Ahead of time deployment in ROM of a Java-OS

Kévin Marquet, Alexandre Courbot, Gilles Grimaud, David Simplot-Ryl

Ahead of Time Deployment in ROM of a Java-OS

Kevin Marquet, Alexandre Courbot, and Gilles Grimaud

IRCICA/LIFL, University of Lille I, France.
INRIA Futurs, POPS research group.
{Kevin.Marquet, Alexandre.Courbot, Gilles.Grimaud}@lifl.fr

Abstract. This article shows how it is possible to place a great part of a Java system in read-only memory in order to fit with the requirements of tiny devices. Java systems for such devices are commonly deployed off-board, then embedded on the target device in a ready-to-run form. Our approach is to go as far as possible in this deployment, in order to maximize the amount of data placed in read-only memory. Doing so, we are also able to reduce the overall size of the system.

1 Introduction

Deploying a Java system dedicated to be embedded into a tiny device such as a sensor involves producing a ready-to-run binary image of it. This binary image is later burnt into a persistent memory of the device, usually Read-Only Memory (ROM), to produce the initial state of the system on the device.

In addition to ROM, tiny devices include several types of writable memories such as RAM, EEPROM, or Flash memory. All these memories have different access times, physical space requirements, and financial costs. For instance, ROM is very cheap and takes few physical space on the silicon, which usually makes it this memory the most significant one in terms of quantity; but it cannot be erased. Writable memories on the other hand are a rare ressource because of their cost and physical footprint.

The memory mapping of data into these different memories is computed off-board, when producing the binary image. A correct placement of the system data at that time is critical for embedded systems. In a domain where the software and hardware productions are tightly tied, placing more data in ROM can divide the final cost of the device and makes the other writable memories available for run-time computations.

Our approach is to go as far as possible in the off-line deployment of the system to maximize ROM usage while decreasing the overall size of the system. We operate at different steps of the deployment process. For each step, we measure the amount of data that can safely be placed in ROM, as well as the overall size of the system, thus obtaining an evolution of these two measurements all along the deployment. Our experiments have been performed on the Java In The Small (JITS[1]) Java-OS toolkit.

The remainder of this paper is organized as follows. Section 2 presents some work related to this paper. Section 3 then introduces the issues related to the placement in ROM of an embedded Java system. The deployment scheme of a JITS system is then briefly described in section 4. In particular, we detail the steps that are important for maximizing ROM placement and reducing the size of the final system. Section 5 details our results, by showing the amount of data it is possible to put in ROM and the size of the system for every deployment step. Finally, we conclude on our results.

2 Related Work

Embedding Java systems into tiny devices while minimizing their size has been studied using different approaches. Rayside [2] and Tip [3]'s approach is to extract the minimal necessary subset to run an application from a Java library. They use abstract interpretation to determine classes, fields and methods that may be used by the application and discard the rest. JITS uses a similar mechanism to extract the needed parts of the library and core system according to the threads that are being deployed. Squawk [4] is a CLDC-compliant implementation of the Java virtual machine targeted at next-generation smart cards. It uses an indirection table containing the references of all objects. This implies a run-time performance reduction, and the use of a part of the writable memory to store this table. Java 2, Micro Edition [5] is a stripped-down specification of the Java platform for small devices. It includes JavaCodeCompact, a class pre-loader and pre-linker that allows classes to be linked with the virtual machine.

These works doesn't take into account the specifics of the *physical type* of memory that tiny devices use. In particular, such devices generally include a high quantity of read-only memory. This important parameter is at the heart of our deployment approach.

3 Placing Data in ROM

A tiny device of the range of the smart card includes different kinds of memories. Their respective cost and physical footprint properties lead to the following proportions: about hundreds of kilobytes of ROM, dozens of kilobytes of persistent, writable memory and kilobyte(s) of RAM. Larger, more expensive devices can embed more memory - but these proportions are usually respected.

In a traditional Java Virtual Machine (JVM), the loading of applications is clearly defined: classes must be loaded, linked, and initialized [6]. In the case of an embedded Java-OS, these phases can be made partly during the off-line deployment, in order to embed a partially deployed system [7]. This results in a faster start up of the system, but it is also possible to take advantage of various steps during the deployment to increase the amount of data placed into ROM, as well as reducing the size of the system. Indeed, some Java objects involved in the deployment process reach their definitive form during these steps, and can

then be considered as immutable. Others are just useful to initialize the system and can be removed.

As placing objects in ROM prevents any further modifications of them, it is impossible to place an object that the system needs to change at run-time in a read-only memory. This leads to the definition of immutability of an object [8], in relationship to the semantics of the program: *an object is immutable if it is never modified by the code of the program.* Our approach is to detect all immutable objects and to place them in ROM.

Among the objects that are needed at run-time, some are always immutable. Their immutability does not depend on the deployment process. For instance, the `String` objects and their associated character arrays are objects that can never be modified. Other objects are created during the loading process and are either not modified or even not used at run-time.

The next section describes the deployment process of JITS. It details how it is possible, for each step of the deployment process, to increase the number of objects in ROM and to decrease the size of the overall system.

4 Deploying Embedded Java

Before being embedded into a small device, a Java-OS is deployed off-board. All the initializations that have not been made during this phase are performed when the device starts up, in order to place the system in a state where it is ready to run applications. While these operations are made at run-time in a traditional JVM (section 4.1), they can also be performed during the offline deployment of the embedded Java-OS in order to improve the size of the system and the amount of data in ROM (section 4.2).

4.1 Java Class Loading Process

A Java platform initializes itself before being able to run applications. In particular, classes must go through different loading states described by the Java virtual machine specification [6] before being ready to use.

Loading During this phase, the class structure is read (from a stream on a file or a network connection for instance) and the internal structures for classes, methods and fields are created. All the external references are still symbolic. Classes are marked as `LOADED` after this step.

Linking The linking step transforms the external symbolic references into direct ones. This step can either be performed once and for all (all the symbolic references are resolved, which involves loading the classes that are referenced) or just-in-time during runtime (each reference is linked when the bytecode interpreter meets it for the first time). During this phase, methods are modified in order to replace the non-linked bytecodes with linked counterparts.

Initializing Before being ready to run, the static statements of the classes must be executed. Once this phase is performed, the class is granted state `READY`.

This class loading scheme is tightly linked to the upper-level application deployment process.

4.2 Application Deployment Process

The pre-deployment phase of JITS is able to perform the class loading process. At each step, useless objects can be removed and some others considered as immutable. All these steps are performed by a tool called *romizer*). As the JITS *romizer* initializes the system before producing a binary image of it, it differs from JavaCodeCompact (JCC, [9]), the J2ME deployment tool, which only loads and links Java classes and lets the initializations be made at run-time.

Loading After this step, apart from objects that are always immutable such as strings, very few objects can be placed in ROM. Bytecodes contained in the code associated to a method are subject to modification during linking. However, if the code associated to a method does not contain any mutable bytecode, this code is immutable. In the same way, few objects can be considered useless after this step. Only the objects used to read the class from the streams can be removed, as well as all the information that has been extracted from them. The associated useless classes can be removed as well. A previous study [10] has shown that the constant pools of the classes can be compacted during this phase.

Linking After the linking phase, all external references from the bytecode are resolved, and more parts of the code can thus be considered immutable. All methods are linked which makes them immutable as well. The `LINKED` state also constitutes another important stage regarding the lifetime of classes: at this state, all objects referenced by classes can be placed in ROM excepted the static zones which can be modified at run-time. The classes themselves can be considered immutable if their states is stored outside of them.

Initializing In addition to the loading and linking phases, the JITS romizer is able to execute the static statements of the classes. This avoids spending time to execute the static statements at run-time but also allows to placethe immutable objects attached to a static field in read-only memory. Although there are few objects that are concerned by this in the Java libraries, applications are more subject to use them.

Applications Initialization Once the static statements are executed, our tool instantiates the threads that will begin to run when the device starts up. Doing this during deployment brings one benefit: it is possible to make a static abstract interpretation [11] of the code in order to detect the parts of the deployed system that will be used at run-time. Our approach at this level is the same as in [2] and [12]. We extract a subset of the system containing only the libraries needed for the system. From the `run` method of the selected threads, we perform a

depth-through analysis of the code in order to list all the classes, methods and fields used, discarding the others. Our static analysis makes use of constant value propagation in order to compute a precise control flow graph and detect more unused objects. In addition to the removal of these objects, it is possible to remove their references in the static zones in order to compress them.

The use of a specific installation process such as Java Card or OSGI would allow even better results, mainly placing more objects in read-only memory. Indeed, the installation functions allow to go further in the deployment process. However, we intend to provide a Java platform that is able to load and execute traditional Java applications and not only applications in a specific format.

System projection In order to transform the off-board deployed system into its binary image, the romizer builds the dependency graph of all the objects of the system. All the objects it contains are walked through in order to assign them a destination memory. This computation is made thanks to the the properties of objects (such as types and values) that are retrieved from the graph of objects. This permits for instance to identify all objects whose type is `Class` and whose field value `state` is `READY` as objects that are immutable. This computation also provides the relationships between objects. For example, all the objects attached to the `staticZone` field of an instance of `Class` must be described in the memory mapping as objects that must be placed in writable memory. Building the graph of objects is also an elegant way to discard useless objects. Indeed, the references to these objects are broken, thus making them unreachable and garbage collectable when the graph is built.

All along the loading process, some objects become immutable and others become useless. Next section shows the evolution of the amount of data in ROM and the overall size of the system at each step of the deployment.

5 Results

This section measures the benefits of deploying the system off-board. The size of the system and the amount of data in ROM are measured after each step of the deployment. The parts of the OS written in native code are not included in our measurements, although it will eventually be executed from a read-only memory. Three applications are measured. The first one is a basic *Hello World* application which shows the memory footprint of a minimal application. The second one is Sun's *AllRichards* which executes seven versions of the Richard scheduling algorithm. This application is interesting because it includes a high number of classes (76). Finally, the well-known *Dhrystone* benchmark is measured, showing quite different results because it uses several static data structures.

Details concerning the impact on the run-time features are also discussed.

5.1 Loading

In addition to objects that are immutable as soon as they are instantiated (strings), the majority of objects become immutable once the classes are loaded.

Excepted objects that are immutable as soon as they are created (as strings), the majority of objects become immutable once the classes are loaded. All methods containing a bytecode that will be changed during the linking phase are mutable. However, parts of code that do not contain such bytecodes can be placed in ROM after this step; 9% are concerned for AllRichards. The compaction of the constant pools allows to reduce their initial size by about 160 Kilobytes.

Table 1 gives the size of the system and the size of data it is possible to place in read-only memory if the system were to be embedded just after this step.

Table 1. System sizes (in Kilobytes) after loading phase

Benchmark	**Size (KB)**	**in ROM (KB)**	**% in ROM**
HelloWorld	308	181	59%
AllRichards	411	185	45%
Dhrystone	315	170	54%

5.2 Linking

We have seen that the linking phase is important. After this step, all classes (17 Kilobytes for *AllRichards*) and methods (60 Kilobytes for *AllRichards*) can be placed in ROM. As bytecodes are mutated, a greater number of bytecode arrays become immutable (33 Kilobytes over 87 Kilobytes). In addition, other entries in the constant pools become useless, allowing to re-compact them. This leads to the measuqrements given in table 2.

Table 2. System sizes (in Kilobytes) after linking phase

Benchmark	**Size (KB)**	**in ROM (KB)**	**% in ROM**
HelloWorld	242	181	78%
AllRichards	319	258	81%
Dhrystone	247	194	79%

5.3 Initializing Static Fields

Initializing the static fields turns the classes into state `READY`. Once the classes are in this state, all bytecodes can be replaced by their linked counterparts, leading all fractions of code (87 Kilobytes) to be immutable. The benefits of this phase also include to avoid these initializations when the device starts up. Table 3 presents the measurements we have done when all the classes are in the state `READY`. These results takes account of the sizes of the objects allocated by the static statements. In particular, the overall size of the system for *Dhrystone* is greater than at the previous step because this application allocates 64Kilobytes of static arrays that will be modified at run-time.

Table 3. System size (in Kilobytes) after initialization phase

Benchmark	Size (KB)	in ROM (KB)	% in ROM
HelloWorld	245	236	96%
AllRichards	323	307	95%
Dhrystone	318	238	75%

5.4 Threads Deployment

Deploying threads allows to dramatically reduce the size of the system, as presented in table 4. In particular, only 6440 bytes of classes, 14 Kilobytes bytes of methods and 18 Kilobytes of code remains for *AllRichards*.

Table 4. System size (in bytes) after threads creation and analysis

Benchmark	Size (KB)	in ROM (KB)	% in ROM
HelloWorld	11326	9795	86%
AllRichards	74134	68315	92%
Dhrystone	86558	14316	16%

6 Conclusion

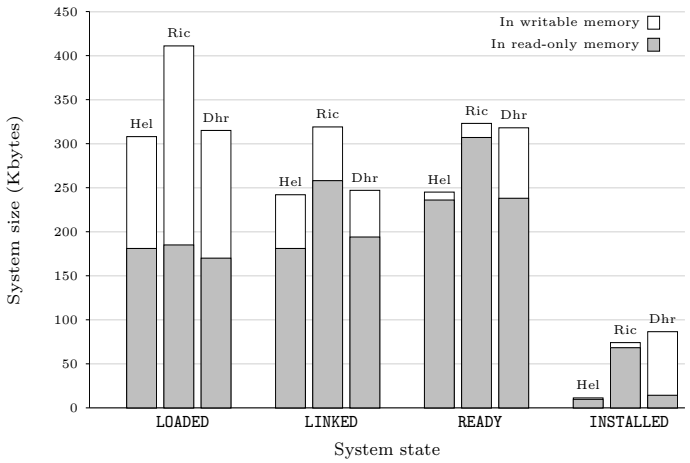

Fig. 1. Memory footprint and repartition across the different states of the system

This article addresses the issue of placing parts of a Java-OS in ROM, which is necessary for tiny devices. At each step of the deployment of the system, it is

possible to place a number of objects in ROM in order to decrease the necessary quantity of modifiable memory. It is also possible to remove objects that have become useless in order to reduce the overall size of the embedded system.

The mechanisms involved in these optimizations take advantage of the particular deployment process of a Java-OS. We gave results concerning the size of data it is possible to place in ROM and the overall size of the system. Figure 1 summaries the evolution of these two measurements. It shows that the more the system is initialized off-board, the higher the proportion of objects in ROM is.

References

1. "Java In The Small." http://www.lifl.fr/RD2P/JITS/.
2. D. Rayside and K. Kontogiannis, "Extracting java library subsets for deployment on embedded systems," *Sci. Comput. Program.*, vol. 45, no. 2-3, pp. 245–270, 2002.
3. F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter, "Practical extraction techniques for java," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 6, pp. 625–666, 2002.
4. N. Shaylor, D. N. Simon, and W. R. Bush, "A java virtual machine architecture for very small devices," in *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pp. 34–41, ACM Press, 2003.
5. S. Microsystems, "Java 2 plateform, micro edition (j2me)."
6. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
7. D. Mulchandani, "Java for embedded systems," *IEEE Internet Computing*, vol. 2, no. 3, pp. 30–39, 1998.
8. I. Pechtchanski and V. Sarkar, "Immutability specification and its applications," in *JGI 'O2: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pp. 202–211, ACM Press, 2002.
9. S. Microsystems, "The k virtual machine (kvm) white paper. technical report," 1999.
10. C. Rippert, A. Courbot, and G. Grimaud, "A low-footprint class loading mechanism for embedded java virtual machines," in *In Proc. of PPPJ'04.*, ACM Press, 2004.
11. P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, ACM Press, 1977.
12. F. Tip, P. F. Sweeney, and C. Laffra, "Extracting library-based java applications," *Commun. ACM*, vol. 46, no. 8, pp. 35–40, 2003.