Cross-Compiling Android Applications to Windows Phone 7

Oren Antebi, Markus Neubrand, and Arno Puder

San Francisco State University,
Department of Computer Science,
1600 Holloway Avenue,
San Francisco, CA 94132
{antebi,mneubran,arno}@mail.sfsu.edu

Abstract. Android is currently leading the smartphone segment in terms of market share since its introduction in 2007. Android applications are written in Java using an API designed for mobile apps. Other smartphone platforms, such as Apple's iOS or Microsoft's Windows Phone 7, differ greatly in their native application programming model. 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 C# for Windows Phone 7. We describe different aspects of our cross-compiler, from byte code level cross-compilation to API mapping. A prototype of our cross-compiler called XMLVM is available under an Open Source license.

Keywords: Android, WP7, Cross-Compilation.

1 Introduction

Android is a software stack for mobile devices maintained by the members of the Open Handset Alliance (OHA) since 2007. It employs Java as a programming language as well as its own API for mobile applications. The Android API offers support for application lifecycle, device management, and UI programming. Apps are offered via the Android Market but can also be downloaded from third-party web sites. In early 2010, Microsoft released its Windows Phone 7 (WP7) platform. The programming environment is based on the .NET framework [4] and applications can be developed in C# and VisualBasic. The WP7 platform offers its own proprietary API for mobile development. Apps are made available through Microsoft's Marketplace.

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. In this paper, we introduce a cross-compilation approach, whereby an Android application can be cross-compiled to WP7. The solution we propose not only

J.Y. Zhang et al. (Eds.): MobiCASE 2011, LNICST 95, pp. 283–302, 2012.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2012

cross-compiles on a language level, but also maps APIs between different platforms. The benefit is that only skill set for the Android platform is required and only one code base needs to be maintained for both devices. In an earlier project we have followed the same approach targeting iOS devices [9].

This paper is organized as follows: Section 2 provides an overview of Android and WP7 followed by a discussion of related work in Section 3. Section 4 presents our cross-compilation framework that can cross-compile Android applications to WP7 devices while Section 5 outlines the API mapping from Android to WP7. In Section 6, we discuss our prototype implementation of this framework as well as an application that was cross-compiled using our toolchain. Finally, Section 7 provides a conclusion and an outlook to future work.

2 Overview of Android and Windows Phone 7

Table 1 provides a comparison of the Android-based Nexus S sold by Google and Samsung's Omnia 7 which uses Microsoft's WP7. The intent of this side-by-side comparison is to show that although both smartphones are relatively similar with respect to their hardware capabilities, they differ greatly in their native application development models.

Android advocates a mobile operating system running on the Linux kernel. It was initially developed by Google and later the Open Handset Alliance. Android is not exclusively targeting smartphones, but is also available for netbooks, tablets and settop boxes. The Windows Phone 7 is a proprietary platform by Microsoft. WP7 stipulates hardware requirements that must be met in order to install Microsoft's mobile OS. Different smartphone manufacturers have licensed WP7. Targeting both platforms requires significant skill sets. Whereas Android uses Java as the development language, WP7 is based on .NET using either C# or VisualBasic. While Java and C# are relatively similar, their respective object models differ in subtle ways. C# features true generics and also supports events and delegates as part of its object model in contrast to Java.

Similar differences exist in the APIs and programming models defined by Android and WP7. To better highlight the different programming environments, the following two sections will show a simple application for both smartphones using their respective native programming language. The application allows to enter a name and upon pressing a button, the name will be echoed in a label. This application is more involved compared to a simple "Hello World", but the purpose is to provide a brief introduction to the programming abstractions employed by the respective platform and also to demonstrate the heterogeneity of smartphone application development.

2.1 Android

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 the selection of a contact from the internal address book. The user

Table 1. Smartphone comparison

	Nexus S	**Omnia 7**
OS	Linux	Windows CE
CPU	Hummingbird S5PC110, 1 GHz	Snapdragon QSD8250, 1 GHz
RAM	512 MB	512 MB
Sensors	Accelerometer, GPS, proximity, ambient light, compass.	Accelerometer, GPS, proximity, ambient light, compass.
IDE	Eclipse	VisualStudio
Dev-Language	Java	C#, VisualBasic
GUI	Android	WP7
VMs	Allowed	Only .NET
License	Open Source	Proprietary

may flip through the contact list or may use a search box. These actions are combined to form an activity. Activities have a well-defined life cycle and can be invoked from other activities (even activities from other applications). The sample application that we introduce here consists of a single activity which uses three widgets: a text input box, a label, and a button. Upon clicking the button, the name from the text input box is read and echoed with a greeting in the label. The following code is the complete implementation of the Android application:

Java (using Android API)

```
1  public class SayHello extends Activity {
2    @Override
3    public void onCreate(Bundle savedState) {
4      super.onCreate(savedState);
5      LinearLayout panel = new LinearLayout(this);
6      layout.setOrientation(LinearLayout.VERTICAL);
7      LayoutParams params =
8            new LinearLayout.LayoutParams(LayoutParams.FILL_PARENT,
9                                          LayoutParams.WRAP_CONTENT);
10     params.setMargins(10, 10, 10, 10);
11     final EditText box = new EditText(this);
12     box.setLayoutParams(params);
13     panel.addView(box);
14     final TextView label = new TextView(this);
15     label.setLayoutParams(params);
16     label.setGravity(Gravity.CENTER_HORIZONTAL);
17     panel.addView(label);
18     Button button = new Button(this);
19     button.setLayoutParams(params);
20     button.setText("Say Hello");
21     button.setOnClickListener(
22       new OnClickListener() {
23
24         @Override
```

```
25         public void onClick(View view) {
26             label.setText("Hello, " + box.getText() + "!");
27         }
28
29       });
30     panel.addView(button);
31     setContentView(panel);
32   }
33 }
```

Every Android application needs to designate a main activity whose implementation has to be derived from the base class `Activity`. The main entry point of an activity is the method `onCreate()` that also signals the creation of the activity. In case the activity was active at an earlier point in time, the saved state of the previous incarnation is passed as an argument to `onCreate()`. Inside the `onCreate()` method, a `LinearLayout` is instantiated that acts as a container and allows to automatically align its children vertically (line 5). Next, some general layout parameters are created (lines 7–10) that define padding of 10 pixels, tell that a widget's horizontal size should use up all available space of the parent (`FILL_PARENT`) and its vertical size should be the widgets natural size (`WRAP_CONTENT`). Those layout parameters are applied to all widgets created subsequently.

Three widgets are created in total: a text input box (line 11), a label (line 14), and a button (line 18). A click listener is added to the button (line 21) in the form of an instance of an anonymous class. Whenever the button is clicked, the click listener will read the name from the text input box and echo a greeting via the label. The remaining code of the "Say Hello" application defines the view hierarchy by adding the widgets to the `LinearLayout` and then setting the `LinearLayout` as the main content view (line 31).

Besides a variety of widgets, Android also allows the declarative description of user interfaces. XML files describe the layout of a user interface which not only simplifies internationalization but also allows to render the user interface on different screen resolutions.

2.2 Windows Phone 7

This sections shows how the same "Say Hello" application can be implemented for WP7. The primary languages offered by Microsoft for WP7 development are C# and VisualBasic, hence the "Say Hello" application for this device has to be written in one of those languages.

C# (using WP7 API)

```
1 public class SayHello : Application {
2   private TextBox box = new TextBox();
3   private TextBlock label = new TextBlock();
4   private Button button = new Button();
5
```

```
6   public SayHello() {
7     this.Startup += new StartupEventHandler(Main);
8   }
9
10  public void Main(object sender, StartupEventArgs args) {
11    label.Foreground = new SolidColorBrush(Colors.White);
12    label.Margin = new Thickness(10);
13    button.Content = "Say Hello";
14    button.Click += new RoutedEventHandler(Click);
15    button.Margin = new Thickness(10);
16    box.Margin = new Thickness(10);
17    StackPanel panel = new StackPanel();
18    panel.Children.Add(box);
19    panel.Children.Add(label);
20    panel.Children.Add(button);
21    this.RootVisual = layout;
22  }
23
24  public void Click(object sender, RoutedEventArgs args) {
25    label.Text = "Hello, " + box.Text + "!";
26  }
27 }
```

A WP7 application class needs to be derived from a base class called `Application`. Its constructor adds a new event handler to the `Startup` property (line 7) that will result in method `Application.Main()` to be invoked (line 10). Analogous to the Android version, the three widgets of the "Say Hello" application are called `TextBox`, `TextBlock` and `Button` (lines 2–4). Method `Main()` first sets various properties of those widgets including a 10-pixel margin and a click listener for the button (line 14). Just as with the startup property, the click listener is added via the overloaded `+=` operator in C#. If the user clicks the button, method `Click()` of class `Application` will be invoked. The vertical alignment of the three widgets is realized in WP7 by a so-called `StackPanel` (line 17).

3 Related Work

Several frameworks promise to facilitate the development of cross-platform applications. In the following we briefly discuss the approach taken by PhoneGAP, MonoTouch, and Adobe AIR. Each framework will be classified with regards to the mobile platforms it supports, the programming languages it offers, the API it uses, the IDE it can be used with and finally the license under which it is released.

PhoneGAP is an Open Source project that addresses web developers who wish to write mobile applications. It is available for iOS, Android, BlackBerry and the Palm Pre. Applications need to be written in JavaScript/HTML/CSS. But instead of downloading the application from a remote web server, the JavaScript

is bundled inside a native application. E.g., for iOS devices a generic startup code written in Objective-C will instantiate a full-screen web widget via class `UIWebView`. Next the JavaScript that is embedded as data in the native application is injected into this web widget at runtime. Special protocol handlers allow the communication between JavaScript and the native layer. All iOS widgets are rendered using HTML/CSS mimicking the look-and-feel of their native counterparts. PhoneGAP supports a common API for sensors such as the accelerometer. Platform-specific widgets have their own API. PhoneGAP is available under the MIT Open Source license at `http://phonegap.com`.

Xamarin (formerly Novell) offers with MonoTouch a .NET based framework for mobile applications. MonoTouch allows iOS applications to be written in C#. The C# iOS API is mapped one-for-one from the Cocoa Touch/Objective-C API. MonoTouch is able to read so-called XIB (Xcode InterfaceBuilder) files created by Xcodes InterfaceBuilder. Applications written in C# are compiled to ARM instructions utilizing the Open Source project Mono. Since Apple does not permit the installation of a virtual machine, C# applications are compiled using AOT (Ahead-of-Time) compilation instead of JIT (Just-in-Time) execution. Xamarin recently released a .NET based framework for Android called MonoDroid that allows Android applications to be written in C#. MonoTouch and MonoDroid are available under a commercial license at `http://ios.xamarin.com`.

Table 2. Comparison of Cross-Platform Frameworks

	PhoneGAP	**MonoTouch**	**Adobe AIR**	**XMLVM**
Platforms	iOS, Android, Blackberry, Palm Pre	iOS	iOS	iOS, Android, WP7
Language	JavaScript	C#	ActionScript	Java
API	Common Sensor API	iOS-only	Graphics-only	Android API mapped to iOS, WP7
IDE	Xcode	MonoDevelop	N/A	Eclipse
License	Open Source	Commercial	Commercial	Open Source

Another cross-platform framework is the Adobe Integrated Runtime (AIR) for iOS development. Similar to MonoTouch, Adobe AIR includes an AOT compiler based on the LLVM compiler suite that translates ActionScript 3 to ARM instructions. This facilitates porting of existing Flash applications while not relying on an installation of a Flash player on the iOS device. AIR offers API based on ActionScript to the device's sensors, but does not provide access to the native iOS widgets which limits AIR applications to games. As the only framework, AIR does not depend on Apple's Xcode SDK. AIR applications can be written under Windows. AIR is available under a commercial license at `http://www.adobe.com/products/air/`.

Table 2 summarizes the distinguishing factors of the various cross-platform frameworks introduced in this section. XMLVM is similar in the respect that it offers one programming language (Java) for different mobile devices. It also includes an AOT compiler to translate Java to native applications in order to avoid the installation of a Java virtual machine on the target platform. In contrast to other cross-platform frameworks, XMLVM relies on the Android API for application development. Instead of creating a new API for various functionalities, XMLVM makes use of the Android API that is mapped to the target platform. Besides sensor API, XMLVM is also capable of mapping Android widgets and layout managers such the ones used in the "Say Hello" application. Both the cross-compilation on a language-level as well as the API mapping is discussed in detail in the following section.

4 Cross-Compilation Framework

In this section we introduce our language-level cross-compiler, as a backend extension of a general flexible byte code level cross-compiler. The latter is named XMLVM due to its internal representation of byte code instructions of the virtual machine via appropriate XML tags. In Section 4.1 we give an overview of the general XMLVM toolchain. In Section 4.2 we explain how XMLVM is used to translate Java byte codes to the C# programming language.

4.1 Toolchain

Our general XMLVM framework [10] only assumes familiarity with the Android software stack. Thus, Android developers may use XMLVM toolchain to cross-compile an Android application to other smartphones, even without any iOS or WP7 skills.

The choice of Android as a canonical platform was made since we believe that there is a wider skill set for the Java programming language and there are powerful tools to develop in Java. For instance, we view this as an advantage over Objective-C used for iPhone and iPad development.

Moreover, Android was specifically designed to offer applications the means to adapt to a wider range of mobile devices. Hence, we believe that it serves as a natural umbrella platform over other smartphone platforms. Furthermore, our toolchain is able to benefit from many open source tools offered for the Android software stack. We use some of these tools for our API mapping and for the transformation of Oracle's stack-based virtual machine instructions [8] to register-based byte code instruction set introduced by the Dalvik virtual machine [1] to allow generation of more efficient code in the target language.

As illustrated in Figure 1, our toolchain cross-compiles byte codes, rather than source code of high-level programming languages as done by other tools (e.g., [5]). The choice to transform byte codes has several advantages. First, byte codes are much easier to parse than Java source code. Moreover, some high-level language

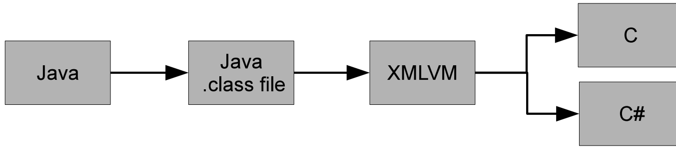

Fig. 1. XMLVM Framework

features such as generics are already reduced to low-level byte code instructions. Furthermore, the Java compiler does extensive optimizations to produce efficient byte codes.

In the following examples we focus on the register-based format of XMLVM which separates the frontend from the backends (`XMLVM` in Figure 1). As we illustrate in the next section, generation of high-level code in the target language, which for our backend is C#, is then a matter of a simple XSL transformation. Other backends have also been considered in earlier work, of which the most complete is the C backend used to build complex iOS applications.

4.2 Byte Code Level Cross-Compilation

To illustrate our approach, consider the following simple Java class `Account`, whose method `deposit()` adds a given amount to the `balance` of an account:

Account.java

```
1 public class Account {
2     int balance;
3     // ...
4     public void deposit(int amount) {
5         balance += amount;
6     }
7 }
```

The source code is first compiled to Java byte codes via a regular Java compiler, and fed into our XMLVM tool. The first transformation converts the stack-based byte code instructions to register-based instructions introduced by Dalvik. The conversion from a stack-based to a register-based machine has been researched extensively [2], [11]. Internally, XMLVM represents the virtual machine via the following XML document based on the `Account` class:

XMLVM

```
1 <vm:xmlvm ...>
2   <vm:class name="Account" ...>
3     <vm:field name="balance" type="int" />
4     <!-- ... -->
5     <vm:method name="deposit" ...>
6       <vm:signature>
7         <vm:parameter type="int" />
```

```
 8         <vm:return type="void" />
 9       </vm:signature>
10       <dex:code register-size="3">
11         <dex:var name="this" register="1"
12                  type="Account" />
13         <dex:var register="2" param-index="0"
14                  type="int" />
15         <dex:iget member-name="balance"
16                   vx="0" vx-type="int"
17                   vy="1" vy-type="Account" .../>
18         <dex:add-int vx="0" vy="0" vz="2" />
19         <dex:iput member-name="balance"
20                   vx="0" vx-type="int"
21                   vy="1" vy-type="Account" .../>
22         <dex:return-void />
23       </dex:code>
24     </vm:method>
25   </vm:class>
26 </vm:xmlvm>
```

On the top-level, there are tags to represent the class definition (line 2), field definitions (line 3), method definition (line 5) and its signature (line 6). The children of the tag <`dex:code`> (line 10) represent the byte code instructions for the method `deposit()`. The attribute `register-size` specifies the number of registers required to execute this method.

In the following we give a brief overview of the byte code instructions generated for the method `deposit()`. Upon entering a method, the last n registers are automatically initialized with the n actual parameters. Since the method `deposit()` has three registers labeled 0 to 2, register 2 is initialized (line 13) with the single actual parameter of that method (the amount). The implicit `this`-parameter counts as a parameter and will therefore be copied to register 1 (line 11). 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 the field `balance` of the account object referenced by register 1 into register 0 (line 15). The <`dex:add-int`> (*add integer*) instruction (line 18) adds 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 (line 19) 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 use XSL stylesheets [13] 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.

Registers can only store integers, floats, doubles, longs and object references. Shorter primitive types such as bytes and shorts are sign-extended to 32-bit

integers. In order to map individual registers to C# variables, we introduce a C#-struct that imitates a C-union and reflects the different data types that registers may contain:

```
C#
1  using global::System.Runtime.InteropServices;
2  namespace org.xmlvm {
3      [StructLayout(LayoutKind.Explicit)]
4      public struct Element {
5          [FieldOffset(0)]
6          public int i;
7          [FieldOffset(0)]
8          public float f;
9          [FieldOffset(0)]
10         public double d;
11         [FieldOffset(0)]
12         public long l;
13     }
14 }
```

Variables representing registers are automatically generated by XSL templates during the code generation process. They are always prefixed with _r followed by the register number, and their definition is based on `org.xmlvm.Element`. However, variables representing registers that store object references are defined separately as `System.Object`, to avoid overlapping by non-object fields which is not permissible in C#.

With the help of these variables, it is possible to map the effect of individual byte code instructions to the target language using XSL templates. As an example, the following XSL template shows how the aforementioned byte code instruction <`dex:add-int`> is mapped to C# source code:

```
XSL template
1  <xsl:template match="dex:add-int">
2      <xsl:text>    _r</xsl:text>
3      <xsl:value-of select="@vx"/>
4      <xsl:text>.i = _r</xsl:text>
5      <xsl:value-of select="@vy"/>
6      <xsl:text>.i + _r</xsl:text>
7      <xsl:value-of select="@vz"/>
8      <xsl:text>.i;</xsl:text>
9  </xsl:template>
```

Applying all XSL templates to the XMLVM of the class `Account` shown earlier yields the following C# source code for the method `deposit()`:

Generated C# for deposit()

```
1 //...
2 public virtual void deposit(int n1) {
3     org.xmlvm.Element _r0;
4     System.Object     _r1_o;
5     org.xmlvm.Element _r2;
6     //...
7     _r1_o = this;
8     _r2.i = n1;
9     _r0.i = ((Account) _r1_o).balance;
10    _r0.i = _r0.i + _r2.i;
11    ((Account) _r1_o).balance = _r0.i;
12    return;
13 }
```

In particular, note that the code in line 10 was generated by the XSL template for the <dex:add-int> instruction explained earlier.

In practice, field and method names have to be mangled to escape C# keywords (such as "out" and "internal"), to avoid clashes of method and field identifiers and to escape $s in Java identifiers. Moreover, the Java object model differs from C# [3]; a C# method will not override its parent unless it is annotated with the `override` modifier and its parent is annotated with `virtual`. Therefore, our tool identifies Java methods that override their base class methods, such as:

InvestmentAccount.java

```
1 public class InvestmentAccount extends Account {
2     public void deposit(int amount) {
3         //...
4     }
5 }
```

and annotates them internally by adding special attribute called `isOverride` to the <vm:method> tag. Subsequently, the stylesheet transforms this attribute into the required modifier:

Generated C#

```
1 public class InvestmentAccount: Account {
2     //...
3     public override void deposit(int n1) {
4         //...
5     }
6 }
```

Furthermore, C# does not allow covariant return types which is permissible in Java. For example, consider an `InvestmentBank` class that overrides an `openAccount()` method of a base `Bank` class:

Covariant return example in Java

```
1 public class Bank {
2     //...
3     public Account openAccount() {
4         return new Account();
5     }
6 }
7
8 public class InvestmentBank extends Bank {
9     @Override
10    public InvestmentAccount openAccount() {
11        return new InvestmentAccount();
12    }
13 }
```

Since Dalvik registers are always of the type `java.lang.Object`, return types of methods can always be replaced with `System.Object` in the cross-compiled C# code:

Generated C#

```
1 public class Bank: java.lang.Object {
2     //...
3     public virtual System.Object openAccount() {
4         global::System.Object _r0_o;
5         _r0_o = new Account();
6         //...
7         return _r0_o;
8     }
9 }
10
11 public class InvestmentBank: Bank {
12    //...
13    public override System.Object openAccount() {
14        global::System.Object _r0_o;
15        _r0_o = new InvestmentAccount();
16        //...
17        return _r0_o;
18    }
19 }
```

Besides the special cases discussed in this section, our code generation tool is fully Java compliant and is also able to handle Java exceptions and arrays (by wrapping them inside their C# counterparts), inner classes, interface fields, reflection API and native interface.

5 API Mapping

While the previous section focused on cross-compiling Java programs to C#, we discuss the mapping of the Android API to the API of the respective target

platform in the following. This section is organized as follows: Section 5.1 provides an overview of the different layers used for the API mapping and discusses the functions they provide and their APIs. Section 5.2 presents the implementation details of the wrapper library layer while Section 5.3 discusses the implementation details of the other two layers: the Android Compatibility Library and the Native Adapter Library.

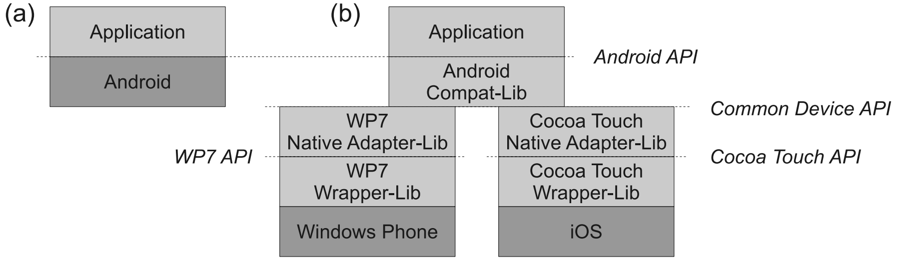

Fig. 2. (a) Classic Android Application (b) Android Application in the XMLVM layer model

5.1 Overview

Figure 2 shows a classic Android application versus the layer model of XMLVM. While a classic Android application makes direct use of the underlying native platform, the mapping of an Android application to a different platform is divided into several different layers.

The highest level of the XMLVM layer model is the *Android Compatibility Library (ACL)*. It offers the same API as the Android platform, but supports multiple underlying platforms through use of our *Common Device API*. The Android Compatibility Library contains the implementation of all parts which can be solved in a platform independent manner. Amongst others this includes APIs like layout manager, XML parsing or the Android application lifecycle. If a part needs access to native functionality it uses our Common Device API to access it.

To provide Android APIs the ACL uses parts of the Android project that is available under an Open Source license. However, Android also uses APIs which are not Android specific, but part of the standard Java SE APIs. The most common are data structures and other parts of `java.util.*`. Like Android itself we use the project Apache Harmony, an Open Source implementation of Java SE, to provide these APIs. Since Apache Harmony is itself written in Java, it is simply cross-compiled with XMLVM.

Native Adapter Libraries are responsible for adapting differences between Android's API and the API of the underlying platform and implement our specified Common Device API. The Common Device API is exposing all platform dependent native functionality needed by the ACL. Typical examples for exposed native functionality are UI widgets, like buttons or text fields, or sensor API. The

adapter library hides platform specific API from the ACL and clearly separates the wrapper libraries from any Android related dependencies.

Wrapper libraries are the lowest level of the XMLVM layer model. As the supported platforms use different programming languages than the Android platform, the wrapper libraries are exposing native API in Java. C# WP7 API, or Objective-C Cocoa Touch API, is represented as a Java API, allowing Java programs to interact with native functionality. Both the ACL and Wrapper Libraries are explained in detail in the following sections.

5.2 Wrapper library

XMLVM uses the C# version of the WP7 API that gets exposed in Java as a wrapper library. C# offers a variety of features like properties, operator overloading or delegates and events, that are not directly supported by the Java object model. In order to represent these constructs, they are emulated by using POJOs with the goal in mind to represent the original C# API as closely as possible.

The following listing shows the Java version of a button from the WP7 wrapper library:

Java: WP7 Button Wrapper

```
1 package Compatlib.System.Windows.Controls;
2
3 @XMLVMSkeletonOnly
4 public class Button {
5   public void setContent(String content) {}
6
7   public final ClickEvent Click;
8
9   private class ClickEvent extends Event {
10    public void __add(EventHandler handler) {}
11    public void __fire(Object sender, RoutedEventArgs args) {}
12  }
13
14  //...
15 }
```

As can be seen in the listing above, the implementation of the class is left empty since its only purpose is to provide a Java API against which the developer can implement an application. C# events and delegates are emulated by the Java classes `Event` and `EventHandler`. Operator overloading is mimicked by specially named methods representing the overloaded operator, like `__add` for += in the example above. Properties, like `Button.Content`, are represented by appropriate getter/setter methods. The package name of a wrapper class is the C# namespace of the class prefixed with `Compatlib` to avoid conflicts after cross-compilation.

Wrapper classes are marked with an `@XMLVMSkeletonOnly` annotation and 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. This code is tying the wrapper class together with the native class it wraps. The following code excerpt demonstrates this concept for the `Button` class.

C#: Cross-compiled WP7 Button Wrapper

```
1 using native = System.Windows.Controls;
2 namespace Compatlib.System.Windows.Controls {
3   public class Button {
4     public ClickEvent Click;
5
6     public virtual void setContent(java.lang.String n1) {
7       //XMLVM_BEGIN_WRAPPER
8       wrapped.Content = Util.toNative(n1);
9       //XMLVM_END_WRAPPER
10    }
11
12    //XMLVM_BEGIN_WRAPPER
13    public native.Button wrapped = new native.Button();
14
15    public Button {
16      wrapped.Click += ClickHandler;
17    }
18
19    public void ClickHandler(object sender, RoutedEventArgs args) {
20      Click.__fire(Util.toWrapper(sender), Util.toWrapper(args));
21    }
22    //XMLVM_END_WRAPPER
23
24    //...
25  }
26 }
```

Note that the wrapper class above is not implementing the widget itself, but only wraps the WP7 API `Button` class (line 13). Code between `XMLVM_BEGIN_WRAPPER` and `XMLVM_END_WRAPPER` comments is manually written C# code which gets injected on either method- or class-level during cross-compilation. The comment markers allow the manually written code to be automatically migrated if it should become necessary to regenerate the wrappers.

Communication between the native API and the wrapper library can happen in both directions. Method `setContent()` is an example for communication from the wrapper library to the native widget. The code above converts a `java.lang.String` instance to a native C# string via a helper function and sets the `Content` property of the wrapped button to the converted string (line 8).

Events represent communication in the opposite direction: from the native widget to the wrapper library. During construction of the wrapper we register

an event handler function for the `Click` event of the wrapped button (line 16). When a native click event is received our event handler function converts the accompanying parameters to wrapper library classes. Afterwards it fires off the `Event` class in the wrapper library which emulates C# events as POJOs (line 20).

To summarize, wrapper libraries are responsible for the communication between the native layer and Java code. We achieve that by injecting hand-written C# code, responsible for this communication, into them during cross-compilation.

5.3 Android Compatibility Library (ACL) and Native Adapter Libraries (NAL)

The purpose of the ACL is to offer the Android API to an application while supporting multiple different platforms for its implementation. The codebase from the original Android sources used for the implementation is modified to use the Common Device API exposed by the Native Adapter Libraries. The Native Adapter Libraries bind the Android API to the previously discussed wrapper libraries.

In addition to the Android codebase, Harmony is used to provide the Java SE APIs. The majority of Harmony's Java SE APIs are themselves written in Java. A useful subset of this API is simply cross-compiled as well to the language of the target platform and used as part of the compatibility library. XMLVM calculates dependencies to J2SE API at compile time and automatically includes needed classes in the cross-compilation process.

On system-level, Apache Harmony uses native Java methods to access functionality of the underlying operating system. These methods are implemented by using native functionality of the target platform (e.g. WP7 API on WP7 or Posix API on iOS). As with wrappers, the code for the implementation of native methods gets injected during cross-compilation.

As the codebase of the Android Compatibility Library is shared between all supported platforms we try to keep all functionality that does not necessarily need access to native capabilities in this layer. For some tasks, like XML parsing, native support would exist on the supported platforms, but to keep the Common Device API as small as possible we chose to implement those parts purely in Java as well. This minimizes the effort needed to add new platforms because these APIs are automatically available through cross-compilation.

Due to the fact that XMLVM supports multiple different platforms, functionality from the Android API must be mapped to several different native platforms. To achieve this the ACL is written platform independent and uses platform specific Native Adapter Libraries that implement an API called Common Device API. Due to the fact that all Native Adapter Libraries implement the same API the actual underlying platform is completely hidden from the ACL.

The following code excerpt shows how an `android.widget.Button` is mapped to the `Button` from the WP7 wrapper library explained in the previous chapter:

Java: excerpts from ACL and NAL

```
1 // Android Compatibility Library
2 package android.widget;
3
4 public class Button {
5   private ButtonAdapter adapter;
6   // ...
7   public Button(Context c) {
8     AdapterFactory f = FactoryFinder.getAdapterFactory();
9     adapter = f.createButtonAdapter();
10  }
11
12  public void setText(String string) {
13    adapter.setText(text);
14  }
15 }
16
17 // WP7 Native Adapter Library
18 public class WP7AdapterFactory implements AdapterFactory {
19   // ...
20   public ButtonAdapter createButtonAdapter() {
21     return new WP7ButtonAdapter();
22   }
23 }
24
25 public class WP7ButtonAdapter implements ButtonAdapter {
26   Compatlib.System.Windows.Controls.Button wrapper;
27   // ...
28   public void setText(String string) {
29     wrapper.setContent(string);
30   }
31 }
32
33 // Cocoa Touch Native Adapter Library
34 public class IOSButtonAdapter implements ButtonAdapter {
35   UIButton wrapper;
36   // ...
37   public void setText(String string) {
38     wrapper.setTitle(text, UIControlState.Normal);
39   }
40 }
```

To communicate with an underlying platform the ACL uses the `FactoryFinder` to get an instance of an `AdapterFactory` (line 8). The `FactoryFinder` will instantiate a platform specific `AdapterFactory`, in case of WP7 a `WP7AdapterFactory`. By using the `AdapterFactory` to instantiate adapter classes the ACL does not know which underlying platform it uses. The `AdapterFactory` is implemented for every supported platform and instantiates adapter classes. Adapter classes are responsible for translating between

Android API and native API exposed by the wrapper library. In the very simple case above the `WP7ButtonAdapter` translates from the Android API method `setText` to the equivalent WP7 API method `setContent()` (line 29). The class `IOSButtonAdapter` shows the same adapter for iOS. In this case the adapter translates from the Android `setText()` method to the `setTitle()` method of the iOS `UIButton` wrapper (line 38).

Adapters are responsible for a variety of simple tasks like converting units or method names. In more complicated cases, where no equivalent of an Android class is available on the native platform, they emulate the class by using several different wrapper classes.

These abstractions give us the advantage that new platforms can be easily added by implementing the specified Common Device API. Another advantage is that all supported platforms can share code in the ACL if it is not platform-dependent. By having a clean adapter layer both other layers, wrapper libraries and the ACL, can be developed without any dependencies to each other.

6 Prototype Implementation

We have implemented a prototype based on the ideas described in this paper. We make use of Dalvik eXchange (DX) [1] 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 for the different backends. The implementation of the Java to C# cross-compilation is fully Java compatible including exception handling, threading, and reflection API. However, our tool does not offer the same kind of completeness for the API mapping, since Android and WP7 API are complex libraries, consisting of thousands of methods. Some UI idioms, such as Android's hardware buttons, cannot be mapped by a tool, but require platform-specific implementations to make the application look and feel native for the respective platform. Nevertheless, the currently supported API already allows for cross-compilation of complex applications, as shown in the following example.

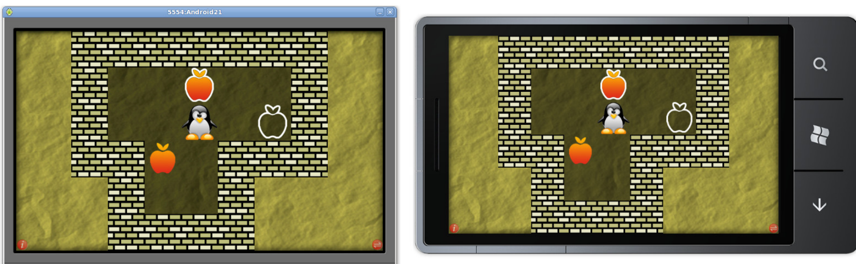

Fig. 3. Xokoban in the Android and WP7 emulator

To demonstrate the capabilities of our tool, we used our prototype to cross-compile Xokoban, an Android application previously used to showcase cross-compilation to iOS, to WP7. Xokoban is a remake of the classic Sokoban puzzle game in which the player has to push objects in a maze. Xokoban makes use of a range of Android APIs and widgets:

- 2D animation.
- Alert views, buttons, checkboxes.
- Accelerometer and swipe interface.
- Saving/loading of preferences.

By using XMLVM, the original Android application was successfully cross-compiled to WP7 as well as iOS devices without any changes. When cross-compiled to WP7, XMLVM generates a turnkey VisualStudio Windows Phone 7 project. The project generated by XMLVM contains all cross-compiled source files and other resources, like images, but no binaries. To compile the application and bundle it as a XAP, using VisualStudio is necessary.

Figure 3 shows the original Android version of Xokoban running in the Android emulator together with the cross-compiled WP7 version running in the WP7 emulator. The original version of Xokoban is available in the Android Market and a cross-compiled version for iOS can be found in the Apple App Store. The cross-compiled version for WP7 has been submitted for review to the Windows Marketplace.

7 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 environments. 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 C# code that can be run on WP7 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. However there are capabilities offered by the Android API, e.g. background services or being able to replace existing system applications like the home screen or the dialer, which are not available in any similar form on WP7 or iOS as of now. This functionality cannot be cross-compiled by our prototype implementation.

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 on the device 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 [12] 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 a WP7 device. 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`.

Acknowledgment. The work described in this paper was supported by a grant from Microsoft Research.

References

1. The Android Open Source Project. Dalvik eXchange (DX), `git://android.git.kernel.org/platform/dalvik.git`
2. 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)
3. ECMA. C# Language Specification, 4th edn. (June 2006)
4. ECMA. Common Language Infrastructure (CLI), 4th edn. (June 2006)
5. El-Ramly, M., Eltayeb, R., Alla, H.A.: An Experiment in Automatic Conversion of Legacy Java Programs to C#. In: ACS/IEEE International Conference on Computer Systems and Applications, pp. 1037–1045 (2006)
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.: Cross-Compiling Android Applications to the iPhone. In: PPPJ, Vienna, Austria. International Proceedings Series. ACM (2010)
10. Puder, A., Lee, J.: Towards an XML-based Byte Code Level Transformation Framework. In: 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation. Elsevier, York, UK (2009)
11. Shi, Y., Casey, K., Anton Ertl, M., Gregg, D.: Virtual machine showdown: Stack versus registers. ACM Trans. Archit. Code Optim. 4(4), 1–36 (2008)
12. Ungar, D., Spitz, A., Ausch, A.: Constructing a metacircular Virtual machine in an exploratory programming environment. In: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, pp. 11–20. ACM, New York (2005)
13. W3C. XSL Transformation (1999), `http://www.w3.org/TR/xslt`