Running Android Applications without a Virtual Machine

Arno Puder

San Francisco State University
Computer Science Department
1600 Holloway Avenue
San Francisco, CA 94132
arno@sfsu.edu

Abstract. Android has gained significant popularity in the smartphone market since its introduction in 2007. While Android applications are written in Java, Android uses its own virtual machine called Dalvik. Other smartphone platforms, most notably Apple's iOS, do not permit the installation of any kind of virtual machine. App developers who want to publish their applications for different platforms are required to re-implement the application using the respective native SDK. In this paper we describe a cross-compilation approach, whereby Android applications are cross-compiled to portable C code. With this approach it is not necessary to have a Dalvik virtual machine deployed on the target platform. We describe different aspects of our cross-compiler, from byte code level cross-compilation, memory management, to API mapping. A prototype of our cross-compiler called XMLVM is available under an Open Source license.

1 Introduction

Android is a software stack for mobile devices initially developed by a company called Android, Inc. before being bought by Google in 2005. Since 2007, members of the Open Handset Alliance (OHA) collaborate on the development of Android which nowadays has become one of the main development platforms for smartphone applications. Although Android employs Java as a programming language as well as a subset of the standard J2SE API, it does not make use of Oracle's (formerly Sun Microsystem's) virtual machine technology for technical and political reasons. Android features its own virtual machine, called Dalvik [5], for running Android-based applications. In contrast to the Oracle virtual machine, Dalvik is based on a register-based byte code instruction set. Running Android applications therefore requires the Dalvik virtual machine on the target platform.

Developers targeting smartphones ideally want their applications to be available on as many platforms as possible to increase the potential dissemination. Given the differences in the way applications are written for smartphones, this incurs significant effort in porting the same application to various platforms. The

Dalvik virtual machine is not available on all smartphone platforms of interest. In particular, Apple explicitly forbids the use of virtual machine technology on their iOS devices, making it impossible to run applications that rely on Dalvik.

In this paper, we introduce a cross-compilation approach, whereby an Android application can be cross-compiled to portable C code. The solution we propose not only cross-compiles on a language level, but also maps APIs between different platforms. Since C is allowed for iOS development, Android applications can therefore be cross-compiled to Apple's devices. The benefit of our approach is that only skill set for the Android platform is required and only one code base needs to be maintained for both devices.

This paper is organized as follows: Section 2 provides an overview of Android and iOS as well as the deficiencies of previous work. Section 3 presents our cross-compilation framework that can cross-compile Android applications to iOS devices. In Section 4, we discuss our prototype implementation of this framework as well as a monitoring application that was cross-compiled using our toolchain. Finally, Section 5 provides a conclusion and an outlook to future work.

2 Background

We first provide a brief overview of Android and iOS from a programmers perspective highlighting the differences in their programming models (Section 2.1) followed by a discussion on the shortcomings of previous work (Section 2.2).

2.1 Overview of Android and iOS

Although smartphones are relatively similar with respect to their hardware capabilities, they differ greatly in their native application development models. Android is a mobile operating system running on the Linux kernel. Android is not exclusively targeting smartphones, but is also available for netbooks and settop boxes. Next to Android, Apple claims a firm position in the smartphone market with their proprietary iOS platform. Its user interface is called Cocoa Touch which is an extension of the Cocoa framework that is used on Apple desktop and laptop computers.

Targeting Android as well as iOS devices requires significant skill sets and overhead. Whereas Android uses Java as the development language, Cocoa Touch uses Objective-C. Those two languages are radically different. While Java features strong typing and garbage collection, the version of Objective-C used on iOS devices supports dynamic typing, but no garbage collection.

Similarly, differences exist in the APIs and programming models defined by Android and Cocoa Touch. An Android application consists of a set of so-called *activities*. An activity is a user interaction that may have one or more input screens. An example for an activity is to select a contact from the internal address book. The user may flip through the contact list or may use a search box. These actions are combined to an activity. Activities have a well-defined life cycle and can be invoked from other activities (even activities from other applications). Besides a variety of widgets, Android also allows the declarative description of user interfaces. XML files describe the relative layout of a user interface which not only simplifies internationalization but also allows to render the user interface on different screen resolutions.

The only official language offered by Apple for iOS devices is Objective-C. Similar to C++, Objective-C is an object-oriented extension of the C programming language. Analogous to Smalltalk, an object can be sent any message. Since the version of Objective-C used for iOS devices does not integrate a garbage collector, the programmer has to use a low-level reference counting mechanism for memory management. The design of Cocoa Touch makes extensive use of Objective-C's dynamic typing. Cocoa Touch offers a variety of UI elements. However, unlike Android, Cocoa Touch offers no layout manager: all UI elements have to be positioned in terms of absolute coordinates on the screen at design time.

2.2 Previous Work

We have worked on cross-compilation in the past [10]. Our cross-compiler, called XMLVM, translates Java byte code instructions represented in XML to a high-level programming language. In the past, the exclusive target for cross-compiling Android applications were iOS devices. For this reason, Java byte code instructions were cross-compiled to the native programming language for iOS devices, namely Objective-C. XMLVM mapped Java classes to Objective-C classes. While this proved to be a simple mapping, it also led to various problems. For one, Objective-C allows overriding but not overloading of methods. Since Java supports both, method names need to be name mangled with their signature when generating Objective-C method names. While name mangling solves the problem of method overloading in Objective-C, the one-to-one mapping of the object models leads to other problems. E.g., Objective-C does not allow to override instance members, so the following Java program cannot be cross-compiled with the Objective-C backend of XMLVM:

```java
// Java
class Base {
    int x;
}

class Derived extends Base {
    int x;
}
```

Another shortcoming stems from the fact that Apple did not include a garbage collector in the version of Objective-C used for iOS applications. XMLVM's backend make use of iOS' reference counting mechanism by inserting retain and release instructions into the generated Objective-C code. One issue with this approach are cyclic data structures which cannot be reclaimed via reference counting. In order to avoid memory leaks, it is the programmer's responsibility to break cycles by using `java.lang.ref.WeakReference`. Instances of `WeakReference` are ignored by the garbage collector when constructing the reachability graph. Another problem is caused by multi-threaded applications: because it is impossible to know what other threads are doing with individual objects, the inserted reference counting instructions need to be conservative and retain objects for the duration of their use. This leads to significant overhead because the majority of the generated code is related to the retaining and releasing of object references.

Because of these shortcomings we decided to replace the Objective-C backend with a code generator for the C programming language. Since C is a strict subset of Objective-C, the cross-compiled code can still be targeted for iOS devices. Generating C code also allows support for the full Java language as well as compile-time optimizations. The following section gives a detailed overview of the C code generator in XMLVM.

3 Cross-Compilation Framework

In this section we introduce XMLVM, a flexible, byte code level cross-compiler, that allows to translate an Android application to iOS devices. In Section 3.1 we give an overview of the XMLVM toolchain. Section 3.2 explains our byte code level cross-compiler to the C programming language. In Section 3.3 we outline the API mapping and finally in Section 3.4 we discuss the integration of a Garbage Collector.

3.1 Toolchain

We chose Android as the canonical platform. This means that a developer only needs to be familiar with the Android system and can then cross-compile an Android application to other smartphones. There are several reasons for choosing Android. First of all, we believe that there is a wide skill set for the Java programming language and there are powerful tools to develop for Java. We view this as an advantage over Objective-C.

The design of Android itself offers various advantages. For one, Android was not exclusively designed for smartphones, but for a wide range of mobile devices. Android's API allows to explore the device's capabilities to give the application the chance to adapt accordingly. Android offers layout managers that can adapt to different screen resolutions at runtime. Cocoa Touch on the other hand expects the programmer to position every widget in terms of absolute coordinates. Since Apple offers with the iPhone 3GS, iPhone 4, and iPad three devices with different screen resolutions, the burden is placed on the programmer who has to manually design three UIs. Since Android applications can more easily adapt to different devices, it makes them ideal candidates to be cross-compiled to other platforms.

The XMLVM toolchain is a byte code level cross-compiler. The output of a Java compiler is first translated to an XML document to allow easy transformations based on XSL stylesheets, as will be shown in the following subsection. XMLVM makes use of Android's Dalvik virtual machine [5] for representing byte code instructions. Dalvik is based on a register-based virtual machine which allows for the generation of more efficient code in the target language compared to Oracle's stack-based virtual machine instructions [8]. On the basis of the intermittent XML representation we generate code in the target language, which in our case is C. In the following examples we focus on the register-based format of XMLVM as the starting point of the cross-compilation. The following three subsections explain in detail the cross-compilation process, the API mapping via a compatibility library as well as memory management via garbage collection.

3.2 Byte Code Level Cross-Compilation

A unique property of our toolchain is that we cross-compile from byte code to high-level programming languages. We make use of the byte code instructions introduced by the Dalvik instruction set to allow more efficient code generation. In a previous project we have used a similar approach to cross-compile byte code to JavaScript for AJAX applications [9]. Using byte codes has several advantages. For one, byte codes are much easier to parse than Java source code. Several high-level language features such as generics are already reduced to low-level byte code instructions. The Java compiler also does extensive optimizations to produce efficient byte codes. To illustrate our approach, consider the following simple Java class:

```java
// Java
public class Account {
    int     balance;
    boolean overdraftProtection;

    // ...

    void deposit(int amount) {
        balance += amount;
    }
}
```

Class `Account` has a method called `deposit()` that adds a given amount to the `balance` of an account. The source code is first compiled to a Java class file via a regular Java compiler. The binary class file is then fed into our XMLVM tool. The first transformation performed on the resulting XMLVM is to convert the stack-based byte code instructions to register-based instructions introduced by Dalvik [5]. The conversion from a stack-based to a register-based machine has been researched extensively [3,11]. Internally, XMLVM generates the following XMLVM document based on class `Account`:

```
1 <vm:xmlvm ...>
2   <vm:class name="Account" ...>
3     <vm:field name="balance" type="int" />
4     <vm:field name="overdraftProtection" type="boolean" />
5     <vm:method name="deposit" ...>
6       <vm:signature>
7         <vm:return type="void" />
8         <vm:parameter type="int" />
9       </vm:signature>
10      <dex:code num-registers="3">
11        <dex:iget class-type="Account" field="balance" vx="0" vy="1" />
12        <dex:add-int vx="0" vy="0" vz="2" />
13        <dex:iput class-type="Account" field="balance" vx="0" vy="1" />
14        <dex:return-void />
15      </dex:code>
16    </vm:method>
17  </vm:class>
18 </vm:xmlvm>
```

The reason our tool is called XMLVM is because the structure of the class file as well as the byte code instructions of a virtual machine are represented via appropriate XML tags. On the top-level, there are tags to represent the class definition (line 2), field definitions (lines 3 and 4), method definition (line 5), and the signature of the method (line 6–9). The children of tag <`dex:code`> (line 10) represent the byte code instructions for method `deposit()`. The attribute `num-registers` denotes the number of registers required to execute this method.

In the following we give a brief overview of the byte code instructions generated for method `deposit()`. Upon entering a method, the last n registers are automatically initialized with the n actual parameters. Since method `deposit()` has three registers labeled 0 to 2, register 2 will be initialized with the one actual parameter of that method (the amount). The implicit `this`-parameter counts as a parameter and will therefore be copied to register 1. The byte code instructions read and write to various registers that are referred to via attributes vx, vy, and vz, where vx usually designates the register that stores the result of the operation. The first instruction <`dex:iget`> (*instance get*) loads the content of field `balance` of the account object referenced by register 1 into register 0 (line 11). The <`dex:add-int`> (*add integer*) instruction in line 12 will add the integers in registers 0 (the current balance) and 2 (the actual parameter) and store the sum in register 0. This instruction performs the operation $vx = vy + vz$. The <`dex:iput`> (*instance put*) instruction in line 13 performs the opposite of <`dex:iget`>: the content of register 0 is stored in field `balance` of the object referenced by register 1.

Once an XML representation of a byte code program has been generated, it is possible to cross-compile the byte code instructions to arbitrary high-level languages such as C, by simply mimicking the register machine in the target language. Individual registers are mapped to variables in C. Since a register can contain different data types, we introduce a C-union that reflects these data types:

```
// C
typedef union {
    JAVA_OBJECT o;
    JAVA_INT    i;
    JAVA_FLOAT  f;
    JAVA_DOUBLE d;
    JAVA_LONG   l;
} XMLVMElem;
```

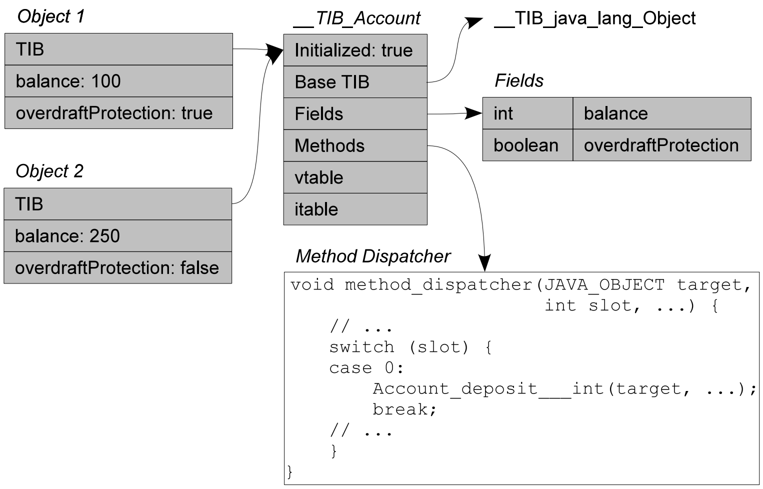
Registers can only store object references, integers, floats, doubles, and longs. Shorter primitive types such as bytes and shorts are sign-extended to 32-bit integers. With the help of the union `XMLVMElem`, it is possible to use XSL stylesheets [12] to produce code in the target language. In the following we show how the aforementioned byte code instruction <`dex:add-int`> is mapped to C source code:

```
<!-- XSL template -->
<xsl:template match="dex:add-int">
    <xsl:text>    _r</xsl:text>
    <xsl:value-of select="@vx"/>
    <xsl:text>.i = _r</xsl:text>
    <xsl:value-of select="@vy"/>
    <xsl:text>.i + _r</xsl:text>
    <xsl:value-of select="@vz"/>
    <xsl:text>.i;</xsl:text>
</xsl:template>
```

Register variables are always prefixed with `_r` followed by the register number. The definition of these helper variables are based on union `XMLVMElem`. These variables are automatically generated by other XSL templates during the code generation process. With the help of these variables, the effect of individual byte code instructions can easily be mapped to the target language. Applying all XSL templates to the XMLVM of class `Account` shown earlier yields the following C source code:

```
// Generated C
void Account_deposit___int(JAVA_OBJECT me, JAVA_INT n1)
{
    XMLVMElem _r0;
    XMLVMElem _r1;
    XMLVMElem _r2;
    _r1.o = me;
    _r2.i = n1;
9      _r0.i = ((Account*) _r1.o)->fields.balance;
10     _r0.i = _r0.i + _r2.i;
11     ((Account*) _r1.o)->fields.balance = _r0.i;
12     return;
13 }
```

The code in line 10 was generated by the XSL template for the <`dex:add-int`> instruction explained earlier. Every method of a class is mapped to a C function whose name is mangled from the class and method name as well as the method's signature. For each class, a so-called *Type Information Block* (TIB) is generated (see Figure 1). The TIB contains all relevant meta-data about a class such as fully qualified name, base class, all implemented interfaces, etc. When a class is instantiated, enough memory is allocated for all the instance members plus a pointer to the TIB. Since XMLVM cross-compiles byte code to C, there is no notion of dynamic class loading as usually found in a Java VM. A particular challenge poses the Java reflection API. In order to support dynamic method invocations via the reflection API, XMLVM creates method dispatchers for each class.



```
void method_dispatcher(JAVA_OBJECT target,
                       int slot, ...) {
    // ...
    switch (slot) {
    case 0:
        Account_deposit___int(target, ...);
        break;
    // ...
    }
}
```

Fig. 1. Type Information Block

Since XMLVM cannot load classes dynamically at runtime, all classes that are needed for an application need to be known at compile time. XMLVM performs a static dependency analysis to determine the transitive closure of all referenced classes. Since XMLVM operates under the *Closed World Assumption* (i.e., all referenced classes are known *a priori*) it is possible to perform powerful compile-time optimizations, especially for method invocations. Java distinguishes between two different dynamic method invocations on byte code level: <`dex:invoke-virtual`> is used for calling methods on an object, where as <dex:invoke-interface> is used to invoke methods via an interface type. The reason for two different invoke instructions has to do with the fact that Java allows single inheritance for classes but multiple inheritance for interfaces. For the former it is possible to build a so-called vtable which essentially is an array of function pointers. An <dex:invoke-virtual> instruction references a vtable index to the appropriate implementation of a method. Because of the Closed World Assumption it is known at compile-time whether a method is overridden in a derived class or not. If a method is not overridden, it can be called directly without the need of allocating an entry in the vtable.

Similar optimizations can be done for <dex:invoke-interface>. Here XMLVM computes an itable (analogous to the vtable) for interface methods. Because of the Closed World Assumption it is possible to compute an itable that can be indexed via one indirection. The technique is known as selector coloring and is described in [4]. The overhead of performing an <dex:invoke-interface> is identical to performing an <dex:invoke-virtual>.

The following code excerpt shows how a method invocation in Java (line 2) is represented in XMLVM (lines 5–10) and how it is mapped to C code. If method deposit() is not overridden in a derived class, it can be invoked via a direct function call (line 13). Otherwise method deposit() needs to be invoked via a vtable (lines 16–17). First, FPTR is defined as a function pointer of the proper type that reflects the signature of method deposit(), then the vtable is accessed via the TIB. In this particular example the function pointer for method deposit() is stored at vtable index 11.

```
 1 // Java
 2 account.deposit(amount);
 3
 4 <!-- XMLVM -->
 5 <dex:invoke-virtual class-type="Account" method="deposit" register="0">
 6   <dex:parameters>
 7     <dex:parameter type="int" register="1" />
 8     <dex:return type="void" />
 9   </dex:parameters>
10 </dex:invoke-virtual>
11
12 // C (direct)
13 Account_deposit___int(_r0.o, _r1.i);
14
15 // C (via vtable)
16 typedef *(void (*)(JAVA_OBJECT, JAVA_INT)) FPTR;
17 (FPTR ((Account*) _r0.o)->tib->vtable[11])(_r0.o, _r1.i);
```

3.3 API Mapping

Any Java program builds upon external API. The most common are the APIs defined as part of J2SE such as data structures. XMLVM leverages the Open Source project Apache Harmony for these purposes. Since much of the J2SE API is itself written in Java, we simply cross-compile this to C as well. On system-level, Apache Harmony uses native Java methods to access functionality of the underlying operating system. In XMLVM we have implemented those native methods based on the Posix API. E.g., the Java thread API is mapped via the native methods to pthreads.

Other APIs such as the one defined by Cocoa Touch are platform specific. Such API needs to be accessible from a Java application. E.g., if a Java-based iOS applications needs to place a label on the user interface, a Java-based API of class `UILabel` is required. The following shows the Java version of class `UILabel` that is part of the XMLVM library:

```
// Java

package org.xmlvm.iphone;

class UILabel extends UIView {
    //...
    void setText(String label) { /* Ignored */ }
}
```

We call classes such as `UILabel` *wrapper classes* since their only purpose is to provide a Java API against which the developer can implement an application. For that reason, the method implementation of wrapper classes can be left empty. Wrapper classes are treated special by XMLVM's cross-compiler. The implementation of a method in a wrapper class is ignored and instead special comment markers are emitted. The programmer can inject manually written code between these comment markers. The following code excerpt demonstrates this concept for method `UILabel.setText()`:

```
 1 // Objective-C (generated with injected code)
 2
 3 void org_xmlvm_iphone_UILabel_setText___java_lang_String(JAVA_OBJECT me,
 4                                                          JAVA_OBJECT n1)
 5 {
 6     //XMLVM_BEGIN_WRAPPER
 7     NSString* text = toNSString(n1);
 8     org_xmlvm_iphone_UILabel* thiz = me;
 9     UILabel* label = thiz->fields.wrappedObject;
10     [label setText:text];
11     [text release];
12     //XMLVM_END_WRAPPER
13 }
```

Note that while the generated wrapper code is C, the code injected by the developer between the comment markers (lines 6–12) is Objective-C. This is necessary because the wrapper effectively wraps an Objective-C object. The code above essentially converts a `java.lang.String` instance to an Objective-C `NSString` via a helper function (line 7) and retrieves the wrapped Objective-C `UILabel` instance (line 9). The field `wrappedObject` is part of the `UILabel` instance and points to the native Objective-C object. Next, the `setText:` message is sent to the Objective-C object (line 10) and finally the `NSString` instance is released (line 11).

Exposing the Cocoa Touch API in Java is the prerequisite for our *Android Compatibility Library* (ACL). The purpose of the ACL is to offer the Android API to an application while using the Java-based Cocoa Touch API for its own implementation. The ACL is therefore written in Java and maps the Android API to the Cocoa Touch API. It offers the same API to the application as Android does but is implemented using the Cocoa Touch wrapper classes. The following code excerpt shows how an Android `TextView` is mapped to a Cocoa Touch `UILabel`:

```
// ACL (Java)
package android.widget;

public class TextView {
    private UILabel label;
    // ...
    public void setText(String text) {
        label.setText(text);
    }
}
```

Therefore, an application using an `android.widget.TextView` is effectively using a `UILabel` on iOS devices via the ACL. The above code excerpt is also cross-compiled to C and is part of XMLVM's implementation of the ACL. In other cases, such as Android's layout manager, we cross-compile source code from the Android project that is available under an Open Source license. This way the layout manager can be used on iOS devices. Other UI idioms cannot easily be mapped such as Android's physical "Back" and "Menu" buttons. Mapping those features is left for future work.

3.4 Garbage Collector

XMLVM makes use of the Boehm Garbage Collector [1]. The Boehm GC is a conservative garbage collector using a mark-and-sweep algorithm. It allows finalization code to be invoked when an object is collected. Whenever a class is instantiated, the `GC_MALLOC()` function of the Boehm GC is used to allocate memory thereby registering the new object with the GC. Making use of `GC_MALLOC()` is therefore all that is necessary in terms of memory management for self-contained Java applications cross-compiled to C.

However, special attention needs to be given to wrapper objects such as UI widgets from Cocoa Touch. Although version 2.0 of the Objective-C programming language introduced a garbage collector, Apple decided to remove this feature for iOS devices. Apple's reason for doing so is most likely related to the undeterministic behavior of a garbage collector that might disrupt the user experience. Instead of garbage collection, an iOS developer has to resort to a reference counting mechanism for memory management.

Base class `NSObject` of the Cocoa Touch library offers methods `retain` and `release`, that increment and decrement respectively the reference count of an object. Objects are created with reference count of 1 and when the reference count drops to 0, the Objective-C runtime automatically deallocates the object. It is the iOS developers responsibility to retain and release objects according to the object ownership rules of Cocoa Touch.

Wrapper classes such as the aforementioned `UILabel` act as a bridge between the generated C code and native objects from Cocoa Touch. In the following we refer to the term "C object" as the cross-compiled version of a Java object. As depicted in Figure 2, each Objective-C object that needs to be accessible from Java (such as `UILabel`), needs to be wrapped by a C object. The C object will retain the Objective-C object to claim ownership and register a finalizer with the GC. When the C object is destroyed by the GC, the finalizer will release the wrapped Objective-C object as well. Special attention needs to be paid to associations between Objective-C objects. Whenever a wrapped Objective-C object holds a reference to another wrapped Objective-C object, the association needs to be mirrored among the wrapping C objects as well (see Figure 2). If this were not the case, the GC could not construct a correct reachability graph since it is unaware of Objective-C associations.

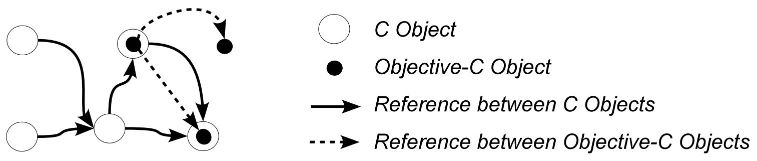
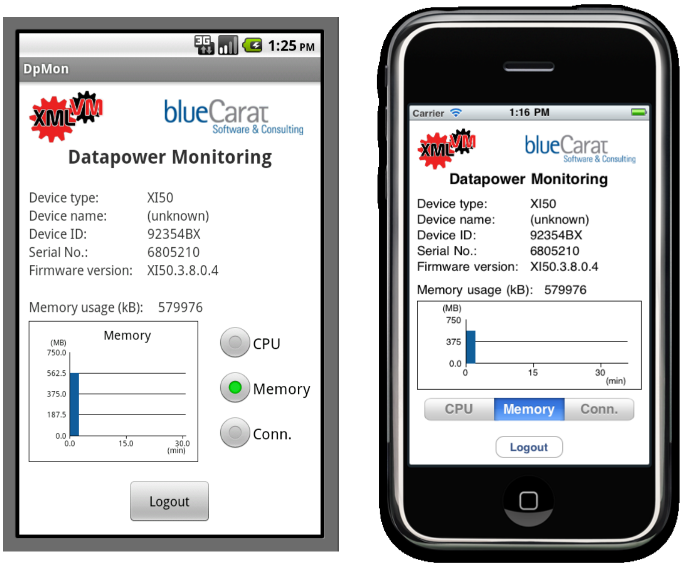

C Object
Objective-C Object
Reference between C Objects
Reference between Objective-C Objects

Fig. 2. Wrapper objects

4 Prototype Implementation

We have implemented a prototype based on the ideas described in this paper. We make use of BCEL [2] and JDOM [6] to parse Java class files and build up the XMLVM files. Saxon [7] is used as the XSL engine to apply the stylesheets that are responsible for the code generation. The implementation of the Java to C cross-compilation is fully Java compatible including garbage collection, threading, and reflection API. Our tool does not offer the same kind of completeness for the API mapping. Considering that both the Android and the Cocoa Touch API consist of thousands of methods, XMLVM currently maps approximately 30% of the API. However, the currently supported API already allows for complex applications.



Device type: XI50
Device name: (unknown)
Device ID: 92354BX
Serial No.: 6805210
Firmware version: XI50.3.8.0.4

Memory usage (kB): 579976

Device type: XI50
Device name: (unknown)
Device ID: 92354BX
Serial No.: 6805210
Firmware version: XI50.3.8.0.4

Memory usage (kB): 579976

Fig. 3. Layout mapping example

XMLVM has been used to cross-compile Android applications to the iOS by various companies. In the following we highlight one of these applications to demonstrate the power of cross-compilation. A German consulting company implemented a monitoring application for IBM datapower devices. A datapower device is a SOA appliance that can be monitored via SOAP requests over HTTPS. The monitoring application was written for Android and cross-compiled to iOS devices using XMLVM. The Android version makes extensive use of layout manager, Android-specific UI widgets as well as custom widgets, HTTPS requests and XPath queries to SOAP replies. All this functionality was successfully cross-compiled to iOS devices.

Figure 3 depicts the original Android version of the monitoring application on the left side and the cross-compiled iOS version to the right. Apart from the obvious mapping of labels, buttons and images, the Android version makes use of a radio button group. The corresponding widget under iOS is called a segmented control that serves the same purpose. Since a segmented control is wider than higher in contrast to Android's radio button group that is aligned vertically, the cross-compiled layout manager places the segmented control underneath the graph-drawing custom widget. As can be seen in Figure 3, the custom widget is automatically stretched in the iOS version to the full width of the screen.

5 Conclusion and Outlook

The popularity of smartphones makes them attractive platforms for mobile applications. However, while smartphones have nearly identical capabilities with respect to their hardware, they differ substantially in their programming environment. Different programming languages and different APIs lead to significant overhead when porting applications to various smartphones. We have chosen Android as the canonical platform. Our byte code level cross-compiler XMLVM can cross-compile an Android application to portable C code that can be run on iOS devices, therefore not requiring the Dalvik virtual machine on the target platform. We have demonstrated that a cross-compilation framework is feasible, thereby significantly reducing the porting effort.

In the future our goal is to support debugging of cross-compiled applications. The idea is that a Java application that was cross-compiled with XMLVM can be debugged with any standard Java debugger such as the one integrated in Eclipse. In order to accomplish this, an implementation of the JWDP (Java Wire Debug Protocol) needs to be available on the target platform. We plan to use the Open Source Maxine project that features a Java implementation of JWDP. With the help of the Java-to-C cross-compiler we will cross-compile Maxine to C to support debugging on any Posix compliant platform. The challenge of this task will be to interface with the generated C code to determine the memory layout (such as stack and heap) at runtime.

XMLVM is available under an Open Source license at `http://xmlvm.org`.

References

1. Boehm, H.: Bounding space usage of conservative garbage collectors. SIGPLAN Notices 37(1), 93–100 (2002)
2. Dahm, M.: Byte code engineering. Java Informations Tage, 267–277 (1999)
3. Davis, B., Beatty, A., Casey, K., Gregg, D., Waldron, J.: The case for virtual register machines. In: IVME 2003: Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators, pp. 41–49. ACM, New York (2003)
4. Dixon, R., McKee, T., Vaughan, M., Schweizer, P.: A fast method dispatcher for compiled languages with multiple inheritance. SIGPLAN Notices 24, 211–214 (1989)
5. Google, Inc. The Dalvik virtual machine, `http://en.wikipedia.org/wiki/Dalvik_virtual_machine`
6. JDOM. Java DOM-API (2004), `http://www.jdom.org/`
7. Kay, M.: Saxon: The XSLT and XQuery Processor, `http://saxon.sourceforge.net/`
8. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley Pub. Co. (April 1999)
9. Puder, A.: A Cross-Language Framework for Developing AJAX Applications. In: PPPJ. International Proceedings Series. ACM, Lisboa (2007)
10. Puder, A.: Cross-Compiling Android Applications to the iPhone. In: PPPJ. International Proceedings Series. ACM, Vienna (2010)
11. Shi, Y., Casey, K., Ertl, M.A., Gregg, D.: Virtual machine show- down: Stack versus registers. ACM Trans. Archit. Code Optim. 4(4), 1–36 (2008)
12. W3C. XSL Transformations (1999), `http://www.w3.org/TR/xslt`