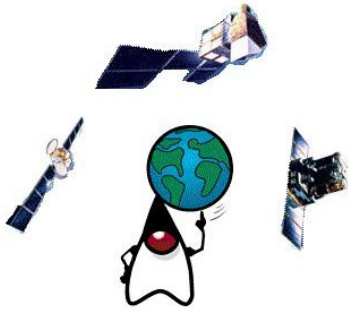
# AERO: Architecture for Enhanced Reprogrammability and Operability

Contract ESTEC 15750/02/NL/LVH



## Technical Note 1

## (Extract of original document)

Fridtjof Siebert, Aicas

Tobias Ritzau, Linköping Universitet
Frederic Deladerriere, Astrium

Reference: AERO/TN1

Issue: 0.2

Date: **2002-05-28**

**Abstract**:

This document gives an analysis of the application of Software JVMs for real-time and safety critical space applications. JVMs are evaluated by their real-time and safety-critical features, by their quality, their performance, the Java language and API they support and the expected complexity of the development.


| **Written by**: | **Name** | **Company** | **Signature** | **Internal reference** |
|---|---|---|---|---|
| | Fridtjof Siebert | aicas | | |


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
Haid-und-Neu-Str. 18
D-76131 Karlsruhe, Germany
Tel: +49 721 663 968-23 Fax: +49 721 663 968-93
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 | 2002-04-11 | | First issue |
| 0.2 | 2002-05-28 | All, new Appendix B | Additions as requested on AERO Meeting in Toulouse, 25.4.2002 |

# Table of Contents

# 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 machines to choose a potential candidate that will be customised, 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 output of task 1.a.1 "Software JVM Analysis".

It gives an analysis of the application of Software JVMs for real-time and safety critical space applications. The document presents technologies used in Java implementations and their implications for the applicability in space context. Criteria for selection of a virtual machine are defined. The main criteria are real-time capabilities, safety-critical features, quality, performance, the level of support for the Java language and APIs and the expected complexity of the development within the AERO project. Two levels of criteria are defined: destructive criteria and selective criteria.

25 Java implementations are then presented and analysed for conformance with the destructive criteria. Those implementations that conform to these criteria are then evaluated in more detail according to the selective criteria giving a preference to the best basis of future development.

## 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. |
| [BKMS98] | David F. Bacon, Ravi Konuru, Chet Murthy, Mauricio Serrano: Thin Locks: Featherweight Synchronization for Java, PLDI, 1998 |
| [DS84] | L. Peter Deutsch and Allan M. Schiff-man: Efficient Implementation of the Smalltalk-80 System, Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pp. 297-302, Salt Lake City, UT, January, 1984 |
| [Peren] | http://www.peren.com |
| [PH] | http://www.plumhall.com |
| [Jacks] | http://www-124.ibm.com/developerworks/oss/cvs/jikes/~checkout~/jacks/ |
| [Mauve] | http://sources.redhat.com/mauve/ |
| [Strou87] | Bjarne Stroustrup: Multiple Inheritance for C++, Proceedings of the European Unix Users Group Conference, pp. 189-207, Helsinki, May, 1987 |
| [SW01] | Fridtjof Siebert and Andy Walter: Deterministic Execution of Java's |

Primitive Bytecode Operations, Java Virtual Machine Research and Tecnology Symposium (JVM'01), Monterey, California, April 2001.

[Yang99] Byung-Sun Yang, Junpyo Lee, Jinpyo Park, Soo-Mook Moon, Kemal Ebcioglu, Erik Altmann: Leightweight Moni-tor for Java VM, ACMComputer Ar-chitecture News, Vol 27/1, March 1999.

## 1.3 Definition of Terms and Acronyms

### 1.3.1 Definition of Terms

None

### 1.3.2 Acronyms and Abbreviations

| | |
|---|---|
| ESA | European Space Agency |
| ESTEC | European Space Technological Centre |
| AERO | Architecture for Enhanced Reprogramability and Operability |
| AOT | Ahead-of-Time (compiler) |
| API | Application Program Interface |
| CDC | Connected Device Configuration |
| CLDC | Connected Limited Device Configuration |
| J2ME | Java-2 Micro Edition |
| JDK | Java Development Kit |
| JIT | Just-in-Time (compiler) |
| JNI | Java Native Interface |
| JVM | Java Virtual Machine |
| RTSJ | Real-Time Specification for Java |

# 2. Evaluation Method

To evaluate existing Java virtual machine implementations for their applicability in the space applications, we use the ADEQUA method. For this, we first define and detail criteria and then evaluate to what extend each candidate conforms to these criteria.

## 2.1 Criteria definitions

There are two levels of criteria: destructive one and selective ones. Destructive criteria lead to the exclusion of the respective candidate from further evaluation, while selective criteria have a weight that influenced to what extend the criteria influences the total result.

The criteria are grouped in six main criteria groups. These main groups themselves consist of a list of sub-criteria detailing the main criteria.

### 2.1.1 Destructive criteria

These are criteria that are prohibitive for the application of the respective candidate for space purposes and hence exclude the candidate from further evaluation.

### 2.1.2 Selective criteria

For each selective criteria, a weight is defined that represents the importance of this criteria. The weight is given in percent such that all selective criteria together have a total weight of 100%. This holds for the main criteria group as well as for any group of nested sub-criteria.

## 2.2 Evaluation procedure

In a first round of evaluation, the destructive criteria are used to exclude candidates that are not applicable for the space domain. This permits a more thorough analysis of the remaining candidates that will be performed next.

In the second round of evaluation, the conformance of each candidate to each selective criteria is determined. A value in the range of 0 (does not conform to the criteria) to 10 (fully conforms to the criteria) is used.

The weights and criteria then permit to calculate a total score for each candidate that summarises the level of conformance to the presented criteria taking into account the importance (weight) given to each criteria.

# 3. Real-time Java Technology overview

The Java programming language originated as part of a research project to develop advanced software for a wide variety of network devices and embedded systems. The goal was to develop a small, reliable, portable, distributed, real-time operating platform. The result is a language that has proven to be ideal for developing secure, distributed, network-based end-user applications in environments ranging from network-embedded devices to the World-Wide Web and the desktop.

The Java programming language hence provides an good environment for efficient development of high quality software that is executed in a safe environment. These features are also desirable in other critical domains with additional requirements such as real-time behaviour. Nevertheless, a number of technical challenges need to be solved to apply Java in these contexts. These challenges and their solutions are presented in this chapter.

## 3.1 Real-time API extensions

The original standard classes for Java do not provide features that are required for the development of real-time systems. These features include mechanisms to control scheduling of tasks and predictable execution in a systems that uses garbage collection. Several suggestions for extensions of the Java APIs to support real-time systems have been made to fill this gap.

The non-proprietary real-time extensions to Java are the Real-time Specification for Java that was defined within SUN Java community process, and the rtCore and RTDA specifications defined by the J-Consortium.

Since real-time programming capabilities cannot be added by simply adding additional libraries, these standards also provide stricter semantics for the Java implementation. The stricter semantics affect the handling of threads and priorities and the semantics of monitors. To permit use of the Java heap within code that cannot be interrupted by the garbage collector, the semantics of assignment operations for references also had to be modified to ensure consistency of the system.

Real-time applications typically occur in embedded devices that require direct access to the hardware (such as memory mapped I/O) and handling of asynchronous events. For these reasons, the real-time extensions also define means to access hardware directly from within Java code and to handle asynchronous events.

### 3.1.1 Real-time Specification for Java

The Real-Time for Java Expert Group (RTJEG) produced the Real-Time Specification for Java (RTSJ) as an extension to the Java Language Specification and the Java Virtual Machine Specification. The RTSJ provides is supposed to provide an Application Programming Interface that will enable the creation, verification, analysis, execution and management of Java thread whose correctness conditions include timeliness constraints, so called real-time threads.

The Real-time Specification for Java was defined not to be restricted to the use by particular Java environments and to be backward compatible with existing Java programs. It was defined to support the "Write Once, Run Anywhere" (WORA) paradigm while recognising the difficulty this poses for real-time systems. It addresses current real-time system practices but shall allow for future inclusion of advanced features. Predictable execution had first priority in the guiding principles, while it allows for variations in the implementations decisions such as algorithms employed or trade-offs that are to be made. The RTSJ does not define any syntactic extensions to the Java language, so standard development tools for Java can be used even for development of code using the RTSJ.

#### 3.1.1.2 Threads and Scheduling

##### 3.1.1.2.1 New thread types

The RTSJ defines new classes RealtimeThread and NoHeapRealtimeThread that have stricter semantics than normal Java threads. These threads have at least 28 different priority levels that need to be distinguished by the implementation. Instances of the RealtimeThread class may have an execution eligibility logically lower than the garbage collector, while instances of the NoHeapRealtimeThread have an execution eligibility logically higher than any garbage collector.

NoHeapRealtimeThreads may not allocate or reference any object on the garbage collected heap nor manipulate any references to objects on the heap.

Even though instances of NoHeapRealtimeThread cannot be interrupted by the garbage collector, synchronising on a monitor that may be held by a RealtimeThread or a normal Java thread that can interrupted by garbage collection may cause the NoHeapRealtimeThread to wait for garbage collection.

##### 3.1.1.2.2 Scheduling

The RTSJ provides classes to control the scheduling of RealtimeThreads and asynchronous events. Priority scheduling is supported. Release of tasks may be period, aperiodic or sporadic, while detailed information on the start, period, cost and deadline.

Feasibility analysis of a schedule is optional, such as detection of overruns and missed deadlines.

#### 3.1.1.3 Memory Management

Since NoHeapRealtimeThreads are not permitted to work with references to objects allocated on the heap, special memory areas classes that can be used with these threads are provided by the RTSJ. These memory areas are scoped and may be nested. Objects allocated within scoped memory are not freed by the garbage collector. Instead, their memory is reclaimed as soon as there is no thread left that uses the scope.

For consistency of the overall system, special assignment rules need to be enforced by the implementation. These rules make it impossible to assign references to an object in scoped memory to either heap memory or a surrounding scope. A special memory can be used for immortal memory that is never garbage collected and than can be used by all threads.

In addition to the scoped and immortal memory areas, the RTSJ defines special memory areas for physical memory. These memory areas can be used to access memory directly, e.g. to control memory-mapped I/O or use DMA memory.

#### 3.1.1.4 Synchronisation

The RTSJ requires the priority inheritance protocol to be used for all Java monitors by default. In addition to this, the priority ceiling protocol can be used as an option.

#### 3.1.1.5 Others

##### 3.1.1.5.1 Timers

Real-time timer classes provide high resolution clocks and timers that use time values with nanosecond precision. Timers can be used to trigger single or periodic events.

#### 3.1.1.5.2 Asynchronous Events

In addition to threads, the RTSJ provides asynchronous events as a means to execute code. These events are lightweight in the way that many (thousands or tens of thousands) potential events might be used in a single real-time system.

Events are triggered either by application logic or by an external that occurred outside of the virtual machine such as an interrupt.

### 3.1.1.6 Summary

The RTSJ was designed to provide real-time capabilities that can be added to existing Java implementation with reasonable effort. More precisely, it does not require more advanced new technologies in areas like memory management to permit the development of real-time applications.

This results in the need to have special classes for threads that perform real-time operations. But even threads of class RealtimeThread can still be stopped by garbage collection activity, which make only NoHeapRealtimeThread applicable for time critical tasks. The application is consequently forced to be split into two completely independent parts for real-time and non-real-time code. Communication between these parts is severely restricted.

## 3.1.2 Other real-time API definitions

Two other real-time API definitions have been defined by working groups of the J-Consortium. This consortium is formed by more then 150 members. It provides a forum for the definition of extensions and standards related to Java technology that is independent of Sun. Unlike the Java Community Process that is dominated by Sun, no single member of the J-Consortium is dominating the process of standards defined within the J-Consortium.

### 3.1.2.1 rtCore

The realtime Core is a specification of the J-Consortium. It defines a 'core' for real-time programming that exists parallel to the Java virtual machine. Within the core, low-level operations that do not interfere with Java functionalities like garbage collection are permitted. Communication between core classes and the standard Java implementation is restricted to specific APIs, making the core a separate world parallel to the non-real-time standard Java world.

The realtime Core even permits to be implemented stand-alone, i.e., without the additional standard Java implementation.

### 3.1.2.2 RTDA

RTDA, Real-Time Data Access specification, is a proposal for an API extension that enables Java code to access hardware directly. It defines APIs that permit accessing memory mapped I/O hardware and to implement interrupt handlers directly in Java.

RTDA requires additional runtime checks by the virtual machine implementation to ensure consistency of the overall system.

## 3.1.3 Comparison of Real-time API Extensions

Comparing the presented real-time API extensions, the Real-Time Specification for Java is clearly the most advanced and stable. It covers the important needs for space applications and it has been implemented in first reference implementations. However, discussion on details of the implementation is still going on and there are several details of the specification that still need further clarification.

The rtCore specification has not been implemented by any Java system so far even though the specification process has started many years ago. It is hence not clear weather it is implementable at all and it must be expected that little support for this specification will be available in the future.

The RTDA provides only minimal features needed in for industrial automation applications. It has been implemented for one commercial product. It does not seem to be used apart from this area and provides less features than the RTSJ.

Consequently, planned or available support for RTSJ has became an important criteria in this Java implementation evaluation.

## 3.2 Deterministic Execution

The main source for indeterministic execution in typical Java implementations is the garbage collector. But also more subtle indeterminism might be introduced by Java's features like dynamic binding and monitors. These areas are and available solutions are presented in the following sections.

Finally, a technique to avoid indeterminism and inefficiency due to memory allocation is presented: static garbage collection. In a system using static garbage collection, code that performs dynamic memory allocation becomes easier to analyse and it will permit significantly better real-time performance.

### 3.2.1 Garbage collection thread problematic

Classic Java implementations perform the garbage collection work within a separate thread parallel to the application threads. This thread needs to be invoked regularly to reclaim unused memory and hence enable the application to continue running and allocating memory. Since the garbage collector accesses and modifies data stored on the heap and in the runtime threads, it needs to suspend all normal Java threads during its activity. This suspension is illustrated in Fig 3.1: Suspension of Java threads by garbage collector thread. In the figure, the garbage collection thread GC periodically suspends all user threads User 1, User 2, etc.

The length of the suspension of the Java threads depends on the garbage collector implementation. A non-incremental implementation requires a full garbage collection cycle to complete. This means that all the memory allocated on the heap and all runtime stacks need to be observed by the garbage collector. This requires a significant amount of time typically in the order of tens or hundreds of milliseconds.

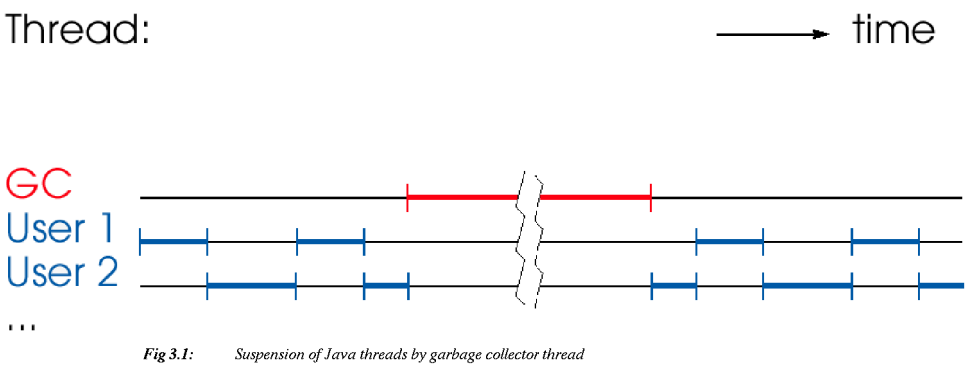



*Fig 3.1: Suspension of Java threads by garbage collector thread*

So-called incremental garbage collectors do not require a full garbage collection cycle at once. Instead, this cycle can be performed incrementally, while one increment consists of analysing one or some single Objects or the runtime stacks. The amount of time that needs to be spent in the garbage collection thread can be reduced significantly in an incremental collector, but the worst-case time may remain in the order of tens or hundreds of milliseconds due to the facts that

- Analysis of the runtime stacks can not be performed incrementally.
- The garbage collector needs to analyse or move an object completely in an incremental step.

- Java objects can be of arbitrary sizes, e.g., large arrays.
- A large amount of garbage collection work is required at once to finish a garbage collection cycle and to free sufficient memory such that the application can proceed execution.

Consequently, the worst-case pause time due to garbage collection can be very large even in incremental garbage collectors and the approach is not applicable to real-time applications with strict timing demands.

To remedy this problem, so called real-time Java implementations provide for a special class of threads, real-time threads, that can interrupt the garbage collector activity. This solution is illustrated in Fig 3.2: Real-time threads pre-empting the garbage collector thread. In this illustration, the garbage collector thread GC suspends all user threads User 1, User 2, etc., but real-time threads rt1 and rt2 are capable of suspending the GC thread and all user threads.

Since these real-time threads may interrupt the garbage collector, they are limited in the extend to which they can use the memory management system. More precisely, they cannot directly allocate objects from the standard heap. In the Real-time Java Specification, this problem is approached by using different heaps that are not under direct control of the garbage collector. The result is that memory management in real-time code is essentially manual, unused memory is not automatically reclaimed by the garbage collector. The benefits of garbage collection (reduction of complexity, no memory leaks, etc.) are not available.

Another important difficulty with the use of real-time threads is communication and synchronisation with user threads. Whenever a real-time thread tries to enter a monitor that might be taken by a user thread, we have the danger of a priority-inversion situation since the user thread might cause activity in the garbage collection thread. Consequently, the real-time thread is interrupted by the garbage collector activity and hence cannot provide strict timing guarantees any longer.

A better solution can be provided if the garbage collector works in even smaller increments that have a small upper bound on the execution time in the order of a few microseconds. This can be achieved by

- Avoiding the need to scan the runtime stacks,
- incrementally analysing larger objects by the garbage collector, and
- using a non-fragmenting object layout such that moving of objects is not required.

The garbage collection work can then be performed directly within the application threads at the time an object is allocated. This technique permits one give an upper bound for the execution time of an allocation while it is ensured that sufficient memory is reclaimed by the garbage collector. The result is a system with equal threads that are all real-time in the sense that they can interrupt one another

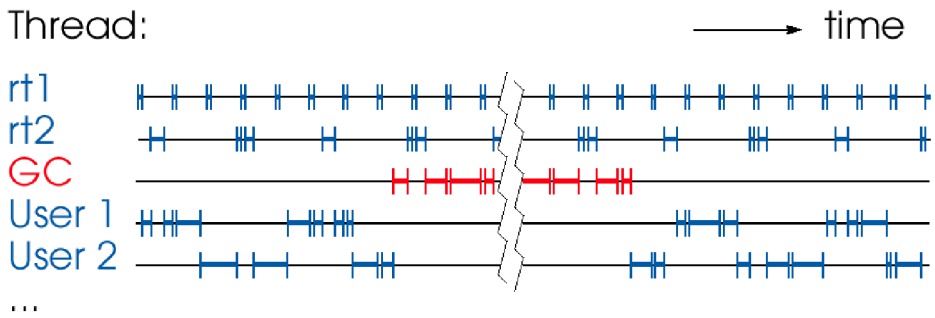



***Fig 3.2:*** *Real-time threads pre-empting the garbage collector thread*

according to their priorities. Even garbage collection work can be interrupted in a short limited time. This scheme is illustrated in Figure 3.3: Performing garbage collection work in application threads.

The described difficulties when communicating between real-time and user threads that we saw in a solution using a garbage collection thread are not present in the approach. All threads are equal and the described priority-inversion cannot occur.

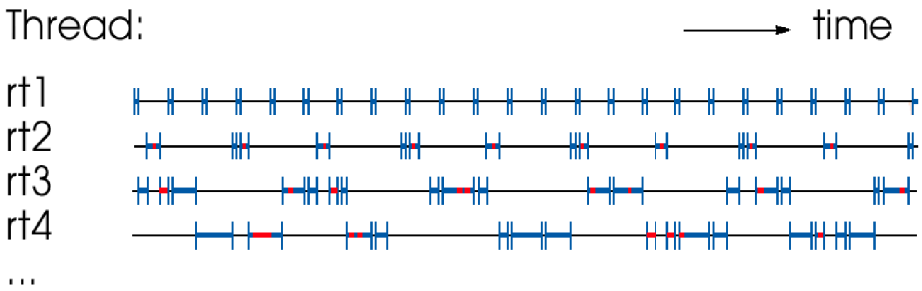



***Fig 3.3:*** *Performing garbage collection work in application threads*

### 3.2.2 Other garbage collection problematics

#### 3.2.2.1 Accuracy

For the garbage collector to be able to find allocated memory, it traverses references from global data structures and runtime stacks to objects and references from objects to other objects. All objects that are directly or indirectly referenced by from a global structure or a runtime stack are called reachable.

The difficulty is to provide information to the garbage collector on where to find valid references. Providing this information for objects on the heap can be done through type descriptors that are associated to all objects. Providing this information for references stored within activation frames on the runtime stacks is significantly more difficult.

Due to the lack of exact information on where to find references within runtime stacks or objects, a number of garbage collector implementations use "conservative" techniques to identify pointers. The idea behind this is that any value that is a found on the runtime stack or within an object and that represents a valid pointer is considered a reference to an object. There are different degrees of conservative garbage collectors: Fully conservative (i.e., conservative on runtime stacks and objects) or partially conservative (i.e., conservative on runtime stacks only). A garbage collector that does not contain conservative techniques is called exact or accurate.

The presence of conservative techniques causes a number of problems to critical systems. A misidentified pointer (e.g., a float value that happens to represent a valid pointer) can keep the garbage collector from reclaiming the memory of the referenced object even though it is unreachable through any pointer variable. Worse, such an object might itself contain references to data structures of arbitrary sizes and hence cause an arbitrarily large memory leak.

Even though the use of conservative techniques is unlikely to cause difficulties, it infringes the security of the system since it opens a means for an attacker to intentionally cause a memory leak on the system. With knowledge about the systems memory structure an attacker might provide data or code to the system that cause the misidentification of pointers such that the resulting memory leak stops the systems normal operation.

For these reasons, it appears inadequate to apply any conservative pointer identification techniques in space systems.

#### 3.2.2.2 Root Scanning

Root scanning is the part of the garbage collection process that identifies references stored within runtime stacks and global data structures. As has just been described, providing exact information on the locations of references within runtime stacks is difficult, conservative scanning techniques are often employed in the root scanning phase.

Another difficulty is that the root scanning phase must be completed as an atomic operation in most garbage collector implementations. Since the runtime stacks can be relatively large, this results in blocking the system for a time that might be too long for time-critical systems.

Some techniques have been developed to perform root scanning incrementally or to do it in an efficient constant-time operation. Unfortunately, these techniques are not widely used due to the difficulty they pose for an efficient implementation.

### 3.2.2.3 Fragmentation

The allocation and freeing of objects of different sizes cause the heap memory to become fragmented. Fragmented memory is free memory that is available in several non-contiguous chunks of memory that hence cannot be used for allocation of larger objects.

Fragmentation typically does not cause difficulties on short running systems or on systems that only use a very limited number of sizes for objects. Some garbage collector implementations consequently accept the danger of failure due to a fragmented heap and take no measures against it.

Two techniques to cope with fragmentation are known: Compaction and non-fragmenting object layout. A system using memory compaction moves all allocated objects during garbage collection such that after a garbage collection cycle the resulting free memory is a contiguous range. Moving of objects typically needs to be done atomically causing a worst-case pause time linear in the size of the largest object. Java objects can be of arbitrary sizes (and Java arrays often are very large), such that this approach causes long pause times that are difficult to accept in real-time systems.

Another difficulty induced by compaction is the need to update all references to a moved object to refer to the object's new location. The number of references to an object can be arbitrarily large such that updating them can be an arbitrarily complex operation. Instead, compacting implementations typically use an additional indirection for all reference accesses through a so-called handle. After an object has been moved, all that needs to be done is to update the reference stored in the handle to refer to the object's new location.

In contrast to compaction, an implementation using non-fragmenting object layout does not move objects. Instead, the available memory is regarded as a large array of blocks of all the same size. Allocated objects occupy at least one such block, while larger objects are composed out of several such blocks that might be non-contiguous in memory. The difficulty of this approach lies in providing good runtime performance using this object model.

### 3.2.2.4 Progress Guarantee

In classical applications of Java a good responsiveness in user interaction is a requirement. This means, the system was optimised such that the average length of garbage collection pauses and the average frequency of long pauses were reduced.

In time-critical applications, this is not sufficient. Instead, it is required that garbage collection work be predictable and bounded while it needs to be ensured that the garbage collector makes enough progress, i.e., the work will be sufficient to reclaim memory to satisfy all future allocation requests.

Basically, two techniques are known to provide such progress guarantees. Both techniques require knowledge about the application's maximum memory use. This amount must be lower than the memory available on the system (if it was higher, the system would fail due to an out-of-memory condition).

The first approach uses allocation rates. In this scheme, the allocation characteristics of all tasks in the systems have to be analysed to know the maximum amount of memory allocated per unit of time (e.g., MB/sec). These allocation rates can then be used to schedule the garbage collector such that it performs sufficient work to reclaim memory at a rate that is at least as high as the maximum allocation rate. This approach requires a good analysis of the application to obtain the allocation rates. A disadvantage of this approach is that it does not dynamically react to the behaviour of the application, i.e., even during times when the actual allocation rates are low the garbage collector needs to run at the selected reclamation rate.

The alternative approach performs garbage collection work at allocation time only. The amount of garbage collection work can be fixed or determined dynamically as a function of the amount of memory currently in use. In both cases, an upper bound for the amount of work that needs to be performed on an allocation can be determined. This approach automatically adopts the garbage

collection activity to the allocation needs of the application; periods with few or no allocations require little or no garbage collection activity. There is no need for further analysis of the application apart from the need to determine the maximum memory used.

### 3.2.3 Deterministic implementation

The garbage collector of a Java implementation poses the biggest difficulty in providing real-time threads and the support for timing- and safety-critical applications. Nevertheless, there are a number of minor issues with the implementation of the Java language itself that should be addressed to make the implementation applicable to this domain. These issues will be described in the following section:

#### 3.2.3.1 Executing Java Bytecodes

Most Java bytecode instructions that are executed by the Java interpreter or compiled into machine code by some compilation technology are straightforward to implement in a way that provides deterministic execution. Bytecodes that pose difficulties are those implementing dynamic dispatch, dynamic type checking and memory allocation.

##### 3.2.3.1.1 Method Invocation

Java knows three kinds of method invocations: Invocation of statically bound methods, invocation of dynamic methods and invocation of interface methods. The invocation of a statically bound method is straightforward since no dynamic retrieval of the method is required.

Java supports single inheritance for classes. This permits the implementation to use simple linear lists for the method tables of all dynamically bound methods. Methods are assigned constant indices within this list. When a method is redefined in a sub-class, the redefined method will inherit the index from the original method and the corresponding entry in the method table will be overwritten. On a dynamically bound method call, it is hence sufficient to access the method table at the called method's index to determine the address of the actual method that is to be called. The call overhead is small and can be implemented in constant time. It poses no difficulty for deterministic execution.

Things are different when calling interface methods. Since multiple inheritance is possible for interfaces, a simple method table is not sufficient to implement these calls. Instead, a dynamic search for the called method is required. Depending on the implementation, this search might require time linear or logarithmic in the number of methods implemented by the target class of a call. Techniques like "inline caching" [DS84] are used to reduce the call overhead in consecutive calls with the same target class, but these techniques do not provide a deterministic worst-case execution time for the call in general.

A deterministic approach to implement interface calls is to use multiple method tables for different interfaces. References to these method tables might either be embedded within the objects themselves [Strou87] or stored within one array per class [SW01]. In these schemes, two indirections are typically required for the retrieval of the implementation of an interface method, in contrast to a single indirection that is needed for a dynamic method call with single inheritance. Nevertheless, the overhead to find an interface method is constant, no dynamic search is required.

The disadvantage of this approach is the potentially high memory overhead for the interface method tables and the references to these tables.

##### 3.2.3.1.2 Type checking

Dynamic type checking is one of the most frequent operations performed in Java code. Type checking either occurs explicitly in statements doing a type cast or instanceof-check as in

```
if (b instanceof T) {
  a = (T) b;
```

```
}
```

or they occur implicitly when storing a reference in an array as in the code

```
Object[] o = ...;
o[i] = b;
```

For this array access, the type check is required to check for the ArrayStoreException that is required to ensure the type safety of Java.

Type checking code needs to be able to determine whether the type of an object is equal to or an extensions or implementation of a given class, interface or array type. Straightforward implementations of type checking would traverse the inheritance tree of classes or the graph of implemented interfaces until the required type is found. This traversal is inefficient and causes difficult to predict runtime overhead. It is not adequate for a deterministic implementation.

Techniques have been developed that permit the implementation of a type check within constant time independent of the actual type of a reference [SW01]. These techniques should be preferred in critical applications.

#### 3.2.3.1.3 Memory Allocation

Memory is allocated in a Java application whenever the "new" operator is used to create new objects, arrays or multi-dimensional arrays.

There are two main requirements for memory allocation operations in a deterministic implementation of Java: These operations must complete in bounded time and they must actually allocate memory. I.e., these operations must not cause out-of-memory errors as long as the memory used by the application stays within the bounds set during the configuration of the system.

To ensure these properties, a suitable garbage collection technique needs to be implemented. The corresponding techniques were described earlier in this document.

### 3.2.3.2 Monitors

In Java, any object has an associated monitor that permits to synchronise on this object for exclusive access by a single thread. The "synchronise" keyword is used to automatically enter and leave a monitor during a statement sequences or a complete method.

Due to the multi-threaded nature of Java, synchronisation is used very frequently throughout Java application and the Java standard libraries. A deterministic implementation of the synchronisation mechanism is hence required for the use of Java in critical applications.

A number of techniques are being employed by current Java implementations that reduce the per-object memory overhead for monitors and to optimise the most frequent operations on monitors such as entering a monitor that has not been entered by any other thread [Yang99, BKMS98]. Nevertheless, these techniques require more overhead in less frequent operations like adding a thread to the waiting list for a monitor that is owned by another thread. These operations require the allocation of additional memory to store the state of the monitor and the waiting list. This memory is typically taken from the garbage collected heap or from a global hash table.

For a deterministic implementation of monitors, the possible use of dynamic allocation or accesses to hashtable for synchronisation is not desirable. It introduces bad worst-case execution times for otherwise efficient synchronisation operations.

The use of dynamic allocation or hash tables can be avoided [SW01].

### 3.2.3.3 Exceptions

Java has a fairly sophisticated exception handling mechanism. Exceptions can be used to divert control flow to any other method on the call stack. Exceptions are represented as Java objects that are typically allocated on the heap at the moment the exception is thrown.

The time required for exception passing is typically hard to predict due to the arbitrary size of the call stack that needs to be traversed to find the destination of an exception and due to the dynamic allocation. However, exceptions are not intended to be used for normal control flow, but for control flow in exceptional conditions like errors only. Consequently, it is not required to have predictable performance for exceptions as long as it is ensured that an exception caused in one thread does not affect the execution of other threads in a system that have to continue normal operation.

## 3.3 Static Garbage Collection

A static garbage collector relieves the work of the dynamic garbage collector by inserting explicit free statements to free objects that are known to be garbage. The dynamic garbage collector must be aware of which objects are handled by the static garbage collector, so it can ignore those objects, otherwise no performance is gained. If it would be possible to find out where each and every object becomes garbage, no dynamic garbage collector would be necessary. However, this problem is very hard and has not been solved.

### 3.3.1 Stack Allocation

A limited static garbage collector finds objects that can be allocated on the stack. Objects that can be allocated on the stack must always become garbage before the function, in which it is allocated, returns. Thus, stack allocated objects must have a life time that is shorter than the life time of its stack frame.

There are three possible approaches when using stack allocation: objects can be allocated inside a stack frame, objects can be allocated using the alloca function, or objects can be allocated on the heap and explicitly freed when the function returns. Allocating memory in stack frames could be problematic if the number of objects that are allocated is not known at compile time, e.g. if objects are allocated within a while loop. Then it is better to use the alloca function that is available in most C runtime environments. A possible problem with using alloca or allocation within stack frames is that the run time stack can grow very large, since objects are moved from the heap to the stack. To work around this problem, normal heap allocation can be used. However, the allocated objects must be remembered so they can be freed when the function returns.

A simple approach to find objects that can be allocated on the stack is to find out if any reference to the object can escape the function. A reference can escape a function by being stored in an object outside of the functions stack frame, by being sent as an argument to another function, or by being returned from the object. This analysis is intra procedural.

The analysis can be extended by allowing references to escape the function, but only to stack frames "below" the stack frame where it is allocated. This analysis requires inter procedural dataflow analysis, which is potentially expensive and cause problems when using separate compilation.

### 3.3.2 Separating Statically Handled Objects from Others

The dynamic garbage collector must be able to separate statically handled objects from other objects. The best case is when a reference always refers to either stack allocated objects or to heap allocated objects. Then the generated code can be designed to handle the different cases. In this case stack allocated objects can be completely ignored by the dynamic garbage collector, and the other objects can be handled as before.

If a reference can refer to both stack and heap allocated objects, a runtime check is needed to separate the different cases. One possibility is to put a flag in the object header, but that could cause extra memory overhead. Another possibility is to separate objects by using the address they are stored at. If stack allocated objects are stored on the heap, as discussed above, these objects have to be allocated in a separate area so it could be decided whether or not objects are handled by the static garbage collector.

## 3.4 VM threads vs. OS threads

The Java implementation has the choice to provide its own thread implementation or to base Java threads on threads provided by an underlying operating system. This advantage of the first approach is a higher independence of the underlying operating system, while the latter provides a simpler Java implementation that can re-use the real-time threads and synchronisation features provided by a real-time operating system.

This is particularly important if code written in other languages such as C need to be addressed within the application (legacy code, drivers, etc.). A solution using threads provided by the operating system provides an easier means to determine the overall schedulability of the system.

## 3.5 VM real-time criteria

The following criteria are used to evaluate the real-time features of the candidates:

**Destructive criteria:**

**D2: Real-time capabilities (implemented or possible to implement)**

It must be possible to support real-time capabilities with reasonable effort. This means that basic real-time support must be present in the implementation or could be added without major difficulties.

**Selective criteria:**

| ID # | weight | description |
|---|---|---|
| 1.1 | 30 | Realtime Java Spec |
| 1.2 | 70 | Realtime Capabilities |
| 1.2.1 | 40 | Real-time thread/scheduling |
| 1.2.2 | 30 | Deterministic garbage collection |
| 1.2.3 | 30 | Deterministic language implementation |

It was considers important to have standardised real-time support through the RTSJ specification, which was hence given a weight of 30%. Among the general real-time capabilities (70%) The presence of real-time threads is most important (40%), while deterministic garbage collection and overall Java language implementation where valued equally important (30% each).

# 4. Safety Critical features

The safety of the execution environment in a Java virtual machine is a major reason for the success of Java in a wide range of applications that require the exchange of active components that contain untrusted code that needs to be executed on the target systems. The Java 'sandbox' and the classfile verification process guarantee that code does not corrupt the overall system.

## 4.1 Exception mechanism

Java uses exception mechanisms to ensure that code cannot access and hence corrupt parts of a running system it is not supposed to touch. Unsafe operations like using undefined pointers or accessing array outside of the legal index range are handled this way.

## 4.2 Memory Management Safety

One important safety aspect is to protect the system from corruption of the heap. Strict type and pointer safety of Java and the use of garbage collection instead of explicit freeing of memory protect the system from unauthorised accesses of internal data structures by Java code.

Nevertheless, a number of potential safety issues with the memory management system remain. These are

### 4.2.1 Memory fragmentation

A long running system might fail due to fragmentation of the heap memory. The heap is fragmented when free memory ranges are scattered throughout the heap in small chunks that are not usable for larger allocations that need to be performed for the system. A system with fragmented memory might hence fail to allocate memory and stop execution.

The memory management system must ensure that fragmentation is solved by either actively defragmenting the memory by moving objects or by using the heap in such a way that it cannot get fragmented (see section 2.2.2.3).

### 4.2.2 Conservative Techniques

Conservative techniques to identify used memory by the memory management system can cause unpredictable loss of memory and hence cause the system to fail due to low memory in an unpredictable way.

A safe implementation must hence use accurate pointer identification (see section 2.2.2.2).

### 4.2.3 Allocation bounds

Malfunctioning or malicious code can cause the allocation of large amounts of memory and hence cause the failure of the system due to low memory.

For the Java implementation, it is not possible to distinguish acceptable allocation from non-acceptable allocation. One means to overcome this problem is provided by the RTSJ APIs that permit to limit the total amount of allocation that is performed within a thread. This limits the maximum amount of memory non-trusted code might take form the system heap and hence protects the system from low-memory situations.

## 4.3 Functionality

The quality of the functionality of the implementation can be measured by the completeness of the implementation and its correctness.

### 4.3.1 Completeness

The completeness criteria selected for this analysis are specification and requirements completeness, implementation coverage and the rates of program units, verification activities and user manual items.

### 4.3.2 Correctness

For correctness testing of Java virtual machine and API implementation, Sun provides Java Technology Conformance Kits (TCK). Due to licensing issues, not all Java implementations can use these TCKs to evaluate the correctness of the implementation.

Instead, a number of independent test suites are available for conformance testing of Java implementations. These include the publicly available Mauve testsuite that contains contributions of a number of Java virtual machine implementor (such as HP, Acunia, etc.). Commercial test suites are available from Plum Hall Inc. (JVS - Java Validation Suite) and Perennial (JETS, $15,000). IBM developed the freely available Java compiler testsuite Jacks [Jacks].

## 4.4 Reliability

To Quantify the reliability of the implementations, the characteristics Recoverability, Robustness, Integrity and maturity where considered in the analysis of the Java implementations.

## 4.5 Security

Security is ensured in Java through the Java classfile verification process and the presence of the Java sandbox with rights restricted according to the installed security manager. The level to which this functionality is provided determined the appropriateness of the Java implementations for the use in AERO.

## 4.6 Safety evidence

To what extend is the safety of the implementation proven.

## 4.7 Safety critical features criteria

The following criteria are used to evaluate the safety critical features of the candidates:

**Destructive criteria:**

none

**Selective criteria:**

| ID # | weight | description |
|---|---|---|
| 2.1 | 30 | Functionnality |
| 2.1.1 | 60 | Completeness |
| 2.1.1.1 | 20 | Technical specification/software requirements mapping rate |
| 2.1.1.2 | 40 | Functional implementation coverage |
| 2.1.1.3 | 20 | Functional requirements/program units mapping rate |
| 2.1.1.4 | 10 | Functional requirements / verification activities mapping rate |
| 2.1.1.5 | 10 | Functional requirements / user manual items mapping rate |
| 2.1.2 | 40 | Correctness |
| 2.1.2.1 | 20 | Module branch coverage |
| 2.1.2.2 | 15 | Test completeness |
| 2.1.2.3 | 15 | Verification coverage |
| 2.1.2.4 | 10 | Successful interface testing rate (100%) |
| 2.1.2.5 | 10 | Interface testing completeness |
| 2.1.2.6 | 30 | Run time errors verification |
| 2.2 | 30 | Reliability |
| 2.2.1 | 20 | Recoverability |
| 2.2.1.1 | 30 | Mean Time to Restart |
| 2.2.1.2 | 70 | Mean Time to Recover |
| 2.2.2 | 20 | Reliability evidence |
| 2.2.3 | 30 | Robustness |
| 2.2.4 | 15 | Integrity |
| 2.2.5 | 15 | Maturity |
| 2.3 | 20 | Security (depending on requirements) |
| 2.4 | 20 | Safety evidence |

The functionality of the implementation (30%) was weighted equally important as the reliability (30%), while security and safety each provide 20% of the total weight of safety-critical features.

# 5. Quality

The quality of the available Java solutions for space applications is measured here using the criteria couplability, portability, reusability, testability, maintainability, adaptability and operability. These criteria are explained here in more detail.

## 5.1 Couplability

The system needs to be capable to run in conjunction with other code on the system that might be related to the Java code or might be unrelated. A solution with good couplability permits easy interfacing and co-existence with other code.

## 5.2 Portability

The portability of the system measures the easy of support for new processor architectures and new operating systems. An important aspect of porting the system for space applications is the stability of the software. To enable porting of the software, it must be largely independent of external software components and specific system hardware.

The presence of interpreter and compiler environments in Java implementations has important effects on the portability of the implementation.

### 5.2.1 Interpreter

A system that contains a Java interpreter is in general easy to port to new architectures as long as the interpreter itself is written in portable C or C++ code. Since the interpreter loop is the most performance-critical part of an interpreted Java environment, it is often hand-coded in assembly language that takes into account the specific processor (size of instruction cache, register sets, etc.).

Such an interpreter is typically much more difficult to port to a new architecture.

### 5.2.2 Compilation strategies and their portability

In addition to the interpreter, Java implementations use compilation techniques to obtain a significant performance enhancement over interpreted code. There are two compilation approaches: just-in-time (JIT) and static (ahead-of-time or AOT) compilation.

Both, JIT and AOT compilers are typically difficult to port to new processor architectures since the code generator needs to be changed to create the code for the selected target processor.

Some AOT compiler use ANSI C code as an intermediate language. This makes it significantly easier to port the compiler to new architectures. All that is needed is a C compiler for the target architecture and possibly some minor adjustments (endianess, etc. ).

## 5.3 Reusability

Two aspects of reusability are taken into account here: The self-contained functionality of the system and its components and portability of components.

## 5.4 Testability

The extend to which the implementation can be tested and how easy is it to perform tests are quantified by the testability.

## 5.5 Maintainability

The maintainability is the effort that is required to locate and fix an error in an operational program. It is influenced by the analysability and complexity of the code.

## 5.6 Adaptability

The possibility for code to evolve for new environments is its adaptability. An important aspect of the adaptability is access to native code via standard of proprietary interfaces.

## 5.7 Operability

The availability of sufficient documentation, tutorials, training, etc. are important aspects of the operability of the solution.

## 5.8 Quality criteria

The following criteria are used to evaluate the quality of the candidates:

**Destructive criteria:**

none

**Selective criteria:**

| ID # | weight | description |
|---|---|---|
| 3.1 | 10 | Couplability |
| 3.2 | 25 | Portability |
| 3.2.1 | 30 | Ease to support new architecture (especially Sparc ERC32/Leon) |
| 3.2.2 | 20 | Ease to support new OS (especially RTEMS and VxWorks) |
| 3.2.3 | 20 | Instability |
| 3.2.4 | 15 | Environmental software independence |
| 3.2.5 | 15 | System hardware independence |
| 3.3 | 10 | Reusability (allows reuse of components) |
| 3.3.1 | 30 | self-contained functionality |
| 3.3.2 | 70 | portability (dependability of components) |
| 3.4 | 15 | Testability (easy to test) |
| 3.5 | 15 | Maintainability (allows corrections) |
| 3.5.1 | 60 | Analysability |
| 3.5.1.1 | 50 | Problem cause understanding |
| 3.5.1.2 | 30 | Cyclomatic complexity or Module complexity |
| 3.5.1.3 | 20 | Code understanding |
| 3.5.2 | 40 | Portability |
| 3.6 | 15 | Adaptability (allows evolution) |
| 3.6.1 | 30 | Allows evolution |
| 3.6.2 | 70 | Access to native code (standard, proprietary etc.) |
| 3.7 | 10 | Operability: usability (Tutorial readiness etc.) |

High importance among the quality features has the portability (25%) and adaptability (15%) since the implementation needs to be adopted for new hardware and a new application domain. Also of high importance for quality insurance and future maintenance are the testability (15%), the maintainability (15%), the couplability (10%), reusability (10%) and operability (10%).

# 6. Performance

The two relevant aspects of performance are the runtime performance and the memory requirements of the implementation.

## 6.1 Runtime performance

Since Java is essentially an interpreted language, the performance is poor compared to compiled language like C as long as no compilation technology is used. To obtain better runtime performance, a first step are improvements in the interpreter like replacement of complex bytecode instructions by faster ones after their first execution or the use of hand-coded assembly code for the inner loop of the interpreter.

A significant improvement of the runtime performance can be obtain by using compilation techniques using JIT or AOT compilers.

### 6.1.1 JIT compilation

A JIT compiler runs on the target system and compiles Java bytecode on the system into machine code. The advantage of a JIT compiler are that all code that is on the system and that is dynamically loaded onto the system will benefit form the enhanced performance. The disadvantages are that the compiler must be present on the target system and requires significant amounts of additional memory (ROM and RAM).

In time-critical applications, a JIT compiler might not be employed since it causes a very hard to predict execution time: Code is either interpreted (slow), or compiled (fast) or in the process of being compiled (very slow).

### 6.1.2 AOT compilation

AOT compilation is done at system build time on the development system, no compiler is required on the target system. The advantages are that more time can be spent on optimisations of the code and no runtime or memory overhead on the target system is needed for the compiler. The disadvantages are that compilation is restricted to applications and libraries that are put onto the system at build time, code that is loaded dynamically cannot be compiled (some AOT compilers even disallow dynamic loading of code completely).

Nevertheless, dynamically loaded code profits from the enhanced speed of library code that was precompiled on build time. This can still lead to a dramatic performance improvement compared to purely interpreted environment.

## 6.2 Memory requirements

The most important types of data that require RAM or ROM memory to execute are storage for Java classes, compiled code (and the compiler itself if JIT is used) the Java heap and runtime stacks.

### 6.2.1 Storage for Java classes

Java classes come from two sources: They are system classes that are stored in ROM on the system itself, or they are loaded dynamically onto the system. In both cases, these classes typically need to be copied into RAM before they can be used by the Java implementation.

To put system classes into ROM, different approaches are used. The simplest solution is to store Java class files in ROM such that they can be loaded as if they were stored on a disk. More memory-efficient solutions use a specific format for classes in ROM that avoids the need to copy all of the class file information into RAM. Instead, these formats enable the use of the code directly from ROM.

Sometimes, these techniques are combined by compaction techniques that reduce the amount of ROM required for these classes.

In any case, dynamically loaded cases need to be stored in RAM.

### 6.2.2 Storage for compiled code

When compilation is used, the compiled code needs to be stored on the target systems. A system using JIT compilation has to store the compiled code in RAM, while a system using AOT compiler can use ROM for the compiled code.

Compiled machine code is significantly larger than interpreted Java bytecode. This makes in impractical to compile all of the application. Instead, it often makes sense to compile only those parts of an application that are executed frequently. In a JIT compiler, this is done automatically using runtime profiling, while a AOT compiler needs to support profiling techniques to discover and compile frequently executed code.

In a system using JIT compiler, the code of the compiler itself and its runtime date structures must be present on the target system. This typically adds an overhead of several hundred KBytes of additional RAM and ROM.

### 6.2.3 Storage for the Java heap

The Java heap basically holds objects used by the Java application. So it is mainly the application's responsibility to use little memory here. But a few aspects of the Java implementation are important here as well:

Additional overhead for each object is required to hold information on the object's type, its monitor, garbage collector information, etc. Since objects in Java are typically very small, This additional overhead must be as low.

Also, the Java implementation has to ensure to use an object layout that is memory-efficient. Some Java implementations use at least one word to store even simple type like boolean values. Memory-efficient implementation use the minimum required number of bits for each type.

### 6.2.4 Storage for runtime stacks

The runtime stacks are needed by the interpreter loop. The memory required for theses stacks largely depends on the application and cannot be influenced by the Java implementation. However, implementation using compilation techniques might require additional runtime stacks for compiled code. In these systems, the memory required for runtime stacks becomes more important.

## 6.3 Performance criteria

The following criteria are used to evaluate the performance of the candidates:

**Destructive criteria:**

**D3: minimal RAM used < 512KB**
**D4: minimal ROM used < 256KB**

The implementation must be able to cope with the limited resources as they are present in the space domain. For this, the minimal footprint within RAM (Java heap, dynamic data) and ROM (code and static data) must be limited.

These minimum memory demands are measured for a minimal application (e.g., HelloWorld) using only a minimal subset of the library classes if such a feature is provided by the system. However, features required for performance enhancements etc. Are included in the values.

**Selective criteria**:

| ID # | weight | description |
|---|---|---|
| 4.1 | 20 | RAM required |
| 4.2 | 15 | ROM required |
| 4.3 | 5 | Compiler available |
| 4.4 | 30 | Performances (ratio CPU used / provided performances) |
| 4.5 | 5 | Mixing of compiled and interpreted code possible |
| 4.6 | 10 | Computational Accuracy (dynamic) |
| 4.7 | 15 | Efficiency |
| 4.7.1 | 30 | Timing margin |
| 4.7.2 | 30 | Memory utilization saving |
| 4.7.3 | 20 | I/O device utilization time |

Most important among the performance criteria are the memory demand (RAM 20%, ROM 15%), the high runtime performance (30%), the possibility to compile bytecode (5%) and to mix this compiled code with compact interpreted code (5%). The accuracy of computation and the efficiency where weighted with 10% and 15%, respectively.

# 7. Java language and Configurations and APIs

For the application of Java in embedded devices, Sun has defined the Java 2 Micro Edition. The Java 2 Micro Edition includes several configurations for Java implementations. On top of these configurations, different profiles have been defined for different application domains.

The configurations define subsets of the Java language implemented by the virtual machine and sets of standard classes that need to be supported. In the context of space applications, two configurations are important: CLDC and CDC.

## 7.1 CLDC

This CLDC (Connected, Limited Device Configuration) is the base for supporting a wide variety of devices that have very limited resources in the range of 160 to 512kB of memory. CLDC defines a subset of the Java language that does not include floating point and 64-bit integer arithmetic. It is intended of small mass-market devices connected to a network with a low-bandwidth connection.

For space applications, the CLDC configuration is clearly too small since it does permit precise calculations.

## 7.2 CDC

CDC is a larger profile than CLDC that includes support for the full Java language including floating point and 64-bit arithmetic. It is intended for system with 2-16MB of memory that might have a graphical user interface and permanent TCP/IP connection.

For space applications, the CDC configuration is too large, a subset excluding a number of features that are not needed will be sufficient.

## 7.3 Java language and API support criteria

The following criteria are used to evaluate the Java language and API support of the candidates:

**Destructive criteria:**

**D1: full Java virtual machine implementation.**

Not supporting the full Java language makes the implementation incompatible with Java development tools and not applicable as a base system.

**D5: File system is not required**

An implementation that does not provide an alternative for using a file system (to load classes etc.) cannot be applied in space systems.

**D7: Access to native code possible**

A system not permitting access to native code (libraries, low-level system functions, hardware access) cannot be applied in space systems.

**Selective criteria:**

| ID # | weight | description |
|---|---|---|
| 5.1 | 70 | Java language specification support |
| 5.2 | 30 | Java standard API support (to details) |

Full support of the Java language is very important for space applications (70%), while only few standard library classes are required (30%).

# 8. Complexity of the development

The complexity of the development that needs to be done to apply Java in space applications. This was measured by the expected complexity of the adoption of the existing code, the maintenance provided by the editor of the Java implementation and the existence of other reference projects in real-time and safety critical areas.

## 8.1 Complexity of the development criteria

The following criteria are used to evaluate the expected complexity of the development for the candidates:

**Destructive criteria:**

**D6: Source code available**

Not being able to access the source code (i.e. neither open source nor a commercial source licence is available) makes it impossible for us to extend the code.

**Selective criteria:**

| ID # | weight | description |
|---|---|---|
| 6.1 | 50 | Expected complexity of adoption (including familiarity) |
| 6.2 | 30 | Maintenance of implementation |
| 6.3 | 20 | Reference project in RT and/or safety critical area |

The expected complexity of the work required to adopt the implementation for space applications is the most important criteria in this group (50%), while the provided maintenance by the vendor (30%) and the proof through reference projects (20%) are also important for the development.

# 9. VM evaluation

## 9.1 Candidates

To obtain a broad overview of the existing Java technologies and to find the most efficient solution that could be embedded in space systems, 25 different Java applications available on the market or as result of research where analysed using the criteria described earlier in this document.

For those Java implementations, that passed the destructive criteria, the more detailed list of selective criteria was taken to measure the applicability of this solution for the AERO project.

The candidate virtual machine implementations are

| | | | | |
|---|---|---|---|---|
| Ameran | AstriumJVM | ChaiVM | EBCI | JBed RTOS |
| GhostVM | J9 | JamaicaVM | JanosVM | Japhar |
| Jeode | JTime | JV Lite2 | JVM CDC | JVM CLDC |
| Kada | Kaffe | Kissme | MachJ | PERC |
| pVM | SimpleRTJ | TinyVM | Waba | Wonka |

**Table 9.1** The evaluated virtual machines

This chapter continues with one page per candidate describing each of the candidates in detail and giving the result of the destructive criteria. The destructive criteria are

D1: full Java virtual machine implementation

D2: Real-time capabilities (implemented or possible to implement).

D3: minimal RAM used < 512KByte

D4: minimal ROM used < 256KByte

D5: File system is not required

D6: Source code available (free or as source-code license)

D7: Access to native code possible

## 9.3 Second round of evaluation

After elimination of the implementations that did not fulfil all destructive criteria, the following possibilities remain:

| | | | |
|---|---|---|---|
| AstriumJVM | ChaiVM | JamaicaVM | JVM CLDC |
| PERC | SimpleRTJ | TinyVM | |

**Table 9.3:** Candidates for second round of evaluation.

These Java implementations were analysed using detailed selective criteria and their weight. The highest level selective criteria are:

| ID # | weight | description |
|---|---|---|
| 1 | 20 | VM real-time |
| 2 | 25 | Safety critical features (implemented or possible to implement) |
| 3 | 20 | Quality: portability, adaptability, testability, maintainab., reusab. |
| 4 | 15 | Performance/Efficiency (limited resources: memory/CPU) |
| 5 | 10 | Java language and API support |
| 6 | 10 | Complexity of development |

**Table 9.4:** Highest level criteria and their weights.

When distributing the weights for the highest level criteria, highest importance was given to the safety-critical features (25%). Next come the real-time features and the quality of the implementations (20% each). Less important are the performance (15%), the language and API support (10%) and the expected complexity of the development within the AERO project (10%).

The detailed results of this evaluation are presented in Appendix A at the end of this document. The weights that were assigned are used to produce an cumulative overall result for each candidate. It lead to the following results. For each implementation, the main advantages and disadvantages are also listed here.

| **Candidate** | **score** | **remarks** |
|---|---|---|
| AstriumJVM | 6.12 | |
| ChaiVM | 5.55 | |
| JamaicaVM | 7.29 | |
| JVM CLDC | 5.87 | |
| PERC | 6.04 | |
| SimpleRTJ | 6.93 | |
| TinyVM | 3.81 | |

## 9.3 Conclusion

Given the presented analysis of available Java implementations, we found that no current implementation provides all the features that are required for the application of Java in space. The amount of modifications and enhancements required are varying widely though.

The candidates that provide the best basis for further development according to the analysis are JamaicaVM, SimpleRTJ and AstriumJVM. These candidates differ in the amount of future adoption that needs to be performed and the run-time performance achievable. JamaicaVM will need to be adopted for the space context, while AstriumVM needs to be adapted for realtime contexts. SimpleRTJ needs to be adapted for both space and realtime support.