

Java-Programmed Bootloading in Spite of Load-Time Code Patching on a Minimal Embedded Bytecode Processor

Thomas B. Preußner, Rainer G. Spallek

Department of Computer Science, Technische Universität Dresden
01062 Dresden, GERMANY

Abstract – *This paper presents the bootstrapping solution used on the embedded bytecode processor SHAP [6]. Although it employs load-time bytecode patching to resolve constant pool indirections, it is itself most comfortably implemented in Java and executed as bytecode by SHAP. Of course, the processor startup sequence initiates the plain loading of the bootloader code – but it does not undertake any conversions on it. The core chicken-and-egg problem is tackled by the bootloader itself, which separates into distinct phases with growing capabilities. This paper describes the operations and conversions performed during these phases and how the operational control is transferred from one phase to another. It further details the tweaks that enabled a high-level coding in Java.*

Keywords: embedded, bootloader, bytecode processor, SHAP

1 Introduction

It is the very nature of bootstrapping that it must activate a system without relying on the functionality provided by this system. While the running system generally offers a high abstraction level making its use comfortable, the bootstrapping implementation must cope with low-level interfaces. As to increase the maintainability and the adaptability of the boot-loader code as well as the productivity of its programmer, it is highly desirable to reach higher abstraction levels fast. Targetting the bytecode processor SHAP [6], the Java-programmed boot-loader to be described meets this goal.

On some embedded platforms, the booted system directly includes the application to execute. In these cases, it is common to merge the system and application loaders into a single entity that merely stores a pre-linked monolithic image into memory before starting the actual execution of the loaded application from a defined memory location. Even quite a few Java processors implement this strategy [2], [5], [7]. The great advantage of this approach is its simplicity. The simple data transfer loop avoiding any complex operations such as code transformations at load time does not create any pressure to introduce an intermediate boot-loader phase programmed in a high-level language. Its drawbacks are (a) that the initial memory image plays a special role in the memory organization throughout the system uptime, and (b) that these systems are fairly resistive against dynamic adaptations or extensions like loading additional application classes at run time.

SHAP allows the dynamic loading of classes and uses regular heap-allocated objects for the runtime representation of classes and string constants. We do not rely on a pre-laid-out heap and can not even do so when adding additional classes. Thus, we require an online means to resolve the references made to these entities to their concrete runtime representatives within the loaded code. This may be achieved by a runtime translation table in the spirit of the Java constant pool that is consulted for such resolutions. Then, a simple identification of classes and strings by table slot numbers can be used within the code, which can often be executed just as loaded. The essential drawback is, of course, the performance loss incurred by this indirection. For its avoidance, it is necessary to transform the code directly when loaded and patch in the resolved references to the named objects. Expanding the application of this technique to the initial system boot holds the benefit that SHAP's high-level memory interface directly working with the object abstraction need not be extended by some kind of an initialization bypass.

An approach quite similar to this idea is pursued by the quick bytecode variants used by traditional Java interpreters dating all the way back to the first JVM implementations by SUN [3]. While having been replaced by JIT technologies in mainstream Java employment, it is still present in clean-room implementations focussing on investigative research such as the SableVM [1]. Latter even resolves and thereby eliminates the constant pool indirection by also inlining other referenced constants of primitive data types. SHAP, indeed, does so too but offloads the elimination of primitive constant references to the offline pre-linker thereby relieving the loader of this burden.

Normally, the substitution of bytecodes accessing the constant pool by their resolved quick counterparts is performed during the execution of the application as part of the implementation of the original bytecode. While this approach enables late class loading not prior to the first use of a class member, it also makes the execution times of code blocks harder to predict: they depend on the execution history. The determination of the worst-case execution time, especially of rarely-taken exceptional code paths, is unlikely to yield desired tight bounds.

Avoiding these drawbacks, the SHAP boot-loader performs

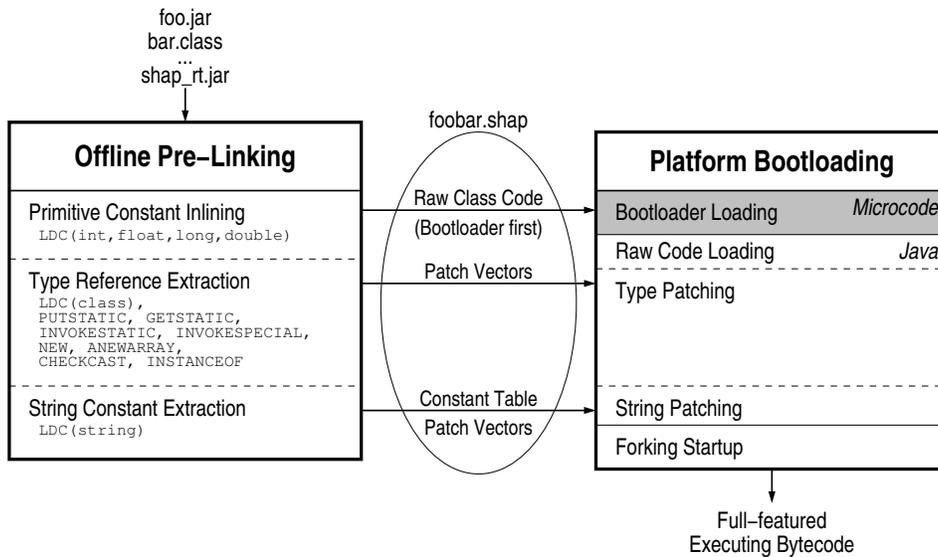


Figure 1. SHAP Process Flow from Class Files to Execution

early class binding and resolves all reference at load time. This comes with the additional benefit that the architecture must implement the quick bytecode variants *only*. These are even significantly simpler than their slow counterparts that would need to include a resolution and a self-patching step before starting their actual operation. On the other hand, a certain limitation is introduced as dynamic application loading can only be performed with self-contained bundles that only reference classes that are already present in the system or contained within the bundle itself.

The remainder of this paper will detail the implementation of the bootloader implemented for SHAP. After an overview on the process flow for programming SHAP has been given in Sec.2, Sec.3 details the design and operation of the platform bootloader. Sec.4 discusses a few side concerns with the integration into the remaining system before Sec.5 summarizes the paper.

2 Process Flow Overview

The complete process bringing compiled Java classfiles to their execution on SHAP is summarized in Fig. 1. It consists of two major stages, the first of which is performed offline as to allow an execution platform with a minimum of resources for execution. A brief description of this stage is necessary here for the understanding of the bootloader to be run on the platform.

As to minimize the work that needs to be performed by the bootloader on the target platform, an extensive pre-linking is performed offline. This step defines the runtime instance layout, resolves primitive constants and substitutes the name-based identification of classes and their members by assigning unique identification numbers and offsets, respectively. It further performs simple optimizations that are valid at link time: the devirtualization or even the inlining of provably

monomorphic call sites, the virtualization of interface method calls as far as possible, and the direct forwarding across bridging methods also within virtual dispatch tables.

The output stream produced by the linker contains individual sections for each phase implemented by SHAP’s platform loader. Each section is again subdivided according to the loaded classes. Their contents will be detailed in the description of the bootloader, which itself must be prepended to the initial system stream providing the runtime implementation.

The transformation of bytecodes accessing the constant pool as performed by the offline linker are summarized in Tab.I. The constant pool as such is eliminated completely. Primitive constants are inlined or loaded from synthetic private final static fields. String, class and member references are represented by numeric identifiers or plain offsets. The locations of such identifiers are collected in patch vectors to be evaluated by the loader on the platform.

3 The SHAP Bootloader

SHAP is bootstrapped in multiple phases, each of which can draw from a greater pool of capabilities. Only the initial phase loading the bootloader code without transformation is driven directly by a microcode sequence. All subsequent phases are implemented as individual Java methods of the bootloader and executed as bytecode.

The separation of the bootloader phases into individual methods is not a mere structuring choice but dictated by the method caching technique used by SHAP [4]. It is necessary so as not to defeat the transformations applied to the code defined by the own bootloader class by caching old versions. Consequently, each phase may safely transform subsequent ones, thereby increasing their spectrum of available functionality. It may, however, not change its own currently executing code.

Opcode	CP Argument	Linked Representation
LDC	int	inlined value
	float	inlined value
	String	numeric string identifier
	Class	numeric class identifier
LDC2	long	GETSTATIC from a synthetic static field
	double	GETSTATIC from a synthetic static field
GETSTATIC	Class.Field	numeric class identifier.fixed class field offset
PUTSTATIC	Class.Field	numeric class identifier.fixed class field offset
GETFIELD	Class.Field	fixed instance field offset
PUTFIELD	Class.Field	fixed instance field offset
INVOKEVIRTUAL	Class.Method	argument count.virtual table dispatch slot
INVOKESPECIAL	Class.Method	numeric class identifier.fixed method offset
INVOKESTATIC	Class.Method	numeric class identifier.fixed method offset
INVOKEINTERFACE	Class.Method	argument count.itable dispatch slot
NEW	Class	numeric class identifier
ANEWARRAY	Class	numeric class identifier
CHECKCAST	Class	numeric class identifier
INSTANCEOF	Class	numeric class identifier

Table I
LINKED REPRESENTATIONS OF CP-ACCESSING BYTECODES

3.1 Basic System Initialization

The SHAP architecture contains a core executing microcode. The individual bytecode instructions are implemented by different entry points. The system starts up at the special entry point at address zero.

At first, the microcode startup routine initializes the source of the code stream, which may be a peripheral UART or a PROM device. Before receiving the first input, it also allocates a thread object on the heap representing the currently executing single control flow. This representative is, of course, no instance of a complete `java.lang.Thread` but merely fulfills the representation of the control flow as needed by the scheduler directly provided by the architecture.

The first data loaded from the code stream are the size and the content of the bootloader class, which are used to allocate and fill the representing class object. This representative contains the unpatched bootloader code as methods implemented by this class. It is, however, not a real instance of `java.lang.Class`, which has not been loaded yet.

Now, the microcode allocates a small table termed the JVM constant pool (JVM CP) and saves a reference to it within a microcode variable. This table will be filled later by the bootloader with entry points to Java methods that need to be invoked from within the microcode. These Java methods, for instance, create and throw JVM exceptions or implement the emulation of floating point operations. Then, a bootloader object is allocated and made a real instance of the bootloader class by initializing its type field appropriately. Finally, the offset of the method implementing the first bootloader phase is received from the code stream and the identified method invoked upon the bootloader object. A reference to the JVM CP to be initialized and a handle to the device providing the code stream are passed as arguments.

3.2 Class Loading

The overall goal of this phase is to load all the class data also including the method implementations defined by each class and to resolve all interrelating references to other classes so far represented by unique class identifiers. These class identifiers are simple serial numbers, which are used to index into a temporary class resolution table used by the bootloader.

Before actually receiving any class data, this phase loads the raw content of the JVM CP. Containing static references to classes, this table will later also be patched with resolved class references.

Having allocated the temporary class resolution table, the individual class definitions are loaded and dumped into memory objects allocated as plain integer arrays. The references to these integer arrays are stored within the appropriate slots of the class resolution table. Its slot #0 is reserved for the bootloader class itself, which, thus, is smoothly integrated into the subsequent processing.

The use of plain integer arrays may seem awkward at a first glance as proper objects of type `java.lang.Class` are required ultimately. It is, however, impossible to assign the correct type at this point of time as the class object representing the `Class` type is not readily available itself. This also rules out the most elegant use of `new Class(...)` as the required `new` bytecode is not yet functional.

Nonetheless, a fixed-sized prefix of the integer arrays representing future classes already matches the layout that will be defined by `java.lang.Class`. In addition to this “official” state, these arrays also contain the dispatch table and the method implementations as defined by a class. This requires the ability to dimension these memory objects dynamically, which is most easily done with arrays.

After the raw versions of the future class objects have been allocated and initialized, it is possible to resolve the numeric class identifiers by references to their true runtime representatives. At first, the JVM CP is patched accordingly.

Then, the patch vectors are received from the code stream that identify the positions of class identifiers within the raw class objects, which are also patched accordingly. Since the provided offsets use byte addresses as to allow to identify an arbitrary position within method bytecode blocks, some bit shifting is required for the patching.

An alternative to using patch vectors would be the symbolic interpretation of the loaded code to determine the start of bytecode instructions and to identify those that take arguments that need to be patched. Keeping the complexity of the loader at a minimum, the current approach is currently preferred.

After creating all class representatives and patching their method implementations with resolved references, nearly all bytecodes have been made functional and, thus, usable for the subsequent loader phases. Only the use of an LDC loading a string constant is still illegal, which will be fixed by the next stage. The handover to this phase is performed through an INVOKEVIRTUAL. Although the exact target method is, in principal, known in advance, a non-virtual invocation through INVOKESTATIC or INVOKESPECIAL would not be valid as the operation of these opcodes relies on the patching just performed by the current loader phase. The class resolution table and the handle to the device providing the code stream are passed on to the next phase.

3.3 String Patching and Class Promotion

This loader phase is responsible for the allocation of String objects used as constants and the following substitution of the numeric identifiers used within the loaded code by the references to these runtime representatives.

The basic structure of this phase resembles the one just seen. After allocating an appropriately-sized String[] array as temporary resolution table, the string contents are loaded from the code stream and dumped into char[] arrays, which are used immediately to initialize proper java.lang.String objects to represent the string constants. Note that these are the first objects whose allocation is backed by the regular use of the NEW opcode. Subsequently, patch vectors are used again to point at the locations to be patched within the class objects.

Having now completed all of the necessary code patching, it is time to transform the plain integer arrays used so far into proper objects of type java.lang.Class. Iterating over the whole class resolution table for a last time, all type identifiers of the former arrays are overridden by references to the now available class object representing java.lang.Class. Doing so, also the fully-qualified names of the classes are resolved from their string identifiers. These names have been transmitted before as regular string constants and are contained in the resolution table. During this procedure, the name to class mappings are entered into a hashtable that will back the implementation of Class.forName(). A few internal system classes, like the bootloader class itself, are excluded from this hashtable. This does not only hide them even from the Class.forName()

method but it also makes the representing class objects eligible for garbage collection when they are no longer needed.

Obviously, the transformation from int[] arrays into Class objects requires some magic that is not and should not be freely available to regular user code. While such magic is often hidden within more privileged native method implementations, the solution turns out to be somewhat simpler in our case. Simply granting the bootloader access to the type and name fields of java.lang.Class and having the linker throw away all CHECKCAST instructions contained in the bootloader code achieves the needed privileges. This, of course, immediately asks for extra caution when programming the bootloader. On the other hand, such caution is imperative for implementing core system functions anyhow. Moreso, the exception machinery, which would be invoked by a failing CHECKCAST, is not fully functional during bootstrapping so that the constraints of using it would be harder to overlook than avoiding it altogether.

At this point of time, both resolution tables have served their purpose and may be discarded. All other objects destined for further use in the running system have been completed by a proper type identification. Only the thread object representing the running control flow is still orphaned and initialized sloppily. While the bootloader circumvented any problems with this object simply by not using it, it is not fit for the execution of user code, which may very well query for Thread.currentThread() and expect to obtain a regular java.lang.Thread object. The bootloader does not even try to fix this thread object. Instead, a startup class extending java.lang.Thread is instantiated and started. The parent bootloader thread is simply terminated.

3.4 Starting the User Code

The startup class has been synthesized by the offline linker. Its first task is to complete the initialization of the runtime system by executing the static class initializers of the loaded classes. As this code is only needed this single time, the linker has moved it from the defining classes to the startup class, which, like the bootloader class, will be made eligible for garbage collection when its work is finished. Finally, it forks a new Thread for each task defined for the booted system. This may be a single application or may also include system services as I/O interfacing or dynamic application loading. Having done so, also this thread is terminated and booting is completed.

4 Side Concerns

4.1 Garbage Collection

SHAP incorporates a memory manager organizing the available RAM as an object heap with autonomous concurrent garbage collection. While its high-level interface provides for object allocation and field accesses, the reclamation of memory is the sole responsibility of the comprised garbage collector (GC).

The complete discussion of the implemented GC is far beyond the scope of this paper. The only issue of concern here is: What keeps the GC from collecting and reclaiming the space of the loaded `Class` objects and of the `Strings` used as constants?

The GC recognized the execution stack and a few microcode variables as root set for the graph of accessible objects. The execution stack contains a frame structure for each method in the active call hierarchy, which holds a reference to the defining class object. This ensures that class objects containing code in use are never reclaimed by the GC¹. This way, the code base of the bootloader and the later running applications are protected. As mentioned before, the bootloader and the startup class are made eligible for garbage collection by terminating their execution threads when their work is finished.

String constants used by a class must be kept alive explicitly by a private array within the class object that duplicates their references. Due to the patching performed by the loader, a reference to each string is, indeed, also contained within the code array using it. These references are, however, not detected by the GC, which would require some means to cope with unaligned reference values located at arbitrary byte rather than word offsets. On SHAP, the references patched in are additionally shortened to a significant length of 16 bit. In the end, concentrating all references to be kept alive for the use of the code within a single array seemed the best option at hand.

Also, simple type constants, `INSTANCEOF` and `CHECKCAST` checks as well as sites of static member accesses may reference other class objects that, thus, must be protected from the GC. Note that this is not necessary for virtual call sites as the runtime object taking the role of the `this` argument must reference the invoked code definition as an instance of some class defining or inheriting this code. While the collection of referenced classes may be collected within the alive array together with the `String` constants, this is not necessary when application bundles are considered indivisible entities. Currently, latter is achieved by the hashtable constructed to back the `Class.forName()` implementation. This keeps alive all application classes as long as a single thread of an application is executing.

4.2 Optimizations

A few simple optimizations are possible and realized by the current bootloader implementation used for SHAP. The most essential ones target the removal of duplicates among the string constants and the references kept within the alive array. Such duplicates are currently recognized by the offline linker and communicated to the bootloader.

¹For non-static methods, there is, in general, an additional indirect reference path to the code-defining class starting from the `this`-argument going over its type field and potentially an arbitrary number of superclasses. This path is, however, not reliable as the slot for the `this`-argument is a regular local variable slot, which may be re-used if the contained value is no longer needed – although current Java compilers seem to be very reluctant to do so.

Concerning the string constants, the offline linker only needs to ensure that all slots of the constructed resolution table contain distinct entries. The request for a slot for a string already within the table is plainly mapped to the existing one. The result is the use of only one single instance of a `String` for each distinct constant within an application bundle. The more thorough dynamic merger of equal strings or merely the `String.intern()` method have not been implemented on SHAP. They are likely to be added when the ongoing work on the weak reference support on SHAP is completed.

Duplicates are also removed from the alive array, as a single reference fulfills the purpose of protecting the target from the garbage collector. As type references display a higher degree of bunching than the use of equal strings, a significant space benefit is, however, only gained when the class references are maintained here locally instead of the application-wide hashtable. While this approach would also allow a fine-grained class unloading, it would jeopardize many uses of `Class.forName()` unless an actively polling class loading is added at the same time. A table of loaded classes would then have to be implemented using weak references.

5 Summary

This paper described the bootloader design implemented for the SHAP microarchitecture. The bootloader is largely implemented in Java although it is also responsible for performing transformations on the loaded code including its own. The design allows to restrict the implementation of bytecodes their quick variants, which are generally even simpler. The phased design of the bootloader allowing the transformation of subsequent phases enables the gradual increase of the available system function until the system is fully booted. The required additional privileges to implement such a low-level system task have been shown to be minimal and to integrate nicely into the implementation language Java. The side issues concerning the concrete implementation on SHAP have been discussed.

References

- [1] E. M. Gagnon and L. J. Hendren. Sablevm: A research framework for the efficient execution of Java bytecode. In *Java Virtual Machine Research and Technology Symposium*, pages 27–40, Apr. 2001.
- [2] S. A. Ito, L. Carro, and R. P. Jacobi. Making Java work for microcontroller applications. *IEEE Des. Test*, 18(5):100–110, 2001.
- [3] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification*. Addison-Wesley Professional, 1 edition, Sept. 1996.
- [4] T. B. Preußner, M. Zabel, and R. G. Spallek. Bump-pointer method caching for embedded Java processors. In *The 5th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES 2007*, 2007.
- [5] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.
- [6] M. Zabel, T. B. Preußner, P. Reichel, and R. G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. IEEE Press, Aug. 2007.
- [7] R. Zulauf. Entwurf eines Java-Mikrocontrollers und prototypische Implementierung auf einem FPGA. Diplomarbeit, Universität Karlsruhe, Apr. 2000. <http://ipr.ira.uka.de/komodo/zulauf/sources/RZDA-2s.pdf>.