An Approach for a Dependable Java Embedded Environment

Gilbert Cabillic Salam Majoul Jean-Philippe Lesot Michel Banâtre
IRISA
Campus Universitaire de Beaulieu
35042 Rennes Cedex
Gilbert.Cabillic@irisa.fr

Abstract

A Java Execution Environment (JEE) presents lots of advantages for embedded architectures. In this paper, we present an approach to build a dependable Java execution environment for Wireless PDAs. It is based on a modular software architecture. Our on-going works focus on reducing software errors, increasing the security of the software and minimizing native software pieces of code to increase the overall stability of the platform by transferring operating system features in Java world.

1. Introduction

A Java Execution Environment (JEE) presents several advantages for embedded architectures. First, Java applications can be dynamically downloaded. Second, as Java bytecode is an intermediate code, the application can be distributed everywhere. Moreover, a Java application cannot explicitly manage memory access because Java bytecode provides no way to express and manipulate pointers in Java world. This increases the stability of all the Java world by avoiding memory access errors due to software faults or illegal intrusion. Of course, this stability depends on the JEE (Java virtual machine (JVM), and APIs implementation), and also on the underlying operating system.

The software architecture we consider is presented in figure 1. Our Java platform (Java virtual machine and Java APIs), named Scratchy [3], runs on the hardware through an operating system. Scratchy is executed in several OS threads in one common space address. Scratchy has the ability to execute several applications at the same time and a scheduler inside Scratchy undertakes the scheduling of all Java threads. Native code, independent of Java world like protocol layers or driver daemons, is executed in other OS threads thanks to the help of the operating system.

In this paper, we present an approach to build a dependable Java execution environment for Wireless PDAs. It is organized as follows. First, we present our Scratchy modular software architecture. Then, features of our development environment for Scratchy are mentioned as well as some evaluations. Finally, we present our perspective works to increase the dependability of our Java execution environment.

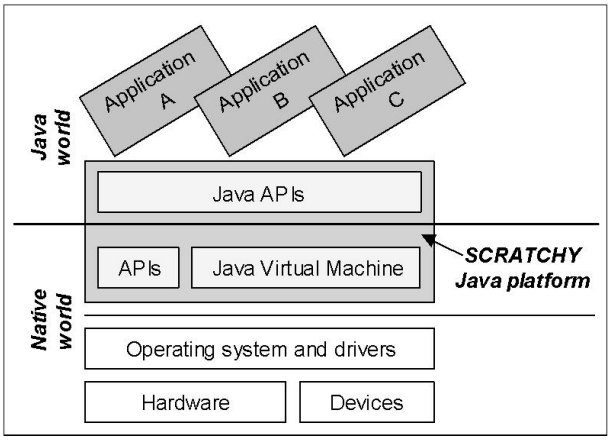

Figure 1. Global software architecture.

2. Scratchy Software Architecture

Memory volume, execution time and energy consumption are critical resources for embedded systems. A flexible Java environment providing a good tradeoff between these resources is required for wireless PDA. Only a modular approach, whose benefits have been already proven in software design, allows to achieve such a tradeoff. In fact, through modularity it is possible to specialize some parts of the JVM for a specific processor by exploiting, for instance, low power features for DSP. With a monolithic-programmed Java environment, it is difficult to change at many levels hardware resource management without rewriting the JVM from scratch for each platform.

2.1 Our modular approach

To reach these goals we designed a modular approach described in the following. A module contains services (functions) and data type structures. Pre-hooks and post-hooks can be attached to a service. The formers are called before a service call to undertake for example resource reservation and allocation. Post-hooks are called after a service call for instance to free the resources and manage errors.

The Scratchy Development Environment (SDE), presented below, is designed to achieve modularity for our Java environment.

2.2 Scratchy Development Environment

This tool is designed to optimize time or memory overhead on the outcome source and to be the most language and compiler independent as possible. SDE takes four inputs:

- a global specification file describes services and data types by using an Interface Definition Language (IDL);
- a set of modules implements services and data types in one of supported language mappings;
- an implementation file inside each module describes the link between specification and implementation;
- target hardware descriptions indicate alignment constraints with their respective access cost for each target processor.

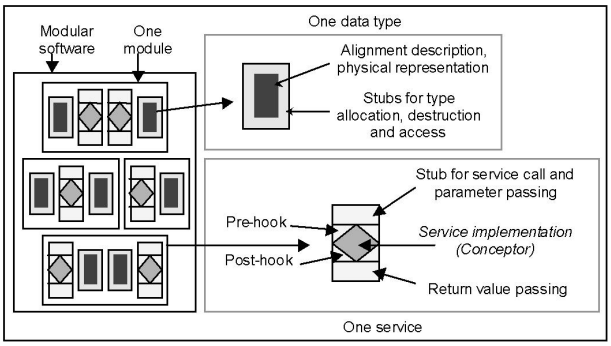

Figure 2. Our modular approach.

SDE chooses a subset of modules according to the targeted hardware criteria. It generates, as shown in figure 2, stubs for services, sets structures of data types, generates functions to dynamically allocate and access data types, etc. SDE works at source level, so it is the responsibility of compilers or preprocessors (through in-lining for example) to optimize a source which has potentially no overhead.

2.3 Some evaluations

The current version of SDE supports the C language as language mapping and generates stubs for services and data type structures in that language.

In order to evaluate memory expansion of SDE data type structures, their memory allocation time and service execution time, we compared the generated stubs to an equivalent program written in C. For this purpose, we compared the assembler codes obtained after optimization for both programs. We used benchmarks when the assembler codes were not comparable. The results obtained were very promising as the generated stubs introduce an insignificant overhead and in some cases no-overhead at all.

For example, the assembler codes of a C program calling the generated stub for a service without parameter and the C program calling directly the C function implementing that service are identical. When the service is specified with parameters, the assembler code for the SDE program contains only one supplementary statement relating to a shift of the stack pointer.

We made some benchmarks to evaluate data type structures generated by SDE. We used a Pentium II processor under Linux system and using a Gnu C compiler and optimizer. As an example, figure 3 shows the time required to allocate an array generated by SDE and an equivalent C array. The time measured to access an array element is the same in both cases.

Array size	C Program (cycles)	Using SDE (cycles)	Overhead (cycles)
20	229	228	+1
80	228	228	0
280	228	228	0
400	228	228	0

Figure 3. Array allocation time comparison.

2.4 Related works

Our modular approach is close to aspect-oriented programming (AOP) that relies on a separation phase of different aspects of a problem and a composition phase. AspectJ [5] is an example of AOP language. It provides some constructs similar to that proposed by SDE such as pre and post-conditions that act like our pre and post-hooks. Another tool that can be compared to SDE is Knit [8] which provides its own language to describe linking requirements. However, both AspectJ and Knit are too limited for our goals because they manage only method/function calls, and not data type. Management of data type is useful to optimize memory consumption and data access performance especially on shared memory multiprocessor hardware.

2.5 Implementation

Scratchy is our modular implementation of the JVM which is written from scratch in order to validate our modular approach. It relies on a CLDC [10] specification compliant. However, Scratchy is closer to CDC than to CLDC because it supports floating-point operations and Java Native Interface. Pentium, ARM and a TI DSP TMS320C55x [11] targets are supported by our JVM as well as a bi-processor Pentium based architecture as mentioned in [3]. To make the link with an operating system, we designed a module named middleware which supports not only a Posix general operating system (Linux, Windows 2000) but also a real-time operating system with Posix compatibility (VxWorks).

3 Perspective works

The stability of the overall platform depends on 3 different points. First, Java applications may contain software errors either deliberately introduced in case of intrusion, or not in case of erroneous code. As all explicit memory manipulations are forbidden, only the involved application is affected when a software error occurs. A software exception (Java language) can be sent and in the worst case, when the error cannot be recovered, the application is killed by the JEE. A Java application can disrupt the global software architecture when its requirements in terms of resources are too important (memory, CPU, network) or when the application uses an API containing a potential software error that could damage devices or lead to a memory error access. Second, the JEE itself can contains software errors in Java or native code part. Third, the operating system and drivers can be instable.

In order to increase the JEE stability, our perspective work relies on two axes both different and complementary. The first one is based on our modular approach, the second one consists in minimizing native code part of the execution environment by transferring it in Java language.

3.1 Exploiting modularity

Besides the ability to specialize some parts of the JEE, the modular approach can be exploited in several ways.

3.1.1 Reducing software errors

In order to reduce software errors the JEE can contain, it is possible to design integrity tests for each service (or a subset) of a module, for a subset of modules and for the whole JEE. This will increase the stability of the overall software. We already have a test suite for Scratchy and we are currently designing more service tests.

Thanks to the automatic generation of call stubs between services, it is also possible to introduce tests to dynamically check whether data pointers are correctly initialized.

At last, because SDE generates data type representation, we can assign to each type a magic word (see figure 4) and check dynamically whether every pointer corresponds to the correct kind of data type (for parameter passing or when accessing a structure). Of course a memory expansion and an execution time overhead are introduced, but this feature of SDE is only used to stabilize the JEE.

3.1.2 Transferring operating system code in modules

As SDE provides a good support to increase the reliability of the Java execution environment software, we could transfer some operating system code parts in modules. Of course, some extensions to SDE need to be done because the current version of SDE is specific to JEE generation. At present, we are working on a small operating system designed to execute Scratchy. This small OS provides basic services for memory, file system and device management.

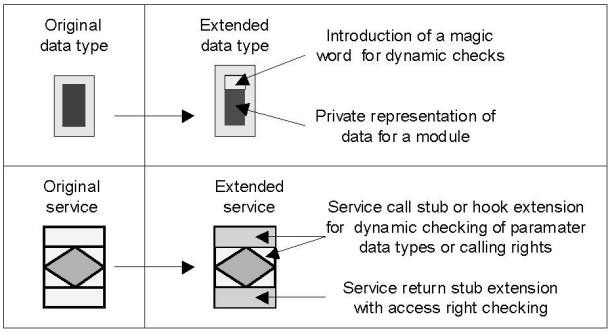

Figure 4. Extending SDE generation.

3.1.3 Increasing security

Several degrees of security, rights or confinement policies can be introduced at the module level and/or at the service level (see figure 4). Each module and/or service could specify the security according to its need. Security policies can be realized in several ways:

- dynamic check of rights before calling a service or using a data type structure belonging to the module. For example, a security policy can be introduced in order to dynamically check rights of native method invocation,
- use of an encryption protocol using keys to communicate parameters between services,
- use of a private representation of data in a module. For example all data of a module could be crypt and only

services belonging to this module can understand the representation.

3.2 Minimizing native code

The second perspective of our approach consists in transferring some native code of the JEE software in Java language aiming at exploiting the software confinement features of Java to increase the overall stability of the platform. At the opposite of current Java operating system projects [2] like J-Kernel, we do not want to enable the direct use of pointers in Java. So, our approach is to provide a java view of low-level resources without any pointer abstraction.

To validate that point, we have exported the native scheduler of Scratchy in Java. Hence, we designed an abstraction for interruption management and threads. The interface *OsInterrupt* specifies an interruption handler and provides a method called handler which is invoked when an interruption occurs. An implementation of this interface can be given at any interruption level. Each implementation has to provide its running parameters through Java fields. For example, a keyboard interruption implementation must define a field representing the value of the pressed key. This value is set using a minimum native interruption handler. In this way, the access of low-level IO control ports of the device is not needed and we still guaranty that no pointer is needed to write any interruption handler in Java side. The class *OsTimerInterrupt* is designed to be associated to the low level timer interrupt and enable the scheduler to realize its policy.

The *OsThread* class abstracts a Java thread context and provides the scheduler a *setThreadActive* method to execute its elected *OsThread* on the CPU. The scheduler performs a round-robin policy that manages all Java threads. It was very easy to be implemented and we showed that it is possible to transfer native code of the JEE in Java world. Our ongoing work focuses on transferring more complex JEE native software (including drivers) like graphics native APIs, sound APIs and also the garbage collector, in order to obtain the smallest native operating system needed for a JEE.

4 Conclusions

Our direction to build a dependable Java execution environment relies on two approaches. First, we exploit and extend the modular features of our JEE. Second, we minimize pieces of native code. This direction is valid only if complementary works aiming at eliminating Java software faults by using other languages or compilation techniques are made and if an effort to low-level design techniques like [6] are done.

At last, transferring operating system features in Java world is conceivable only if the JEE provides good performance. Recent works made in this area [7, 1, 4, 9] are promising on that point and it is clear that Java has a place in embedded environments in the future.

References

[1] Ajile. Overview of ajile java processors. Technical report, Ajile (http://www.ajile.com), 2000.

[2] G. Black, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau. Techniques for the design of java operating systems. In *proc. of the USENIX Annual Technical Conference*, June 2000.

[3] G. Cabillic, J. Lesot, M. Banâtre, F. Parain, T. Higuera, and V. Issarny. *The Application of Programmable DSPs in mobile Communications*, chapter A Flexible Java Environment for Wireless PDA Architectures based on DSP Technology, pages 119–135. WILEY press, december 2001.

[4] Imsys. Overview of cjip processor. Technical report, Imsys (http://www.imsys.se), 2001.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *proc. of the 15th European Conference of Object Oriented Programming (ECOOP)*, Budapest, Hungary, June 2001.

[6] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An idl for hardware programming. In *proc. of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, San Diego, CA, October 2000.

[7] Nazomi. Boosting the performance of java software. Technical report, Ajile (http://www.ajile.com), 2002.

[8] A. Reid, M. Flatt, L. Soller, J. Lepreau, and E. Eide. Knit: Component composition for system software. In *proc. of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 347–360, San Diego, CA, October 2000.

[9] T. Suganuma, T. Ogasawara, T. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journals*, 39(1), February 2000. Java Performance Issue.

[10] Sun Microsystems. *CLDC and the K Virtual Machine (KVM)*, 2000.

[11] Texas Instruments. *TMS320C5x User's Guide*.