Static Closure of Java Dynamic Class Loading*

Andreas Gal gal@uci.edu
Christian W. Probst cprobst@uci.edu
Michael Franz franz@uci.edu

Department of Computer Science
University of California, Irvine
Irvine, CA 92697-3425

September 10, 2003

Abstract

One of Java's most fundamental core concepts is dynamic class loading. While being very practical in the general purpose domain, the runtime cost of dynamic class loading poses a significant challenge for the deployment of Java applications in embedded systems. In this paper we describe a mechanism called *static class loading* which allows to perform the resource-intensive class loading process at compile-time while preserving the full class-loading semantics as defined in the Java specification. This eliminates the need for a byte-code interpreter and allows to translate all Java code to native code ahead-of-time, saving valuable resources on the target platform.

1 Motivation

Out of roughly eight billion processors produced in 2000, nearly 98% were used in embedded systems [Ten00]. Deployment of microprocessors in the embedded systems domain is driven by very much different forces than the desktop PC market. While the PC domain is currently in the process of shifting from 32-bit processors to 64-bit architectures, the embedded system domain is still dominated by 8-bit architecture accounting for well over 50% of all produced units. As far as embedded systems are concerned, 32-bit processors are nearly irrelevant (less than 0.5% market share) and 64-bit processors practically non-existent. One can also not assume that the embedded systems market will

*Parts of this effort are sponsored by the National Science Foundation under ITR grant CCR-0205712 and by the Office of Naval Research under grant N00014-01-1-0854. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the National Science foundation (NSF), the Office of Naval Research (ONR), or any other agency of the U.S. Government.

follow the lead of the desktop PC example and adopt more and more powerful processing architectures just because the technology is available.

Embedded systems are mass produced and savings in the dimension of a few cents per unit sum up quickly. Manufacturers thus always strive to keep expenditures in terms of CPU power and memory as small as possible, but as large as necessary to accomplish the task. Utilizing modern programming languages such as Java [GJS96, LY96] in this domain is a significant challenge.

Besides the omnipresent resource constraint problem, an embedded system is also very different from the common PC in many other ways. Most importantly, embedded systems are usually dedicated systems. While a desktop PC can be used for many different applications (many of which are installed after the machine has been delivered to the customer), an engine control unit will spend its entire lifetime performing exactly that one given task: controlling an engine. Small bug-fixes and minimal feature improvements aside radical, changes are very unusual for embedded systems software.

The Java programming language has been designed to be flexible and extensible — properties that do not necessarily apply to embedded systems in the same sense that they apply to general purpose desktop computers. In particular, the concept of dynamic class loading is deeply rooted in the Java execution model. Class loading allows one to dynamically compose Java programs or to extend them with additional functionality even at runtime. Unfortunately, this degree of flexibility comes at very significant runtime costs as we will show in the course of this paper. Unanticipated evolution as enabled by dynamic class loading rarely occurs in embedded systems. Thus, the involved runtime costs are even harder to justify. In this paper, we present an alternative approach to dynamic class loading that is more suitable for embedded systems. Notably, our approach reduces the runtime cost for class loading while fully preserving the original Java class-loading semantics and is thus fully compatible with existing Java library code and applications.

The remainder of this paper is organized as follows: In Section 2, we discuss existing approaches to Java class loading and analyze their runtime-cost and suitability for the embedded systems domain. Section 3 describes our *Static Class Loading* (SCL) architecture and Section 4 reports about on prototype implementation of this architecture. The paper ends with conclusions in Section 5 and a discussion of potential future work in Section 6.

2 Related Work

In this section we discuss different approaches to dynamic class loading. We look into several classes of approaches to Java execution, from the CVM [Mic03], the virtual machine provided by Sun for small devices, to the smallest available specification, the JavaCard.

Class loading in the JVM [LY96] is controlled by a *class loader*. The JVM uses several of such loaders for initializing the virtual machine and loading system classes and the application itself. Beside those, an application can also

define its own class loaders for special purposes.

In general, the overall class loading process consists of five steps:

1. find and load the appropriate byte-code for the demanded class,
2. verify the byte-code,
3. link and place it into the JVM's data structures,
4. prepare and initialize the class, and
5. execute the constructor.

The result of the class initialization is an instance of `java.lang.Class`, which identifies the class and is used for the creation of new instances of the represented class.

Obviously, class loading is a complex process that is requiring not only a lot of computing resources on the host, but also support from the runtime system. Namely, the virtual machine needs to be able to locate and download byte-code and, even more important, verify them. Byte-code verification is a cumbersome task, and approaches targeting small and resource constrained systems restrict the availability of loading classes at runtime, or even completely prohibit it.

The rules for loading classes are fixed in the JVM specification [LY96]. In principle, a class is only loaded when needed. However, the JVM can decide to load classes ahead of time. The only restriction imposed is the order in which loaded classes are initialized. Since beside the class itself the JVM also needs to transitively load the superclasses and implemented interfaces, the class loading process can be recursive.

The CVM is Sun's implementation of the CDC specification [Mic03]. Regarding class loading, it basically implements the process outlined above. Figure 1 shows a piece of code that is going to load a class, create an object of the class and call a method on the object.

```
Class c = Class.forName("unloadedClass");
Object o = c.newInstance();
(unloadedClass)o.m();
```

Figure 1: Loading a class in Java

The elements needed are a class loader, which is going to look up the class's byte-code, verify it and load it, the implementation for `java.lang.Class`, and the JVM support in order to insert the class at runtime into the virtual machine's data structures. Looking up the class will either load it from a local archive or over a network connection, resulting in a delay. Additionally, the verifier is going to need some time, delaying the class initialization even further. Figure 2 shows the size of some of the parts relevant to the class loading process for some of the virtual machines discussed. The given numbers are approximate in that

some functionality is scattered over several object files. All the numbers have been taken on 32 bit machines, since some of the technologies are not available on 8 or 16 bit architectures. This in contrast to our approach that is able to handle arbitrary architectures.

VM component	file size [kb] CVM	KVM	GCJ
class loading	10	10	9
verifier	43	13	110
interpreter	15	11	21
total	68	34	140

Figure 2: Object-file sizes of components relevant for class loading

Sun's CLDC specification defines the KVM [Mic99] as the virtual machine for devices that provide very restricted resources. While "K" stands for kilobyte, in a useful configuration the KVM needs closer to one megabyte of RAM than merely kilobytes. In contrast to many other virtual machines for small devices, the KVM allows class loading, however only using the standard class loader. Additionally, it uses a two-step approach to verification — all classes need to be preverified, resulting in class files that are annotated with stack maps, which describe the possible stack states at run time. The verifier then only checks these stack maps in linear time. The resulting annotated class files are on average 12.5% bigger than the original ones, resulting in more storage space or communication time needed. The preverifier itself is 115KB in size.

The KVM also supports the *JavaCodeCompact* utility, which combines Java class files and generates a C file, that can be linked with the Java virtual machine. This allows reducing an application's footprint at the cost of restricting dynamic class loading.

GCJ is the Java extension for the Gnu C compiler [Fre03]. It is a portable, optimizing, ahead-of-time compiler and compiles Java source code and byte-code to native machine code. The compiled applications are linked with the GCJ runtime, namely `libgcj`, which provides the core class libraries, a garbage collector, and a byte-code interpreter. Using this interpreter, the resulting native code can dynamically load and interpret class files.

However, this offsets the benefits of native code generation. On the one hand the resulting binary is small and fast, on the other hand an 8MB library is needed to enable dynamic class loading.

As our prototype implementation, JPure [BBM+01] translates mobile code in its entirety into machine code before execution. While being originally designed for small embedded controllers, the JPure project endeavored to scale Java down to devices typical of that domain. The main reasons were the inefficient code generation performed by the GNU Java Compiler (GCJ) and the non-availability of small Java API subsets at that time. JPure did not deal with dynamic class loading at all and instead partially rewrote Java libraries. However, this is a cumbersome and error prone task that, even more important,

hinders the compilation of applications out of the box.

Even smaller Java implementations such as TinyVM [Sol03b], leJOS [Sol03a], and JavaCard [Sun03] succeed in further reducing the VM overhead by dropping dynamic class loading entirely . The lack of dynamic class loading support causes the same engineering problems that have been previously observed in JPure.

3 Static Class Loading

In this section we describe our Static Class-Loading approach, which maintains the class loader semantics of the Java Virtual Machine (JVM), but saves resources on the target system by performing class-loading at compile-time in *anticipation* of actual class-loading requests at runtime.

As discussed in the previous section, in a traditional Java Virtual Machine the runtime system performs a number of complex operations when a class is to be loaded: First, the VM looks up the class in the file system, then parses the class file and readies it for execution, which involves in many cases some form of just-in-time (JIT) compilation. As a final step, static class members are initialized if the class-loading request called for it.

Java class-loading requests are ubiquitous in the standard Java library implementations distributed by Sun Microsystems. Even the smallest available Java library version, the Java 2 Platform Micro Edition (J2ME) in its smallest configuration, the Connected Limited Device Configuration (CLDC), contains 7 dynamic class-loading requests. The requested target class is in many cases not unambiguously predictable.

Figure 3 contains a section from the implementation of the `Character` class in the J2ME runtime library. In this example, dynamic class loading is used to select dynamically at runtime an appropriate case converter. The actual target class is selected using a user-configurable option (*property*).

```
ccName =
    System.getProperty("java.lang.Character.caseConverter");
if (ccName == null) {
    ccName = "com.sun.cldc.i18n.uclc.DefaultCaseConverter";
}
Class clazz = Class.forName(ccName);
cc = (DefaultCaseConverter)clazz.newInstance();
```

Figure 3: Example for dynamic class loading in the J2ME runtime library

As discussed in the Section 2, previously many implementors of Java solutions for small embedded systems decided to rewrite the runtime libraries and application code to not use dynamic class loading. This approach has two significant shortcomings. First of all, forking the J2ME runtime library requires the implementor to track all future changes by Sun Microsystems to the library

as they have to be applied manually to the internal version. Unfortunately, Sun Microsystems is notoriously known for frequent and incompatible changes to Java's standard APIs. Secondly, the resulting new library and application code becomes suboptimal if it is used on a more powerful VM setting which actually does offer dynamic class loading. In such environments, dynamic class loading can be actually very beneficial as it reduces startup delays and overall memory consumption as only classes really in use are instantiated. Thus, the programmer is effectively forced to maintain two separate versions of the system, one with dynamic class loading and one without. Such parallel code maintenance is not only error prone but also not well supported by Java paradigms and tools. For instance, the common C/C++ approach of using configurable preprocessors directives is not available in standard Java.

To overcome this problem, we propose to use a very lightweight class-loading approach which acknowledges the specific requirements of embedded systems but still preserves the original Java class-loading semantics.

The major source for the overhead of dynamic class loading in terms of resource consumption is the inherent support for unanticipated code evolution. In a Java environment geared towards a desktop PC it is not unusual to independently evolve parts of deployed software by exchanging certain specific system parts. This is also often refered to as component-oriented programming (COP) [Szy98]. For a Java implementation for embedded systems it would cause little limitation to drop support for dynamic evolution of system parts after deployment. Instead, our approach requires all class files to be known at compile time in their final version. Updating single class files after deployment is not supported, but this is in practice no limitation for the vast majority of embedded applications.

By having all class files available at compile-time, we are able to compile all Java code to native machine code for the target device. Similar to *JavaCode-Compact* (JCC) for KVM [Mic99] we discover the minimal subset of classes needed for execution of the final Java application using Rapid Type Analysis (RTA) [Bac98]. However, simply starting with the entry point of the Application does not guarantee that all required classes will be available at runtime. As shown in Figure 3, the application can use the class loader to refer to Java classes by the textual representation of their name. Such relations would not be discovered by the basic Rapid Type Analysis.

Instead, our approach requires the programmer to supply a list of classes which the application could potentially request through the class loader. The compiler merges this list with the classes the Java library is known to create and performs a RTA on these classes as well to ensure complete full code coverage for all potential execution paths.

Thus, in contrast to the traditional class-loading approach, in our architecture the compiler *anticipates* the loading of certain classes based on user-supplied and discovered information and performs the actual class loading already at compile-time. All Java code of classes potentially subject to class loading is compiled ahead-of-time just as the code of classes directly referenced by the Java application.

To allow the runtime system to resolve classes by name, the compiler generates a table containing information about all classes which might become subject to class loading. The layout of this *class descriptor table* (CDT) is shown in Figure 4. We chose to store this information in a dedicated table instead of building our approach on top of the Java reflection mechanism, because the minimal Java standard J2ME in which we are most interested does not offer any reflective capabilities due to its high runtime cost.

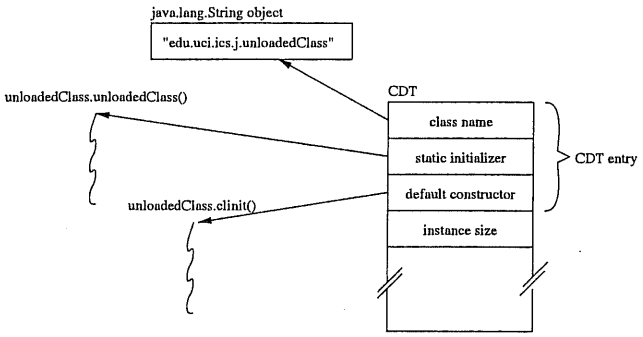

Figure 4: Class Descriptor Table (CDT) to allow runtime resolution of classes by name

For each `Class.getByName(...)` request, the runtime system scans the CDT for matching entries and then initializes the static class members by calling the class initializer from the CDT entry. If a matching class is found, an appropriate `Class` object is returned.

The `Class` object can be used to create new instances of that type using `newInstance(...)`. As the object size in allocation units is unknown at compile-time for `newInstance(...)` invocations, and actually can even vary at runtime depending on what particular class name is given, the CDT contains a field providing this information. After allocating memory for the new object, the runtime system invokes the default constructor provided by the CDT entry to initialize the object.

The application (and the Java libraries) observe the same behavior as they would see in case of a virtual machine with true runtime class loading. The only limitation is that the compiler has to be able to anticipate classes which could be requested at runtime. Requesting unanticipated classes returns a `null Class` object to the caller of `Class.getByName(...)`. As explained previously, we do not consider this to be a significant limitation for the majority of embedded applications.

4 Implementation and Benchmarks

In this section we report on our implementation of a Static Class Loader as part of the byte-code compiler in our ProxyVM framework [VWG+03]. The main

emphasis here is on the size of the resulting class descriptor table and the code needed to load it into the run time system.

The size of the class descriptor table is determined by the data structures used to store information on classes, methods and fields. Figure 5 gives the respective sizes for the K Virtual Machine (KVM), the Sun Microsystems CVM, the GNU Java Compiler (GCJ), as well as for our Static Class-Loading approach (SCL).

All sizes are calculated assuming 32-bit pointers as some of the technologies we use for comparison are not available for 16-bit processors. Figure 5 shows the advantage of ahead-of-time compilation, namely the ability to completely remove information on fields and methods at compile time. We only store information for virtual methods, that is the address, in the virtual method table. Thus, the overall size of the complete class descriptor table can be minimized. During the execution of `HelloWorld`, the KVM loads 125 classes and the standard Java2SE virtual machine loads 278 classes. Even for small applications the memory savings can thus be tremendous. To shrink the needed run time support further, we ensure that each entry in the class descriptor table corresponds to the layout of `java.lang.Class`. A call to method `forName` simply scans the class descriptor table and returns a pointer to the entry instead of creating a new object.

data structure	KVM	CVM	GCJ	SCL
class	80	88	112	20
field	16	8	16	0
method	32	16	20	4

Figure 5: Data structure sizes in bytes for classes, methods, and fields on a 32 bit architecture

As pointed out in Section 2, another important number is the size of the code that locates, loads, and verifies the actual class. Since in our compiler this is done at compile time, we can optimize all parts beside the class descriptor table and the code for finding a class. E.g., for `HelloWorld`, the size of the actual class descriptor table as generated by our compiler together with the code needed to search it is 3KB. The table itself contains 41 classes and 351 method references.

5 Conclusions

Current approaches to deal with dynamic class loading in embedded systems are either to include a complete virtual machine into the target device or to abstain from using class loading altogether. The virtual machine approach generates a very significant runtime overhead, abstaining from class loading causes a number of engineering and code-maintenance problems. We have presented *Static Class Loading* as an alternative approach which preserves the class-loading semantics

of the Java Virtual Machine at a fraction of the runtime cost of traditional approaches. At the same time Static Class Loading guarantees that the compiler sees all classes which ever get instantiated at runtime and thus whole-program compilation and optimization can be performed on the entire application.

Sun Microsystems is currently developing the Micro Edition of the Java 2 Platform in the opposite direction we advocate in this paper. Their approach to overcome the performance problems of the KVM [Mic99] implementation is to introduce a JIT compiler into the smallest Java standard J2ME/CLDC. Even being much smaller than equivalent technologies for desktop PC, the CLDC HotSpot Virtual Machine still requires well over 196kB RAM to execute minimal applications. We believe that this overhead is not acceptable for a large subset of embedded applications. We believe that the approach presented in this paper is a viable alternative for small embedded systems, which still by far dominate the embedded systems market.

6 Future Work

So far we have looked at class loading only in static settings where no new classes are introduced after deployment. However, the presented approach does not dictate this limitation. We are currently investigating the feasibility of downloading native code chunks and additional CDT entries at runtime. This will allow us to add additional Java classes to an already running system. We are also currently exploring the feasibility of adding SCL to the JavaCard standard. This endeavour is hampered by the limited availability of modifiable JavaCard-compatible Virtual Machines for academic research purposes.

References

[Bac98] David Francis Bacon. Fast and Effective Optimization of Statically Typed Object-Oriented Languages. Technical Report CSD-98-1017, University of California, Berkeley, October 5, 1998.

[BBM+01] Danilo Beuche, Lars Büttner, Daniel Mahrenholz, Wolfgang Schröder-Preikschat, and Friedrich Schön. JPure - Purified Java Execution Environment for Controller Networks. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES'2000).* Kluwer Academic Press, October 2001.

[Fre03] Free Software Foundation. The Gnu Compiler for the Java Programming Language, 2003. http://gcc.gnu.org/java.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[Mic99] Sun Microsystems. *KVM - Kilobyte Virtual Machine White Paper.* `http://java.sun.com/products/kvm/wp/`. Palo Alto, CA, 1999.

[Mic03] Sun Microsystems. *CDC: An Application Framework for Personal Mobile Devices.* Palo Alto, CA, 2003. White Paper.

[Sol03a] Jose Solorzano. leJOS, 2003. http://lejos.sourceforge.net/.

[Sol03b] Jose Solorzano. TinyVM, 2003. http://tinyvm.sourceforge.net/.

[Sun03] Sun Microsystems, Inc. Java Card 2.2.1 Specification, Public Review Draft, 2003. http://java.sun.com/products/javacard/JavaCard221.html.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley / ACM, 1998.

[Ten00] David Tennenhouse. Embedding the Internet: Proactive Computing. *Communications of the ACM*, 43(5):43–43, May 2000.

[VWG+03] Vasanth Venkatachalam, Lei Wang, Andreas Gal, Christian W. Probst, and Michael Franz. ProxyVM: A Network-based Compilation Infrastructure for Resource-Constrained Devices. Technical Report 03-13, University of California, Irvine, School of Information and Computer Science, 2003.