Application-Driven Customization of an Embedded Java Virtual Machine

Alexandre Courbot, Gilles Grimaud, Jean-Jacques Vandewalle, David Simplot-Ryl

Application-Driven Customization of an Embedded Java Virtual Machine*

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

[1] IRCICA/LIFL, Univ. Lille 1, France, INRIA futurs, POPS research group
{Alexandre.Courbot, Gilles.Grimaud, David.Simplot}@lifl.fr
[2] Gemplus Systems Research Labs, La Ciotat, France
Jean-Jacques.Vandewalle@research.gemplus.com

Abstract. Java for embedded devices is today synonym of "embeddable pseudo-Java". Embedded flavors of Java introduce incompatibilities against the standard edition and break its portability rule. In this paper, we introduce a way to embed applications written for Java 2 Standard Edition. The applications are pre-deployed into a virtual Java execution environment, which is analyzed in order to tailor the embedded Java virtual machine according to their runtime needs. Experiments reveal that this method produces customized virtual machines that are comparable in size to existing embedded Java solutions, while being more flexible and preserving standard Java compatibility.

1 Introduction

Many solutions exist as of today for using Java on small and restrained devices [1], like Java 2 Micro Edition (J2ME) and Java Card. To become embeddable, these flavors deviate from standard Java and only offer a subset of its features. Java 2 Standard Edition (J2SE [2]) is the original edition of Java, and as such has the widest applicative spectrum of all the Java implementations. However, its resource requirements limit it to desktop workstations or strong PDAs. Lighter devices have to turn to degraded versions of Java such as J2ME. These Java flavors come with APIs that cover a limited range of the J2SE APIs, and are sometimes incompatible with it. In addition, their virtual machine doesn't cover all the features range of the J2SE specification. A Java derivative is therefore only suitable for a given kind of applications and a given range of devices, and enforces the application programmer to cope with an environment that is not J2SE-compliant. The portability gold rule of Java is thus broken.

Obviously, using Java on restrained devices requires a degradation of the Java environment at some point to make it fit. However, imposing restrictions right from a specification tend to make the environment either suitable for the general use and inefficient for dedicated tasks, or good for one domain and inapplicable

* This work was partially supported by a grant from CPER Nord-Pas-de-Calais/FEDER TAC MOSAQUE.

to others. It also multiplies the number of incompatible implementations of Java that a developer has to choose from. Our approach is to tailor the most suitable customized Java environment from a standard Java environment on a per-case basis, according to the applications that are to be run, and the specifics of the target device. As efficient customization of software requires knowledge about its runtime conditions, the customizations take place during an off-board pre-deployment phase of the system, called *romization*.

We identify two kinds of customizations that are applicable during romization. The first one, automatic reduction and specialization of the J2SE APIs to get light and efficient custom-build APIs, has been studied in previous work. In the present paper, we are interested in the specialization of the Java virtual machine that is embedded into the target device. We propose and evaluate a method for determining and removing the virtual machine features that are not necessary to the embedded applications. This approach has the advantage to retain J2SE compatibility for the programmer, and to provide an adequately-tailored virtual machine to the applications.

The remainder of this paper is organized as follows. In section 2, we make an overview of Java on embedded devices, introduce the romization concept, and summarize our previous work on it. Then, section 3 explains how deployment-time analysis of the system can be useful to customize the embedded virtual machine. Section 4 experimentally measures the memory gained by removing unused virtual machine features, and we conclude on our approach in section 5.

2 Overview of Java on Embedded Devices

In this section, we overview some existing solutions for using Java on small and restrained devices. Then, we present the romization process, its advantages for embedded Java systems, and summarize our previous work on it.

2.1 Java on Embedded Devices

Java offers features like compact program bytecode and safe execution that make embedded Java a hot topic. As of today, many embedded Java environments are available. Java 2 Micro Edition (J2ME [3]) specifies a Java-like virtual machine specification and APIs, and is derived into two *configurations*. The *Connected Device Configuration* (CDC) is designed for network appliances, while the *Connected Limited Device Configuration* (CLDC) is targeted at small and mobile networked devices, like mobile phones. Both CDC and CLDC come with a small subset of the J2SE APIs and bring new, incompatible APIs. Moreover, CLDC imposes restrictions on the virtual machine: no support for reflection, objects finalization, floating point numbers, and limited error handling. Java Card [4] is another Java derivative from Sun that targets smartcards. It has more limitations than CLDC, since it also drops support for garbage collection, 32-bits operands, and strings. Java Card also deviates by the firewalling mechanism, and its `.cap` preloaded class format. TinyVM and LeJOS [5, 6] are community

projects for enabling Java on the Lego Mindstorm platform. They propose two differently sized and featured implementations, with additional non-standard APIs and limitations on the virtual machine.

Java's promise is "*compile once, run everywhere*". But as we can see, all the embedded solutions considered here are incompatible with J2SE and violate this rule. Moreover, they offer a rather static virtual machine configuration, which features may not all be exploited by the embedded applications, thus wasting silicon and questioning the relevancy of writing small applications in Java. To address these issues, we propose not to adapt the applications to a specific Java environment, but on the contrary to tailor a standard J2SE environment according to its applications and targeted device. Such a tailored Java system becomes embeddable and provides the right subset of runtime features needed by the applications. Our approach, which customizes the J2SE APIs as well as the embedded virtual machine, relies on the romization process.

2.2 The Romization Process

Romization is the process by which a Java system is pre-deployed by a deployment host, for a target device. In this particular form of deployment, the device that runs the system is not the device that deployed it. Romization differs from distributed deployment methods like Java Card `.cap` format or JEFF [7], which are pre-loaded alternatives to the `.class` format. The romizer deploys the Java Runtime Environment within a virtual execution environment, and then dumps a memory image of it. This memory image containing the deployed system is then copied to the target device where it will continue its execution. Romization can therefore be characterized as an "in-vitro" form of deployment (figure 1).

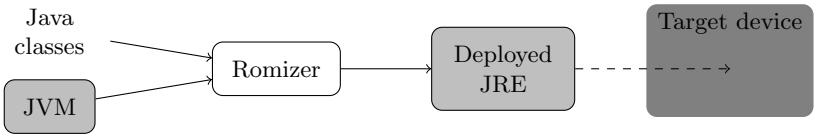

Fig. 1. The romization process

Romization brings two interesting properties for restrained devices willing to run Java. First, the device does not need to support the cost of deployment. This point is important because Java class loading is too costly a process for many small devices. Second, the output of the romizer serves as the initial state for the device (the state it is in when powered on). Since this state comprehends the deployed Java virtual machine, the device is ready to use it immediately, which reduces startup times. These points make romization a very common practice, not to say a mandatory step, in the embedded Java world.

2.3 Previous Work on Romization

As of today, romization is primarily used to pre-load Java classes and provide a service quite similar to distributed class formats. In previous work [8], we have overridden this classical usage of romization and shown the benefits of going further in the system deployment during romization: the romizer can perform very aggressive customizations on the system if the latter is deployed far enough. In particular, call graph analyses [9] on the threads allow unused parts of the APIs to be removed using library extraction techniques [10, 11], and the remainder to be specialized for runtime usage. This results in APIs that are custom-tailored on a per-case basis for the system, and have low memory footprints.

This previous work did only cover the specialization of the deployed applications and Java APIs. In this paper, we take advantage of the advanced deployment state of the system to customize the embedded Java virtual machine.

3 Customization of the Java Virtual Machine

We have seen in the previous sections that many features of the J2SE virtual machine are not supported by restrained devices. We are however interested in developping our applications using the standard J2SE, and degrading it according to the applications needs and the capabilities of the target device. This section evaluates how the necessary runtime features of a Java program can be figured out, while the next one gives experimental results on this approach.

The purpose of the Java virtual machine is to execute Java programs: i.e, to provide an implementation for every bytecode used by a program, in such a way that its semantic is conform to the Java specification. If a bytecode is not used by the virtual machine, support for it can safely be dropped.

3.1 Unused Bytecodes Support Removal

The full Java instruction set covers a large spectrum of operations: integer and floating point arithmetic and logic for 32 and 64 bits operands, objects allocation, methods invocation, threads synchronization, and so on. But few Java programs use all the bytecodes – this is especially true for small programs. For instance, many embedded applications have no use for floating point arithmetic. Critical applications, if deployed far enough within the romizer, often never allocate memory.

Figure 2 shows the bytecodes usage spectrum of various benchmark programs, as stated by the call graph analysis done in our romizer. `AlarmClock` is a simple alarm program that waits for a given time to be reached. `Dhrystone` is the well-known integer operations performance benchmark, and `Raytracer` is a multi-threaded image rendering benchmark from the SPEC JVM98 suite [12].

It is striking that every benchmark is far from using all the bytecodes of the Java instruction set. A very small application like `AlarmClock`, which scope is limited to integer arithmetic, has no use for the majority of them. `Dhrystone`

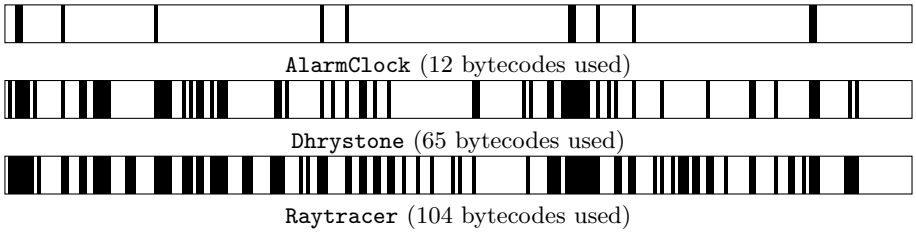

Fig. 2. Bytecodes usage spectrum for different programs. The horizontal axis parses the whole Java instruction set. Bytecodes present in the program call graph rise a black bar, whereas a white gap indicates an unused bytecode

uses strings and is already a more complete program, but there are still more white sections than black ones on its spectrum. `Raytracer` heavily uses floating point arithmetic in addition to integers, as well as threads synchronization and memory allocation. However, it still uses less than half of the bytecodes set.

Once the set of useful bytecodes is determined, support for unused ones can be removed from the bytecode interpreter. This may give the opportunity to remove some services provided by the virtual machine. For instance, the `new` bytecode is responsible for allocating memory for an object of a given class. To do so, it uses a virtual machine function that allocates memory on the heap. If there is not enough memory available, this function triggers a garbage collection to recover memory. This mechanism is reproduced for other memory allocation bytecodes, like `newarray` or `anewarray`. If the `new` bytecode is never to be met by the virtual machine at runtime, the object allocation function of the virtual machine can be dropped. If none of the memory allocation bytecodes is present in the program code, then not only can their corresponding allocation functions be dropped, but also the garbage collector since it is never going to be useful to recover memory: the gain of removing all the memory allocation bytecodes is greater than the cumulated gain of removing each of them individually.

All the bytecodes are not equally interesting to remove. Memory allocation bytecodes are great candidates, because they rely on heavy mechanisms. On the contrary, removing an arithmetic bytecode leads to a poor gain. We noticed that two-thirds of the memory footprint of the virtual machine serves for implementing one-tenth of the bytecodes: those responsible for memory allocation, threads synchronization, exception throwing, and method invocation.

Removing support for bytecodes in the virtual machine is a good way to eliminate some of its useless features. However, all the virtual machine features are not exclusively dependent on the presence of some bytecodes. For instance, threads switching may be triggered by a bytecode (`monitorenter` or `monitorexit`), but also by other events (a thread used its time slot, a native method put the current thread in sleeping state, ...). Such features require a deeper analysis of the system in order to be decided useful or not.

3.2 Analysis of the Deployed System

Some virtual machine features, such as threads management, would always be present in the virtual machine no matter the bytecodes included. The reason is that these mechanisms are called by the virtual machine itself: for instance, when a time slice is elapsed, the virtual machine requests a thread switching. This is unfortunate because threads management is responsible for a non-neglectable part of the virtual machine memory footprint, and some Java systems have no use for threads (for instance, Java Card). In particular, systems that never have more than one Java thread simultaneously still perform in accordance with the Java specification if they don't include multithreading.

In a virtual machine developed with configurability in mind, threads management can easily be disabled through compile-time definitions. In such a configuration, the bytecode interpreter executes the current thread without switching and ends with it. It is possible for the program analyzer of the romizer to detect in which case this configuration is possible. The virtual machine can be purged of threads management if it fulfills the following conditions:

- There is only one active thread into the system at the time of analysis,
- The program analysis reveals that the method `Thread.start()` is never called,
- No additional code is loaded from the outside.

In such a case, it is assured that no more than one thread will ever run, and the threading mechanisms of the virtual machine can safely be discarded by a compilation flag.

The system analysis might enter in conflict with the bytecodes support removal in some cases. For instance, consider a system which fulfills the conditions to be mono-threaded. However, its execution path meets the bytecode `monitorenter` at some point (for instance, by calling a synchronized method). The implementation of `monitorenter` triggers a thread switching if the current thread doesn't own the monitor, thus including the thread switching functions into the virtual machine. To override this problem, all the instances of `monitorenter` and `monitorexit` in the Java code can be eliminated during romization, which also has the beneficial side effect of reducing the code size.

Removing threads management when it is useless is just an example amongst others, although it is probably the one that offers the most significant memory gains. Similar analyzes can be performed for other customizations, like disabling support for exceptions.

4 Experimental Evaluation

The previous section explained how to detect and remove features of the virtual machine unneeded for a given Java program. In this section, we evaluate the effective memory footprint gained by this tailoring.

4.1 Methodology

All our measurements have been performed on the Java In The Small (JITS [13]) Java-OS. JITS comprises J2SE-compliant APIs and virtual machine, and a romization architecture that allows to execute the system off-board and to perform analyzes on it. The binaries are obtained by romizing the benchmark programs mentioned in section 3, and by compiling the tailored JITS virtual machine using GCC 3.4.3 with optimization level 2, for the x86 architecture. The linker is then asked to eliminate dead code.

The JITS virtual machine is made of several compilation units. The bytecode interpreter engine, when including support for all bytecodes, is 15350 bytes big. The interpreter loop itself is 11670 bytes, the rest are peripheral features like method frame creation or exception throwing. The full threads management mechanisms take 6967 bytes, and the complete memory manager 10915 bytes, of which 7036 are for the garbage collector. A non-customized JITS virtual machine therefore has a memory footprint 33232 bytes, to which one must add the target-specific code, native methods, and Java classes that are needed for the virtual machine to function. Indeed, many core features of JITS, like the class loader, are written in Java. We are not including these features in our measurements because they are covered by the customization of the Java classes that has been addressed in previous work. In this paper, we are just interested in tailoring the natively-written part of the runtime environment.

4.2 Results

Table 1 shows the sizes obtained for virtual machines capable of running our benchmark programs.

`AlarmClock` uses 12 bytecodes, and its engine is reduced to 2895 bytes. This program never allocates memory, which makes the memory manager unnecessary, and the threads management operations can also be highly reduced since the program never creates new threads. Moreover, the set of bytecodes used is quite "ideal", with only low-cost bytecodes (stack manipulation and integer arithmetic). This explains the very low footprint of this virtual machine.

`Dhrystone` uses 65 bytecodes, for an engine size of 6992 bytes, and is also mono-threaded. One question can be raised about why the memory manager is not included in the binary, since this benchmark allocates arrays in its source code. The answer is, because all the allocations have already been performed within the romizer. `Dhrystone` allocates memory at two points of its execution: during the initialization of the classes (for initializing static fields), and at the very beginning of the benchmark where it allocates one-sized integer arrays (which are a trick to simulate passing integers by address). These memory allocations are not performed at runtime because the romizer dumped the state of the system *after* their execution. It doesn't change the runtime semantic of the program because at this point the benchmark algorithm has not yet started. Therefore, these memory allocations can be considered initialization work that is safe to be performed off-board. This is a typical example of the advantage of

bringing the system to an advanced state of deployment within the romizer: if our romization architecture were only capable of pre-loading the classes, none of these initializations would have been performed. Our virtual machine would then have suffered a penalty of several kilobytes for the memory manager; not to mention the footprint of the class initialization mechanism, and the increased startup time of the system.

Table 1. Size (in bytes) of the obtained virtual machines for the different benchmark programs

Benchmark	`Reference`	`AlarmClock`	`Dhrystone`	`Raytracer`
Number of bytecodes used	242	12	65	104
Engine size	15350	2895	6992	8576
Memory manager size	10915	0	0	7986
Threads management size	6967	1908	1908	6854
Total size	**33232**	**4803**	**8900**	**23416**

We should also mention that the customization of the Java APIs done during romization is essential for efficiently removing bytecodes. For instance, the `String.charAt` method used by `Dhrystone` allocates and throws an exception if the given index isn't within the range of the string. But since `Dhrystone` always calls this method with well known values and on strings we statically know the size of, the romizer can infer that the exception is never thrown, and improve the code of the method accordingly. Without this APIs customization pass, the exception throwing would be a plausible program path and the `new` bytecode marked as used, requiring a part of the memory allocation module and the whole garbage collector to be included.

Our last benchmark, `Raytracer`, requires 104 bytecodes for an engine size of 8576 bytes. There is no way to completely drop the memory manager since it allocates objects at runtime. Being multithreaded, it also requires almost all the threading mechanisms. Its virtual machine size is therefore of 23416 bytes, which is only 8 Kbytes less than the fully-featured reference virtual machine. Indeed, `Raytracer` doesn't even use half of the bytecodes set, but within the used bytecodes are a good part of the "critical bytecodes" that require the heaviest features of the virtual machine, notably memory allocation bytecodes.

To complete these experiments, we have generated 300 virtual machines, each one supporting a random number of randomly-chosen bytecodes. Whether the virtual machine is mono or multi-threaded is also determined randomly. These virtual machines are not designed for a particular application, but give an overview of the possible memory footprints for a customized virtual machine.

Figure 3 shows the memory footprints obtained for virtual machines supporting a given number of bytecodes and for our benchmark programs. The grayed area is a theoretical range of the possible virtual machines, based on the individual cost of the bytecodes: the upper curve follows the worst possible

case (costly bytecodes included first), while the lower one shows the best case (cheapest bytecodes first). As we can see, the memory footprint varies a lot for virtual machines with the same number of bytecodes. We also notice that the dots tend to group into lines that grow linearly, each line corresponding to the inclusion of a "critical" virtual machine feature: namely, the memory and threads managers. After 150 bytecodes, chances are very low not to include at least one memory allocation bytecode, and the dots form two parallel lines: the lower line for mono-threaded virtual machines, the upper line for multi-threaded ones.

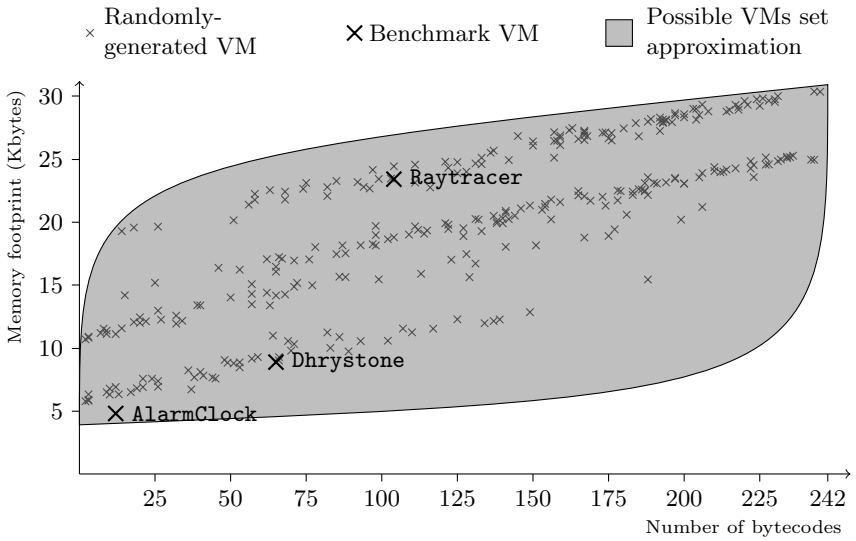

Fig. 3. Memory footprint against number of bytecodes supported, for 300 randomly-generated virtual machines. The grey area gives a theoretical approximation of the range of virtual machines that can be generated

We can compare these results with existing embeddable virtual machines. A standard KVM supporting the CDC configuration is about 40 Kbytes of code when compiled for x86. Recent work on the Squawk virtual machine [14], which aims at providing an efficient CLDC-compliant virtual machine for next-generation smart cards, resulted in a virtual machine memory footprint of 26 Kbytes. Our results obtained by customizing a J2SE virtual machine are therefore quite comparable with these more static solutions. It should be noted, when comparing these sizes with our measures, that the KVM and Squawk footprints comprehend system parts like the class loader which are not included in our virtual machines. This is because the JITS class loader is implemented in Java and is not a direct part of the virtual machine.

5 Conclusion

We gave a proposal solution to the problem of embedding Java on embedded and restrained devices. Current solutions consist in statically-degraded Java virtual machines that are incompatible with J2SE. On the contrary, our approach let the programmer use a full-fledged J2SE virtual machine, which is then customized during romization according to the applications it is going to run and the target device that will host it. The "right" virtual machine is thus generated on a per-case basis, which efficiently reduces its memory footprint.

Put together with our previous work of [8], which tailors the J2SE APIs, these results make it possible to use J2SE for programming embedded Java applications, while providing lower memory footprints than traditional solutions.

Since the bytecodes set is chosen according to the romized applications, this solution is particularly suitable for closed systems that do not load code dynamically. Open systems can define a set of "authorized" bytecodes that are to be included into the virtual machine regardless of their usage by the romized applications; this is especially pertinent if this set only comprehend low-cost bytecodes which gain is negligible. For cases where the Java system has already been deployed, the device memory can also be flashed with another, more featured Java virtual machine.

References

1. D. Mulchandani, "Java for embedded systems," *Internet Computing, IEEE*, vol. 2, no. 3, pp. 30 – 39, 1998.
2. T. Lindholm and F. Yellin, *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.
3. Sun Microsystems, *J2ME Building Blocks for Mobile Devices*, 2000.
4. Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2000.
5. "TinyVM." http://tinyvm.sourceforge.net/.
6. "LeJOS." http://lejos.sourceforge.net/.
7. J-Consortium, *JEFF Draft Specification*, March 2002.
8. A. Courbot, G. Grimaud, and J.-J. Vandewalle, "Romization: Early deployment and customization of java systems for restrained devices," Tech. Rep. RR-5629, INRIA Futurs, Lille, France, July 2005.
9. 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.
10. 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.
11. F. Tip, P. F. Sweeney, and C. Laffra, "Extracting library-based java applications," *Commun. ACM*, vol. 46, no. 8, pp. 35–40, 2003.
12. "SPEC JVM98 benchmarks." http://www.spec.org/jvm98.
13. "Jits : Java In The Small." http://jits.gforge.inria.fr.
14. N. Shaylor, D. N. Simon, and W. R. Bush, "A java virtual machine architecture for very small devices," in *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pp. 34–41, ACM Press, 2003.