

# Code Generation At The Proxy: An Infrastructure-Based Approach To Ubiquitous Mobile Code

Deepak Chandra, Christian Fensch, Won-Kee Hong, Lei Wang, Efe Yardımcı,  
Michael Franz

University of California, Irvine  
Department of Information and Computer Science  
Irvine, CA 92697, USA

{*dchandra, fensch, wkhong, wangglei, yardimci, franz*}@ics.uci.edu

## Abstract

Current approaches to mobile code are ill-suited for resource-constrained devices such as mobile phones and personal digital appliances — yet it is exactly these devices that are the most attractive targets of mobile-code technology. Our approach to reconciling this apparent contradiction is to off-load much of the required mobile-code support machinery into the (wireless) infrastructure, rather than the mobile device itself. We are implementing a system involving a powerful proxy which is in continuous contact with the mobile device. This interaction can be utilised to perform runtime improvements on processes running on the mobile device. We believe this approach will lead to various benefits such as execution speed, optimisations for power efficiency, and the ability to execute beyond hardware limitations such as strict memory constraints. In this paper we describe our development plan, the current state of our prototype, and the directions we will take in the future.

## 1 Introduction

Mobile code allows us to execute software which does not have to be installed on the local platform. This makes it ideal for usage in mobile devices that are limited in resources such as storage memory. The most popular mobile code format is Java bytecode. It is executed on the Java Virtual Machine (JVM) [9].

However, executing mobile code on a mobile device has a lot of inherent problems. The main limitation of mobile devices are power consumption and size restrictions which lead to several other constraints like processor speed and memory size (faster processor consumes more power and requires bigger heat-sinks, more memory also needs more power and requires more space). These limitations make impractical or impossible the application of techniques [4, 6, 8] used in desktop environments to increase the performance of mobile code. In some cases the mobile program simply cannot be executed since the mobile device is lacking the required resources.

Our main idea to solve this problem is to use a powerful (i.e., relatively unconstrained) stationary computer to support the mobile device. The first idea is that the stationary computer, subsequently known as a *proxy*, performs the compilation (and verification) of the mobile code into the machine code format of the mobile device. That way all techniques used in today's compilers could be applied. The possibilities to support the mobile device do not end here. For example, running a lightweight profiler on the mobile device enables the proxy to optimise hot-spots, replacing the original unoptimised code on the mobile device. The profiling results might be piggybacked to data which might have to be sent through the proxy anyway. In an even more radical scenario some parts of the application would be executed on the proxy. In this case the application would be partitioned between the proxy and the mobile device.

Using this approach we hope to obtain improvements in the following areas:

**Power** During the verification of the mobile code the proxy will definitely save power, even compared to hardware-supported JVM on the mobile device. Another saving in energy could be observed through switching from a stack-based machine (like the JVM) to a register-based approach due to reduction in code size and memory access.

**Speed** Considering the speed improvements Just-In-Time compilers achieved in desktop environments compared to traditional interpretative approach, a similar effect should be possible by generating optimised native code at the proxy.

**Physical Limits** By offloading computation to the proxy, it could be possible to execute applications on the mobile device that otherwise could not possibly be executed due to hardware limits (such as memory size).

This paper is organised as follows: Section 2 discusses the related work. The overall architecture of the prototype system and features that are considered to be implemented are discussed in Section 3. Section 4 describes the proxy and the runtime environment of the current prototype system. Some final remarks and future work will be drawn in Section 5.

## 2 Related Work

In general there are three methods of executing Java bytecode. The first of these involves the most common method of using *interpreters*. These execute the Java bytecode instructions one by one. As a second method, Just-In-Time (JIT) compilers perform on-the-fly code generation. These are intended to help the Java Virtual Machine (JVM) to increase the speed of applications by replacing frequently executed code with native equivalents. This may highly improve execution speeds for loop- and recursion intensive applications. There have been many developments on JIT compilers. Ebcioğlu, et al. [4] described a JIT compilation scheme for the JVM. The first JITs were implemented by Netscape and Microsoft for their browsers. Sun Microsystems has released JIT compilers for several platforms. Krall and Grafl [8] developed a 64 bit JIT for the Alpha platform. A third approach, that of translation, has been taken by Hsieh, et al. [6], whose Caffeine system compiles a complete program in advance. As far as we have been able to see from the current literature, usage of non-interpretation based schemes on an embedded device such as a handheld has not proved to be very practical. Including and running a compiler alongside a Java program, especially with such comparatively limited hardware has been very difficult. We believe that a promising method to implement efficient execution of Java programs in mobile devices is partitioning.

One angle towards partitioning in Java platform is to partition the JVM itself. One such effort is a distributed JVM designed by Sirer, et al. [11], which decomposes a JVM in a runtime, a verifier, an optimiser, a performance monitoring service and a security manager. However they are taking a much broader approach than appropriate resource constrained environment, by not limiting themselves to just mobile devices. They also have realized that having a distributed JVM reduces the resource requirements on the client site.

Several further research efforts have been made towards Java application partitioning. JavaParty [10] and Pangaea [12] are designed for partitioning Java applications at source code level. And more recent projects, Addistant system [13] and J-Orchestra [14] partition applications at Java bytecode level. Those projects aim at enabling distributed execution of centralised Java programs in order to adapt them to distributed systems. However one major drawback is that users have to configure ahead of time how to do the partitioning. Even in the automatic approach proposed by J-Orchestra, the user still needs to clearly define the network location of resources and the corresponding Java classes. Obviously this is not flexible and practical in the real world. Considering this, our goal is to bring up an approach to partition applications on the fly at run time.

## 3 Research Plan

### 3.1 Architectural Overview of the Prototype System

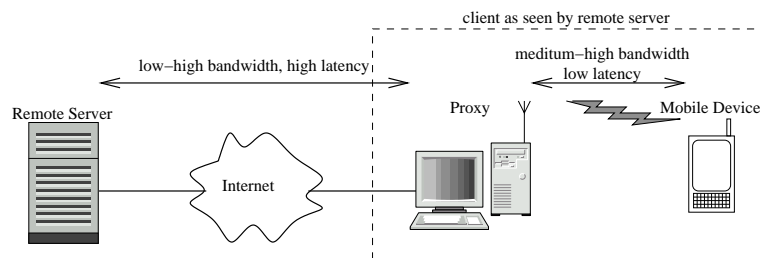


Figure 1: Overview of the Prototype System

As shown in Figure 1, the prototype system being developed is a three-level architecture, where an active proxy is interposed between the mobile code generator and the eventual target device executing the code. A highly configurable Just-in-Time compiler/bytecode translator runs on the powerful interposed proxy, which could be a router or wireless base station. In order to make it practical, the prototype system will use the ordinary Java class

files as the mobile code. As the code generating router, a personal computer with a off-the-shelf 802.11b wireless network device will be used. The ARM RISC processor is used for a target processor, which is widely used in not only several commodities [2] but also a variety projects under DARPA's Power Aware Computing/Communication Program [3]. The relatively low-cost off-the-shelf handheld "PDA-class" devices running mobile Linux are used. The key characteristic of this model is that there is a high-bandwidth, low latency connection between the proxy and mobile device.

From the server side, the proxy-mobile pair acts as an atomic entity that is physically located wherever the mobile device happens to be. The mobile device issues all of its requests to the server through the proxy. The proxy intercepts such requests and re-issues them. When the response of the server to such a request is a web page, it is promptly forwarded to the mobile device. But if it is a Java executable, it will get no further than the proxy. Any Java executable returned by the server is at first verified on the proxy, and if it passes verification, subsequently compiled on-the-fly into the native code of the mobile device. The native program for the mobile device is then transmitted to the mobile device and linked with the libraries and code files already present<sup>1</sup>.

## 3.2 Application Partition in Two Levels of the Execution

Keeping our main goal in mind, which is to achieve an efficient mobile application environment, we argue that the key is to flexibly partition the execution between the proxy and mobile device in order to offload the resource- or computation-intensive parts from the mobile device appropriately. However the partitioning criteria may differ when aiming at different goals such as low power consumption, fast execution and ability to run applications larger than the available memory at the mobile device. Our approach is completely software based which will complement expected improvements in hardware of the mobile devices in next few years. As described in the previous sections, our test-bed is a proxy — mobile device architecture running Java bytecode applications. We propose a two-level application partition structure, as shown in the following sections.

### 3.2.1 Partition of Execution Environment (JVM)

Partition at this level is relatively straightforward, separating compilation from execution. The computation-intensive verification, compilation and optimisation can be done at the proxy, and the mobile device only executes the generated native code it retrieves from the proxy.

**Dynamic Compilation and Optimisation** Compilation in our proposal has two stages. The first stage emphasises on quick generation of code so that the mobile device can start execution as early as possible, whereas the second stage tries to dynamically optimise code further. Dynamic compilation and optimisation are achieved by combining the result of both static analysis of the control flow and the runtime profiling of the execution. The code generated in the second stage can then be hot swapped with the currently running code.

**Linking** The linker can be located either at mobile device or at the proxy. Linking at the mobile device is easy to implement with our current system, and we don't have to interfere with the memory management of the OS at the mobile device. Linking at the proxy would necessitate the guided placement of compiled code. However, this latter approach has significant advantages in the memory constrained handheld environment since linking remotely eliminates the need for the storage of large symbol tables at the mobile device.

### 3.2.2 Partition of Execution

To more aggressively achieve goals like low power consumption, high efficiency and larger "virtual memory" partitioning at this level is crucial. If properly partitioned, the user may experience a far more powerful mobile device than what it actually is. Not only can we achieve much higher computational capabilities and be able to execute programs which are presently constrained by the size of the memory but will also be able to reduce the power needs of the mobile device.

For the purposes of application level partitioning we suggest to exploit the boundaries at the graphical user interface (GUI) level or the logical partitioning that the object oriented languages inherently provide in the form of classes.

We believe that efficient partitioning can be achieved at object boundaries. The granularity of the partitioning will however depend on the type of application. For coarse granularity a whole object or even a package can be

---

<sup>1</sup>Our current testing format is Java bytecode. We will extend this frame work also to the emerging .NET platform using Microsoft Intermediate Language (MSIL) format. This will provide us with an alternative environment to evaluate our system. Another possible mobile code format is SafeTSA [1].

moved over to the base station whereas for finer granularity only execution of specific methods of the class can be shifted to the base station. Possible partitioning criteria can be processing time and memory needs and execution frequency of a function, all of which can be obtained by profiling.

## 4 Our Prototype System

We are currently developing the infrastructure that we will use to implement this system. For a system with the mentioned properties this involves doing development on multiple fronts, but the interaction between the proxy and the mobile device forms the most crucial part of the system. Thus we have started off by implementing a compiler on the proxy, a supporting runtime system on the mobile device and a simple network connection between the proxy and mobile device. The status of the first two parts will be discussed in the following sections in greater detail. As for the network connection, both devices are still connected over a wired connection which supports the Transmission Control Protocol. As our first mobile code format we will be implementing the widespread Java bytecode format.

### 4.1 The Proxy

The proxy answers a class request by the mobile device. It tries to load and set up the requested class and all super classes recursively. The setup step involves gathering information about the class such as the fields and methods it contains and the access permissions to these members. During this phase the *class descriptor* for the loaded classes is created. The class descriptors hold information on the position of the class' static fields and method handles, a reference to the descriptor of the superclass, the size of instances of this class, and information which is necessary in order to support Java interfaces and reflection.

After the creation of class descriptors, the proxy compiles all methods to native ARM machine code, by replacing each byte code instruction with a sequence of ARM instructions in order to simulate the behaviour of the Java stack machine. Although this stack based approach is not very efficient (both in execution performance and power consumption, due to heavy memory access), it enables us to implement a full featured JVM very quickly. If during the compilation it becomes obvious that additional classes are needed, the compilation is interrupted and these classes are loaded. During the compilation the proxy does some basic bytecode verification, such as checking that access rights to class members are not violated.

Currently the proxy tries to load every class and compile every method, which might be used during the program execution. A later version might only do this for classes or methods which are definitely needed.

After the proxy has completed the compilation the native code is sent as one block to mobile device. The mobile device returns the address at which it had mapped the code, with which the proxy updates the method references inside the class descriptors. Then the parts of the class descriptor which are needed by the runtime environment are sent to the mobile device.

At this point we are supporting most Java byte code instructions except instructions which deal with the primitive data types float, double and long. Also *invokeinterface*, *tableswitch*, *lookupswitch*, *athrow*, *jsr* and exception handling still need to be implemented.

### 4.2 The Runtime Environment

The other part of the project at which we have been directing our efforts is the construction of an efficient runtime environment on the mobile device. This runtime environment would be responsible for supporting the object-oriented aspect of the executed programs by maintaining the support structures and providing services such as thread support, access to the underlying operating system, garbage collection and maybe exception handling. This involves more than simply converting Java bytecodes to sequences of native ARM instructions, we need to have full support for executing a program stored in a object-oriented mobile code format such as Java bytecode.

The runtime environment provides the services necessary to fulfil this task. It implements the services required by Java Virtual Machine Specification [9] and forms an interface between the program and the OS. Currently we are able to run basic object-oriented java programs with support for method invocations, java objects and arrays, and synchronisation with locks and monitors. We are close to having full multi-threading capabilities and will soon implement a memory management module with garbage collection.

Besides requesting and loading code from the proxy, the runtime environment offers support to the running program mostly through a library of *runtime services*. These range from implementation of Java bytecode instruction such as *instanceof*, *checkcast* to *invokevirtual*, *new*, *newarray*, *monitorenter* (which are

mostly built on top of the underlying operating system) to services having to do with the internal structures the runtime environment maintains for proper object-oriented execution. It also has to implement certain native methods in the Java API.

The main method of supporting object-oriented program execution is by maintaining *class descriptors* for each class that has been loaded to the mobile device. All class descriptors are placed in a contiguous *class descriptor block*, whose location can change as long as we are able to obtain the starting address through a PDA runtime service. Our class instances do not consist of two pointers as in the Sun JDK representation [6], or a single pointer included with the data (a common feature in JITs [6, 8]), but the offset of the object's class descriptor inside the class descriptor block. Having an offset into the class descriptor block instead of the actual address allows us to grow the class descriptor block very easily inside the heap segment when further classes are loaded dynamically.

Whenever the runtime environment receives a code block of compiled methods, it maps this to memory using the system calls of the underlying operating system (our present implementation platform is the Sharp SL-5000D PDA using the Embedix embedded Linux operating system). The address of the mapped code block is then sent back back to the proxy, which updates the method handles inside the *class descriptors* for all affected classes. The modified class descriptors are then sent and appended to the existing class descriptor block.

We are using the PThreads [7] user-level thread package to implement monitors and locks. The `monitorenter` and `monitorexit` Java byte code instructions are supported, and we are very close to having a multi-thread capabilities through the implementation of classes `Thread` and `ThreadGroup`.

Completion of these services and support structures according to the complete specifications will then allow us to proceed implementing lightweight profiling on the running applications. Results of profiling will be used to apply target optimisations with the support of the powerful proxy.

## 5 Conclusion

In this paper, we have explained our strategy for implementing efficient execution of object-oriented mobile code. We have given an explanation of why we think state-of-the-art Java technologies are not suitable in their current state for small mobile devices. We have explained our vision of a distributed system where powerful proxy will take significant portions of the workload off the mobile devices.

Our proposed system will involve continuous runtime profiling of the running code. The system will rely on close interaction between the mobile device and the proxy on many levels, using the data gathered at runtime from profiling. One obvious way is reconstructing observed hot regions and hot swapping to optimise for lowpower or execution speed. Another is to do dynamic partitioning of the process to lighten the load on the mobile devices and utilise the powerful proxy.

We hope to have a completely running prototype of this system and begin testing our optimisation strategies by the beginning of June.

Finally, as a long term perspective we are also going to explore the following areas:

**Process Migration in “Handover”** Our object system is a distributed system with clusters of proxies and mobile devices. When a mobile device roams to a different proxy, the current running process should not be interrupted. Thus in order to achieve a smooth “handover”, we need to migrate all the related execution states and connections from the current proxy to the new one.

**Remote Memory Management** We currently use Linux on the mobile device, which limits the fully remote control of the mobile device's memory. Achieving remote memory management can not only lead to a more memory efficient execution model, but can also facilitate an easier implementation in remote linking, remote execution, and remote garbage collection. Another idea is to reduce instruction cache conflicts by observing and acting upon the interaction between methods. This strategy has been implemented before by Hashemi et al. [5], although their method calls for going through a separate profiling stage beforehand.

## References

- [1] Wolfram Amme, Niall Dalton, Jeffrey von Ronne, and Michael Franz. SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIG-PLAN'01 conference on Programming language design and implementation*, pages 137–147. ACM Press, 2001.
- [2] <http://www.handhelds.org/platforms.html>.

- [3] <http://www.darpa.mil/ito/research/pacc/index.html>.
- [4] K. Ebcioglu, E. Altman, and E. Hokenek. A java ilp machine based on fast dynamic compilation. In *MAS-COTS'97 - International Workshop on Security and Efficiency Aspects of Java*, 1997.
- [5] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–182, 1997.
- [6] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen-mei W. Hwu. Java bytecode to native code translation: the caffeine prototype and preliminary results. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture*, pages 90–99. IEEE Computer Society Press, December 1996.
- [7] IEEE. Threads Extension for Portable Operating Systems (Draft 6), February 1992. P1003.4a/D6.
- [8] Andreas Krall and Reinhard Graf. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, January 1997.
- [9] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
- [10] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.
- [11] Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Symposium on Operating Systems Principles*, pages 202–216, December 1999.
- [12] Andre Spiegel. PANGAEA: An automatic distribution front-end for JAVA. In *IPPS/SPDP Workshops*, pages 93–99, 1999.
- [13] Michiaki Tsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of ‘legacy’ java software. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.
- [14] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. Technical report, Georgia Institute of Technology, July 2001.