REAL-TIME JAVA IN SPACE: POTENTIAL BENEFITS AND OPEN CHALLENGES

Tomas Kalibera[(1)], Marek Prochazka[(2)], Filip Pizlo[(1)], Martin Decky[(4)], Jan Vitek[(1)], Marco Zulianello[(3)]

[(1)]Purdue University, Dept. of Comp. Science, 305 North University St., West Lafayette, IN 47907, USA,
Tel +1 765 494 6531, Fax +1 765 494 0739, Email:{kalibera, filip, jv}@cs.purdue.edu
[(2)]SciSys UK Ltd, 23 Clothier Road, Bristol, BS4 5SS, United Kingdom,
Tel +44 117 916 5179, Fax +44 117 916 5299, Email: marek.prochazka@scisys.co.uk
[(1)]ESA/ESTEC, Keplerlaan 1 - P.O Box 299, 2200 AG Noordwijk, The Netherlands,
Tel +31 71 565 8933, Fax +31 71 565 5420, Email: marco.zulianello@esa.int
[(1)]Charles University, Dept. of SW. Engineering, Malostranske nam. 25, 118 00 Praha 1, Czech Republic,
Tel +420 221 914 266, Fax +420 221 914 323, Email: martin.decky@dsrg.mff.cuni.cz

ABSTRACT

In this paper we analyze potential benefits of using the Java programming language in spacecraft on-board applications as well as problems with current Real-Time Java implementations that would have to be solved to make this possible. We base our experience on porting the Ovm Real-Time Java Virtual Machine to RTEMS/LEON2 and also our findings in the Real-Time Java Assessment Project funded by ESA.[1]

1. INTRODUCTION

Java is a popular, mature, widely accepted programming language; it features extensive library support, high quality implementations, tool support for the complete software development cycle, and it has the benefit of being familiar to a large segment of the programmer population. While Java has traditionally been relegated to non-safety-critical software, the acceptance of real-time and safety-critical Java technologies is increasing steadily. The Real-Time Specification for Java (RTSJ) [4] has largely met its promises – commercial implementations exist, real applications are being deployed (including the DDG-1000 Zumwalt class destroyer [9], multiple UAVs [1], audio applications [2,10], and numerous uses for industrial control [13]). New technologies such as real-time garbage collection (RTGC) allow for an even easier programming model than the one originally stipulated by the RTSJ, and are already implemented and in use.

A natural question thus is: Can Java be used in space? This question has been asked in the past, but we believe it is valid to ask it again due to two reasons. Firstly, progress has been made in the past years (several RT Java implementations have been ported to RTEMS/LEON and they also have been evaluated systematically). Secondly, our reasoning about this issue is somewhat more practical and combines points of view of RT Java VM developers, spacecraft on-board software developers and customers. In this paper we analyze potential benefits of using Java in space applications and summarise our findings gathered during the Real-Time Java Assessment Project. We also point out technical issues of current Java implementations that have to be solved to make this possible.

2. HOW CAN SPACE APPLICATIONS BENEFIT FROM JAVA

Space applications can benefit both from features in the Java language itself and the Java ubiquity. The Java virtual machine lifts much of the burden of unnecessary low-level details from the programmer. The ubiquity of Java leads to broader tool support and a greater availability of programmers familiar with the language.

The wide spread of Java suggests that Java is a well balanced environment. Java itself is a simple language – much simpler than for instance C, C++ or Ada. Its syntax and semantics were specified after decades of research and practical experience. It is coherent, yet powerful and easy to use for most programmers. Java has a built-in support for class-based object-oriented programming, generics (a safer and simpler replacement for C++ templates), exception handling, and vertically integrated thread management. Java has a wide range of libraries as part of its thorough standard API, making it unnecessary for software engineers to seek out third-party libraries for most tasks. Furthermore, Java allows dynamic allocation and has automatic garbage collection, which takes even more burden as well as opportunities to make mistakes from programmers. Java has been proven by experience to be suitable for a wide range of applications.

The Java language is type safe. Type safety provides a solid ground for both security and spatial partitioning of different threads running in an application. Thanks to reference safety each thread can only access its properly allocated memory. Type safety is largely enforced at compile time and partially by the runtime environment – the Java Virtual Machine. In particular, type safety

1 Porting of Ovm to the RTEMS/LEON2 platform has not been funded by ESA, however ESA provided support to make this effort possible.

allows for the implementation of spatial partitioning without hardware support, such as a hardware MMU. The spatial separation is not of a great concern when Java runs on top of a general-purpose OS with memory protection, but it is vitally important for real-time operating systems such as RTEMS, which do not support hardware memory protection, even when running on hardware that provides it (i.e. LEON3). Moreover, type safety and further bytecode checking of the Java Virtual Machine makes the Java platform safer compared to native applications – attacks such as stack overflow exploits are not possible if the JVM implementation is bug free and can afford to perform on-line bytecode checking [16].

The Java Virtual Machine supports dynamic class (code) loading, providing a base mechanism for updating applications at run-time or loading needed components at start-up time. Conceptually this is not different from dynamic linker support present in most general purpose operating systems, except that safety is ensured: whereas a dynamic linker for C applications will not check the compatibility of various loaded modules, the Java class loader will prove that at worst, compatibility issues will be isolated and will not result in a failure of the system at large. In the COrDeT project funded by ESA [6], support for dynamic updates of software components at runtime has been identified as one of the requirements for future spacecraft on-board software. Dynamic class loading would therefore be an important feature, as RTEMS does not support dynamic libraries and it adds the aforementioned benefits comparing to dynamic linking. Java on-board software can thus in principle be updated at run-time.

Since Java programs conceptually all run on the same platform, the Java Virtual Machine, most of the portability issues are taken away from the application programmers. This is also known as WORA (Write Once Run Anywhere). In the case of space applications this has the nice feature that most of the code (except hardware-dependent parts) does not need to be developed and debugged in specialized, expensive and usually slow simulators, but instead in a fairly common desktop environment.

The actual portability of the Java applications then depends on how well the individual virtual machine implementations follow the specification. The API is part of the specification, so the programmers cannot accidentally use virtual machine specific features. However, it is their responsibility to make sure they only rely on the specified behavior of the API. With new Java releases, issues affecting portability in both the specification and implementations are being fixed – specification is being clarified, methods with unfixable portability issues deprecated and complemented by better ones. As a result, it is in general easier to write portable programs in Java than in languages like C or even C++, but still programs should be tested on multiple virtual machine implementations. Similar observations apply to the portability of Ada language, however, as Java is being recently used in more areas, the progress of portability issues clarification is swifter.

Despite the good supply of Java libraries, applications sometimes need to interact with native code, mainly to access low-level OS dependent functionality or native libraries. Java provides an interface to native code for applications that need it. Spacecraft on-board applications thus can potentially be formed by both native code and Java code. This could be exploited for instance in the development of Basic Software (Fig. 1).

According to an experimental study published in [11], programmers are more productive in Java than in C++ and they also create less bugs in Java than in C++. We are not aware of any similar study that would compare Java and Ada.

A language-level comparison of Ada and RT Java in the context of safety critical applications is given in [5]. Unlike Ada, dynamic memory allocation and automatic garbage collection are integral parts of standard Java.

3. CHALLENGES FOR SPACE READY RT JAVA

In order to provide suggestions for using RT Java in spacecraft on-board software, an analysis of this domain has to be taken into account. This work has been performed recently in projects funded by ESA, e.g. CorDeT [6] and DOMENG [7]. In the Real-Time Java Assessment Project [14] funded by ESA and carried out by SciSys with help by Charles University, we took into account high-level considerations only. We provided suggestions for applicability of RT Java for platform and payload applications based on our evaluation of selected RT Java products, which followed the guidelines drawn in [12]. We used a high-level spacecraft on-board software architecture as shown in Fig. 1.

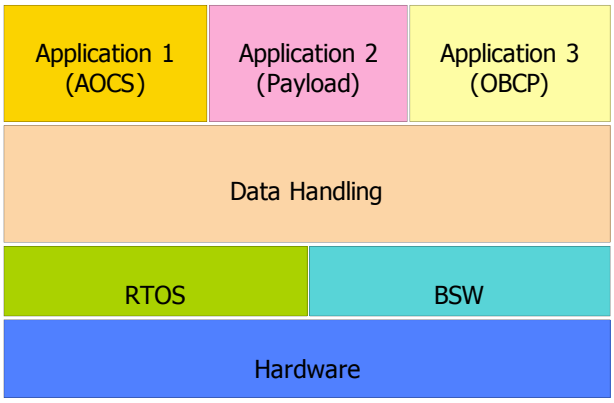

Figure 1. On-board software high-level architecture

Our benchmarking results in the project suggested that although RT Java system latencies on RTEMS/LEON were above acceptable limits, they were comparable to C on platforms for which RT Java implementations have been better optimised (e.g. Linux/x86). From the perspective of using RT Java for particular spacecraft on-board software applications, our summary was as follows:

- Basic Software:
  1. Adequate support in RT Java to access hardware is missing (e.g. interrupt handling);
  2. Too high system latencies (on the target RTEMS/LEON platform);
  3. It is worth considering using JNI or other mechanisms to access Basic Software written in C. The difference in performance between BSW written in Java on one side and written in C and accessed via JNI/other tools on the other side should be quantified.
- Data Handling:
  1. RT Java appears being close to provide all necessary functionality;
  2. Improvements in performance and predictability are needed on the target platform.
- AOCS:
  1. RT Java appears being close to provide all necessary functionality (attention must be paid to implementing the trigonometric functions with higher precision as in the java.lang.StrictMath library which should use algorithms compliant to the Freely Distributable Math Library (fdlibm), but also make a faster but less precise implementation relying on hardware support in java.lang.Math);
  2. Improvements in performance are yet needed.
- OBCP
  1. RT Java dynamic class loader combined with an interpreter provides the necessary functionality;
  2. Doubts remain as to whether using the Java interpreter as the OBCP interpreter is actually beneficial;
  3. In our experience, dynamically loaded interpreted code performs up to two orders of magnitude worse than the AOT-compiled code on the target platform. Just-in-time compilation is not a solution, as it requires huge amount of computation time and memory.
- Payload Software:
  1. No specific obstacles have been found; however all the general observations apply;
  2. Improvements in performance and predictability are needed on the target platform.

We complement these earlier results based on Purdue's experience with porting their Ovm [1] RT Java implementation to RTEMS/LEON and RTEMS/x86. In this paper we focus more on memory management, as this appears to be a critical aspect for real-time guarantees in Java.

In standard Java, the garbage collector could cause pause times as large as 100 ms or more. The garbage collector is however an integral part of Java and cannot be bypassed. An additional problem is that the base Java thread scheduler does not fully enforce priorities. Real-Time Specification for Java (RTSJ), an extension to standard Java, provides real-time scheduling features as well as different modes of using memory that bypass the garbage collector: scoped memory and immortal memory. The new programming model requires explicit distinction of these types of memory. Immortal memory is never released. Scoped memory is similar to stack-allocated local variables of a function: variables are allocated when a function is entered and freed when it is left. The difference is that in RTSJ a scope is orthogonal to a function: a scope is entered explicitly as well as it is explicitly left. To preserve type safety, references from outside a scope cannot point into a scope, as they would be turned invalid when the scope is left.

RTSJ has been implemented both in commercial and open-source virtual machines (Java RTS, WebSphere Real Time, PERC, Ovm, and JamaicaVM/AeroVM). It is known to be used in a battleship computing environment (US Navy Zumwalt-class Destroyer by Raytheon/IBM, 5 mil. lines of Java code, Aegis Weapon System Open Architecture Program [9,3]), avionics (Zedasoft's Java flight simulator, Boeing ScanEagle UAV [1], EADS Barracuda UAV), audio processing [2,10], industrial control [13], trading and visualization. However, to the best of our knowledge none of these systems uses scoped memory – in each application, developers opted to use either a provider-specific real-time garbage collector or statically allocated memory. This is largely due to the fact that scoped memory takes

away the simplicity of the base Java – explicit entering/leaving the scopes and keeping the invariant that references do not point to scopes from the outside introduces hassles that arguably make Java no better than C or C++. Real-time garbage collectors strive to avoid the need for scoped memory: although less efficient than the non-real-time ones, they currently provide pause times in hundreds of microseconds. Even shorter pause times seem theoretically possible. A real-time GC is a part of JVM implementations of IBM, Sun, Aicas, and Purdue's Ovm.

A unique restriction for spacecraft on-board software is the amount of memory. Due to extensive costs of radiation hardened RAM, future ESA systems are expected to have only 32M of RAM, although the hardware could support 128M. The limit of 32M is very restrictive for any realistic Java application. If we only focus on the heap, due to the use of GC, we typically need 3 times more memory than the live data really spans. It seems that space applications would benefit from a GC that would require less memory, most likely for the price of decreased performance. Compression techniques or just reduction of extensive aligning are obvious candidates to be explored, as well as modification of the GC algorithm to reduce the three-fold overhead. On the other hand, existing C or Ada space applications also incur space overheads due to the use of static memory allocation or memory pooling, which introduces an a priori fragmentation of the heap. The memory pool code is often heavily used by most subsystems of the on-board software and not enough attention is paid to its efficiency and flawlessness. Thus, it is possible that performance of existing RTGCs is already comparable with legacy space applications in memory usage.

Code size is also an issue with respect to memory usage – the future ESA hardware has (only) 8M of non-volatile memory in addition to the 32M of RAM. Even if we restrict the dynamic class loading to classes known at compilation/deployment time, the amount of code used just by the start-up of class libraries can be significant. Although much of the initialisation may not be used in a particular application execution, stripping down the Java features would make us lose most of the benefits of Java. Potential solutions to apply include rewriting the class libraries to initialise more lazily, while keeping real-time properties, as well as code/bytecode compression.

Lastly, the VM footprint itself can be significant. Non-RT JVMs implement just-in-time compilation, which is hardly applicable in small embedded devices due to limited memory and computational power. Ahead-of-time compilation is usually preferred. However, the binary code is significantly larger than the bytecode.

Space applications could greatly benefit from dynamic updating of a running application by loading code not present when the application was started. The built-in Java support for this represents much safer and technically more sound option comparing to the current practice of patching binary images of software applications. A sensible solution with respect to available memory would probably thus be based on ahead-of-time compilation of most of the classes and on interpreting bytecode of the dynamically loaded classes. Our experience from the RT Java Assessment project shows that we can identify pieces of on-board software where the performance penalty of the interpreted code is not significant (e.g. the time to perform a system call is – not surprisingly – comparable in both native and interpreted execution modes).

4. THE OVM EXPERIENCE

Purdue Ovm is a research JVM that implements most of the RTSJ specification and has a real-time garbage collector. It was primarily designed for Linux and worked on similar POSIX systems with x86, SPARC, and PowerPC processors. It can employ the Xenomai RT-Linux to expose hardware interrupts and I/O devices to Java programs. The core of Ovm has also been ported to RTEMS/LEON2, RTEMS/LEON3 and RTEMS/x86. We were able to run the SPEC JVM 98 compress benchmark on a system with 8M flash memory and 32M RAM.

Ovm provides an ahead-of-time compiler that compiles Java bytecode into C code, which is in turn compiled by GCC into machine code. The same compiler is used for the application bytecode, the Java libraries bytecode, and the bytecode of the VM runtime itself (the majority of the VM is implemented in Java). The use of C as the output language has the advantage of leaving some low-level optimizations (register allocation, instruction selection, redundant code elimination, some of inlining) to GCC as well as making the VM more portable. Additionally, the generated C code is relatively easy to debug.

As Ovm already had support for SPARC, porting to LEON2 and LEON3 CPUs did not require much work; only some calling convention details had to be resolved. On the other hand, porting to the RTEMS operating system required new threading control code, some I/O fixes, fixes to the boot process and extensions of the build system. The build system had to be extended for cross-compilation and for running code in an emulator. Because Ovm already used GNU Autotools, this did not require massive changes. The VM boot code had to be adapted for RTEMS memory layout. Ovm uses a pre-compiled memory image with the VM code, which has to be at fixed memory address. As RTEMS does not

have MMU, some relatively minor changes were needed to make this possible.

The port of Ovm to RTEMS brought to light limitations that have to be eliminated to make the use of Java for the development of spacecraft on-board software possible. The memory usage is still too large, due to both the binary (the VM code and pre-compiled class libraries) and heap requirements. The problem of the large binary cannot be alleviated through interpretation, as our real-time configuration of Ovm currently does not support bytecode interpretation (earlier versions of Ovm supported a simple interpreter which was used for debugging). Ovm has also a very limited support for interfacing C and Java code; this is both beneficial (full-fledged C-to-Java interfaces such as JNI tend to be quite large) and detrimental: end-users cannot quite easily write their own C code and link it with an Ovm program.

5. ROADMAP TO JAVA IN SPACE

Spacecraft on-board software could benefit from Java once Java is improved in certain aspects. Some of the suggested improvements are generally accepted as requirements for RT embedded Java systems – shorter RTGC pause times, better CPU utilization. Somewhat unforeseen are the stringent memory requirements due to hardened RAM (up to 8M for code, up to 32M for heap). An important factor is also a limited timer resolution due to limited clock rate, as compared to desktop systems. The limited timer resolution makes RTGC implementation more challenging.

In short time, it is unlikely that with current Java implementations the whole spacecraft on-board software could be written in Java. However, Java could be used as an isolation platform for software that has not been assigned the highest criticality, while the critical code would still be written in Ada or C. The Ada/C code will communicate with the Java code using well defined native interface, allowing both calls from Java to Ada/C and vice versa. Java could never bypass the native interface to corrupt or break the Ada/C code thanks to the Java type-safety. Indeed, the Ada/C code still would have to be robust against erroneous or adversary use of the native interfaces it would provide to Java. The Java code could be updated in flight (thanks to dynamic class loading). And indeed the Java code could use all mentioned Java features that make programming easier and safer.

This scenario of Java as an isolation platform would still require some engineering of the Java VM. The Java native interface should be fast and the Java VM, especially the RTGC, should not consume excessive resources when not in use. The dynamic class loading would require an interpreter with reasonable performance and limited memory requirements. A *logical partitioning* approach could be used to make the memory and CPU utilisation of Java under control. One possible solution to this is to use a separation microkernel such as PikeOS, which was recently selected for Securely Partitioning Spacecraft Computing Resources, a project funded by ESA [15]. In addition to the separation of Java control code from critical native code, this scenario can be employed for isolation of different Java control applications from one another.

Surprisingly, certain limitations of hardware for spacecraft on-board applications can be exploited by the JVM. In particular, SMP or multi-core systems are not expected to fly in the short term (ESA is only starting projects to investigate the consequences of the use of multi-core systems). The lack of real parallelism makes the VM implementation simpler and puts unique requirements for the GC as well. The extremely small heap size (up to 32M as opposed to gigabytes in desktop systems) would also impact decisions on GC design. Also, limitations of software, such as the limited SMP support found in RTEMS, can also be overcome by the VM; VMs typically implement much of the heavy-lifting required to support SMPs, potentially allowing parallel Java code to run on top of not fully parallel-aware operating systems.

Once the sandbox scenario is proven to work and the RT GC technology allows writing code with acceptable latencies, more and more parts of the space systems could be implemented in Java.

We also believe that Java design and coding patterns should be investigated for the real-time embedded systems development. For instance, our experiments with different implementations of the System Data Pool have shown that simple design issues such as method signatures could have significant influence on the overall throughput and response times. Attention has to be paid when Java exceptions or synchronisation are used.

6. CONCLUSIONS

In the future, we would like to focus on two parallel tracks.

One of them is developing a RT Java VM for RTEMS/LEON, which meets the requirements of spacecraft on-board software. Ovm can run on RTEMS/ LEON, but has two important drawbacks: does not presently support dynamic loading and it's footprint is too large. Adding dynamic loading to Ovm would be possible since it existed in earlier versions for debugging, but reducing the footprint will require significant changes or writing a new virtual machine from scratch. We would like to focus on Safety Critical

Java [6], an updated subset of the Real-Time Specification for Java and to explore optimizations reducing footprint in this context.

The second track is the evaluation of existing RT Java VMs for their suitability for spacecraft on-board software. This will include incremental building of code base of prototype on-board software in RT Java. Here we would like to build upon our efforts in the Real-Time Java Assessment project, as well as other studies. At the same time, this will include further benchmarking of different RT Java products such as Ovm, AeroVM and PERC Pico. We currently have a test suite available which we can easily port to the other VMs in order to compare their performance and predictability.

7. REFERENCES

1. Armbruster, A., Baker, J., Cunei, A., Flack, C., Holmes, D., Pizlo, F., Pla, E., Prochazka, M. & Vitek, J. (2007). A Real-Time Java with Applications in Avionics. *Trans. on Embedded Computing Sys.* **7**(1), 1-49.
2. Auerbach, J., Bacon, D. F., Bömers, F. & Cheng, P. (2007). Real-Time Music Synthesis in Java Using the Metronome Garbage Collector. In *Proc. International Computer Music Conference (ICMC)*.
3. Berry, R. F., McKenney, P. E. & Parr, F. N. (2008). Responsive Systems: An Introduction. *IBM Systems Journal*. **47**(2), 197-206.
4. Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S. & Turnbull, M. (2000). *The Real-Time Specification for Java*, Version 1.0.2, Addison-Wesley.
5. Brosgol, B. M. & Wellings, A. (2006). A Comparison of Ada and Real-Time Java for Safety-Critical Applications. Reliable Software Technologies – Ada-Europe 2006, LNCS 4006, Springer, 13-26.
6. Component-Oriented Development Techniques (COrDeT) (2007-2009), ESA/ESTEC Contract Nr 20464/06/NL/JD and ESA/ESTEC Contract Nr 20463/06/NL/JD.
7. Domain Engineering (DOMENG) (2007-2008), ESA/ ESTEC Contract Nr. 20001/06/NL/JD/jk.
8. Henties, T., Hunt, J. J., Locke, D., Nilsen, K., Schoeberl, M. & Vitek, J. (2009). Java for Safety-Critical Applications. In *Proc. 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCer)*, Elsevier.
9. IBM (2007). IBM and Raytheon Deliver Technology Solution for DDG 1000 Next Generation Navy Destroyers. Online at *http://www.ibm.com/press/us/en/pressrelease/21033.wss* (as of 18 May 2009).
10. Juillerat, N., Arisona S. M. & Schubinger-Banz, S. M. (2007). Real-Time, Low Latency Audio Processing in Java. In *Proc. International Computer Music Conference (ICMC)*, 99-102.
11. Phipps, G. (1999). Comparing observed bug and productivity rates for Java and C++. *Softw. Pract. Exper.* 29(4), 345-358.
12. Prochazka, M., Ward, R. & Wellings, A. (2007). A First Step towards Using Real-Time Java for Spacecraft On-board Software. In *Proc. Data Systems in Aerospace Conference (DASIA)* (Ed. L. Ouwehand), ESA SP-638 (CDROM), ESA Publications Division, European Space Agency, Noordwijk, The Netherlands.
13. Robertz, S. G., Henriksson, R., Nilsson, K., Blomdell, A. & Tarasov, I. (2007). Using Real-Time Java for Industrial Robot Control. In *Proc. 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, ACM, 104-110.
14. RT Java Assessment Project. (2008). Final Report, Contract ESTEC 20474/06/NL/JD/na.
15. Securely Partitioning Spacecraft Computing Resources Resources (2009-2010), ESA SoW TEC-SWS/08-163/SOW.
16. Vertanen, O. (2006). Java Type Confusion and Fault Attacks. In *Proc. 3rd Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, LNCS 4236, Springer, 237-25.