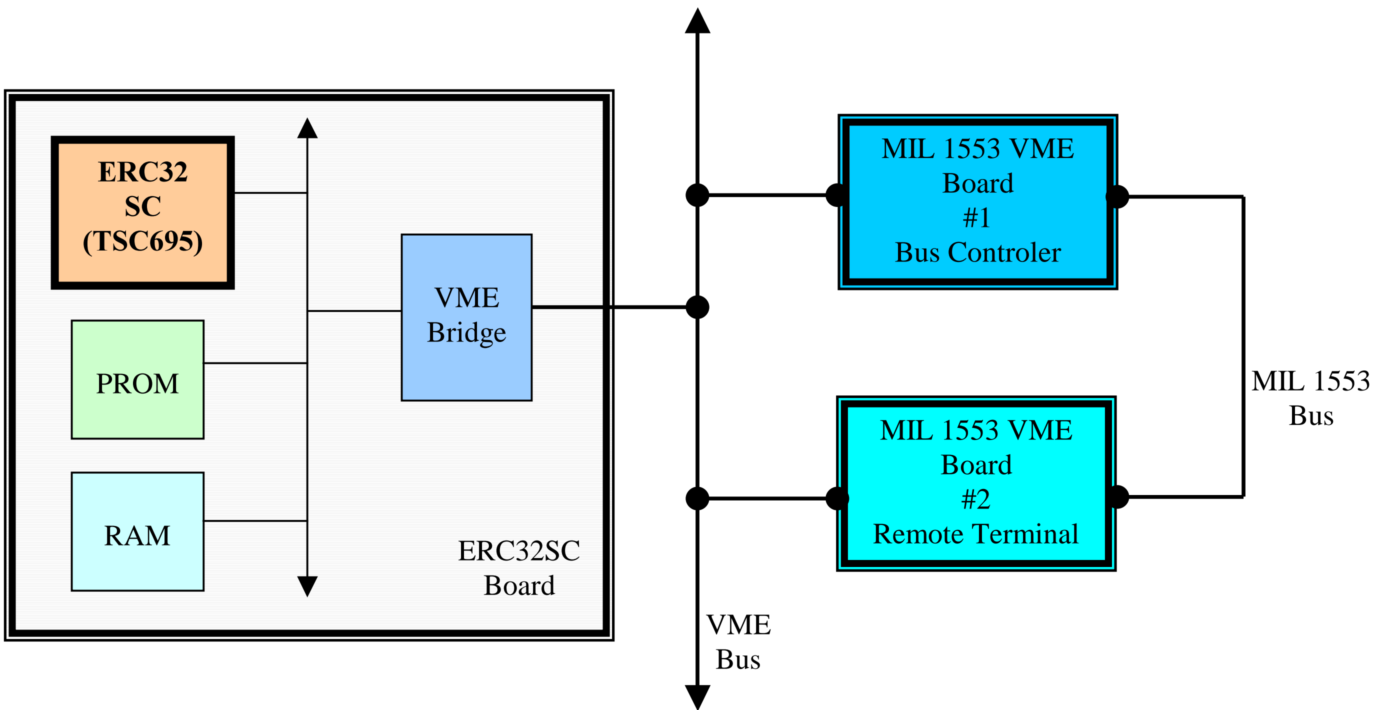
  The tests involve a user client, and a server, the server runs on ERC32 Board, on AERO-VM, and the client on a standard JVM with graphical support (MS-Windows, Linux, Solaris etc.). The client send TC to the server, that calculate orbit propagation, and returns elements that are displayed by the client.

  Communication is made using network socket connexion.

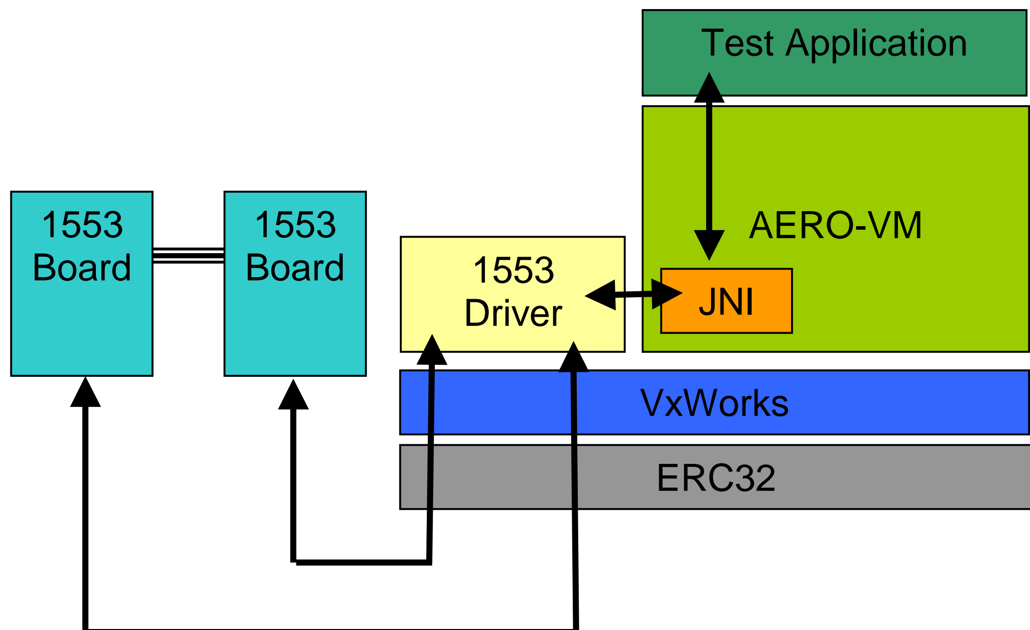


2.3.4 Low level functions test

This test control 1553 bus real-time exchanges. It involve two MIL 1553 Board, that are controlled by a Java application, using a low level driver. The application send and receive message through the two 1553 bus and check that exchanges are correct, and in the time.



The objective of the test is to verify the use of JNI on ERC32 board, then the 1553 Bus driver works fine over AERO-VM. Finally, speed (data rate) will be evaluate.



3. Validation Results

3.1 Introduction

The Java Virtual Machine (JVM) used in the AERO-Project is the AERO-VM developed at aicas GmbH, Karlsruhe. AERO-VM is being continuously extended in order to support all of Standard Java as well as Java extensions like the Realtime Specification for Java (RTSJ).

During this process it has to be ensured, that the development is done in small steps and that they are being validated daily by testing the whole AERO-VM. In order to do this, an automated test suite,the Jamaica test suite, has been developed for regression tests and earliest possible detection of problems.

The test suite is used only internally, namely at aicas. Any developer, using Jamaica in its system, will not get in touch with this test suite. It is used for quality reasons within the developement team.

3.2 Test environment

For the support of the Standard Java classes in AERO-VM, implementations of GNU Classpath GNU_CP are being used. That is a project of the open source community whose goal is to provide a free replacement for Sun's proprietary class libraries. Out of this community the Mauve Project MAUVE was established to provide a test suite for the GNU Classpath implementations. The Mauve test suite is freely available. A detailed description about the project follows in the next section.

The test suite, implemented as a shell script, performs nightly test runs without any user interaction. More comprehensive tests are run on weekends. The current version of the test suite does not include any benchmarks measuring run-time performance. In a future version, this is meant to be included.

3.2.1 The Mauve Project

The Mauve Project is collaborative project whose goal is to create a free suite of functional, black box tests for the Java core libraries. The initial group of contributors come from the GNU Classpath project, GNU Compiler for Java project and Transvirtual Technologies' Kaffe project. All three groups are working on free open source clean room implementations of the Java core libraries.

Although they are working on independent library implementations, they have decided to collaborate on a single test suite. The Mauve Project is the product of this collaboration. Collaborating on a single test suite will help to raise the quality bar for all independent implementations.

The Mauve Project test suite will help make possible thelibraries all behave the same way. Checking conformance of application with specification, but which one? There are a number of them out there: the JLS 1.0 and its "amendments", as well as the different flavors of Java, like PersonalJava and JavaCard. There are also the various JDK implementations (1.0, 1.1 and 1.2/2.0) that occasionally vary from the Java

Language Specs in subtle ways. One of the design goals of the test suite is to support multiple specs. A simple test case tag scheme is used to identify which specs individual test

cases work against. The tester selects the set of tags they wish to test withwhen they run the suite. "JDK1.2", for instance, is one of the current tags -but the tag system was defined to be flexible and extensible.

The test suite basically consists of a whole set of test classes. These classes are Java source files which test a specific feature of Java. This can be a simple addition of two integers or writing to and reading from a file. The Mauve Project does not include a framework for automating test runs and generating statistics. Since these are two of the main requirements for an automated test suite, it has been developed at aicas using the Mauve test classes.

3.2.2 Tests for the RTSJ

The Mauve Project only covers the Standard Java APIs. In order to meet realtime requirements, the AERO-VM needs to support the RTSJ (see RTSJ). The classes related to this specification are in the package javax.realtime. The implementation of these classes are not provided by GNU Classpath, but are being developed at aicas. Own tests covering these realtime classes have been written as well. A list of all these additional classes can be found in Appendix A. They follow the way Mauve test classes are implemented.

That makes it easy to integrate new tests in the given test framework.

3.2.3 Jamaica/AERO-VM test framework

Given these circumstances, the following points can be stated:

- the Mauve Project provides a set of classes to be used for testing the Standard Java API.
- Own tests have to be written to cover features like RTSJ, which is not part of the Mauve Project.
- a test suite is needed to support automated testing and statistics.

The test suite was specified covering the following features:

- any number and set of classes or packages can be given as an input to the test suite and they will be tested one after another in a single test run.
- different types of tests, e.g. compiled or interpreted, can be applied to a test. This is important due to covering the features of AERO-VM itself.
- the test suite has to work autonomously performing the scheduled test runs without any user interaction.
- a test schedule of daily or weekly test runs can be defined. These test runs will then be started automatically.

- in case of any failure, the current test will be terminated and the remaining tests are performed.
- after every test run, a statistic about passes and fails of all tests has to be generated and sent to the developers in an email.

Additional tests

Besides the Mauve test classes own tests were written. The list shows the tests used for the javax.realtime package :

- HeapMemory
- HighResolutionTime
- ImmortalMemory
- LTMemory
- NoHeapRealtimeThread
- POSIXSignalHandler
- PeriodicTimer
- RealtimeThread
- ScopedMemory
- SizeEstimator
- Timed
- VTMemory

Conformance matrix: validation plan tests/ Aero tests

Validation Plan test	AERO tests	PASS	FAIL	Comment
JVM_API_01	gnu.testlet.java.*	5558	79	
JVM_API_02	gnu.testlet.java.*	5558	79	
JVM_API_03	aerojvm.validation.JVM_API_03.Double.Test	31		
JVM_API_03	aerojvm.validation.JVM_API_03.WeakHashMap.Test	29		
JVM_API_03	aerojvm.validation.JVM_API_03.clinit.Test	9		
JVM_API_03	aerojvm.validation.JVM_API_03.exceptions.Test	38		
JVM_API_03	aerojvm.validation.JVM_API_03.instof.Test	64		
JVM_API_03	aerojvm.validation.JVM_API_03.monitors.Test	27		
JVM_API_03	aerojvm.validation.JVM_API_03.ref.Test	16		
JVM_API_03	aerojvm.validation.JVM_API_03.reflect.Test	117		
JVM_API_03	aerojvm.validation.JVM_API_03.serialization.Test	9		
JVM_API_04	OBJA evaluation applications	1		
				9953 out of 10000 events executed. timing/timeout?
JVM_ASY_01	aerojvm.validation.JVM_ASY_01.Test	1	1	See annexe 8 for non covered requirements list
JVM_ASY_02	aerojvm.validation.JVM_ASY_02.Test	4		
JVM_ASY_03	aerojvm.validation.JVM_ASY_03.Test	37		
JVM_ASY_04	aerojvm.validation.JVM_ASY_04.Test	23		
JVM_ASY_05	aerojvm.validation.JVM_ASY_05.Test	10		
JVM_ASY_06	aerojvm.validation.JVM_ASY_06.Test	8		

Validation Plan test	AERO tests	PASS	FAIL	Comment
JVM_ASY_07	aerojvm.validation.JVM_ASY_06.Test	8		See annexe 8 for non covered requirements list
JVM_ASY_08	aerojvm.validation.JVM_ASY_06.Test	8		
JVM_ASY_09	aerojvm.validation.JVM_ASY_06.Test	8		
JVM_BRA_01	Gnu Gcov	1		
JVM_CAPA_01	aerojvm.validation.JVM_CAPA_01.Test	130		
JVM_CAPA_02	aerojvm.validation.JVM_CAPA_01.Test	130		
JVM_EXE_01	aerojvm.validation.JVM_EXE_01.Test	34	1	bug in RTSJ specification
JVM_EXE_02	aerojvm.validation.JVM_EXE_02.Test	20		
JVM_EXE_03	aerojvm.validation.JVM_EXE_03.Test	10		See annexe 8 for non covered requirements list
JVM_GEN_01	aerojvm.validation.JVM_GEN_01.Test	2		
JVM_GEN_02	aerojvm.validation.JVM_GEN_02.Test	4		
JVM_GEN_03	aerojvm.validation.JVM_GEN_03.Test	345		
JVM_GEN_04	gnu.testlet.java.*	5558	79	
JVM_GEN_05	Jacks tests	4175	197	“Fail” as standard JDK
JVM_GEN_06	SpecJVM98 test	8		
JVM_GEN_07	SpecJVM98 test	8		
JVM_GEN_08	Source code inspection	1		
JVM_GEN_09	Manual inspection	2		Work on Linux and ERC32
JVM_JNI_01	aerojvm.validation.JVM_JNI_02.Test	1		
JVM_JNI_02	aerojvm.validation.JVM_JNI_02.Test	1		

Validation Plan test	**AERO tests**	**PASS**	**FAIL**	**Comment**
JVM_MEM_01				Checked through rest of validation tests
JVM_MEM_02	aerojvm.validation.JVM_MEM_02.Test	4		
JVM_MEM_03	Code inspection	1		
JVM_MEM_04	aerojvm.validation.JVM_MEM_02.Test	4		
JVM_MEM_05	Code inspection	1		
JVM_MEM_06	Code inspection	1		
JVM_PERF_01	CafeineMarks	1		
JVM_SCH_01	aerojvm.validation.JVM_SCH_01.Test	129		
JVM_SCH_02	aerojvm.validation.JVM_SCH_02.Test	2		
JVM_SCH_03	aerojvm.validation.JVM_SCH_03.Test	41		
JVM_SCH_04	aerojvm.validation.JVM_SCH_04.Test	16		
JVM_SCH_05	aerojvm.validation.JVM_SCH_04.Test	16		
JVM_SCH_06	aerojvm.validation.JVM_SCH_04.Test	16		
JVM_SCH_07	aerojvm.validation.JVM_SCH_04.Test	16		
JVM_SCH_08	aerojvm.validation.JVM_SCH_04.Test	16		
JVM_SCH_09	aerojvm.validation.JVM_SCH_04.Test	16		
JVM_SYNC_01	aerojvm.validation.JVM_SYNC_01.Test	11		
JVM_THR_01	aerojvm.validation.JVM_THR_01.Test	1		
JVM_THR_02	aerojvm.validation.JVM_THR_02.Test	7		
JVM_THR_03	aerojvm.validation.JVM_THR_03.Test	142	54	requires realtime OS

Validation Plan test	**AERO tests**	**PASS**	**FAIL**	**Comment**
JVM_THR_04	aerojvm.validation.JVM_THR_04.Test	24		
JVM_THR_05	aerojvm.validation.JVM_THR_05.Test	9		
JVM_THR_06	aerojvm.validation.JVM_THR_06.Test	7		
JVM_TIM_01	aerojvm.validation.JVM_TIM_01.Test	91		
JVM_TIM_02	aerojvm.validation.JVM_TIM_01.Test	91		
EXEPROD_01	Manual test	1		
EXEPROD_02	Manual test	1		
SIM_01	Manual test	1		Simulator provided through Linux, Solaris and Windows releases
Total	**Total**	**1450**	**56**	

3.2.4 Test framework

Since the test suite is only used at aicas, it was designed to tightly fit in the given development environment. The version management system used at Aicas is cvs and will not being described, however, it is part of the test suite. Any given path correspondents to the directory structure used at aicas.

Overview

The test suite basically consists of two shell scripts which control the sequential flow of a whole test run. This includes checking out the current source files from cvs, compiling the given class(es), creating the wrapper around the test class to make it executable, building the test application with Jamaica as well as writing out log files and composing statistics.

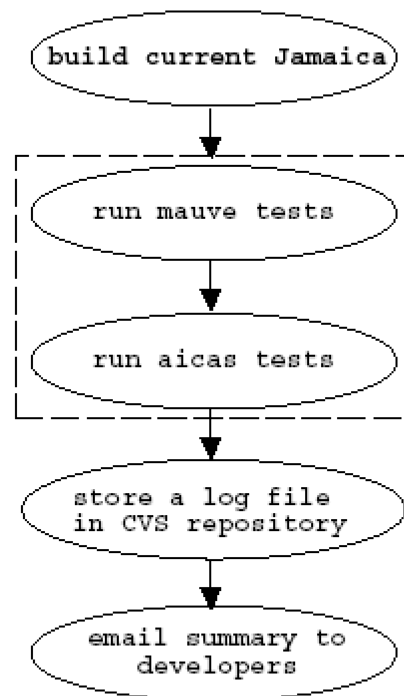
Test sequence

The fairly extensive task of a test run is split into two scripts in a way that the 'administration' of a test does not depend on the real test process and its result. That means, if a single test or even the AERO-VM crashes, the following tests still can be performed.

The sequence of a test:

- The test suite is checking whether the command line arguments is a test class (or several) or a package that has to be tested.
- In the next step the tested class file is being compiled. A wrapper, which will instance this class, is being built and compiled. Then the test is started. After finishing the single test, the next source file is taken, compiled and a new wrapper will start the next test. This procedure happens with all given source files. Firstly, all mauve tests and then all aicas tests are being performed.
- After every test of a single class the results are written to a log file. At the end of a test run these files are inspected to make the overall statistic. The total passes and fails of all tests are calculated and then can be compared to yesterdays statistic. At last an email with the statistic is sent to everyone involved.

The following figure illustrates the single steps needed:



Exception handling

To make sure a failure in a single test can not stop the whole test run, the wrapper starts the test in a new thread and terminates it if a timeout occurs. In case the AERO-VM has a failure and it would not be able to terminate the thread running the test and itself, then a functionality a level higher takes care of that. Before every single test, a new process is started which waits for the end of testing a single class. If this sleep-time ends before the test has finished, then the process belonging to the class-test will be killed. Therefore the sleep-time has to be longer than the time of the test which takes the longest.

Test types

The AERO-VM is able to perform interpreted Java byte code as well as compiled code. These two different execution modes have to be tested separately. The test suitedistinguishes between these two modes and runs either 'testinterpreted' or 'testcompiled'. Besides this major difference in how AERO-VM executes code, there is another main option that can be set to control the execution, it is the debugging mode. In this mode, several checks are being done during execution, e.g. reference type checking.

The following table gives an overview of the implemented test types and their options set.

Test type	**Command**	**Additionnal option(s)**
Interpreted	Jamaicavm_slim	None
Compiled	Jamaica	None
Partly compiled	Jamaica	-XcompileOnly 1
Interpreted Debug mode	Jamaicavmdb	None
Compiled Debug mode	Jamaica	-debug
Partly compiled, Debug mode	Jamaica	-debug –XcompiledOnly 1

The standard options are:

`/-heapSize 32M -smart -numDynamicTypes 300 -compile -numThreads 10/`

-debug/: turns on the debug mode

-XcompileOnly 1: only half of the methods are compiled. That forces the AERO-VM to switch between interpreted and compiled execution.

Test automation

In order to start tests periodically without any user interaction, a so called cron-job is started. In the according crontab-file all daily/weekly test runs are defined. A crontab-file is a list of commands where every row is a command, that will be performed automatically at the given time defined in that row.

3.3 Test results format

For every class that is being tested, a new directory-tree is created which consists of the class' full package name (package-to-directory- convention). In this directory all the log files are stored, written while performing different types of tests, e.g. testcompiled.out or testinterpreted.out.

The final result of a test run is presented as follows:

```
Log Message:
Fri Jan 10 23:25:03 CET 2003
      INT/DBUG: PASS: 5038 FAIL: 49 TIMEOUT: 3 HARDKILL: 0
Fri Jan 10 23:25:04 CET 2003
      CMPLD/DB: PASS: 5032 FAIL: 62 TIMEOUT: 1 HARDKILL: 0
Fri Jan 10 23:25:05 CET 2003
      aicas INT/DBUG: PASS: 86046 FAIL: 26 TIMEOUT: 3 HARDKILL: 0
Fri Jan 10 23:25:05 CET 2003
      aicas CMPLD/DB: PASS: 66586 FAIL: 30 TIMEOUT: 1 HARDKILL: 0
```

Description of the four possible test results:

HARDKILLHARDKILL

PASS: n tests completed successfully

FAIL: n tests failed

TIMEOUT: n tests timed out and were terminated by the Virtual Machine

HARDKILL: n tests timed out and could not be terminated by the Virtual Machine but by the test suite itself.

The generated statistic over a whole package looks as follows:

```
Fri Jan 10 23:25:03 CET 2003
gnu.testlet.java.lang.ref: PASS: 6 FAIL: 2 TIMEOUT: 0 HARDKILL: 0
```

The messages are then mailed to all developers through the cvs commit command.

Test strategy

The AERO-VM and the Jamaica Builder provide several options to control their execution. The number of all different combinations of these options is very large and makes it impossible to test them all through. Therefore a number of standard options have been chosen. The following table shows the types of tests run at what frequency. The tests are started by a cronjob.

Test Type	Frequency
interpreted, debug mode	daily (weekdays)
compiled, debug mode	daily (weekdays)
partly compiled, debug mode	daily (weekdays)
interpreted	weekly (weekend)
compiled	weekly (weekend)
partly compiled	weekly (weekend)

3.4 Test results

The set of APIs supported by AERO-VM is compliant of specified one.

The full list of supported and validated APIs is provided in annexe

4. Compiler test, Cyclomatic Complexity & code coverage

4.1 Jacks Test

Jacks is a free test suite designed to detect bugs in a Java compiler. The editor (IBM Corp.) guarantee that it will find at least one bug in any supported Java compiler, in fact it may find more than one. Jacks is not designed to test a Java runtime (JVM) or Java class libraries. The Mauve project already aims to do that.

Jacks includes a collection of Java compiler tests contributed by people on the net. Jacks is free software licensed under the terms of the GPL. A test case must be licensed under the terms of the GPL to be suitable for inclusion into Jacks, but authors retain the original copyright. The license used by the Jacks test suite does not affect ability to run the suite with any free or non-free Java compiler

The most interesting feature of Jacks is the ability to generate regression reports. These reports show how the current test results have changed with respect to the previous results. This is very useful since to run the tests and check the results to see if any changes made cause regressions. This really makes a big difference when there are a large number of tests, since otherwise it could not be possible to know if a failing test was failing before or is now failing because of changes.

Jamaica with javac:

Total 4617 Passed 3985 Skipped 264 Failed 368

JDK:

Total 4617 Passed 4079 Skipped 144 Failed 394

jikes:

Total 4617 Passed 4269 Skipped 125 Failed 223

jamaica with jikes:

Total 4617 Passed 4175 Skipped 245 Failed 197

This test has a number of failures and skipped cases with JDK and IBMs jikes. The results are similar with AERO-VM, but we could not go into detail of where we see differences.

4.2 Cyclomatic complexity

JavaNCSS was used to compute cyclomatic complexity, it is a simple command line utility that measures two standard source code metrics for the Java programming language. The metrics are collected globally, for each class and/or for each function.

Features and Metrics JavaNCSS Provides:

- Metrics can be applied to global-, class-, or function-level.
- Non-Commenting Source Statements (NCSS).
- Cyclomatic Complexity Number (McCabe metric).
- Packages, classes, functions and inner classes are counted.

Average values are calculated

Result summary is:

average Cyclomatic complexity: 2.81

maximum Cyclomatic complexity: 303

In total, there are 1446 functions in 138 classes.

The average Cyclomatic complexity is acceptable, maximum is encountered in specific case, non representative of onboard space applications (high network exchanges). Space domain code standard specify a maximum average complexity of 8.

4.3 Code Coverage

The coverage of the C code of the AERO/JVM was analysed using the open-source tool gcov:

http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html

Coverage analysis was performed only on interpreted code, built applications could not be run directly using gcov. The resulting coverage hence does not cover code that is required for

- loading code from ROM
- executing compiled code
- the Java Native Interface JNI
- the analysis of the memory demand using the builder option –analyse
- a fixed garbage collector work using the builder option -constGCwork

Also, the detailed coverage was not made for C code that belongs to the native part of the class libraries from the GNU classpath project, but the summary of the coverage for these files is given.

Files	***Coverage***
classe.c	84,75%
interpreter.c	80,21%
jamaica_vm.c	49,80%
jamaica_threads.c	77,80%
jamaica_gc.c	74,07%
jamaica_debug.c	0,00%
jamaica_generic_file.c	0,00%
jamaica_generic_io.c	50,00%
jamaica_generic_thread.c	82,76%
jamaica_lang_Debug.c	0,00%
jamaica_lang_Process.c	80,00%
jamaica_target.c	33,33%
jamaica_utf8.c	57,30%
jni.c	30,66%
nativecode.c	31,78%

stubs.c	35,77%
jcl.c	25,35%
java_lang_Class.c	92,39%
java_lang_Classloader.c	25,42%
java_lang_Double.c	100,00%
java_lang_Float.c	100,00%
java_lang_Math.c	90,91%
java_lang_Object.c	87,50%
java_lang_String.c	93,53%
java_lang_StringBuffer.c	100,00%
java_lang_system.c	71,80%
java_lang_thread.c	77,36%
java_lang_Throwable.c	91,14%
java_lang_ref_JamaicaReferences.c	100,00%
java_lang_ref_References.c	100,00%
java_lang_reflect_AccessibleObject.c	80,33%
javax_realtime_MemoryArea.c	86,24%
javax_realtime_POSIXSignalHandler.c	41,94%
javax_realtime_PriorityScheduler.c	100,00%
javax_realtime_RawMemoryAccess.c	0,00%
javax_realtime_RawMemoryFloatAccess.c	0,00%
javax_realtime_RealtimeClock	93,10%
java_lang_Runtime.c	23,36%
Average coverage	61,80%

The average coverage do not give the real coverage result, in fact some part of files are not used in AERO-VM, and some others represent option or native implementation. Annexes give the details of coverage per files, with each methods and explanation concerning the none coverage of methods.

After analysing the justification of non covered functions the coverage of AERO-VM is about :

<u>84,73%</u>

For a beta product, this level of coverage is quite acceptable, industrialisation phase will require to gets 100%. Task complexity will depends of supported (required) functions, some current uncovered functions are not strictly required for onboard space application.

5. Evaluation results

5.1 Functional evaluation summary

Functionnal evaluation does not give direct measurement results, tests show a comportement of the code that have to be complient with expected one.

OBJA

Evaluation start by running the OBJA Manager, that wait sometimes before loading several OBJA. This OBJA runs in parallel and interact between them. 64 OBJA was runned in the same time, to check capacity of management and loading of the JVM. OBJA applications themselves are not representatives of space systems, but functional interaction (start, stop, suspend, load, unload) are the same. Messages exchange have been tested successfully. Some time was spend to have correct ClassLoader, even if it is not the space one but a customized one for the ERC32 bench.

Real-time comportment

Simple measurement was made using Tornado tools, basic tests applications show that the real-time comportment is garanteed at first level. Real time thread synchronization works fine, and real-time delays are effectively measured.

Industrialisation phase of the project will have to investigate more precisely on this aspects, and characterize requirement of space applications.

1553 Bus interface

The close loop between 1553 card works fine, the test application check that sending byte value are the same received on the other channel. Driver is a C application, that is linked using a specific header generated by AERO-VM tools (jamaicah). The specific syntaxe of JNI require some work to understand the correspondance between C type and Java one.

Data debit rate is limited by the JNI interface itself, probably optimization are possible using variable buffer size, it was not the purpose of the application. Tests gives an average data rate of: **5 Kbps**

Garbage Collector

Static garbage collector comportment was verified by restreint available memory on the ERC32 board, and checking the functional comportment of the application is strictly the same. Then, EmbeddedcaffeineMark tests check the Garbage Collector comportment, and some validation tests (see Annexe 7.2) verify it. All tests show the correctness of the static Garbage collector of AERO-VM.

At the current state, the static GC has only been validated by verifying the correctness of the algorithms. The static GC is not enabled in the compiler yet, but if everything goes according to plans, a working prototype should be available in AERO-VM/Jamaica soon.

5.2 Performance evaluation

5.2.1 SpecJVM98

SPECjvm98 is a benchmark suite that measures computer system performance for Java virtual machine (JVM) client platforms.

The SPECjvm98 benchmark suite contains eight different tests, five of which are either real applications or derived from real applications that are commercially available. The tests measure the time it takes to load the program, verify the class files, compile on the fly if a JIT compiler is used, and execute the test. Each test is run several times and two scores are generated: a "worst" score for the slowest time and a "best" score for the fastest. A geometric mean is used to compute a composite score for all tests. Test scores are normalized against a reference machine - a midrange IBM PowerPC 604 with a 133-MHz processor. Higher scores indicate better performance.

SPECjvm98 takes advantage of Java's byte-code format to provide tests based on a variety of applications from independent software vendors (ISVs). Byte-codes allow ISVs to contribute to SPEC benchmarks without releasing the secrets of their proprietary source code.

5.2.2 CaffeineMark

The original CaffeineMark benchmark consisted of four tests: a prime number sieve, a tight integer loop, an image blasting test and a BitBilting test. The tests gave a fairly accurate measure of java performance. CaffeineMark 2.01 have attempted to address some of the shortcomings of the 1.0 version by incorporating nine tests instead of four and by changing the formula for the overall CaffeineMark. Two tests were also added to measure Allocation/Garbage Collection and JIT compiler speed.

Finally, version 2.5 fixed some problems and was unable to test embedded systems, e.g., systems intended for use in consumer electronics.

The CaffeineMark theorize about the types of optimisations that are being performed by the Virtual Machine. In AERO-VM case the embbeded tests was runned.

5.2.3 Specifics tests

Several tests have been ported in Java, they're runned under Linux on standard PC, and on ERC32 board. This components have been rewritten using strictly same algorithm.

Performance measurments are made using VxWorks instruction : timexN. This instruction made many measure to find a convergent measure, independently of other running applications.

5.2.4 Results

CaffeineMark

	ERC32 14Mhz, 12 MB VxWorks	PII 350Mhz, 128 MB Linux Mandrake
EmbeddedCaffeineMark	168 (overall score)	3791 (overall score)

SpecJVM98

	Overall Score	Reference score
200_check	1.93 sec	N/A
213_javac	7 min 39 sec	7 min 08 sec
222_mpegaudio	28 min 15	18 min 33
202_jess	724 ms	N/A
201_compress	33 min 13 sec	19 min 58 sec
209_db	12 min 5 sec	8 min 41 sec
228_jack	3 min 6 sec	7 min 58 sec
999_checkit	15 min 2 sec	N/A

Remark: these tests require more than 20 MB to run, it is not possible to run them on ERC32 board. Results are turn up to same system frequency. They're runned in interpreted mode (SpecJVM98 base mechanism) on Linux – Reference = PowerPC 604 133 Mhz

Standard Aicas test

	ERC32 14Mhz, 12 MB VxWorks	PII 350Mhz, 128 MB Linux Mandrake
JVMTest 1	5.39 sec	2.71 sec
JVMTest 2	28.6 sec	12 sec
JVMTest 3	5.8 sec	2.93 sec
Net	1.75 sec	0.66 sec
JNI	7.2 sec	1.51 sec

Compiled mode, this tests includes computations, displays, types conversions, network tests, JNI call etc.

Specific Astrium tests

Remark : JDK and JIT running process include a long start phase of loading and verification of bytecode. Thanks to elaboration process of AERO-VM, there is no latency time during the application start.

	ERC32 14Mhz, 12 MB VxWorks		**LEON (1)**	**PII 350Mhz, 128 MB Linux Mandrake 8.2**		
Test	***AERO-VM Interpreter***	***AERO-VM Native***	***AERO-VM Native***	***AERO-VM Interpreter***	***AERO-VM Native***	***JDK 1.4.1 JIT compiler***
Attitude	70.8 sec	6.9 sec	N/A	4.44 sec	0.71 sec	1.3 sec
BarySST	496 sec	14 sec	17.4 sec	42.18 sec	1.62 sec	11 sec
BigPi	3.2 sec	2.8 sec	N/A	310 ms	270 ms	920 ms
FFT	1h40	3.6 min	2.9 min	7.47 min	9.88 sec	9.57 sec
Fibonacci	20 min	17.9 sec	13.25 sec	2.4 min	4.02 sec	1.65 sec
Life	8.15 sec	3.3 sec	2.6 sec	950 ms	880 ms	1 sec
Math_Matrix	21 min	48 sec	50 sec	6.55 sec	4,12 sec	1.62 sec
SST	3.18 sec	2.81 sec	N/A	270 ms	230 ms	960 ms
Thread	4.25 sec	3.13 sec	N/A	280 ms	250 ms	1 sec
APS WinSort	4.23 sec	3 sec	N/A	340 ms	280 ms	1 sec
Attitude Control Syst.	110 sec	28.15 sec	N/A	84.8 sec	2.38 sec	2.15 sec

(1) LEON: FPGA prototype LEON board, 20MHz, 12MB VxWorks, compiled Sparc v7 (specific Sparc v8 optimization not included)

Overall ratio ↱	Interpreted ERC32	Compiled ERC32	Interpreted Linux	Compiled Linux	JDK Linux(1)
Interpreted ERC32		0.017	0.066	0.002	0.0032
Compiled ERC32	59.2		1.95	0.12	0.19
Interpreted Linux	15.0	0.51		0.035	0.048
Compiled Linux	47.64	8.22	28.13		1.34
JDK Linux	313	5.29	20.8	0.74	

(1) Linux Red Hat running standard JDK 1.4.1 on Pentium II 350 Mhz, 128MB Ram

Overall ratio have to be corrected depending of the application type, on ERC32 the difference between compiled and interpreted mode are :

	AERO-VM Interpreted Mode / JDK Linux(1)	AERO-VM Compiled Mode / JDK Linux(1)
Functionnal applications	6.41	4.26
Simple mathematical applications	70	3.45
Complex mathematical applications	390	9.45

(1) Linux Mandrake 8.2 running standard JDK 1.4.1 on Pentium II 350 Mhz, 128MB Ram

5.3 Performances trade off (OBCP/Astrium ERC32-VM/AERO-VM/Native C)

Interpreted systems provide great flexibility to develop high-level function, with simplified system update from the ground; when interpreted, Java gives a complete solution with robustness and fault tolerance. Many ideas give to think that interpreted Java is slower that native or other interpreted systems. Even if it is interpreted Java use specific mechanism to gives good performance, but the loading phase of application and their start is a bite longer due to preverification that Java makes to ensure security of execution. Some performance tests have been made to evaluate interpreted Java performance compared with previous technology (onboard interpreter) and with native execution.

OBCP are intrepreted procedure used on :

- Astrium E3000 telecom satellite familly
- Rosetta spacecraft

Astrium ERC32-VM is the first JVM developed on ERC32 processor, based on KVM CLDC

5.3.1 Bytecodes

OBCP bytecode (called APIC) is a set of 60 instructions. This set include sometime the same instructions for each type of data.

Java bytecode is designed to answer a lot of requirement that APL does not have to comply; the bytecode consists in a set of 200 instructions. Inside the bytecode instructions set, some instructions have higher speed; this restricted set (80 instructions) involves the most used instructions. The Java bytecode includes complex instructions, with up to 7 operands; a single instruction provides capability to create a multi-dimension array for example.

Bytecode usage

Some experimentation made on ten applications gives *a ratio of x4 less instructions less used with the Java bytecode* for the same application (identical source code) compare with the OBCP bytecode.

Source codes are similar, the produced bytecode is very different (without variable/constant definition). Experimentation made at Astrium show a typical ratio of 0.25.

OBCP use x4 more bytecode for the same application than Java bytecode.

"Compiled " applications sizes

Size of the resulting application when compiled is not in direct proportion of bytecode usage, due to the fact that the Java bytecode use a predefined structure that includes static definition and optimizations with shortcuts inside the bytecode himself.

Compiled files size comparison show that *Java bytecode files have 40% of the size of APL bytecode files*, except for very small application, where the java static header is more important than the bytecode …

Execution of one bytecode instruction

Java bytecode instruction are more complex than APL bytecode, it's also interesting to evaluate the required time to execute a single bytecode instruction with each interpreter in a same context.

Thanks to complex evaluations for ATV project and internal purposes, it is possible to compare the execution of a single bytecode instruction between the interpreters :

Average OBCP interpreter execution times are:

Interpreter Compilation	Processor	Average time to execute one
OBCP IPE interpreter ADA no-check	i1750 1 Mhz	250 µs / instr.

Previous trade-off have shown that:

From	To	Ratio
ADA no-check i1750	ADA no-check ERC32 SPARC 14Mhz	3.8
ADA no-check ERC32 SPARC 14Mhz	C –o2 opt. ERC 32 SPARC 14Mhz	1.2

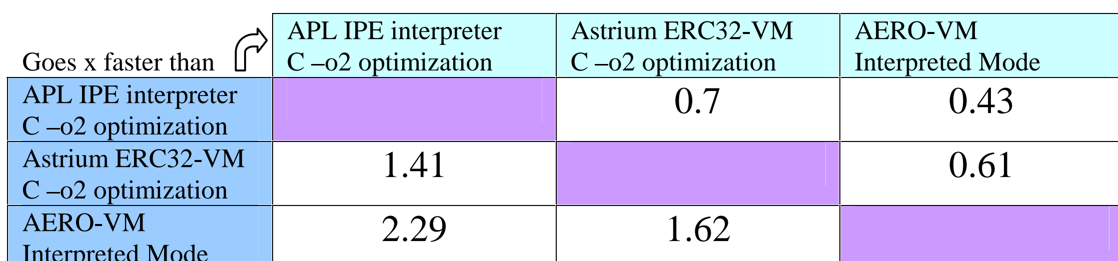
Extrapolated OBCP interpreter on ERC32 should be:

Interpreter Compilation	Processor	Average time to execute one
APL IPE interpreter C –o2 optimization	ERC32 SPARC 14Mhz	55 µs / instr.

Average Java interpreter execution times are:

Interpreter Compilation	Processor	Average time to execute one
Astrium ERC32-VM C –o2 optimization	ERC32 SPARC 14Mhz	39 µs / instr.
AERO-VM Interpreted Mode	ERC32 SPARC 14Mhz	24 µs / instr.

Results: ratio between interpreter on ERC32 SPARC 14Mhz on bytecode execution time



Goes x faster than	APL IPE interpreter C –o2 optimization	Astrium ERC32-VM C –o2 optimization	AERO-VM Interpreted Mode
APL IPE interpreter C –o2 optimization		0.7	0.43
Astrium ERC32-VM C –o2 optimization	1.41		0.61
AERO-VM Interpreted Mode	2.29	1.62	

5.3.2 Performances of execution

Plate-form

Performances tests was made using :

- the ERC32-VM on ERC32 14 Mhz, 12 Mb board
- the AERO-VM interpreted mode on ERC32 14 Mhz, 12 Mb board
- the AERO-VM compiled mode on ERC32 14 Mhz, 12 Mb board
- the GNU C compiler for native C version of tests on ERC32 14 Mhz, 12 Mb board

Process

Each test was run 5 times, lowest and highest execution times are rejected, the final time is the average execution time of the third values measured.

Source code are similar for each language, it could be noted that the Java version use, like other version, array for manipulating data structure, even if the object technology gives better performances in Java.

The C code, when compiled, is directly loaded in memory and executed. In the case of interpreter, the interpreter is loaded in memory, then the application to execute and associated libraries (APIs for Java). The execution time is corrected from the required delay to load the application, before starting execution. This correction was made to compare pure execution performance.

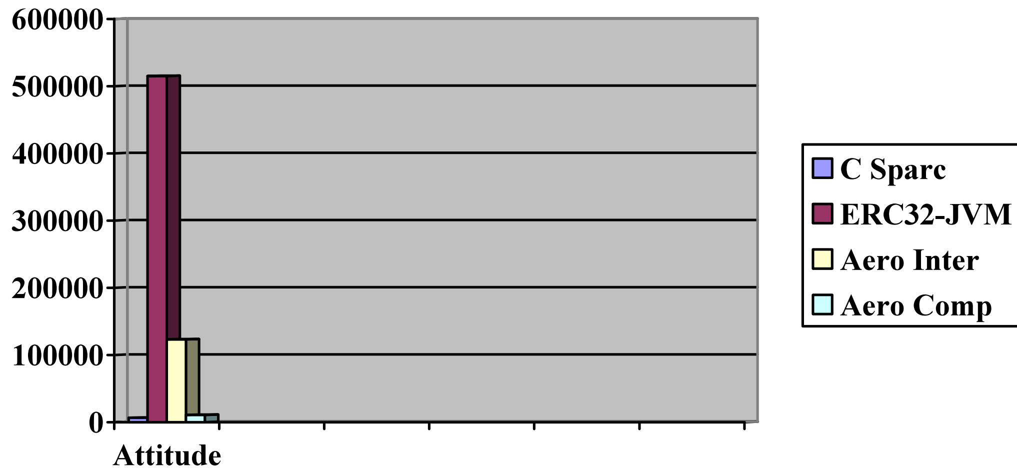
C Sparc: it is the native version of the application, compiled with maximum optimization

ERC32-VM: it is the java version of the application run under the ERC32-VM

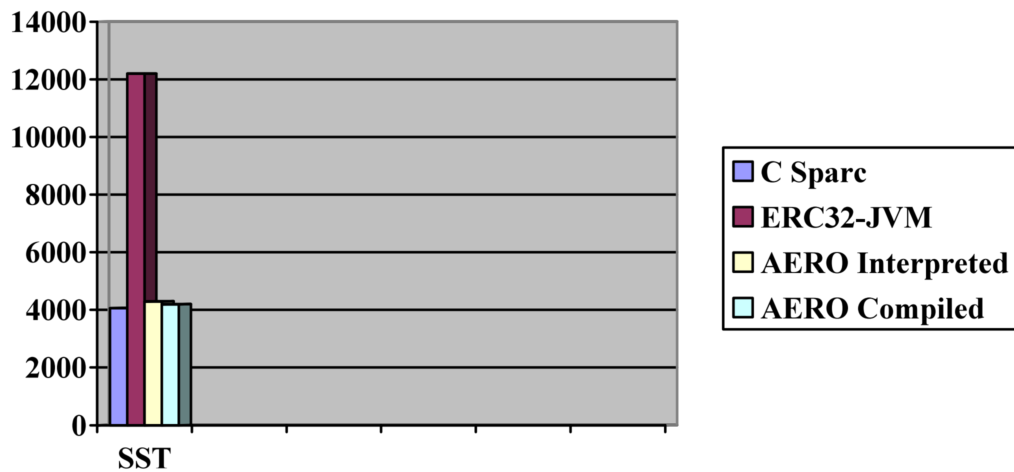
AERO Interp: it is the java version of the application run under the AERO-VM in interpreted mode

AERO Comp: it is the java version of the application run under the AERO-VM in compiled mode

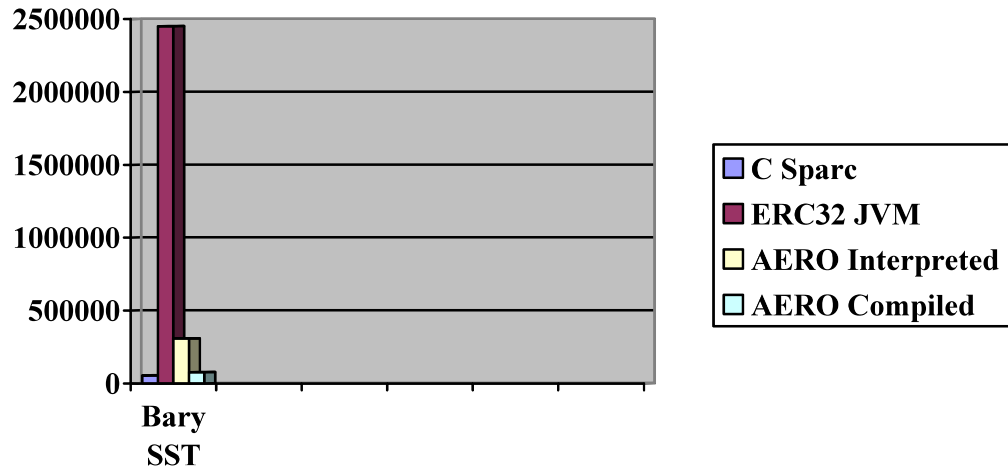
Time are in milliseconds



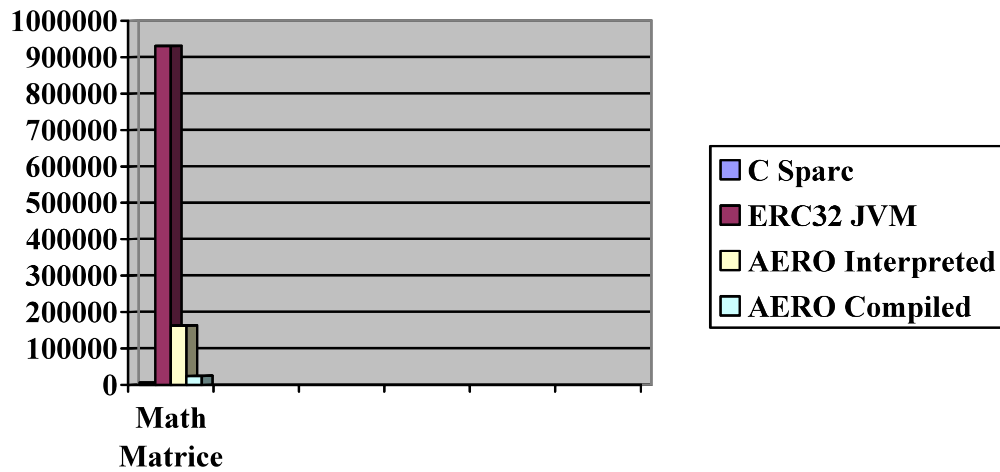
The C native version runs x80 times faster than ERC32-VM, and x20 to x1.6 times faster than AERO-VM. This is due to the fact that this application includes only calculation, with array access. Java is not optimised for pure calculation, for array access, and it seems it could be a bite faster using objects instead array.



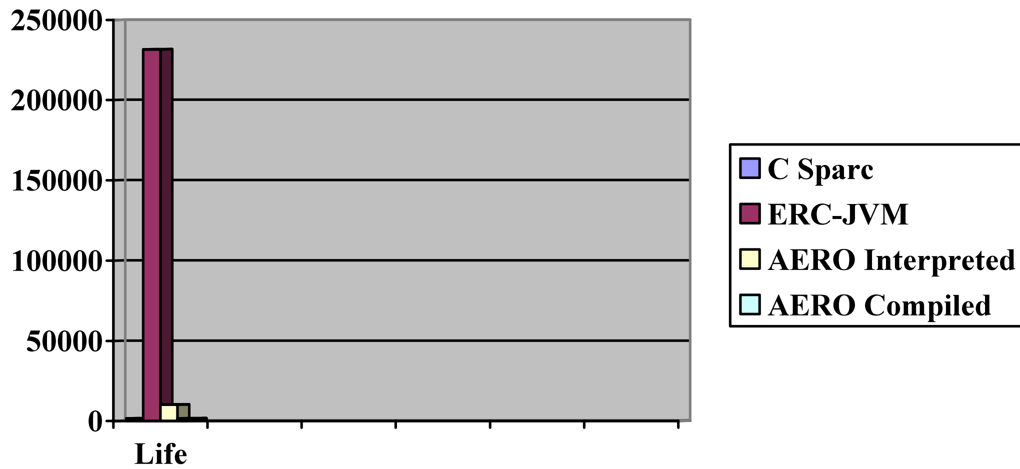
C version goes x3 times faster than basic ERC32-VM interpreter, and at same speed than AERO-VM. In case of functional application, with small or simple calculation the AERO-VM runs very fast compared to native C version.



C version goes x30 times faster than base ERC32-VM interpreter, and x3 faster than interpreter version of the AERO-VM, and approximatively same speed that compiled release of AERO-VM. This application include functional and calculation methods with double.



C version goes x50 times faster than base ERC32-VM interpreter, and x5 faster than AERO-VM interpreted release, and x2 faster than AERO-VM compiled This application integrates more calculation than previous one and the same number of functional methods.



C version goes x30 times faster than base ERC32-VM interpreter, and x3 faster than AERO-VM interpreted release, and same speed than AERO-VM compiled This application integrates simple calculation; it's recursive, but use a great number of functional methods with many memory accesses.

Conclusion Java interpreter vs native C

The execution mode used in ERC32-VM is based on an interpreter, who's wrote himself in C; the interpreter could not provide better performance than the language on which it's based.

AERO-VM release works differently, interpreted release include many native parts (APIs, Garbage collector etc.), and provide good performances, compiled release include all in native form, spend times compared with C, is taken by the security and secure mechanisms that ensure robustness of execution.

On applications with similar source code (same functions and architecture used), it seems that :

Application type	C	ERC32-VM	AERO-VM Interpreted	AERO-VM Compiled
Full calculation	1	x50 to x80	x10 to x20	x1.5 to x2.5
Calcul & functional	1	x8 to x20	x3 to x5	x1.25 to x1.5
Functional	1	x3	x1.5	x1.05

6. Summary of supported and validated API

The following Java packages are supported and tested

Java Packages:

java.lang
java.lang.ref
java.lang.relect
java.io
java.math
java.net
java.security
java.text
java.util

RTSJ packages:

javax.realtime

The specification is the Java API specification [Java_API1.2] and the Real-time Specification for Java [RTSJ].

[RTSJ] The Real-Time Specification for Java by Gregory Bollella
(Editor), James Gosling, Benjamin Brosgol, James Gosling, Peter Dibble,
Steve Furr, Mark Turnbull, 2001

[Java_API1.2] The Java 2 Platform Standard Edition 1.2.2 API
Secification, Sun Microsystems, Inc.

7. 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 modelling and object development) and a better reusability of the existing components leading to an important reduction of the software cost and development time.

However the Virtual Machines available on the market are not compatible with the soft or hard real-time constraints of the space systems. To solve this problem and embed a Java Virtual Machine in these systems, the AERO-VM was developed. This Virtual Machine is able to replace the existing interpreters and permit to decrease the development and validation cost of typical Interpreted Procedures with a same or higher level of performance and safety. This product should be reusable by all the missions including an interpreter.

Validation gives a good knowledge in embedded capacities of the AERO-VM, and evaluation show that performances are very good. The level of validation allow to qualify the current AERO-VM release as a **Beta product**, candidate to be industrialized with an objective of use in operational space system.

8. ANNEXES

CHAPTER REMOVED
FROM THE AERO PROJECT
ORIGINAL TN4 DOCUMENT