Romization: Early Deployment and Customization of Java Systems for Restrained Devices

Alexandre Courbot, Gilles Grimaud, Jean-Jacques Vandewalle

Romization: Early Deployment and Customization of Java Systems for Restrained Devices*

Alexandre Courbot[1], Gilles Grimaud[1], and Jean-Jacques Vandewalle[2]

[1] Laboratoire d'Informatique Fondamentale de Lille
UMR CNRS 8022 - Bâtiment M3
59655 Villeneuve d'Ascq Cédex - France
Alexandre.Courbot@lifl.fr
Gilles.Grimaud@lifl.fr
[2] Gemplus Systems Research Labs
La Vigie - ZI Athélia IV
13705 La Ciotat Cedex - France
Jean-Jacques.Vandewalle@research.gemplus.com

Abstract. Memory is one of the scarcest resource of embedded and restrained devices. This paper studies the memory footprint benefit of predeploying embedded Java systems up to their activation using romization. We find out that the more the system is deployed off-board, the more it can be efficiently and automatically customized in order to reduce its final size. This claim is validated experimentally through the production of memory images that are between 10% and 45% the size of their J2ME CLDC counterparts, while using the J2SE API and being ready-to-run without any further on-board initialization. Embedded solutions like J2ME degrade the Java environment and API right from their specification, limiting their usage perspectives. By contrast, our romization scheme generates and specializes a custom-tailored Java API for embedded applications deployed in a full-fledged J2SE environment.

1 Introduction

Embedded and restrained devices programming is evolving towards more sophisticated and secure programming languages. In particular, strong efforts have been made during the last years to allow embedded applications to be written in Java. However, the low amount of memory and safety constraints of these devices make heavy runtime environments such as J2SE inapplicable to them. For these reasons, stripped-down versions of the Java environment have been defined, like Java 2 Micro Edition or Java Card.

Unfortunately, these special editions of the Java environment are incompatible with J2SE. For instance, Java Card does not support floating point numbers

* This work is partially supported by grants from the CPER Nord-Pas-de-Calais TACT LOMC C21, the French Ministry of Education and Research (ACI Sécurité Informatique SPOPS), and Gemplus Research Labs.

and features a firewall that restricts access to methods and data. Also, the APIs of these editions define new packages and are thus not compatible with J2SE.

These incompatibilities and restrictions over the original Java environment break the Java gold rule "*compile once, run everywhere*". J2ME and Java Card have been tailored in order to support a pre-defined range of applications. Therefore, they are unable to run standard Java applications which requirements go beyond their restrictions. For instance, J2ME CLDC doesn't provide the JDBC API, that offers a standard way to access databases. However, embedding a local database or database client on a small device makes sense for some businesses; but in order to provide Java derivatives that fit on a given range of embedded devices, a choice has to be made as to which parts of the API are kept.

Obviously, embedding Java into small devices such as mobile phones or sensors implies a degradation of the environment at some point. Anyway, even if the whole J2SE environment would fit into an embedded device, it would still be desirable to avoid embedding unused packages and features in order to save silicon and production costs. Providing a pared-down version of Java for these devices is therefore inevitable. But conversely to the J2ME and Java Card approaches, we support the idea that the specialization of the Java environment should not be imposed by a specification. Instead, it should be done on a per-case basis, according to the applicative domain of the Java programs that will run on it.

This paper is about a new deployment scheme that allows J2SE-compliant applications to be embedded into closed, restrained devices. To allow this, the Java API of the system is tuned and reduced according to the applications needs during an off-board deployment phase called romization. This phase offers a complete view of the deployed system to the customization tools, which can thus perform much more efficiently than classical library extraction tools.

The remainder of this paper is organized as follows: in section 2, we discuss the deployment process and the specifics of Java applications deployment for embedded devices. Section 3 describes romization, an off-board form of deployment suitable for embedded devices, and section 4 presents our romization architecture that allows the Java API to be tailored according to the applications that are deployed. We discuss experimental results and compare with related work in section 5, before concluding on our approach.

2 Deployment Schemes for Embedded Devices

Software deployment can take various forms and interpretations. This section presents a simple yet comprehensive model of deployment that maps the Java class loading process. We then observe the limitations of the Java class loading scheme for small and restrained embedded devices, and consider the existing solutions addressing this issue.

2.1 Application Deployment Model

The deployment process covers all the steps that bring a software component from a state where it is ready to be loaded (usually a software package) to a

state where it is ready to run on a particular environment. This includes the installation and configuration tasks, but also any kind of adaptation applied to the software being installed.

An exhaustive definition of the deployment process can be found in [1]. For the purpose of this paper, we only cover the deployment tasks that are needed to bring a software component up to a runnable state.

As shows figure 1, a software component needs to successfully go through several steps before being usable. After being *Released* (made available for installation by a software packager), a software component is *Transferred* into a target system and *Configured* in order to operate within its new environment. Put together, these two steps correspond to the *Installation* task. Finally, the component becomes operational by being *Activated* (*Activation* task). We name this state when a component is activated the *Useful Initial State* of the component, because it is from this state that its installation is complete and it can be used by the system without any other preparation.

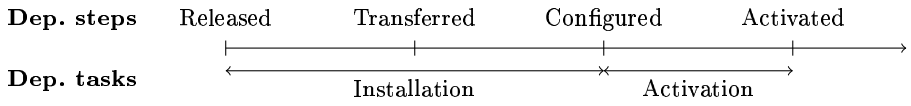

Fig. 1. The software deployment time-line. Before being used, a software component must go through installation and activation tasks. Note that the component starts being useful at step *Activated*

The above-mentioned tasks have corresponding opposite operations: an activated component can be de-activated, then re-activated, and a component can be removed from the system by being uninstalled. The software packager can also choose to de-release the software by stopping its distribution and support.

This deployment model is mappable to the Java deployment scheme, which consists in loading classes into a Java virtual machine.

2.2 Java Applications Deployment Scheme

Several full-fledged software components deployment schemes are available for Java, like OSGi[2] or J2EE[3]. Because they all rely on the lower-level class loading mechanism, we center our deployment study on it.

Java Class Loading. The class loading process is one of the core mechanisms of Java. It can be seen as a classical software deployment solution that allows a software package (a Java class) to be deployed into an operational system (the Java virtual machine). We describe how the above-mentioned deployment tasks map with the class loading stages.

The Java Virtual Machine Specification[4] states that between the released `.class` file and the loaded and operational class, several distinct stages are passed over:

At first, a class loader is told to create a new class from a binary representation (the `.class` format). The class is read from a binary container (usually a file) and its internal representation in the JVM is created. This stage gives the state *LOADED* to the class, and corresponds to the transfer of the class within the virtual machine (*Transferred* step).

Then, the external references of the class are resolved during the linking stage. Linking can trigger the loading of several other classes that are referenced. This stage also verifies the bytecodes of every method for type-safety. The class is given state *LINKED* once this operation is finished, a state that is equivalent to the *Configured* step in our deployment model: the class is set up so that it can be used by the virtual machine. It should be noted, that the external references resolution can either be done once and for all during linking (*early linking*), or be delayed to be performed just-in-time when the bytecode is executed (*late linking*). Late linking can trigger the loading of classes during runtime, when an unlinked reference is met by the bytecode interpreter. Early linking prevents this, but at the cost of loading all the classes that are referenced in the code at once, regardless of whether the interpreter will actually meet them or not. On the contrary, late linking only loads the classes referenced if they are used at runtime. Desktop Java implementations run on comfortable machines and have all the necessary `.class` files at hand (either on disk, or from the network), so they usually adopt late linking. Embedded Java environments dispose of limited processing power, few storage space and intermittent (if any) network connections, and therefore tend to use early linking for their class loading model in order to avoid having to load classes during runtime.

Finally, the class is initialized by interpreting its static statements (or *class initializer*), which are mainly used to set the initial values of static variables. The class is *READY* (*Activated* step in our deployment model) once this stage is complete, and can be used thereafter. Contrary to loading and linking, the specification imposes a precise time for initialization to occur: right before the first *active use* of the class. The first active use is basically the first time the bytecode interpreter meets a reference to the class, for instance via a static method invocation or instance creation.

The figure 2 shows the mapping between the deployment and Java class loading steps.

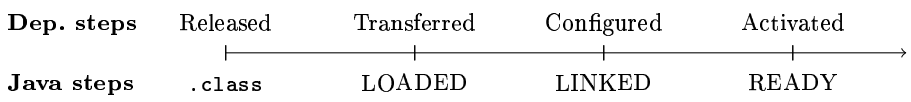

Fig. 2. Java class loading steps mapped to their counterparts in our deployment model

It is only when the class loading is entirely done (i.e., when the class has reached the state *READY*) that a class can actually be used by the virtual machine. Just as the *Activated* step for a component, the *READY* state corresponds to the useful initial state of a Java class.

Java System Initialization Phase. Similar to a class, the whole Java virtual machine follows a time-line, which begins at its invocation. The Java virtual machine first performs bootstrap activities to initialize itself. Then, in order to execute a program, it creates a Java thread and loads the class that contains the entry point of the program; typically, a static method named `main`. It is only once the entry point class reaches state *READY* that the system can start executing the `main` method and has reached its own useful initial state (figure 3).

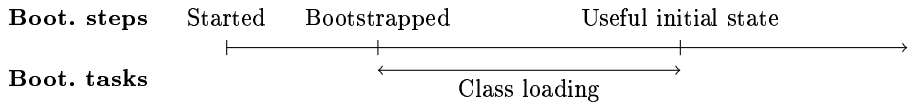

Fig. 3. The Java system time-line

We label the task performed before the virtual machine reaches its useful initial state the *system initialization task*. This definition embraces all the activities that are performed identically every time the virtual machine is invoked with the same program, before the program is actually run.

The system initialization task (especially class loading) is a quite heavy process, and can hardly (if at all) be performed by small and restrained devices[5]. Such devices have to turn to alternate class loading schemes.

Alternative Java Class Loading Schemes. The Java class loading process is so inadequate for embedded devices that research has been undertaken to provide smaller, easier to load class formats or loading schemes. EJVM[6] uses a client/server model to distribute the class loading burden and allow only useful methods of a class to be loaded. Several pre-loaded class formats have also been developed[7], and some have been made available on the market, like the `.cap` format of Java Card[8] or JEFF[9]. They define an almost ready-to-run format for sets of Java classes, where all the symbolic references are resolved and the constant pools are merged. In Java 2, Micro Edition, classes needed during runtime can also be pre-loaded into the virtual machine when the latter is being compiled. Such a process is called *romization*.

All these class loading schemes distribute the class loading process so that the hardest work has not to be performed in the target device. Romization in particular brings some interesting opportunities to efficiently tailor the system, that pre-loaded class formats cannot bring, as we will see in next section.

3 The Romization Process

Although widely used by the embedded devices industry, romization has evoked few interest from the scientific community so far. To our knowledge, no publication ever studied in depth or formalized romization. In this section, we define the general principles of romization and analyze the limitations of existing romization techniques.

3.1 Principles of Romization

Romization is the process by which a software system is pre-deployed by a specific tool (the *romizer*), running on a *deployment host*, for a *target device*. The inputs of romization are the bare system and a set of components to pre-deploy on it. From them, the romizer creates a memory image suitable for the target device that contains the system with the components already deployed on it. Romization can therefore be qualified an "in-vitro" form of deployment: the software is not deployed on its actual target, but rather inside a "test tube", before being transferred already-deployed to its runtime device.

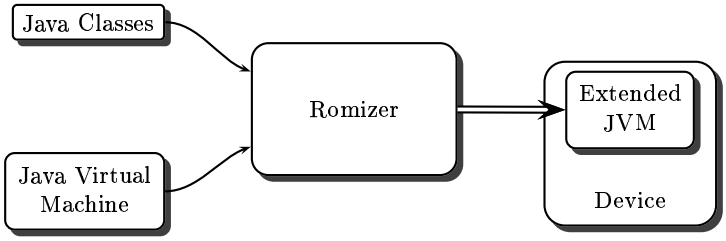

Fig. 4. The romization process applied to a Java virtual machine

Figure 4 illustrates the romization process applied to Java: the system is the embedded Java virtual machine, the components to deploy are the Java classes, and the resulting output is a virtual machine for which all the given classes are already loaded. This extended virtual machine can thereafter be transferred to the target device for being run.

The memory image produced by the romizer is completely ready-to-run and mappable to the physical memory of the device. It is intended to be placed in Read-Only Memory, hence the name *ROMization*, although other memories can be used. In the embedded devices industry, romization is used in order to instantiate the initial program of a device, that is invoked by the boot loader. The initial state of the system on the target device is the state of the system when it is dumped by the romizer (figure 5).

Ideally, the initial state on the device is as close as possible to the useful initial state. That way, the system is immediately active and useful when the device is powered on.

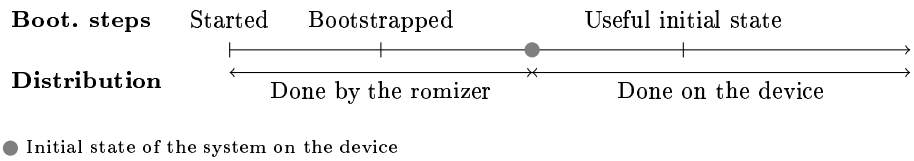

Fig. 5. Example distribution of the Java system initialization between the romizer and the target device. The romization process let the system be started on a deployment host before being transferred to the target device

For Java systems, romization is mainly used to relieve the target device from loading the applications and system classes: as we said, this phase requires a lot of resources to be performed on a restrained device. Moreover, the class loading would unnecessarily be identically repeated every time the device is powered on, increasing startup times. Today's existing romization solutions for Java are designed to address this issue.

3.2 Available Romization Solutions

All the romization solutions studied hereafter are industry responses to the need of deploying Java applications on small and limited devices. They aim at preventing the embedded device to load the classes, by providing them already loaded within the embedded virtual machine.

Java 2 Micro Edition (J2ME). J2ME[10] is a configurable derivative of Java targeted at embedded devices with at least 128KB of memory. It features the Kilobyte Virtual Machine (KVM), a low-footprint Java virtual machine.

J2ME is divided into several *configurations* that reflect the differences between ranges of restrained devices. The Connected Device Configuration (CDC) is designed towards strong PDAs and set-box boxes, while the Connected Limited Device Configuration (CLDC) is more adapted for devices like mobile phones. The CLDC API is a strict subset of CDC API, which is itself a (non-strict) subset of the J2SE API. In addition to API restrictions, CLDC also limits the virtual machine capabilities by removing support for reflection, objects finalization and by limiting error handling.

J2ME also provides a romization tool called *JavaCodeCompact* (JCC), that is capable of pre-loading classes against the KVM so that they are immediately available upon invocation. JCC performs the loading and linking operations of the classes (figure 6). The classes initializers still have to be executed on the target system in order to finalize class loading, and there is no way to request the execution of code during romization.

As output, JCC produces a C file representing the loaded and linked form of the classes for the KVM. This file is thereafter compiled and linked with the KVM binary.

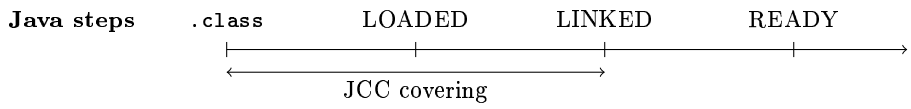

Fig. 6. The class loading activities covered by JCC

Java Card. As the tiniest flavor of the Java technology, Java Card[11] is targeted towards devices so limited that they cannot even support the lightest configuration of J2ME. As its name suggests, it is primarily designed towards smart cards. Although based on Java by concept, Java Card only gives a slight taste of it. The restrictions on the virtual machine are very drastic (optional 32-bits integers, no automatic memory collection, multithreading, 64-bits or floating point operands), and the high safety needs of smart cards applications led to the addition of new security mechanisms such as the firewall. The system API, which need to exploit the resources of these tiny devices with poor communication capabilities, has also very few in common with J2SE. Nonetheless, Java Card met success in the smart card industry thanks to its high safety, portability and ease of programming when compared to past smart card development toolkits.

The deployment of classes into a Java Card requires an additional step to be performed outside the device: the classes are pre-loaded into a `.cap` file in a nearly ready-to-run, yet portable across Java Cards, representation of the classes. This chewed-up form is then given to the device which has little more to do than verbatim-copying the class data to memory. For this reason, it is quite easy to produce a Java Card virtual machine with classes already romized in it.

The `.cap` file format contains classes that are in state *READY*: The class initializers are evaluated by the `.cap` production tool, and static variables are initialized on the card after a data array. However, and because of this, the static initializers in Java Card are limited to (arrays of) primitive compile-time constant values. One cannot, for instance, create new objects using the class initializers, which strongly limits the pre-deployment possibilities.

3.3 Evaluation of Existing Romization Solutions

Both J2ME and Java Card are good answers to the romization problem for their respective range of devices, as their commercial success witness. However, they are not going far in the deployment process regarding our definition of romization.

In section 2, we defined the state where the system actually start to run applications as the *useful initial state*. Our definition of romization in section 3.1 then states that the purpose of romization is to approach this state as much as possible outside the target device, so that the latter doesn't suffer from the cost of deployment. J2ME and Java Card approach the useful initial state by pre-loading classes against the virtual machine. The motivations behind this are to avoid embedding the heavy class loading mechanism if not necessary, to

reduce the virtual machine startup time, and to avert the need of having a copy of the classes available (either locally or from a network).

However, by limiting their romization capabilities to these sole points, J2ME and Java Card are unable to do complete software pre-deployment. The deployment cost is indeed reduced by the class pre-loading, but nothing is done regarding higher-level components initialization or activation. For instance, it is possible to pre-load the classes of an OSGi component during romization, but not install or activate it in the OSGi sense. Moreover, class loading is not even completely covered by JCC, which leaves the classes initialization to the target device, and by Java Card, which considerably limits the class initialization capabilities. In the end, and although the most costly part of the deployment is done by the romizer, a non-neglectable part of the system and applications deployment must still be performed on the target device (figure 7).

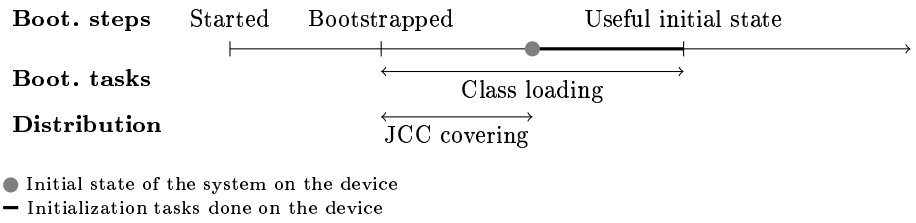

Fig. 7. The part of the Java system initialization covered by JCC. JCC only covers class loading partially, and leaves a consequent part of the initialization task to the target device

This incomplete system initialization during romization has more consequences on the final system than the minor annoyance of a longer startup time. Indeed, it is common to apply customizations to the system being deployed during romization: JCC for instance can be given a list of the classes, methods and fields to romize. Elements not mentioned in this list are omitted in the system memory image. This selection allows the romized system to keep a reasonable memory footprint by not including useless elements of the system.

Let's consider that we want to romize the classical `HelloWorld` program, that uses the standard output stream (`System.out`) to display a constant string. The `System` class contains the standard output stream, but also refers to several other kinds of streams through `System.in` and `System.err`, to the system `Properties` and `SecurityManager`, and so on. Since the romizer works with early linking, all these references are recursively loaded when the romizer loads the `System` class. This means that for a single "Hello, World" displayed on the screen, one would need to load dozens of classes that are not used at runtime and occupy many kilobytes of precious memory. With JCC, the system producer can decide by hand to limit the romization of the class `System` to the static field `out`, disregarding useless references to other classes, fields and methods.

Determining which parts of the system and applications classes are needed can be done using a call-graph resolution algorithm[12], run from the entry points of the system. Many library-extraction tools[13, 14] use this technique to extract a minimal subset of a library, that will behave identically with respect to the original library for a given set of applications. Call graph analyzes are used to explore all the possible paths of a program and mark the classes, methods and fields that are likely to be accessed by it. A call graph analysis requires the knowledge of the program entry points, and give better results if it is provided static information about the system. For instance, knowing the initial values of some variables can help removing paths in the call graph, and thus keeping less elements. The romizer looks like a good place to perform call graph analyzes, because it has a complete view of the system being deployed: if the romizer were capable of going as far in the system initialization as creating the applications threads, it would know the entry points necessary to compute the call graphs. If it could initialize the classes and deployed software components, it would dispose of static information useful for further paths elimination and code simplification. Moreover, in addition to providing a great context for call graph analyzes, the romizer would also be the direct beneficiary of their results.

We can see that there is a great promise of system customizability for a romization scheme capable of handling deployment in a more complete manner. The next sections describes and evaluate a romization framework proposal that targets this purpose.

4 A New Romization Scheme

As we have seen, the romization solutions presented in the previous section limit their activities to part of class loading. By contrast, in order to increase the customization potential of the system, we need a solution that can not only handle the class loading completely, but also any software component layer that could be used on top of it. This way, the deployment could be covered in a more complete manner by the romizer, and the useful initial state of the system could effectively be approached off-board. Doing so would not only reduce the startup time, but more importantly would open the way to efficient call graph analyzes that allow to remove unused parts of the system and greatly reduce the final memory footprint.

4.1 System Initialization Task Distribution

The romization solutions studied earlier were simple class pre-loaders, that only perform a small part of the system initialization. In order to cover a larger range of initialization activities, a romizer has to include more features according to the additional operations it needs to support.

The following is a list of all the initialization tasks performed in order to reach the useful initial state of the system, and an evaluation of their applicability within a romizer:

Initializing the hardware: Since the romizer has no access to the hardware of the target device, it cannot perform this very first step. Anyway, the hardware loses its state when the device is switched off, so this operation needs to be done every time the device is powered on.

Bootstrapping the system: This operation consists in initializing the virtual machine internals in order to make it usable: for instance, setting up the heap or the bytecode interpreter. If the romizer can determine the suitable post-bootstrap system state, it can dump a binary image with these values already set up. Some system initializations may rely on the execution of Java code, especially if deep parts of the system are programmed in Java. In this case, the romizer must include a bytecode interpreter to execute them.

Loading, linking and initializing classes: This part is the only one covered by existing romization tools like JCC. However, without a bytecode interpreter, the classes cannot be initialized (which would involve executing the class initializers). Also, the class loading mechanism is often just a layer above which a real software component framework like OSGi relies. In this case, the components deployment is only partially covered by the romizer unless an execution environment is provided to cover the registration and activation of the software components.

Creating the threads: Once the classes of the applications are loaded, their threads can be created. The only requirements are properly initialized classes and objects creation capabilities from the romizer, so that a thread object can be created. This step is crucial for further system customizations: With the applications threads at its disposal, the romizer can infer information about the system that are useful for a customizer, like the call graph.

Running the threads: The Java threads might need to run until a given point before the useful initial state is reached. A common situation is a system for which the entry point is an OSGi implementation, which needs to be run in order to deploy the applications bundles. Executing the threads requires a full-fledged system to be performed safely: the romizer must have implementations for the native methods, and must be capable of running the Java code as if it was the target device itself.

In order to completely deploy the Java applications, the romizer thus needs to be able to perform runtime operations like executing bytecode or creating objects. A romization architecture covering the deployment activities we listed must therefore be based on a complete virtual execution environment.

4.2 General Architecture

We divide our romization architecture into three main parts that interact with each other (Figure 8). The *Environment* is a *virtual execution environment* (VEE), in which the system to romize is prepared. The *Dumper* takes the Java objects pool of the Environment as input and, as its name states, dumps them into a representation that is suitable to be used with the runtime environment. Finally, the *Builder* receives the output of the Dumper and creates the final system by assembling it with the runtime environment.

The Environment: A virtual execution environment that appears like a real Java runtime environment to the applications. It includes a class loader, a memory manager and a bytecode interpreter. The Environment allows the user to deploy and execute the system up to the point considered to be the "useful initial state". It also provides means to introspect the objects graph and to modify the objects of the system.

The Dumper: Its purpose is to create a memory representation of the objects contained in the Environment, correctly mapped with the physical memory of the target device. The Dumper must know about the device memory mapping (quantity, location, access means and properties of the different memories). It parses every object of the Environment and decides a destination memory for each of them according to placement policies. Then, it dumps a representation of the different memory sections and their objects that is passed to the builder for being linked with the runtime support layer of the target device.

The Builder: The Builder coordinates the actions of the two other romization parts with the building tools like the compiler to produce the memory image of the deployed system. It controls how the system is built according to the target device: which compiler to invoke, with which options, which set of native methods to use, and so on. It also decides the parts of the runtime support layer to include and how to tune them according to system properties provided by the environment. For instance, if the environment asserts that the code it contains never allocates a single object during runtime, it is useless to include a memory manager in the generated system. The embedded bytecode interpreter can also be tuned in order not to support bytecodes that are absent in the code.

In a typical scenario, a user who wants to create a memory image of an embedded Java system that runs the `HelloWorld` OSGi bundle will proceed as follows: First of all, the building profile and memory mapping of the target device are given to the Builder and the Dumper. Then, the user asks the Environment to run the OSGi framework, by invoking the former with the latter as parameter, as with a regular Java runtime environment. The OSGi framework starts running into the Environment: the bundle can be loaded and initialized using the interaction means provided by the OSGi framework. The user then sets a "breakpoint" (similar to a debugger breakpoint) on the main method of the bundle (which marks the useful initial state of the bundle) and asks OSGi to activate the bundle. This action requests the execution of the main method, and causes the environment to freeze when meeting the breakpoint: the system has reached the state desired to be the initial state on the device. At this point, the bundle is totally deployed and the OSGi deployment facilities are no more used.

When the Environment has reached the initial state that the user wants for the target device, it can be dumped. But prior to doing so, it is opportune to take advantage of all the static informations that are provided by the fully-deployed system to customize it. The customization opportunities and their effects are

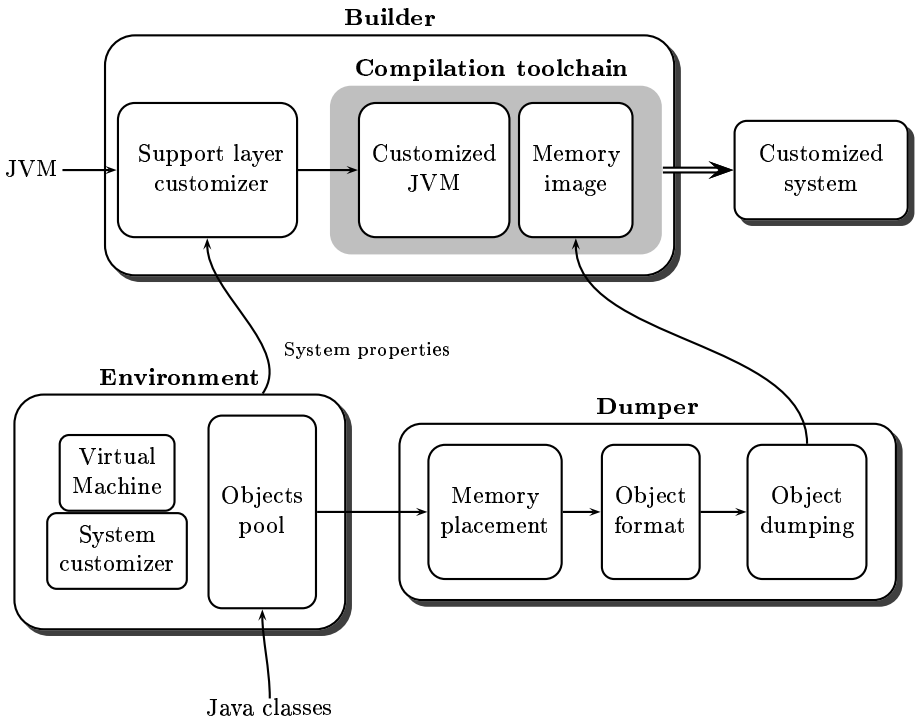

Fig. 8. The proposed romization architecture

described in detail in section 4.3. They affect the Environment by suppressing its useless parts, and by transforming some objects (for instance, objects for which an unused field can be removed, or methods that have been specialized to their calling context). The customized system also provides information to the Builder, like which classes are to be eventually included in the memory image, or which bytecodes are used (for tuning the embedded bytecode interpreter).

This final Environment state is given to the Dumper which, using the memory layout provided earlier and its memory placement policies, creates the memory image of the Environment objects, correctly spread between the different memories of the target device. It is then up to the Builder to tune the hardware support layer according to the Environment properties, and to compile the necessary part of the support layer along with the memory image given by the Dumper. The result of the compilation process is the ready-to-run memory image of the embedded system, at the state it had when the Environment was given to the Dumper. This memory image is finally burnt on the physical memory of the target device. When powered on, the target device immediately executes the main method of the `HelloWorld` bundle that has been deployed, as it simply continues the execution of the system from its final state within the Environment.

It should be noted that not all system states are safely dumpable. Typically, it has no purpose to dump states that contain non-serializable objects like opened network sockets or file descriptors.

The next subsection looks at the customization opportunities before the objects are passed to the dumper.

4.3 Customization Opportunities

As stated in the previous subsection and on figure 8, there are two kinds of customizations that can be performed on the romized system:

1. The customization of the Environment, done by tailoring or suppressing some of its objects,
2. The runtime virtual machine customization, done by the Builder, that selects and tailors the parts of the support layer to keep according to the Environment properties.

They are to be performed in the given order, since tailoring the Environment might influence on how the runtime virtual machine is to be customized. The customization of the virtual machine is planned for future work, we are concentrating in the present article on the customizations applied to the Environment.

Customization of the Environment occurs right before its objects are given to the Dumper: when the frozen system has reached its initial state on the device. The customizer figures out a transformation of the Environment that will perform the further execution of the system identically with respect to the original Environment, but is optimized for size and performance. The customization process starts by a call graph analysis from the current threads states to mark all the possible paths the embedded bytecode interpreter could go through. The advanced deployment state of the system helps removing unreachable paths: first, the call graph runs on live stacks, which contain objects of known type and known value. This allows, for instance, to resolve virtual methods invocations[15, 16]. Moreover, at the time the system is running, we have many instantiated static objects that won't change during the program execution, which allows their values to be inlined.

The call graph obtained also gives information about which classes, methods and fields are potentially needed by the program. All the fields, methods and classes not referenced in this call graph can safely be removed from the Environment, unless they are to be called later by dynamically loaded classes (in this case, the call graph can be completed with a list of "potential" entry points). The call graph can actually tell much more than just which objects are reached by the program flow: it can also provide additional information about the objects, like whether they may be written or not. Such information is useful for the placement manager of the Dumper to determine the destination memory of objects (objects that are never written can safely be placed in Read-Only Memory).

A second customization pass is then run on the remaining objects, in order to tailor them for their runtime usage. The most frequently concerned objects are Java methods, which can be simplified using partial evaluation with the static information of the Environment. For our `HelloWorld` example, the method `println` is only called from one context, with a static argument: it can therefore be specialized for this unique usage, and the string argument can be removed. Following the same principle, variables can be replaced by constants where applicable, conditionals on known values can be removed, virtual methods call can be turned into static ones, and so on.

Finally, decisions may be taken as to how the objects are to be outputted by the Dumper. Java methods which are found out to be critical (either by the code analyzer or the user) can be compiled into native code by an ahead-of-time compiler for maximum efficiency, at the cost of a larger memory footprint. The compiler can take advantage of all the static informations gathered during the call graph analysis to optimize the code, like omitting runtime exceptions checks for proven sites.

All these customizations affect the deployed Environment and aim at producing the most compact memory image of the Java system tailored for the applications that are (or are to be) deployed on it. The efficiency of applying these customizations at activation time is evaluated in the next section.

5 Experimental Results

Our romization architecture has been implemented into the Java In The Small (JITS[17]) platform. This section evaluates our approach by measuring the memory footprint of the image generated by the Dumper (including all the Java objects of the system, loaded classes, methods, etc.) at different stages of the deployment process.

5.1 Methodology

The memory footprint generated by JITS has been evaluated on three benchmarks. The `HelloWorld` benchmark is the typical example of a minimal program which final memory footprint should be very low. `AllRichards` is a much bigger benchmark (78 classes) that simulates 7 different implementations of an operating system kernel task dispatcher. Finally, `Dhrystone` is a small benchmark made of a few classes, mainly used for integer performance evaluation. Its interest for our measurements resides in its memory allocations within the class initializers.

The customizations implemented in the Environment customizer are the removal of unreferenced objects, classes, fields or methods, and the modification of the classes structure to suppress entries for unused fields and methods. The call graph computer implements constant propagation and static class hierarchy analysis[18] in order to detect inapplicable paths. It also marks all the objects likely to be accessed during runtime.

The measurements are performed as follows: for each benchmark, the main class is loaded into the Environment. The class loader then resolves and loads all the necessary dependencies. After this, the system is brought up to the desired state, and dumped into a C file containing the definition of all its Java objects. This C file is thereafter compiled using GCC 3.4.3 for the i386 platform with optimization level 2, and stripped. The final measurement is the size of the stripped object file.

We compare our measurements obtained using JITS with the equivalent memory image generated by running JCC on the CLDC configuration of J2ME, version 1.1, including the standard CLDC API as well as the benchmark classes. The C file generated by JCC is compiled and evaluated using the same protocol.

The measurements performed on JITS cover, for each benchmark, the size of the memory image generated by the Dumper at the following stages of the system deployment:

Transferred All the classes have state *LOADED*,
Configured All the classes have state *LINKED*,
Activated All the classes have state *READY*, and the benchmark thread is created (but not started). This stage is considered to be the useful initial state of the system for these benchmarks.

5.2 Results

Table 1 shows the sizes of the memory images generated by JITS at the different deployment stages, as well as the equivalent images generated by JCC.

The column labelled *Transferred* shows the size of the system when all the classes necessary for the benchmark are in state *LOADED*. The size of all the bare `.class` files necessary for the benchmarks to load and run is of 636 Kbytes for `HelloWorld`, 1014 Kbytes for `AllRichards` and 664 Kbytes for `Dhrystone`. We can see that the loaded form of the classes is much lighter than the original `.class` files, mainly because many constant pool entries containing symbolic references can be discarded at that point, depending on the class loading mechanism used[19]. However, at this stage no static information is available that would allow the customizer to suppress objects.

Linking the classes together brings the system to state *Configured*. This stage leaves more constant pool entries unreferenced (those whose sole purpose is to give the symbolic name of references), which can be suppressed. Another beneficial side-effect of this state is that more system objects reach their final state. This stage, which is the stage at which JCC dumps the classes it loaded, gives us memory images of 242 Kbytes for `HelloWorld`, 321 Kbytes for `AllRichards` and 248 Kbytes for `Dhrystone`.

The *Activated* column gives the size of the memory dump obtained when going further in the deployment process: the classes initializers are executed by the Environment, and the applications threads are created. This allows the customizer to compute the call graph, remove useless objects, and tailor the

Table 1. Sizes (in Kbytes) of the memory images generated by JITS and J2ME CLDC

Benchmark	JITS		
	Transferred	*Configured*	*Activated*
`HelloWorld`	307	242	11
`AllRichards`	410	321	70
`Dhrystone`	314	248	86

Environment. The final size obtained for `HelloWorld` is 11 Kbytes: the minimalist semantics of the program leads to a system of minimal size. `AllRichards` displays a size of 70 Kbytes, of which 58 Kbytes are made of its many classes and methods. Since all the code of this huge program is executed at runtime, it is romized entirely; however, we found out that the amount of system classes kept in the memory image is almost as low as for the `HelloWorld` benchmark. `Dhrystone`'s final 86 Kbytes may seem surprising when considering that its memory footprint for the others deployment stages is very similar to `HelloWorld`'s. They are however understandable regarding the class initializers: this benchmark, amongst other smaller allocations, allocates an array of 65 Kbytes in its class initializers. If we disregard this dynamically allocated data, we find a size of 14 Kbytes for all the others objects of the system. Would the system have been romized earlier, this memory would anyway have been allocated by the device.

These results are very supportive to our initial claim: going further in the deployment process during the romization of the system provides all the information needed to tailor it efficiently. In particular, the customization process allowed us to get rid of the useless references in the J2SE API, and to obtain the fitting derivative of it for the benchmark being deployed.

The graph of figure 9 compares the footprint of the memory images generated by JITS at state *Activated* against their J2ME CLDC counterparts. JITS generates closed memory images that are up to 90% smaller than their open J2ME equivalent: this is explained by the customization phase which tailors the API and benchmark code in order to extract and customize a minimal subset of it. The remaining Java API is finely adapted to the runtime needs, and therefore better-suited than the static J2ME API. It should be noted, that since J2ME doesn't initialize the romized classes, the memory allocated dynamically in the class initializers is not included in the dumped image. This detail is significant for the `Dhrystone` benchmark, so we represented this amount of memory, which is allocated on the target device, by the dashed part of its graph bar. We thus gain about 90 KBytes on J2ME for the `AllRichards` and `Dhrystone` benchmarks, while the J2ME systems retain the possibility to load classes.

Comparing our results at stage *Configured* against JCC is also interesting, because at this stage the classes have equivalent states on both systems. J2ME get better results, which is explained by several factors. First, JITS has many core parts of the system written in Java, which are therefore included in the memory image. For instance, the system class loader of JITS is written in Java, whereas

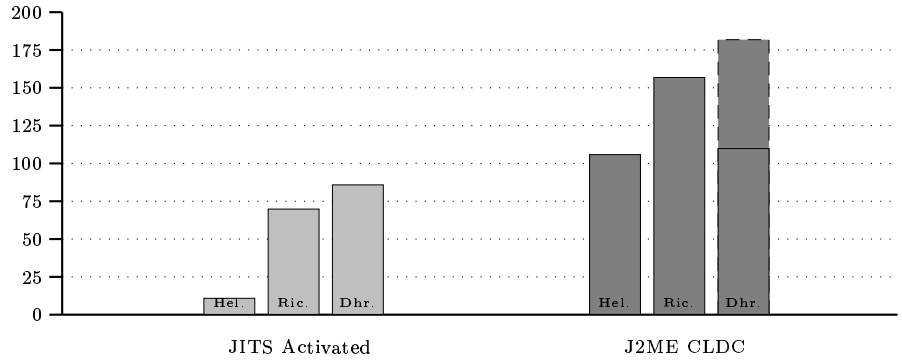

Fig. 9. Comparing the system size of JITS at state *Activated* with its J2ME CLDC counterpart

the J2ME one is written in C and is therefore not counted in our measurements. The main reason, however, is that the system deployed by JITS uses the J2SE API, which has much more classes than J2ME and much more links between them. On the `HelloWorld` benchmark, JCC romized 99 classes, while loading and linking the `HelloWorld` class in JITS resulted in loading 142 classes. The customization phase, that frees the J2SE API from all the useless elements, can not be performed efficiently at this stage. Therefore, at this point of the deployment process, or for systems that need to remain open, using light APIs like J2ME is justified and indeed more efficient - besides, they have been specified to address these precise issues.

It is also pertinent to compare our results with classical library extractors. In[13], Rayside et al. obtained an extracted library size of 328 Kbytes for the HelloWorld program, using the J2SE API. While the entity measured (the unloaded bytecode) is not directly comparable with our results, the subset still comprehends 122 classes and one can predict that the loaded extracted library will still occupy between one or two hundreds of kilobytes in memory once loaded into the virtual machine. The library extractor operates on a non-deployed system and therefore has few clues about the possible runtime behavior of the system. On the opposite, the JITS customizer operates on a deployed system and knows all the types and values of entry points parameters (which are on the stack), as well as many static objects.

The conclusion of our experiments is that the more the system is deployed within the romizer, the more it can be tailored for its runtime needs. In particular, if the applications threads are available, the romizer is able to use this information to extract and customize the fitting subset of the system APIs that is necessary for runtime: in our experiments, the customization phase always leaded to a final memory image that is much lighter than its J2ME counterpart, while the romizer worked on the whole J2SE API. Contrary to J2ME, which restricts the system API right from its specification, our approach holds the API specialization until the deployment of the system, resulting in an per-case cus-

tomization that is more adapted, and only comprehends the features useful for runtime. However, our approach forbids loading classes on a customized system that have not been evaluated (and therefore known) at the time of customization.

Our experiments also confirmed that, if the system is only partially deployed during romization, a dedicated API like J2ME clearly outperforms the J2SE API in terms of memory image footprint.

6 Conclusion

We presented a romization architecture capable of generating very small memory images of closed Java systems for applications written using J2SE. Experiments show that going further in the system deployment within the romizer allows the latter to perform very precise analyzes on the deployed Java system. These analyzes can then be used by a customizer to extract a custom-tailored subset of the large J2SE API for the applications being deployed. The resulting system is therefore broadly adapted to its runtime needs and shows a much smaller memory footprint than the equivalent system obtained with static solutions like J2ME, while preserving full J2SE compatibility for applications development.

Contrary to solutions like J2ME or Java Card, our architecture makes no initial assumption about the kind of applications that it will run, or the kind of device that will be used. Therefore, it imposes no upper-limit to the system capabilities: the generated system is the smallest possible Java subset that allows the deployed applications to run on the given device.

We see many perspectives from this work. They include the implementation of more customization tools (particularly code specializers), in order to study how they behave in the favorable romization environment. Another short-term study point is the efficient customization of the embedded Java virtual machine the romized applications are linked with. The system informations brought by the romizer could also probably be used in order to improve the results of other algorithms, like Worst Case Execution Time computation. Finally, an open problem is the extensibility of our solution. Our current implementation gives very small closed systems, yet it may be desirable to extend them.

Acknowledgments

The authors would like to thank Dorina Ghindici, who implemented the call graph analysis that allowed us to run our experiments.

References

1. A. Carzaniga, A. Fuggetta, R. S. Hall, A. van der Hoek, D. Heimbigner, and A. L. Wolf., "A characterization framework for software deployment technologies," Tech. Rep. CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998.

2. OSGi Alliance, *OSGi Service Platform, Release 3.* IOS Press, Inc., 2003.
3. R. Searls, *Java 2 Enterprise Edition Deployment API Specification, Version 1.1*, August 2002.
4. T. Lindholm and F. Yellin, *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., 1999.
5. D. Mulchandani, "Java for embedded systems," *Internet Computing, IEEE*, vol. 2, no. 3, pp. 30 – 39, 1998.
6. D.-W. Chang and R.-C. Chang, "Ejvm: an economic java run-time environment for embedded devices," *Software Practice & Experience*, vol. 31, no. 2, pp. 129–146, 2001.
7. D. Rayside, E. Mamas, and E. Hons, "Compact java binaries for embedded systems," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, p. 9, IBM Press, 1999.
8. Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley Longman Publishing Co., Inc., 2000.
9. The J-Consortium, *JEFF Draft Specification*, March 2002.
10. Sun Microsystems, *J2ME Building Blocks for Mobile Devices*, 2000.
11. *The Java Card Virtual Machine Specification*, 2003.
12. D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (New York, NY, USA), pp. 108–124, ACM Press, 1997.
13. 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.
14. F. Tip, P. F. Sweeney, and C. Laffra, "Extracting library-based java applications," *Commun. ACM*, vol. 46, no. 8, pp. 35–40, 2003.
15. A. Diwan, K. S. McKinley, and J. E. B. Moss, "Using types to analyze and optimize object-oriented programs," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 1, pp. 30–72, 2001.
16. V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (New York, NY, USA), pp. 264–280, ACM Press, 2000.
17. "Java In The Small." http://www.lifl.fr/RD2P/JITS/.
18. J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, (London, UK), pp. 77–101, Springer-Verlag, 1995.
19. C. Rippert, A. Courbot, and G. Grimaud, "A low-footprint class loading mechanism for embedded java virtual machines," in *3rd ACM International Conference on the Principles and Practice of Programming in Java*, (Las Vegas (USA)), 2004.