# ON-THE-FLY METADATA STRIPPING FOR EMBEDDED JAVA OPERATING SYSTEMS

Christophe Rippert
*INRIA Futurs, IRCICA/LIFL, USTL — Lille 1* *
Christophe.Rippert@lifl.fr

Damien Deville
*INRIA Futurs, IRCICA/LIFL, USTL — Lille 1* *
Damien.Deville@lifl.fr

**Abstract** Considering the typical amount of memory available on a smart card, it is essential to minimize the size of the runtime environment to leave as much memory as possible to applications. This paper shows that on-the-fly constant pool packing can result in a significant reduction of the memory footprint of an embedded Java runtime environment. We first present JITS, an architecture dedicated to building fully-customized Java runtime environments for smart cards. We then detail the optimizations we have implemented in the class loading mechanism of JITS to reduce the size of the loaded class constant pool. By suppressing constant pool entries as they become unnecessary during the class loading process, we manage to compact constant pools of loaded classes to less than 8% of their initial size. We then present the results of our mechanism in term of constant pool and class size reductions, and conclude by suggesting some more aggressive optimizations.

**Keywords:** Java class loading, constant pool packing, embedded virtual machine

## Introduction

Embedding Java applications on resource-limited devices is a major challenge in a highly heterogeneous world where computing power is found in all kind of unusual devices. The portability of Java is an invaluable asset for the programmer who needs to deploy applications on

---

*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.

these heterogeneous platforms. However, embedded Java virtual machines are typically very restricted because of the limitations of the underlying hardware. For instance, the Java Card virtual machine [Chen, 2000] does not support multi-threading or garbage collection due to the typical computing power and memory space available on smart cards [Rippert and Hagimont, 2001]. Memory is an especially scarce resource in most embedded systems due to technical constraints and prohibitive costs which prevent the miniaturization of large memory banks. For instance, a smart card typically includes 1–4 KB of RAM used as working space, 32–64 KB of persistent writable memory (usually EEPROM) used to preserve data when the card is not connected to a power source (and sometimes also a working space if the RAM is not large enough), and 256 KB of ROM which usually contains the kernel of the runtime environment. Thus, reducing the size of the virtual machine and its runtime memory consumption are critical objectives if complex applications are to be executed on the system.

Reducing the memory space consumed by classes obviously means trying to obtain smaller code and smaller data. Previous work has shown that bytecode compression can be used to reduce the memory space used by the code [Bizzotto and Grimaud, 2002]. However, the compressed code size usually cannot be reduced to more than $\frac{2}{3}$ of the initial code size, which does not result in a significant reduction of the overall class size considering that most classes include much more data than code. So it seems interesting to try and compress the constant pool of each class, which stores most of the data used by the class *(e.g.* immediate values, external method names and prototypes, etc.). A careful analysis of the constant pool shows that many of its entries are only needed during the class loading process and can be removed before execution. Moreover, some data is duplicated in different classes and could be factorized. Thus, we have devised a new class loading mechanism which compacts the constant pool on-the-fly by suppressing entries as soon as they are unnecessary, and implemented it in JITS, our architecture for building customized Java operating systems. A valuable asset of our mechanism is that it does not imply disabling important features of the virtual machine, such as dynamic type checking or garbage collection. JITS advocates a very different philosophy than the Java Card environment, since Java Card can be seen as a customization of the specification of Java, whereas JITS implements the standard Java specification while making it possible to customize the code of the environment to fit to the underlying hardware.

We first present the JITS platform we have developed to build customized Java virtual machines for embedded systems. We then detail

the class loading scheme we have chosen in JITS and present the optimizations we have implemented to reduce the memory space needed by loaded classes. Some evaluations of the memory consumption of various loaded classes are then presented, and we conclude by detailing the future optimizations we plan to implement in JITS. A trace showing the evolutions of the constant pool of a classical embedded application is described in an appendix.

# 1. JITS: Java In The Small

JITS is an architecture dedicated to building embedded Java operating systems. JITS is composed of a full-featured virtual machine (including garbage collection, multi-threading, etc.) and a complete Java 1 API. Developers can use the services and packages provided to build a tailored Java Runtime Environment fitting the needs of the application and exploiting the resources available in the best way. Developers can therefore choose which services they want to include, which contrasts with other embedded environments which usually provide a restricted Java runtime environment with little support for customization [Deville et al., 2003]. For example, a developer building a Java Card compliant environment does not need to include a TCP/IP stack and can replace it by a much smaller APDU automaton.

JITS also offers some tools dedicated to help building the embedded environment, as presented in Figure 1. These tools include a program dedicated to generate the binary image of the environment which will be embedded in the device (we call this binary image a “Rom” though it can be stored in other kinds of memory on the embedded device). This program, called the Romizer, first loads all classes selected to be part of the embedded API, and brings them to an initialized state using the loading scheme presented below. After loading the classes, the Romizer takes a snapshot of the objects created in memory and dumps it to a C file which will be compiled with the core of the virtual machine to build the binary image of the runtime environment. The Romizer is a program entirely written in Java and can be run on any virtual machine[1], which differs from standard romization schemes which usually impose a dedicated building environment [Sun Microsystems, 2000] [Sun Microsystems, 2002], Similarly, the JITS API can be used as any other Java API by programs executed on a standard virtual machine. JITS is programmed mostly in Java, so as to ease its porting to various hard-

[1]Though it needs the part of the JITS API in charge of loading classes, namely the classes `Class`, `ClassLoader`, `Field` and `Method`.

ware platforms and to reduce as much as possible its memory footprint. Native code is limited to parts which cannot be programmed in Java and implemented using strict ANSI C to ease the porting.

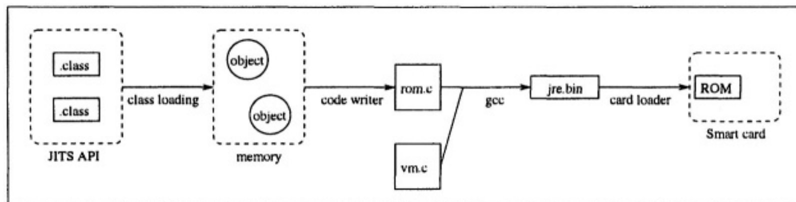


*Figure 1.* The romization process

It is important to note that JITS is not a replacement of Java Card since it is not dedicated only to smart cards. It is meant to build dedicated Java operating systems for various types of hardware. It can be used to build a Java Card compatible runtime environment, but the generated system will be implemented in a very different way than standard Java Card environments. A major difference concerns the class loading mechanism, since in Java Card class loading is done outside the card, when converting classes to `.cap` files [Schwabe and Susser, 2003], whereas in JITS the class loader is part of the runtime environment. This means that the same class loader is used during romization and when dynamically loading new classes. Thus, the class loader in JITS takes into account the limitations of the underlying hardware and is devised to minimize its memory consumption when optimizing the memory footprint of loaded classes, whereas the `.cap` file converter can use as much memory as needed since it is always executed off-card. This on-card reduction of the memory footprint of loaded class is to our knowledge very rarely supported by embedded Java runtime environments.

# 2. Class loading in JITS

## 2.1 Principles

The class loading mechanism in JITS is different from the one implemented in a standard Java runtime environment [Lindholm and Yellin, 1999]. In Java, classes are loaded and linked only when they are actually used (*i.e.* when one of their methods is called or one of their fields is accessed). On the other hand, in Java Card, classes are compacted in `.cap` files containing closed packages, which means that all classes are prelinked when inserted in the `.cap` file. In JITS we chose an intermediate scheme. JITS class loading mechanism supports the standard Java class

loading scheme, but also permits to recursively load and link all classes referenced by the class currently being loaded[2]. This scheme is useful for an embedded Java runtime environment executing on a platform which might not be connected permanently to the network and which therefore needs to load all classes available when actually connected. Another difference with the standard Java class loading scheme is that JITS provides both the standard `defineClass` method which takes a `.class` file stored in a byte array as parameter, but also a `defineClass` method taking an `InputStream` as a parameter. This permits to create the internal representation of the class being loaded on-the-fly without having to load the whole `.class` file in memory, thus preserving memory.

Most classes loaded by JITS go through the four states presented in Figure 2. Primitive types and arrays are exceptions to this scheme, since they are directly created by the virtual machine without having to load any class file. This class loading scheme is used both for classes loaded during romization and for classes dynamically loaded when the virtual machine runs on the embedded system, except for the initialization of static fields as explained below. For classes loaded during romization, our mechanism permits to reduce the footprint of the binary image which will be loaded in ROM. For classes loaded during the execution of the environment, the loading scheme we propose permits to reduce the space consumed in EEPROM where dynamically loaded classes are stored. So we are able to preserve memory both when the environment is created and during its execution.

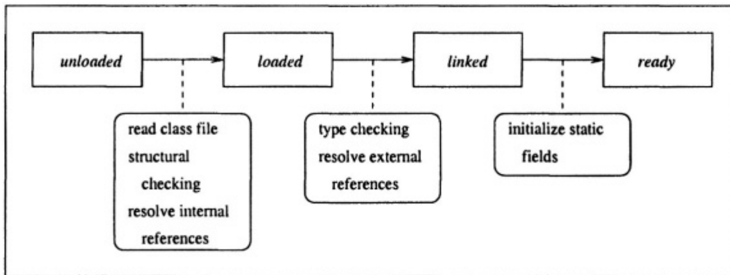


*Figure 2.* The four states of class loading

[2]Recursive linking is also supported in Java Card through the export file mechanism, though this linking is not done automatically as in standard Java.

## 2.2 State *unloaded*

A class is *unloaded* when its `Class` object is first created by a class loader (*i.e.* by using a `new Class()` instruction). This `Class` object is basically an empty container which is filled as the class is loaded from its `.class` file. Figure 3 details the structure of a `.class` file as specified in Section 4.1 of [Lindholm and Yellin, 1999]. A class in state *unloaded* is typically a class which is referenced by another class but which has not yet been loaded by a class loader. This differs from the applet loading scheme in Java Card where a `.cap` file is made of all classes needed by the application. In JITS, classes are loaded one by one and are only required to be loadable when they are actually used.

```
ClassFile {
  id_info;          // magic and version number
  constant_pool;    // stores all constants used by the class
  base_info;        // access flags, class name and superclass
  interface_list;   // interfaces implemented by the class
  field_list;       // fields of the class
  method_list;      // methods of this class (including their bytecode)
  attribute_list;   // attributes (e.g. debug info, etc.)
}
```

*Figure 3.* `.class` file structure

## 2.3 State *loaded*

A class is loaded when the `loadClass` method of a class loader is called. After having checked that the class has not already been loaded and having found its class file in the classpath, the class loader calls the `load` method of class `Class`. This method first reads the basic information of the class (*i.e.* its version number, name, superclass, etc.) before loading its constant pool. When loading a class, JITS ignores attributes not useful during execution of the program (*e.g.* line number table, source file, etc.). This can save a significant memory space, especially if the class file contains lots of debugging information.

The constant pool of classes is loaded from the `.class` file in two tables, named `atable` and `vtable`. The `atable` is an array of `Object` which is used at first to store `Utf8` constants (represented as `String` objects), whereas the `vtable` is an array of `int` in which immediate values are encoded. The `atable` is used later on to store other kinds of objects, such as `Class`, `Method` or `Field` objects.

The constant pool is then prelinked, which consists in resolving the accesses to the structures which represent metadata in the constant pool (*i.e.* the `Constant_info` structures defined in Section 4.4 of [Lindholm and Yellin, 1999]). For instance, a class is represented in the `.class` file by a structure (called `Constant_Class_info`) containing an index pointing to an array of characters (a `Constant_Utf8_info` entry) representing its name. In JITS, a class constant is represented by a corresponding `Class` object stored in the `atable`. Thus, the `Constant_Utf8_info` entry does not need to be mapped in memory. The `Class` object is created if it does not already exist (which means that the referenced class has not yet been loaded or referenced). If the class has already been loaded, we use the `Class` object created when the referenced class was in state *unloaded.* If it has already been referenced, we use the `Class` object created when the class was first referenced. Thus, we preserve memory since the `Class` object is needed to load the referenced class anyway and we do not create any intermediate object to represent it. We apply the same transformation to metadata representing strings and name-and-type constants. Figure 4 details the transformation applied to `Constant_NameAndType_info` entries. These structures are used to describe fields and methods. A name-and-type is composed of the name of the entity (field or method) and a string representing its type (using the convention presented in Section 4.3 of [Lindholm and Yellin, 1999]).

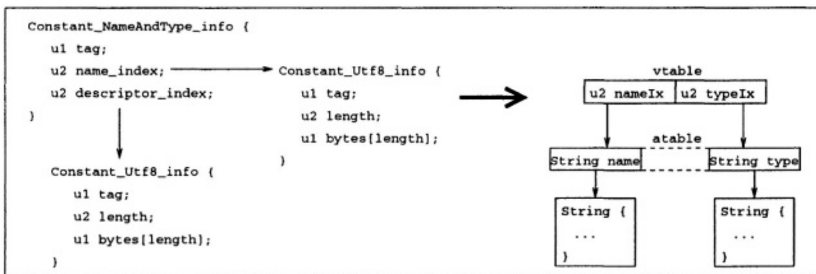


*Figure 4.* Constant pool prelinking (phase one)

A second pass of the prelinker transforms the metadata representing fields, methods, and interface methods (*e.g.* `Constant_Fieldref_info`, `Constant_Methodref_info` and `Constant_InterfaceMethodref_info`) into an `int` stored in the `vtable`. This `int` is composed of the 16-bit index of the corresponding `Class` object and the 16-bit index of the `Field` or `Method` object representing the constant. These objects are

added to the atable. Once more, we are able to preserve some memory by discarding unused Constant_Utf8_info entries. Figure 5 details the transformation applied to Constant_Fieldref_info entries.

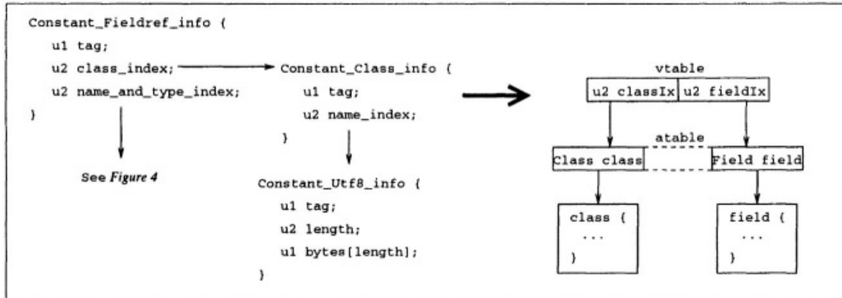


*Figure 5.* Constant pool prelinking (phase two)

After loading the constant pool, the load method reads the interfaces implemented by the class, then its fields and its methods. The static fields of the class are stored in two tables, aStaticZone which contains reference fields, and vStaticZone for immediate values. Reading the methods consists of loading the bytecode, reading the exception table, loading stack maps if they are included in the class file, and finally building the class virtual method table. When loading the bytecode of a method, some instructions are replaced by an optimized version which will be interpreted faster at runtime and can also save some memory space. These optimized instructions are usually known as *quick* bytecodes. For instance, the anewarray instruction includes a constant pool index pointing to the type of the elements of the array. This instruction is replaced by anewarray_quick, which takes as a parameter an index pointing to an entry in the atable containing a Class object of the array component type. Thus, we can suppress the Constant_Class_info and Constant_Utf8_info entries representing the type of the elements of the array.

Another interesting example of instruction replacement concerns the ldc, ldc_w and ldc2_w instructions which are used to load constants from the constant pool onto the operand stack. When loading the bytecode, these instructions are replaced by their *quick* counterparts which directly access the immediate value stored in the vtable without needing the Constant_Integer_info, Constant_Float_info, Constant_Long_info and Constant_Double_info structures which can be discarded. Thus, a ldc instruction is replaced by a ldc_quick_a instruction if the constant is a reference, a ldc_quick_i if the constant is an int, and a

ldc_quick_f if the constant is a float. It would be possible to use the same instruction for both int and float constants since they are both 32-bit immediate values, but that would compromise the type-checker which needs to be able to differentiate int and float. By replacing ldc instructions by a type-specific opcode, we can preserve necessary type information without keeping complete constant pool entries, and so preserve both memory space and functionalities of the virtual machine.

## 2.4 State *linked*

Classes reach the *linked* state after being linked to each others. The linking process starts by recursively loading all the classes referenced by the constant pool of the class being linked. Then every method of the class is prelinked, which consists of type-checking its bytecode, compacting invokevirtual instructions, and marking the constant pool entries used by the method code. During prelinking of a method, invokevirtual instructions are compacted if the index of the method in the constant pool and the number of arguments of the method are both less than 256. Compacting these instructions simply consists of replacing the index of the method in the constant pool, which is encoded in 16 bits in the instruction, by the number of arguments of the method and its index in the virtual method table of the class declaring it. Thus, at runtime the interpreter can call the method directly without accessing the constant pool, which speeds up the calling process. It also saves memory space, since the constant pool entry representing the called method can be deleted. During method prelinking, constant pool entries which are used by the bytecode are marked so that unused entries can be detected during the compaction of the constant pool.

Static fields referenced in the vtable are then converted to references pointing to the vStaticZone and aStaticZone. Static fields are treated differently than virtual fields since their value can be accessed directly since we know the class to which they belong (whereas finding a virtual field requires looking up the inheritance tree to find the first class defining that field). The index pointing to the Field object representing the field is replaced by a 16-bit immediate value containing the 13-bit offset of the field in the corresponding static zone and the 3-bit type of the field (which is necessary in order to know which static zone contains the field and how many bytes should be read). Thus, constant pool entries representing static fields can be suppressed. Figure 6 presents the compacting of static fields.

The constant pool is then packed and resized, thereby losing all unused entries. Finally, each method is linked, which basically means modifying

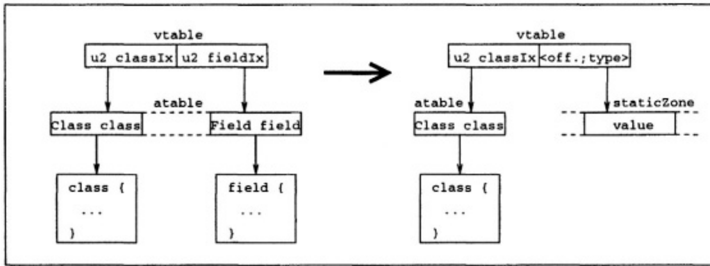


*Figure 6.* Linking of static fields

the bytecode by replacing indices to the original constant pool entries by indexes to the corresponding compacted constant pool entries.

## 2.5 State *ready*

A class reaches the final state *ready* after initializing its static fields to their initial values. If the class is loaded during romization, this is done by using the underlying virtual machine class loader to load the class and then copying the values set by the static initializer to the JITS instance of the class. This rather heavy mechanism is necessary since the `<clinit>` method of a class cannot be called directly from a Java program executing on a standard virtual machine. On the other hand, if the class is being dynamically loaded by a running JITS virtual machine, `<clinit>` methods are called directly by the virtual machine as specified in [Lindholm and Yellin, 1999]. A final optimization can be done here, as the `<clinit>` method of each class can be removed after it has been used to initialize the static fields of the class. This is done simply by removing the `Method` object representing the `<clinit>` method from the linked list of the methods included in each class, and letting the garbage collector free the corresponding memory space.

# 3. Benchmarks

We monitored the memory footprint of the JITS API when loaded using the scheme presented above. The API currently contains most classes from the base package `java.lang`, and some classes from `java.awt`, `java.io` and `java.net`, including a full TCP/IP stack.

We first counted the number of constant pool entries discarded while loading the classes. Results are presented in Figure 7, with state *unloaded* refering to the number of entries in the `.class` files.

| Class state | *unloaded* | *loaded* | *linked* |
|---|---|---|---|
| Number of entries | 8,416 | 3,067 | 1,426 |
| % of initial number | 100% | 36.44% | 16.94% |

*Figure 7.* Number of constant pool entries for the whole JITS API

These results show that most of the reduction of the number of constant pool entries is done while loading the class, i.e. when resolving accesses to the constant pool and removing unnecessary indirections. We still manage to divide by two the number of entries while linking, i.e. by compacting `invokevirtual` instructions and packing static fields (which implies suppressing unreferenced metadata for methods and fields).

We then monitored the memory footprint of the constant pool in bytes. We tried and suppress as many strings as possible since they are the most space-consuming data in the constant pool. Unfortunately, some of them (*e.g.* field names, method descriptors, etc.) are needed by the `java.lang.reflect` package, so we need to keep them if we want to support introspection. Figure 8 presents the size of the constant pool with and without those strings to illustrate the cost of supporting introspection.

| | | with introspection | | without introspection | |
|---|---|---|---|---|---|
| Class state | *unloaded* | *loaded* | *linked* | *loaded* | *linked* |
| Size in bytes | 152,154 | 48,203 | 40,455 | 19,435 | 11,687 |
| % of initial size | 100% | 32.68% | 26.59% | 12.77% | 7.68% |

*Figure 8.* Size of the constant pool for the whole JITS API

The size of the constant pool can be reduced to less than 8% of its original size if introspection is not supported. This is due to the fact that direct references to `Constant_String_info` represent only a small part of all the `Constant_Utf8_info` constant pool entries, so most of them can be eliminated during loading. If those strings are not removed, we manage to pack the constant pool to nearly one fourth of its original size, while preserving a complete support for introspection.

Since most of our optimizations concern compacting the constant pool (apart from disregarding unused attributes, which are seldom included in `.class` files except when debugging), we can use the results presented in Figure 8 to compute the size reduction for entire classes of the JITS API.

The total size of all .class files is 271,117 bytes[3], which includes 152,154 bytes for constant pools. Since we manage to reduce the size of constant pools to 40,455 bytes with support for introspection and to 11,687 without introspection, we obtain a memory footprint for the API which is only 58.8% of the total size of the .class files (48.19% without support for introspection) in state *linked*. Suppressing the <clinit> method of each class allows to save 54812 bytes for the whole API (including 50KB from java.awt), which means that the final memory footprint of JITS API in state *ready* is only 38.58% of the total size of the .class files with support for introspection and 27.97% without it. Figure 9 sums up the reduction of the total size of classes with and without support for introspection.

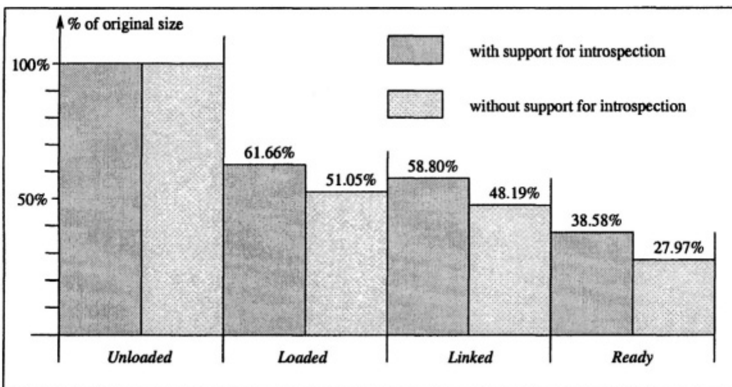


*Figure 9.* Reduction of the total size of classes

These results are similar to those obtained using the *JEFF* class format [The J-Consortium, 2002], which reduces the size of the .class files by merging constant pools of different classes. We make a similar optimization when factorizing entities used to represent data. For instance, most classes in a Java API includes an entry in their constant pool representing the class String (since most classes implement the toString method). In JITS, we replace each entry by a reference to one Class object representing the class String. Thus we are able to benefit from optimizations similar to those done in the *JEFF* class format,

[3]The total size of the whole API is over 260KB. However, this includes packages like java.net and java.awt which will probably not be included on a very constrained platform such as a smart card.

while loading standard `.class` files and staying compliant with Sun's specification.

## 4. Future work

An optimization similar to the one applied to static fields can be done for private virtual fields and methods. In JITS, objects are implemented as a C structure containing a pointer to the related class and the virtual fields of the object. When the `getfield` and `putfield` bytecodes are interpreted, the virtual machine accesses the required field by adding the offset stored in the bytecode to the base address of the object. Thus, it is possible to suppress all constant pools entries referencing private fields since all accesses to these fields are made in the class declaring them and so the `getfield` and `putfield` instructions can be modified to contain the proper offset. Similarly, entries describing private virtual methods can be removed from the constant pool. This optimization cannot be applied to protected, package-accessible or public fields or methods, since they could be accessed by a method of a class loaded dynamically after romization. In that case, the constant pool entry representing the target field or method would be necessary to link the new method.

However, if we define an additional state, called *package-closed,* we can apply this optimization to all non-public fields. The state *package-closed* is reached by classes in a package when no new class can be added to that package. Locking a package this way can be useful for instance to prevent an application from modifying a fundamental package such as `java.lang`. If a class is *package-closed,* all constant pools entries corresponding to its non-public fields can be suppressed since all accesses can be linked before romization of the package.

Similarly, it is possible to define a state *closed* to be able to extend this optimization even to public fields. A class reaches the state *closed* if we can assure that no dynamically loaded class will need to be linked to this class. In practice, this state is most useful for embedded virtual machines romized with all the applications and that do not need to dynamically load new classes. These last two optimizations implies disabling some features of the virtual machine (namely restricting or even forbidding dynamic class loading) so they will be made optional when implemented in JITS.

Preliminary results show that these optimizations would allow a reduction of the constant pool to below the 7.68% lower limit presented in Figure 8. In state *closed,* all `Constant_Utf8_info` representing name or type metadata would become useless, as well as all metadata representing fields or methods in the constant pool. Thus, we can assume that

7.68% is the upper limit of the results we can expect when state *closed* is reached by a class, noting of course that closing a class prevents loading of any new class referencing it.

Another optimization especially interesting for smart cards would consist in minimizing the reorganizations of the constant pool of dynamically loading classes. These classes are stored in EEPROM, a type of memory much slower than RAM and which life expectancy diminishes with each write. So it would increase the performance of the dynamic loading process and prolong the lifetime of the smart card if we could devise a loading mechanism which compacts the constant pool of a class with only the minimal amount of entry reorganizations.

## Conclusion

This paper shows that it is possible to greatly reduce the memory footprint of the class loading mechanism by applying on-the-fly packing of the constant pool of loaded classes. This allows saving memory space on the embedded system without sacrificing functionality of the virtual machine, since for instance we can still type-check the bytecode of the class while suppressing type information from the constant pool. Coupled with the flexibility of JITS which permits to choose precisely which components need to be included in the runtime environment (as a garbage collector would not be relevant to a Java Card compliant platform for instance), this makes it possible to generate a Java virtual machine fully tailored for the target device, thus exploiting the limited resources in the best possible way.

## Acknowledgments

We would like to thank our shepherd, Erik Poll, for his helpful advice on enhancing this paper.

## Appendix

We present in Figure A.1 a piece of the classical example of the Purse application to illustrate the reduction of the constant pool during romization. This partial example shows the drastic reduction of the constant pool which can be achieved by discarding unnecessary entries.

Figure A.2 presents meaningful information from the constant pool (*i.e.* `vtable` and `atable`) of class `Purse` after it has been loaded. Method prototypes and type descriptions are still included in the `atable` since they are necessary to link the class. Besides the initial value for the field `Purse.id`, the `vtable` only contains references to entries of type class which are stored in the `atable`.

Figure A.3 shows the constant pool after linking the class, which permitted discarding most method prototypes and type descriptions. The `vtable` still includes the

```
public class Purse extends Object {
    private static int id;
    private Float sum;
    static { id = 0x12345678; }
    public Purse(float b) {
        this.sum = new Float(b);
    }
    final public void credit(float n) {
        this.sum = new Float(this.sum.floatValue() + n);
    }
}
```

*Figure A.1.* The original Java source of the `Purse` class

```
vtable:
 [0]: 0                  [3]: 18
 [1]: 17                 [4]: 19
 [2]: 0x12345678
atable:
 [0]: null
 [1]: id                 [14]: fr/lifl/rd2p/jits/test/Purse
 [2]: java/lang/Float    [15]: java/lang/Object
 [3]: sum                [16]: floatValue
 [4]: ()F                [17]: class fr/lifl/rd2p/jits/lang/Float
 [5]: <init>             [18]: class fr/lifl/rd2p/jits/test/Purse
 [6]: (F)V               [19]: class fr/lifl/rd2p/jits/test/Object
 [7]: I                  [20]: JMethod <init> ()V access=0x1001
 [8]: ()V                [21]: JField private fr/lifl/rd2p/jits/-
 [9]: Ljava/lang/Float;        lang/Float sum
 [10]: credit            [22]: JMethod <init> (F)V access=0x5001
 [11]: <clinit>          [23]: JField private static int id
 [12]: getSum            [24]: (F)Ljava/lang/Float;
 [13]: JMethod floatValue ()F access=0x1001
```

*Figure A.2.* The constant pool in state *loaded*

```
vtable:
 [0]: 0x12345678
atable:
 [0]: class fr/lifl/rd2p/jits/lang/Float
 [1]: class fr/lifl/rd2p/jits/test/Purse
```

*Figure A.3.* The constant pool in state *linked*

value used to initialize the static field `Purse.id` since this has not yet be done by the

Romizer. Similarly, the `atable` contains the fully qualified name of the class `Purse` since it is needed by the static initializer to find the `Purse.id` field.

```
vtable:
 empty
atable:
 [0]: class fr/lifl/rd2p/jits/lang/Float
```

*Figure A.4.* The constant pool in state *ready*

Figure A.4 shows the final state of the constant pool, after the static initializer of class `Purse` has been executed and discarded. The constant pool entries which were associated with it have been removed too, resulting in a constant pool with only 1 entry left. This entry describing class `Float` could probably be suppressed in most Java runtime environments since class `Float` is part of the package `java.lang` which is typically completely romized. However, we chose to keep it in case the programmer decides not to include support for floating point arithmetics in the base system and then load it dynamically during execution.

## References

[Bizzotto and Grimaud, 2002] Bizzotto, G. and Grimaud, G. (2002). Practical Java Card Bytecode Compression. In *Proceedings of RENPAR14 / ASF / SYMPA.*

[Chen, 2000] Chen, Z. (2000). *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison Wesley.

[Deville et al., 2003] Deville, D., Galland, A., Grimaud, G., and Jean, S. (2003). Smart Card operating systems: Past, Present and Future. In *The 5th NORDU/USENIX Conference.*

[Lindholm and Yellin, 1999] Lindholm, T. and Yellin, F. (1999). *The Java Virtual Machine Specification, Second Edition.* Addison Wesley.

[Rippert and Hagimont, 2001] Rippert, C. and Hagimont, D. (2001). An evaluation of the Java Card environment. In *Proceedings of the Advanced Topic Workshop "Middleware for Mobile Computing".*

[Schwabe and Susser, 2003] Schwabe, J. E. and Susser, J. B. (2003). *Token-Based Linking.* US Patent Application number US 2003/0028686 A1. http://www.uspto.gov/.

[Sun Microsystems, 2000] Sun Microsystems (2000). *J2ME Building Blocks for Mobile Devices.* http://java.sun.com/products/kvm/wp/KVMwp.pdf.

[Sun Microsystems, 2002] Sun Microsystems (2002). *The CLDC Hotspot Implementation Virtual Machine.* http://java.sun.com/products/cldc/wp/CLDC_HI_WhitePaper.pdf.

[The J-Consortium, 2002] The J-Consortium (2002). *JEFF Draft Specification.* http://www.j-consortium.org/jeffwg/index.shtml.