Java in Real-time Applications [1]

Edy Bertolissi and Clive Preece
School of Engineering, University Of Durham, South Road, Durham, DH1 3LE, U.K.
edy.bertolissi@durham.ac.uk clive.preece@durham.ac.uk

Abstract

This paper addresses the use of the Java programming language for real-time applications, giving an overview of its main advantages and current limitations. Java can implement at language level several of the constructs defined by the POSIX1.b standard for real-time applications. However, if Java is to realise its potential in this field, appropriate additions to the language will need to be made, and further work will be necessary.

I. Introduction

Java, a programming language developed by Sun Microsystems, represents a new candidate for a robust, architecture independent programming language. Although originally developed for software consumer electronics, it is becoming well known as the language for programming applets, small executable programs which can be embedded into an HTML page on the World Wide Web. Even if the main use of Java is related to the Internet world, it still retains the characteristics of a programming language for consumer electronics. The software for these types of applications must have special features; it must be able to be ported to different architectures and processors without changes even though the underlying hardware is up-graded, and it must be have a high degree of reliability to avoid failure of the device with possible hazardous consequences.

Java has several features which may be advantageous in a language to be used for real-time development. It is a simple, easy to learn and use, object oriented programming language. It defines only a limited number of language constructs which are similar to those used in C or C++. It has eliminated several of the features of C and C++ which tended to encourage programming errors such as pointers. It is designed to be suitable for use in distributed applications on networks, allowing easy access to remote resources. It is robust; it is a strongly typed language which performs several run time checks, such as that of array access within the bounds. It has built-in multi-thread support and the ability to load dynamically classes as they are needed. It has been developed with the aim of creating a programming environment which makes programming errors less likely to occur. It is designed as an interpreted language and this means that the code generated by the compiler is architecture independent, and can run on any system which implements the Java Virtual Machine. Recently, however, alternatives to interpreted Java have been developed, and Just in Time (JIT) compilers and Java compilers have been introduced. The language standard is maintained as Sun Microsystems reserves the right to alter it.

Java presents several new and positive features, and it is clear that it is much more than a language for adding animations to Web pages. The next sections will give an overview on the main advantages and current limitations in the use of Java in embedded and real-time systems and examples of projects involving real-time applications of Java will be discussed, together with comments on appropriate extensions to the language.

II. Java for the real world

In programs such as editors and compilers the element of time may not be critical to their operation, there are large classes of applications which are involved with time and deadlines, requiring to a greater or lesser degree some elements of real-time performance. There are two distinct classes of applications, those which interact just with the computer and its peripherals, and those which need to interact also with the world outside the computer. Examples of the first class of programs can be found in the increasingly expanding market of multimedia systems, teleconferencing and video games. Of the second class, aircraft control systems, or real-time digital diagnosis systems are typical applications which interact with the external world. These are often implemented as embedded systems on chips which form a component of the greater system.

One of the main advantages of Java consists in the fact that it is a platform independent language. Developers can write an application on one platform and then run it on any platform which implements a Java Virtual Machine. This eliminates the problems associated with cross-compiler development systems, multiple platform version maintenance and rewriting and retesting the software every time it is ported to a new processor [1]. This fact is extremely attractive since programs such as video games could be developed on one platform and then be ready to be commercialised for a whole range of computers. In the fast moving scene of embedded systems where new and improved chips are introduced frequently on the market, applications which could be simply ported from one chip to another offer immediate advantages. Old silicon could be quickly replaced without a large investment of effort in adapting the software to the new chip. The platform independence feature of Java does present some complications in time critical applications as developers may not be able

[1] Included in the Proceedings of the 10th IEEE Real Time Conference '97 Beaune, France, 22-26 September 1997

to test their applications in each environment where they are expected to run. A real-time application would have to run within the same constraints on platform with different features and speed (for example a 33 MHz Intel 40486 and a 300 MHz Digital Alpha) [1]. Moreover it is possible to run the application as a process on a time shared operating system where it would compete with the other tasks for the CPU time.

Clearly the expectation of using Java as the solution for implementing real-time programs which are able to meet the same deadlines on any platform is unreasonable. This is especially true when hard real-time performances are required. It cannot be assumed that a real-time application which just manages to achieve its goals on a fast machine such as a Sun Ultra would be able to provide the same behaviour on a Intel 486 platform. In many applications the development of a real-time system must be carried out closely allied to the underlying hardware since performance must be verified and guaranteed to meet constraints. It is important to recognise that a language that can run on any platform does not mean that it will be able to produce the same time performance on all of them.

The term Java is used to identify three different entities;

- Java the language: the programming language;
- Java the Bytecode: the outcome of the compilation of a Java program;
- Java the Virtual Machine (JavaVM): the application required to run the Java Bytecode on a specific platform.

While the first two entities are strictly defined by Sun Microsystems and any alteration in them would result in the creation of a non-standard dialect of Java, the Java Virtual Machine is not completely specified: a fact which leaves some freedom for implementation.

III. Real-time features

While the platform independence of Java is a great advantage of the language, there are other positive features which are desirable in a language which is to be used for real-time development.

First Java allows easier code development since it is object oriented, but without some of the complexity of C++. Its platform independence permits initial code testing and debugging on other host platforms before starting work on the target machine.

Another important advantage is in memory management. Java relies on automatic garbage collection while C and C++ generally require the programmer to free up unneeded memory. Memory leaks (and, worse, freeing an object which is still being referenced) are some of the most common programming errors.

Java's type safety is another feature which decreases the likelihood of programming errors. However it prevents direct access to hardware registers, as is done in C and C++ by the use of casts. Allowing Java to access the underlying hardware would produce two types of problems. First of all it would raise security issues, and second it would prejudice platform independent features. This could be seen as a problem in embedded applications where it is necessary to access the hardware. However Sun Microsystems has announced the development of a Java version for embedded systems, but at the time of writing full details have not yet been released. At present the only solution to this problem is to take advantage of the feature of Java which permits the inclusion of methods written in native code into Java programs. This means that it is possible to write procedures which access the hardware in some local native system language, typically C or C++, and include them in the Java application. These methods cannot provide the security guarantees typical of a Java program. Moreover native code cannot guarantee portability and any Java code that relies on native methods must be ported to each type of platform on which that code is to run.

Other useful Java features include its fixed-size longs, ints, shorts and bytes which allow the programmer to write more portable code, although the lack of unsigned types may be a problem in some cases.

At a higher level, Java allows change of the code "on the fly" since Java classes can be dynamically loaded into the application. This feature is especially useful for systems which need a lot of flexibility. At the present this capability is used for loading applets across the Internet. The ability to add new applets or update old ones on the fly is extremely powerful, and can add to the functionality of any system easily and quickly. Users can acquire new functionality or new user interfaces by simply connecting to the local server or to the network. This is becoming widely used in multimedia applications. However using the same facility in safety critical system might raise security issues. In addition the design of some embedded systems might require changes to the hardware to cope with new software features loaded in this way. It is nevertheless a powerful capability and one which could exploited by systems which have a server or network connection.

Java's built in support for multithreading and multitasking represents an advantage during the development of real-time applications. However the capability of predicting the behaviour of an application written in Java in a multitasking environment requires support from the run-time system. One possibility of ensuring predictable behaviour consists in developing a JavaVM which takes advantage of the features of a real-time operating system (RTOS). In this context Java with its VM is analogous to C or C++ running on an RTOS kernel. These other languages provide support for multitasking but not in the integrated way that it is done in Java. At present the developers of the RTOS QNX have expressed their intention to develop a JavaVM which will run on their operating system. However commercial RTOS's generally provide a wider range of features for process control than Java and its class libraries, and therefore some additional code would have to be written to make Java competitive in this market. Perhaps if web applets were not so important a part of current Java applications, this is a direction that the language might have taken. At present the standard POSIX.1b [2, 3] defines a set of

support functions for the development of real-time applications in real-time multitasking environments. If Java is to be used for developing real-time systems it has to provide a set of functions comparable to the ones defined in the standard. The files `<unistd.h>` and `<limits.h>` describe the operating system's POSIX support at compile time. Since Java is supposed to be machine independent the local configuration of the system should not be an issue of the application, but eventually only for the JavaVM. A function for checking the run time environment, like `sysconf` defined in POSIX, does not make sense in the Java language for the same reason. The problem of *namespace pollution* addressed in POSIX is eliminated in Java since there are no global variables or functions, and a unique package naming scheme that is based on the domain name of the organisation where the package has been developed has been proposed [4]. Java allows the creation of child processes by calling the method `Process.exec()`. The object returned by the `exec()` call provides methods to access the input, output and error streams of the child process. In addition methods for waiting for the process termination, killing it or retrieving its exit value are provided.

POSIX defines a series of mechanisms which allow the communication and co-ordination of processes; signals, messages, shared memory, and synchronisation. Signals are used for many purposes including exception handling, process notification of asynchronous event occurrence, process termination in abnormal circumstances, emulation of multitasking and interprocess communication. Java does not have any support for signals, but several of the tasks implemented by signals can be done in other ways in Java, for example by using the Java built in exception handling mechanism or threads. Signals for interprocess communication cannot be implemented easily (in any case signals are not an efficient and reliable interprocess communication system even if POSIX.1b gives them some useful features). Messages are used to pass information between processes in a more efficient way than signals. Three different mechanisms for message passing are available; pipes, FIFO and message queues. Pipes work like streams; one process simply writes data at one end of the pipe and the other one reads at the other end. A FIFO is simply a pipe with a name. A message queue is a more sophisticated version of a FIFO since it consists of a priority queue of discrete messages. In Java since it is possible to access the input and output streams of the child process, the implementation of a pipe is a simple operation. If it is necessary to prioritise the messages it is possible to create a thread in the sending process which sorts the messages and forwards the proper order to the receiving process.

The creation of named entities is a more problematic issue. In other situations, shared memory provides a low level for processes to communicate with each other. Unfortunately since Java is not able to address memory directly it is not possible to implement shared memory. A higher level solution for implementing shared memory would consist in declaring common segments of memory within a process. However this solution would rely on pipes for communication, which would defeat one of the main reasons of using shared memory, namely fast access to data. Since it is impossible to implement shared memory in Java at present the problem of synchronising multiple processes for accessing data will not be apparent. If a process needs to read data held in another process the second one can decide when to provide the information, thus eliminating the problem of generation of partially updated data. POSIX provides a set of functions for setting the scheduling policies of the processes. Since Java (the language) has no control over the OS where the application is running, it is not possible to define a similar set of facilities. A partial solution to this problem could be implemented at the level of the JavaVM. It could be written in such a way that it would be possible for it to define the scheduling policy and the priority of the process which is going to run when it is started. In this way no alteration of the language would be necessary, but the solution would not provide all the flexibility that the POSIX approach allows.

Java has access to the system clock (with millisecond resolution). It provides basic timer facilities which could be used inside the `Thread` class. These are not as sophisticated as the timers defined in POSIX, but they could provide sufficient support in many applications. Scheduling and timers are the means for making applications perform operations on time. However the OS may create some obstacles to letting this happen. The first problem is related to the performance of the OS. If the OS is too slow the only solution to this problem is to use a faster OS and/or platform. The second problem consists in the fact that many modern operating systems make extensive use of virtual memory. If not enough physical memory is present the system performs automatic paging moving parts of physical memory onto hard disk making new space available. In addition swapping may occur which involves dumping the entire process out to disk and reusing its physical memory. The operating system decides what and when paging or swapping occurs. The problem is that the time of accessing data which has been transferred to hard disk takes an order of magnitude greater in comparison with that when data is present in the physical memory, and therefore the expected response time cannot be guaranteed. The solution is to lock down data or the entire processes in memory. At the language level Java does not allow this. However again the solution for locking entire processes down in memory could be found at the level of the JavaVM which could give the OS the instruction to avoid the swapping out of memory of executed process. Data locking could be achieved by extending the meaning of the `volatile` keyword. At present a variable is defined as `volatile` if it changes asynchronously and therefore the compiler must read the variable's value from memory every time and not attempt to save its value in a register. The JavaVM could extend this definition by locking down in memory data which is defined as `volatile`.

POSIX defines support for synchronised I/O. When I/O operations are synchronised with the underlying device it means that they return only when the device is appropriately updated. This is not always the case since several systems for efficiency reasons use a buffer cache to hold I/O for later flushing on disk. This provides an increase in performance.

These functions are not present in Java, but could be implemented at the level of the JavaVM. Unfortunately in this case all the I/O operations (or classes of them) would be treated as synchronised with associated performance problems. Since I/O synchronised methods are necessary to access devices, the inclusion of a method written in native code using the POSIX features would be a better solution. Finally POSIX provides support of asynchronous I/O, which means that I/O operations are executed in parallel with the application. Java allows this to be achieved using threads.

At the present Java lacks of some of the features required by the POSIX standard, but some of the problems could be solved modifying the JavaVM. However Ken Arnold and James Gosling, the creators of Java, write [5]:

> Each implementation of Java must provide one or more appropriate extended classes of `Process` that are able to interact with processes on the underlying system. Such classes might have extended functionality that would be useful for programming on the underlying system. The local documentation should contain information on this extended functionality.

Therefore it is possible to expect some developments in the future in this part of the language, which may however undermine its platform independence.

There are other problems connected with Java which are shared by any type of application designed to achieve real-time goals. Overall performance is an important issue. Real-time is not concerned with execution speed of programs as such, but faster execution allows deadlines to be met more easily. The interpretation of the Java Bytecode makes sense on Internet applications which must run on any platform, but it is generally undesirable in other cases. However Java need not necessarily be used as an interpreted language, even if at present implementations are based on interpreters. There are other two possible alternatives which increase performance. The first option involves the precompilation of Java to native machine code. After a product has been developed, the Java code is compiled directly to native machine code prior to shipping. At present there is only one example of Java compiler, Toba [6], developed at the University of Arizona, but it is likely that their number will increase soon. The second option is to apply Just In Time (JIT) compilation. This type of compilation, known also as “on the fly” compilation, works in the following way. Once a Bytecode has been loaded into a particular virtual machine environment it can be translated at run time (by the virtual machine environment) into the host machine’s target instruction set just before the code is actually executed. An example of such type of JavaVM has been developed with the Kaffe VM [7]. The use of a JavaVM is problematic for embedded applications since they are often are implemented on small cheap and power frugal chips. Memory on these systems is usually an expensive option which is always kept to the minimum. Since the code of the JavaVM can be measured in hundreds of kilobytes its use is not a practical option in small applications. The size of the JavaVM itself is of the order of 64 Kbytes, but standard class libraries occupy a large amount of memory. It is possible to think of implementing a JavaVM for embedded use which does not include the libraries since they are not part of the language, thus reducing the size of the interpreter. The implementation of a JIT compiler is a feasible approach, but only for larger systems since it requires more memory to allocate the Bytecode, the JIT compiler and the translated native code, in comparison with an interpreter or the native code approach. For applications targeted to be executed on small dedicated systems it is most likely that the best solution is to compile to native code, while for applications which run on more powerful machines the use of JIT compilers might be the preferred option since they do not require the developers to compile the code individually for the specific platforms. At the time Java was launched, Sun Microsystems announced the introduction of a series of Java chips: microprocessors with different features, which could run Java Bytecode without the need of a JavaVM. Unfortunately Sun has not released the full details on these chips and therefore their assessment is difficult. However the idea did not find enthusiasts in the area of embedded system since it is not clear how would be possible to develop device drivers (which need to address the hardware) using the current version of Java. Moreover at the high end chip market it is difficult to see the advantages of a Java chip since Java applications could be implemented using a JIT or recompiler. Even if Java chips provide a good performance, they will still have to compete with the economies of scale of the general-purpose microprocessor industry and in the past similar approaches have proved not to be successful (e.g. the SOAR chips).

At present Newmonics Inc. is developing an ambitious project for a clean-room implementation of Java especially designed for reliable embedded real-time systems [8]. The proposed JavaVM implementation, called PERC, would allow loading and analysing of the Java code at run time. When a new real time activity is loaded a configuration manager would assess the requirements of the code (execution time, memory requirements etc.). Then a resource negotiator would analyse the data from the configuration manager and decide if the new application can be accommodated. If this is not possible, other applications would be terminated to free resources for the new one. If it is not possible to terminate any of the active applications the problem would be reported. The activities related to the timing analysis of the code and its schedulability performed by the PERC are complex and many people have raised doubts on the feasibility of the project. It seems that the company aims as a first step to implement a JavaVM for soft real-time applications and then to develop a hard real-time version. However the majority of the critics of the project question the decision of the developers to introduce two new keywords; the `atomic` statement for providing a new type of synchronisation mechanism and the `timed` statement for defining the upper bound of CPU time allowed to the code included in this statement. The reasons of their introduction are to provide greater efficiency in comparison with `synchronised` statements, to make the code more

readable, and to simplify the development of the Bytecode verifier. It is believed that these extensions to the language are dangerous since they will create a non-portable Java dialect. They also seem unnecessary since it would be possible to develop class libraries with equivalent semantics, which used in conjunction with a run time environment able to interpret their semantics, could provide the same functionality. The company recognises this since it plans to develop a compiler which would translate extended code in plain Java (code or Bytecode), however it is not known if it would be possible to achieve the same performance.

Another example of development of Java classes for the support of real-time applications has been done by James Young at the University of Berkeley [9]. He has developed support for "real-time tasks", which are directly analogous to threads, except that they have attached to them some additional timing information related to worst case execution, expected execution time, and deadlines. It is possible to specify hard real-time, soft real-time and non real-time tasks. Tasks are co-ordinated by the JavaRealTime executive which chooses to accept new tasks or not. Unfortunately sometimes browsers and JavaVMs running applications which make use of JavaRealTime terminate themselves. The package has been proved to be very reliable on some platforms (such as Pentium 133 with Win95, using Sun JDK 1.0.2), but immediately kills the JavaVM on other platforms (for example DEC Alpha, using Netscape 2.02). In other cases the behaviour is inconsistent and depends on the runs. Young presumes that this may be due to the combination of bugs in the code of the JavaRealTime and bugs in Java VM implementations on the vendors' part.

IV. CONCLUSIONS

Java offers several interesting features for the development of real-time applications. However it cannot yet be considered as the language which will resolve all the problems related to real-time programming. It is important here to distinguish Java the language from the other parts of Java implementation. As a language, Java has great advantages over C and C++ in terms of safety and simplicity, but it lacks of their flexibility in interfacing with the hardware. This means that at present Java is more suitable for applications which do not require interaction with external devices (multimedia, games). The future of the Java chips is not clear, we need to know their exact potential and limitations. Garbage collection is a problem for small applications, but the knowledge required to implement real-time garbage collectors is available at present. The built-in support for threads allows simple development of concurrent applications. The language does not have all the features which are defined in the POSIX1.b standard, but several of them can be implemented at language level, and others at the level of the JavaVM. The performance of the interpreted version of Java is currently slow, but it is rapidly improving with the introduction of native and JIT compilers. There are already in the literature some examples of projects using Java in real-time applications, but they are still at the development stage. In any case the introduction of new keywords in the language for the support of real-time features should be done by Sun Microsystems in order to eliminate the problems of introducing non-standard Java dialects. Extensions to, and implementation of a JavaVM designed for the real-time time world would be a more satisfactory alternative in order to allow Java to achieve real-time performance. The recent announcement of a version of Java for embedded suggests that Sun Microsystems is still pursuing the application of Java in consumer electronics, and that a certain level of real-time support is likely to be included in future releases of the language.

V. REFERENCES

[1] K. Nilsen, "Embedded real-time development in the Java language," in *Embedded Systems Conference East*, (Boston), April 1996.

[2] IEEE, *POSIX.1b; Application Program Interface - Real Time Extensions*, 1995. Draft Version.

[3] B. O. Gallmeister, *POSIX.4; Programming for the Real World*. O'Reilly & Associates Inc., 1995.

[4] D. Flanagan, *Java in a Nutshell*. O'Reilly & Associates, Inc, 1996.

[5] K. Arnold and J. Gosling, *The Java Programming Language*. Addison Wesley, 1996.

[6] T. A. Probsting, P. B. G. Townsend, J. H. H. Newsham, and S. Watterson, "Toba: Java for Applications, A way Ahead of Time (WAT) Compiler," in *Proceedings of the 3rd Conference on OO Technologies and Systems*, 1997.

[7] J. Project, "Kaffe: A free virtual machine to run Java code." available from http://www.kaffe.org/, 1997.

[8] K. Nilsen, "Issues in the design and implementation of real-time Java," *Java Developer's Journal*, 1996.

[9] J. S. Young, "Integration of a real-time programming system for dynamic reactive systems in an object-oriented language," tech. rep., Departement of EE and CS, University of California, Berkeley, 1997. available at: http://www-cad.eecs.berkeley.edu/~jimy/java/JavaRealTimeReport.pdf.