Bytecode level cross-compilation for developing web applications

Arno Puder\textsuperscript{a,∗}, Sascha Häberling\textsuperscript{b}

\textsuperscript{a} San Francisco State University, Computer Science Department, 1600 Holloway Avenue, San Francisco, CA 94132, United States
\textsuperscript{b} Google Switzerland GmbH, Brandshenkestrasse 110, 8002 Zurich, Switzerland

A R T I C L E   I N F O

Article history:
Available online 13 February 2009

Keywords:
Web applications
Byte code
Cross-compiler

A B S T R A C T

JavaScript provides the technological foundation of Web 2.0 applications. AJAX (Asynchronous JavaScript And XML) applications have received wide-spread attention as a new way to develop highly interactive web applications. Breaking with the complete-page-reload paradigm of traditional web applications, AJAX applications rival desktop applications in their look-and-feel. But AJAX places a high burden on a web developer requiring extensive JavaScript knowledge as well as other advanced client-side technologies. In this paper, we introduce a technique that allows a developer to implement an application in Java or any .NET language and then automatically cross-compile it to an AJAX-enabled web application.

Published by Elsevier B.V.

1. Introduction

The initial intent of the World-Wide Web (WWW) was to provide access to remote documents. The HTML standard that is used to describe the content for the WWW was quickly extended to include user interface elements such as buttons and input fields that allowed the construction of web applications where the web browser acts as a generic client. Although enhanced by user interface elements, the look-and-feel of web applications was still far from the user experience of a regular desktop application.

With the release of Google Maps, a new era of web applications began. Instead of viewing a web application as a sequence of mostly static HTML pages that are loaded in response to user interaction, the latest generation of web applications made extensive use of JavaScript inside a browser to create highly interactive applications that rival desktop applications with their look-and-feel. In fact, some new web applications using this paradigm implement applications such as word processors or spreadsheets that traditionally have only been available on the desktop. The browser as a generic client slowly becomes the next desktop. The acronym AJAX (Asynchronous JavaScript And XML) \cite{12} is used to describe these new generation web applications.

The core of every AJAX application is JavaScript used for implementing a web application. Unfortunately JavaScript places a high burden on developers due to its nature as well as cross-browser portability issues. JavaScript is a prototype-based, dynamically typed programming language that lacks powerful development environments. Therefore, writing AJAX applications requires extensive knowledge of JavaScript, DOM manipulations, and awareness of portability issues across different browsers. The lack of IDEs and appropriate skill-set among current IT developers makes the development of AJAX applications a daunting task.

The basic assumption of this paper is that JavaScript is the assembler language of the web. Ideally one does not want to be exposed to it. In this paper, we introduce XML11 that features a cross-compiler capable of translating regular Java and .NET applications to portable JavaScript applications. The outline of this paper is as follows: Section 2 introduces a taxonomy

\textsuperscript{∗} Corresponding author.

E-mail addresses: arno@sfsu.edu (A. Puder), haeberling@google.com (S. Häberling).

0167-6423/$ – see front matter. Published by Elsevier B.V.
doi:10.1016/j.scico.2009.01.009
for web applications. Section 3 provides an overview of the XML11 framework describing its various components. Section 4 gives a detailed description of the byte code level cross-compiler part of XML11. Section 5 gives an overview of the prototype implementation of XML11. Section 6 discusses related work based on our taxonomy and in Section 7 we provide a conclusion and outlook.

2. Taxonomy for web applications

Shortly after the invention of the World-Wide Web the desire arose to use a web browser not just for viewing static documents, but to also use it as a generic client to access remote applications and services. New HTML markup was introduced for buttons, input fields, listboxes, and other widgets that allow the description of user interfaces rendered by a browser. In parallel, browsers began to allow extensions through plugins that can handle special content. Two examples are Java applets and Flash that also make it possible to build user interfaces inside the browser.

Different programming models have emerged that support developers to implement web applications. The purpose of this section is to introduce a taxonomy for web applications to allow a systematic classification and comparison of various programming models. In the following we discuss different categories that help us to distinguish between various paradigms. The first distinction can be made with respect to where the business logic of the application is executed. Some parts of the application typically always reside on the server-side. But it is possible to migrate some portions of the application to the browser as well.

Migrating parts of the application to the client leads to the next category of our taxonomy, namely the type of execution platform used on the client-side. Migrating code to the browser requires a general-purpose execution platform. Using the aforementioned browser plugins different programming languages such as Java or ActionScript can be supported. Virtually all major browsers offer a JavaScript interpreter. JavaScript has the benefit of being ubiquitous and is the lowest common denominator in terms of prerequisites. JavaScript does not require a special browser plugin and because of this it serves as the foundation of AJAX programs. The downside of JavaScript is cross-browser portability issues. It requires significant effort to write portable JavaScript applications.

To facilitate the development of JavaScript applications, some web application development environments offer a cross-compiler. The idea is that a compiler creates an abstraction by hiding the complexities of portable JavaScript just like a regular compiler hides the complexities of low-level assembler. Instead of having to write a web application in JavaScript, a developer can use a high-level programming language. For the scope of our taxonomy we distinguish between different permutations of languages supported by the cross-compiler. The front-end of the cross-compiler may only accept one source language or it may support several different source languages. Likewise the back-end can generate code for one or more target languages. While JavaScript is certainly the target language of choice especially for AJAX applications, it is also possible to emit ActionScript for Flash applications.

Cross-compilation itself only provides the translation from one language to another. For a complete development environment it is also necessary to offer a library that provides the building blocks for a user interface such as buttons and listboxes amongst others. Here our taxonomy distinguishes between the way the API of the library is designed. By utilizing a standard API the developer can make use of familiar building blocks from existing libraries. Likewise it is possible to create a completely new custom API for this library that can be tuned for certain special needs of web applications. The downside of this approach is that a developer first has to learn a new API.

The way the user interface is updated inside a browser provides another category for our taxonomy. Older web applications typically reload a complete page whenever the user interface is updated. With the advent of AJAX it has become fashionable to support partial updates of the user interface. Applications using this technique have more the look-and-feel of desktop applications. Partial updates require modification of the Document Object Model (DOM) inside the browser. DOM manipulations likewise suffer from portability issues between different browsers. The majority of AJAX frameworks encapsulate the complexity of such manipulations inside their library.

The final category of our taxonomy deals with the way the client- and server-portion of a web application communicates with one other. The transport mechanism depends on the execution platform used inside a browser. While a Java applet can use plain TCP sockets, a JavaScript application only has access to the so-called XMLHttpRequest object to issue HTTP requests. For both of these cases we refer to the communication using a raw transport mechanism. Some web application toolkits offer RPC functionality by automating the generation of stubs and skeletons by means of some interface definition language. Other toolkits go so far to make the generation of proxies completely transparent and do not require the developer to provide an explicit description of the interface through some interface definition language. We refer to this approach as implicit middleware. Table 1 summarizes the different categories and criteria of our taxonomy for web applications. In Section 6 we will apply the taxonomy to discuss related work.

3. XML11 framework

XML11 is our framework for developing AJAX applications. Its name is a tribute to the old X11-Windows protocol developed by MIT in 1984 [27]. Just as an X11 Server can render any user interface, a web browser likewise serves as a generic client that can render arbitrary user interfaces. Whereas the X11-Protocol focuses on graphics (i.e., the X11-Protocol has no notion of buttons or listboxes) the XML11-Protocol supports widgets. The client-side components of XML11 are
Table 1

<table>
<thead>
<tr>
<th>Category</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business logic</td>
<td>• Client/Server</td>
</tr>
<tr>
<td></td>
<td>• Server</td>
</tr>
<tr>
<td>Client-side execution</td>
<td>• Applet</td>
</tr>
<tr>
<td>platform</td>
<td>• Flash</td>
</tr>
<tr>
<td></td>
<td>• JavaScript</td>
</tr>
<tr>
<td>Cross-compilation</td>
<td>• None</td>
</tr>
<tr>
<td></td>
<td>• 1 to 1</td>
</tr>
<tr>
<td></td>
<td>• 1 to n</td>
</tr>
<tr>
<td></td>
<td>• n to 1</td>
</tr>
<tr>
<td></td>
<td>• n to m</td>
</tr>
<tr>
<td>Library API</td>
<td>• Standard API</td>
</tr>
<tr>
<td></td>
<td>• Custom API</td>
</tr>
<tr>
<td>GUI updates</td>
<td>• Page reload</td>
</tr>
<tr>
<td></td>
<td>• Partial updates</td>
</tr>
<tr>
<td>Communication</td>
<td>• Raw transport mechanism</td>
</tr>
<tr>
<td></td>
<td>• RPC</td>
</tr>
<tr>
<td></td>
<td>• Implicit middleware</td>
</tr>
</tbody>
</table>

Fig. 1. Architecture of XML11.

implemented in JavaScript and are loaded after visiting a respective URL pointing to the XML11 Server. Unlike in the X11-Windows world, a web browser contains a ubiquitous Turing complete execution platform through a JavaScript interpreter. Note that we do not consider Java or Flash extensions because the idea of AJAX is to only use what is commonly available in all browsers. For this reason the client-side portions of XML11 are implemented in JavaScript and XML11 cross-compiles those parts of the application to JavaScript that are to be migrated to the browser. A general description of the XML11 framework can be found in [29]. In the remainder of this section we provide a brief overview of its main components before describing the cross-compilation process in detail.

3.1. Architecture overview

Fig. 1 depicts the overall architecture of XML11. It follows the Client/Server model where the client is the browser and the server can be accessed through standard HTTP. The client uses HTTP requests to interact with the remote server. Upon visiting an XML11-enabled application, the complete client-side JavaScript implementation of XML11 is first downloaded into the browser. Also those parts of the application that should be migrated to the client, are automatically cross-compiled to JavaScript.

As can be seen in Fig. 1, the client and server are largely symmetrical in their internal structure. The client is completely implemented in JavaScript, whereas the server-side of XML11 is implemented in Java. In the following we give a brief
Fig. 2. Architecture of XMLOB.

overview of the various layers making up the XML11 architecture beginning with the lowest layer. Some of those layers will be discussed in detail in subsequent sections.

The bottom layer shown in Fig. 1 implements the transport mechanism for XML11. Since XML11 runs inside a browser, the transport mechanism uses the XMLHttpRequest object on the client-side to issue HTTP requests to the server. The server accepts incoming HTTP requests from the client. The layer above the transport mechanism is XMLOB, an XML-based Object Broker. The purpose of XMLOB is to create an abstraction from the raw transport mechanism by offering a simple message-based, bi-directional communication model between objects. XMLOB is capable of sending messages asynchronously in both directions. Section 3.2 describes XMLOB and in particular how the server can send messages asynchronously to the client.

One component that only exists on the server-side is the cross-compiler. This component is called XMLVM because it is based on an XML-based virtual machine model. The task of XMLVM is to translate the client-side portions of an application written in Java or one of the languages supported by .NET to JavaScript. The cross-compiled application is then migrated to the client. The application shown in Fig. 1 on the client-side is the result of this translation process. Since we focus in this paper on XML11’s cross-compilation framework, XMLVM is covered extensively in Section 4.

The core part of XML11 is based on a micro-kernel architecture. The idea is to create an architecture where new functionality can be plugged into the system dynamically at runtime. The XML11-core is mainly responsible for the plugin management. Various plugins enrich XML11 with additional functionality. One plugin implements the core widgets that an application can use for its user interface. In the current prototype implementation of XML11 those core widgets have the same API as the Abstract Windowing Toolkit (AWT) for Java and WinForms for .NET. As a consequence, AWT or WinForms applications that were originally developed for the desktop can run under XML11.

Some plugins implement custom widgets (such as a Google Maps panel) that serve as the foundation of mashups in XML11. Other plugins are created dynamically. One example is the application-specific proxies that support the transparent distribution of the client- and server-side of the application. The idea of dynamically creating the proxies at runtime leads to our notion of an implicit middleware (XMLIM) that is further explained in Section 3.3.

3.2. Object broker (XMLOB)

XMLOB offers middleware services to the applications running on top of it. It is not meant to be a full-fledged middleware layer such as CORBA or Web Services, but rather intended as a light-weight solution sufficient for AJAX applications. Fig. 2 depicts the basic architecture of XMLOB. Objects register with an Object Broker that is responsible for message transmission and dispatching. Objects on the side of the client are implemented in JavaScript whereas the objects on the server-side are implemented in Java. Objects are addressed by a priori known unique identifiers to avoid the need of a naming service.

One distinguishing feature of XMLOB is that its messaging service is completely symmetrical: objects on either side can send any other object a message. This is obvious for JavaScript objects calling Java objects; the Object Broker inside the browser can make use of the XMLHttpRequest object to send a message to the server. The question is how an object on the server-side can send a message to an object on the client-side. The problem is that HTTP is a client/server protocol and only the client can initiate HTTP requests.

The solution to this problem is a technique called deferred-reply: the Object Broker on the client-side issues a HTTP request, but upon receiving this request on the server-side, the Object Broker simply defers the reply until there is a message to be sent back to the client. This guarantees immediate transmission of messages from the server to the client. The downside of this approach is that at any point in time there is an open HTTP request in order to guarantee that messages can immediately be sent to the client. This solution does not scale for large number of clients when each of the clients is using this deferred-reply technique.

Deferred replies allow asynchronous updates that can be pushed from the server to the client. Many applications do not need this feature and therefore it is not necessary for using deferred replies. In order to accommodate different application
needs, XMLOB supports three different communication models:

- **Asynchronous**: This model employs the aforementioned deferred-reply technique. At any point in time, XMLOB keeps one HTTP connection open so that messages originating on the server-side can immediately be transmitted to the client.

- **Synchronous**: Using this model, XMLOB will only issue HTTP requests when a message is to be sent to the server. Any messages pending on the server-side will be piggy-backed onto the HTTP response. While this model scales to a large number of clients, messages originating at the server will be queued until the client issues a HTTP request.

- **Polling**: The polling model is a hybrid between the aforementioned asynchronous and synchronous communication models. Using the polling policy, the XMLOB client will periodically poll for messages pending for transmission at the server. This is a good compromise between interactivity of applications and scalability of the communication infrastructure.

Every application using XMLOB can choose between one of those three communication models that best suits its needs. A highly interactive application with asynchronous updates will use the asynchronous communication model. An application where the user interface only updates in response to interaction from the end-user will use the synchronous communication model.

The code fragment below shows how to send a message from the server (written in Java) to the client (written in JavaScript). XMLOB offers an appropriate API to make construction and handling of messages as efficient as possible.

```java
// Java - Sending a message
Message msg = new Message("awtManager",
  "createWidget");
msg.put("id", "ELEM_1");
msg.put("type", "button");
msg.send();

// JavaScript - Receiving a message
function AWTManagerClass {
  this.createWidget = function(msg) {
    var id = msg.id;
    var type = msg.type;
    // ...
  }
}
XMLOB.registerObject("awtManager",
  new AWTManagerClass());
```

The Java code in lines 2–6 constructs and sends a message. Every message has a target object identifier (awtManager) and method that is supposed to be called on the target object (createWidget). Those parameters are specified in the constructor of class Message. Actual parameters can be added in the form of name/value pairs. In the example above, parameters id and type are added to the message before the message is sent to the client. Depending on the communication model chosen by the application, the message will eventually be transported to the client-side. The Object Broker on the client-side will unmarshall the message and invoke the appropriate method on the target object. Lines 9–18 in the code fragment above show the JavaScript code for instantiating and registering the new object with XMLOB. Note that the actual parameters can be referenced as properties of a JavaScript object. This provides a natural mapping and makes best use of JavaScript as an interpreted language.

The above example only shows very simple actual parameters. The data model supported by XMLOB is inspired by JSON [5]. There are two different types of parameters:

**Objects**: unordered name/value pairs.

**Arrays**: lists of values.

Both data types can be arbitrarily nested. While this data model does not provide the same flexibility as a full-fledged middleware layer such as CORBA, in our experience this is sufficient for AJAX applications. XMLOB supports two different marshalling engines as shown in Fig. 2. The JSON codec marshalls the actual parameters as a JavaScript data type and the XML codec marshalls the actual parameters as an XML document. The fragment below shows the resulting PDU that would be observed on the wire based on the simple example discussed earlier where a message is sent to an object whose identifier is awtManager:

```javascript
// JSONCodec
{xmlb:v:
  message:[
    {method: "createWidget",
      target: "awtManager",
      id : "ELEM_1",
    }
  ];
}
```
3.3. Implicit middleware (XMLIM)

The code migration framework introduced in the previous section cannot generally be used to migrate a complete application. The reason for that is that most applications are not self-contained but require access to fixed resources that cannot be migrated. One example is databases that need to reside on the server. An application can therefore be partitioned into classes that can be migrated to the client and classes that need to remain on the server-side. The decision which classes to migrate is currently determined by a configuration file that the application programmer has to provide.

Problems arise when a class that has been migrated to the client is referencing a class whose implementation resides on the server. What is needed in this case is middleware that marshallsthe actual parameters provided by the client and sends them to the server. This task is typically done by a proxy that is linked with the application and that appears to the application as if it were the remote object. The proxy can use XMLLOB to transport the marshalled actual parameters.

The problem still remains how this proxy is generated. The proxy has the same API as the remote object and is therefore application specific. That is, depending on the signatures of the methods offered by the remote object at its public interface, a specific proxy has to be generated. Traditionally, the generation of the proxy is dependent on some external artifact describing the public interface of an object. E.g., CORBA has the Interface Definition Language (IDL) for that purpose and Web Services use the Web Services Definition Language (WSDL) to achieve the same. A tool generates the proxies based on the IDL- or WSDL-specifications.

As will be seen in the section explaining XMLVM, detailed information on the signatures is already present in an XMLVM program. Using this information, it is possible to automatically generate the appropriate proxy. The reason we call this approach implicit is because the proxy can be generated automatically without requiring an external specification such as IDL or WSDL. From an application programmer’s perspective, middleware is implicitly inserted to connect remote objects.

Proxies have to be generated in the same language as the application that uses them, because they need to be linked to the application. Since XML11 requires JavaScript on the client-side and Java or one of the .NET languages on the server-side, proxies also need to be generated for these languages. The way this is done is by expressing the implementation of the proxy through XMLVM and then cross-compiling it to the desired target language. Based on the signature of a method, the implementation is replaced with one that marshals the actual parameters and that sends it to the remote object via XMLOB. By making use of XMLVM and its cross-compilation capabilities, the proxy is generic with respect to the implementation language.

4. Cross-compilation framework

A key component of the XML11 architecture is the cross-compilation framework that allows the migration of application logic to the browser. This is done by cross-compiling the client-side portions of the application to JavaScript and then migrating the generated code to the browser. We are able to cross-compile both Java class files as well as .NET executables to JavaScript. The translation process happens in several stages. Fig. 3 depicts the complete toolchain. The binary executable (either a Java class file or a .NET executable) is first translated to an XML representation we call XMLVM because it is modeled after a stack-based machine common to both the Java and .NET virtual machines. We refer to XMLVM_JVM as an XMLVM program using Java byte code instructions [22], whereas XMLVM_CLR uses .NET’s byte code instructions [9].

As shown in Fig. 3 XMLVM_JVM serves as a canonical format that separates front- and back-end. Therefore XMLVM_CLR has to be cross-compiled to XMLVM_JVM which requires a data flow analysis. This intermediate step is referred to as XMLVM_CLR- DFA in Fig. 3. Once an XMLVM_JVM program has been created, the next step is to remove goto-statements. This step is necessary because JavaScript does not feature appropriate jump statements. The intermediate artifact is referred to as XMLVM_FC in Fig. 3. Finally the resulting XMLVM_FC can be translated to JavaScript. The XMLVM toolchain is heavily based on XML tools such as XSL stylesheets and XPath expressions. The following subsections describe each stage of the toolchain in more detail.

4.1. XML representation of byte code instructions

Another way to look at XMLVM is that it defines an assembly language for the Java and .NET virtual machines. Although both platforms share many things in common, their byte code differs nonetheless. XMLVM uses XML-namespaces to clearly distinguish between byte code instructions stemming from either platform. The following template shows the general structure of an XMLVM translation unit:
Fig. 3. XMLVM cross-compilation toolchain.

An XMLVM program consists of several classes, each contained in a separate translation unit. Each class can have one or more fields and methods. The attributes of the XML-tags, that are not shown in the template above, give more details such as identifier names or modifiers. A method is defined through a signature and the actual implementation, denoted by the tags `<vm:signature>` and `<vm:code>` respectively. The byte code instructions used within the `<vm:code>` element depend on the platform from where the XMLVM program was generated. The following two subsections show a specific example for the Java Virtual Machine (JVM) and the Common Language Runtime (CLR).

### 4.1.1. JVM byte code instructions

Consider the following simple Java class whose only static method determines if an integer is odd [3]:

```java
// Java
class Check {
    static public boolean isOdd(int i) {
        return i % 2 != 0;
    }
}
```
Class Check has one static public method called isOdd. The method returns a boolean value indicating whether the actual integer parameter is odd or not. Although this is a very simple example, it allows us to show all basic aspects of an XMLVM program. The following simplified XML document shows the representation of class Check in XMLVM:

```
<vm:xmlvm xmlns:vm="http://xmlvm.org"
         xmlns:jvm="http://xml11.org/jvm">
  <vm:class name="Check">
    <vm:method name="isOdd" isStatic="true"
              isPublic="true">
      <vm:signature>
        <vm:return type="boolean" />
        <vm:parameter type="int" />
      </vm:signature>
      <vm:code>
        <jvm:iload type="int" index="0" />
        <jvm:iconst value="2" />
        <jvm:irem />
        <jvm:ifeq label="0" />
        <jvm:iconst value="1" />
        <jvm:goto label="1" />
        <jvm:label id="0" />
        <jvm:iconst value="0" />
        <jvm:label id="1" />
        <jvm:ireturn />
      </vm:code>
    </vm:method>
  </vm:class>
</vm:xmlvm>
```

It should be emphasized again that the above XMLVM program is essentially an XML representation of the contents of the Check.class class file generated by a Java compiler. The top-level tags are identical to the XML template shown earlier. The <vm:method>-tag has three attributes for the method name as well as the two modifiers static and public.

The more interesting part of the XMLVM program shown above is the actual implementation of the method isOdd. The bytecode instructions are prefixed with XML-namespace jvm: to indicate that they belong to the Java virtual machine. The <jvm:iload> (integer load) instruction pushes the actual parameter passed to the method onto the stack. Instruction <jvm:iconst> (integer constant) pushes an integer constant referred to by the attribute value onto the stack. The next instruction <jvm:irem> (integer remainder) pops off the last two values (the actual parameter and the constant 2) and pushes their remainder after division back onto the stack. The <jvm:ifeq> (if equal) instruction pops the last element off the stack and performs a conditional jump if its value is equal to 0. Note that flow control is represented in XMLVM through gotos. The <jvm:ireturn> (integer return) instruction pops off the top of the stack and returns the value to the caller of the method isOdd.

The XMLVM instruction set features a mix of low-level and high-level virtual machine instructions. In addition to the low-level instructions mentioned above, there exist high-level instructions such as <jvm:new> (for instantiating new objects) and <jvm:invokevirtual> (invoke a virtual method). These instructions go beyond the capabilities of normal (hardware) machine languages and therefore require substantial runtime support.

### 4.1.2. CLR byte code instructions

To show some details of CLR byte code instructions, consider the C# version of class Check introduced in the previous section:

```
// C#
public class Check
{
  static public bool isOdd(int i)
  {
    return i % 2 != 0;
  }
}
```

Apart from a different name of primitive type bool, the C# version is syntactically identical to the Java version. The XMLVM version of this class using CLR byte code instructions looks as follows:

```
<xmlvm xmlns:clr="http://xmlvm.org/clr"
       xmlns="http://xmlvm.org"
       class name="Check">
  <vm:method name="isOdd" isStatic="true">
    <vm:signature>
      <vm:return type="boolean" />
      <vm:parameter type="int" />
    </vm:signature>
    <vm:code>
      <jvm:new type="int" index="0" />
      <jvm:iconst value="2" />
      <jvm:irem />
      <jvm:ifeq label="0" />
      <jvm:iconst value="1" />
      <jvm:goto label="1" />
      <jvm:label id="0" />
      <jvm:iconst value="0" />
      <jvm:label id="1" />
      <jvm:ireturn />
    </vm:code>
  </vm:method>
</xmlvm>
```
The above XMLVM<sub>CLR</sub> program represents the contents of the Check.exe executable generated by a C# compiler. The top-level tags are identical to the XML template shown earlier. The <method>-tag has the same attributes as in the XMLVM<sub>JVM</sub> version. The main differences are the different byte code instructions. Since they belong to the CLR, the respective instructions are placed in the XML-namepace denoted by the prefix clr:. The <clr:ldarg> (load argument) pushes an actual parameter that was passed to the method onto the stack. In this particular example the integer parameter referenced with index 0 will be pushed onto the stack. The <clr:ldc> (load constant) pushes a constant of the specified type denoted by attribute type onto the stack. One special instruction only available in the CLR is <clr:ceq> (compare equal). It will pop the last two integer values off the stack and push 1 onto the stack if those two values are equal and 0 otherwise. Because of this instruction the CLR version of class Check does not require conditional jumps.

The <clr:rem> (remainder) instruction computes the remainder after division. This instruction gives no indication of the type of the operands. In this particular example, the <clr:rem> instruction will compute the remainder of two integers since the two top elements of the stack are of type integer. The same <clr:rem> instruction would have been used for computing the remainder of the two long values. The CLR states that the virtual machine has to determine the correct type through some mechanism. This could be accomplished by either a static data flow analysis of the program, or by maintaining a type stack at runtime.

4.2. CLR to JVM cross-compilation

Within the XMLVM toolchain, the JVM instruction set serves as the canonical format. That is, the code generation for JavaScript is solely based on the JVM instruction set. For that reason, an XMLVM<sub>CLR</sub> program first needs to be cross-compiled to an equivalent version only using JVM byte code instructions. While both the JVM and the CLR are stack-based virtual machines, they differ in several details. As outlined earlier, the CLR only features un-typed instructions. During the cross-compilation process it is necessary to know the type of the operands. Without this knowledge it would be impossible to map the CLR-remainder instruction to one of the typed remainder instructions supported by the JVM. As a prerequisite, one has to determine on which specific argument types the un-typed instructions operate. This can be accomplished by a data flow analysis. This task is similar to what a byte code verifier has to do when loading a program into the virtual machine [14].

During a data flow analysis, all execution paths through a program are traced, but instead of pushing and popping specific arguments onto the stack, only the types of those arguments are stored on the stack. For this reason it is also called a type stack (vs. an argument stack). By doing so, it is possible to determine the type of the arguments that will be stored on the stack at any point during the execution of the program.

As a first step towards a data flow analysis, we introduce new markup of an XMLVM<sub>CLR</sub> program that can capture the effects an individual byte code instruction has on the type stack. For each instruction we need to record the state of the type stack before and after execution of that instruction. Those two states will be marked up with the tags <stack-pre> and <stack-post> respectively and appended as children to the XML-tag of the respective byte code instruction. The elements contained on a stack are denoted by the <elem> tag. We refer to the resulting XMLVM program that contains these type-stack annotations as XMLVM<sub>CLR-DFA</sub>. The following excerpt shows the first three instructions of the original XMLVM<sub>CLR</sub> program for class Check with the computed type stack transitions as a result of the data flow analysis:

```
<!-- ... -->
<clr:ldarg index="0"/>
<stack-pre/>
<stack-post/>
<elem type="int"/>
```
In the XMLVM<sub>CLR-DFA</sub> excerpt above it can be seen that the data flow analysis added tags for each CLR instruction that reflects the content of the type stack before and after the execution of that instruction. As shown in lines 17 to 25, the CLR instruction &lt;clr:rem&gt; pops off two integers and pushes an integer back onto the type stack. It should be noted that the data flow analysis done here is much less complex than what a byte code verifier typically is required to check and can easily be implemented by an XSL stylesheet. In particular, since we are only interested in primitive types, it is not necessary to compute the LUB (least upper bound) of object types of different execution paths [21].

The next step of the XMLVM toolchain consists in translating the CLR instructions to JVM byte code. In some cases this mapping is trivial as there is a one-to-one correspondence between CLR and JVM instructions. One example is the unconditional branch instruction &lt;clr:br&gt; (branch) that can be mapped directly to &lt;jvm:goto&gt;, as can be seen by the corresponding XSL template:

```xml
<xsl:template match="clr:br">
  <jvm:goto/>
</xsl:template>
```

The &lt;xsl:copy-of&gt; expression in line 3 copies all XML-attributes from &lt;clr:br&gt; to &lt;jvm:goto&gt;, which in this case will copy the branch target denoted through the attribute label. In other cases the mapping has to rely on the data flow analysis. The following two XSL templates demonstrate the mapping of the CLR instruction &lt;clr:rem&gt;:

```xml
<xsl:template match="clr:rem">
  <jvm:irem/>
</xsl:template>
```

The XPath expressions in the match-statements of the XSL templates check the top of the type stack and map the &lt;clr:rem&gt; CLR instruction to either the JVM &lt;jvm:irem&gt; instruction (line 4) or the JVM &lt;jvm:lrem&gt; instruction (line 10) depending on whether the top of the type stack is of type int or long. This is an example of how un-typed CLR instructions can be mapped to type-specific JVM instructions via declarative XSL templates.

The aforementioned &lt;clr:ceq&gt; instruction can be mapped to a short sequence of JVM instructions via the following XSL stylesheet:

```xml
<xsl:template match="clr:ceq">
  <jvm:if_icmpne label="A"/>
  <jvm:iconst value="1"/>
  <jvm:goto label="B"/>
  <jvm:label id="A"/>
  <jvm:iconst value="0"/>
</xsl:template>
```
The `<clr:ceq>` instruction is replaced by the XSL stylesheet during the cross-compilation from XMLVM<sub>CLR</sub> to XMLVM<sub>JVM</sub> with the JVM instructions embedded in the XSL template (lines 2–7). The `<jvm:if_icmpne>` (if integer compare not equal) pops off two integers and branches if they are not equal. Depending on the outcome of the comparison, either 0 or 1 will be pushed onto the stack to achieve the same behavior as the `<clr:ceq>` instruction. Note that the above template introduces jump instructions to the resulting XMLVM<sub>JVM</sub> program. Special care needs to be taken with the labels for the jump instructions (labels “A” and “B” in the XSL template above). If the `<clr:ceq>` instruction is used multiple times in one method, this would lead to duplicate labels. The above XSL template is therefore only a simplified version to explain the principle, whereas the actual version not shown here has to generate unique labels.

Applying the CLR to JVM cross-compilation templates for the .NET executable Check.exe yields to the following XMLVM<sub>JVM</sub> program that only uses JVM instructions. For clarity purposes we have inserted the original CLR byte code instructions as XML-comments to show how each individual CLR instruction is mapped to the JVM:

```xml
1 <vm:code>
2  <!-- clr:ldarg index="0" -->
3  <jvm:iload index="0" />
4  <!-- clr:ldc type="int" value="2" -->
5  <jvm:iconst value="2" />
6  <!-- clr:rem -->
7  <jvm:irem />
8  <!-- clr:ldc type="int" value="0" -->
9  <jvm:iconst value="0" />
10  <!-- clr:ceq -->
11  <jvm:if_icmpne label="19" />
12  <jvm:iconst value="1" />
13  <jvm:goto label="20" />
14  <jvm:label id="19" />
15  <jvm:iconst value="0" />
16  <jvm:label id="20" />
17  <!-- clr:ldc type="int" value="0" -->
18  <jvm:iconst value="0" />
19  <!-- clr:ceq -->
20  <jvm:if_icmpne label="23" />
21  <jvm:iconst value="1" />
22  <jvm:goto label="24" />
23  <jvm:label id="23" />
24  <jvm:iconst value="0" />
25  <jvm:label id="24" />
26  <!-- clr:return -->
27  <jvm:ireturn />
28 </vm:code>
```

Note that it is possible to create a Java class file based on this XMLVM<sub>JVM</sub> that can be executed on a standard JVM. This part of the XMLVM toolchain is depicted in the lower right corner of Fig. 3. It is therefore possible to use XMLVM not only for generating web applications, but also for cross-compiling regular desktop applications. We will highlight this fact again when we discuss the prototype implementation of XMLVM.

### 4.3. Goto elimination

Whether the application was written in Java or in one of the .NET languages, the resulting intermediate artifact is an XMLVM representation based on JVM byte code instructions, referred to as XMLVM<sub>JVM</sub> in Fig. 3. Flow control on byte code level is based on goto-statements. However, JavaScript does not feature jump instructions for flow control. JavaScript only offers the usual while, do-while, if/break/continue statements. Hence there is no straightforward way to map the XMLVM<sub>JVM</sub> `<jvm:ifeq>` instruction discussed earlier to JavaScript via an XSL stylesheet. To solve this problem, the XMLVM toolchain provides a transformation tool that removes goto-statements from a program and replaces them with loop-, break-, and continue-instructions. The goto-elimination algorithm is based on a classic paper by Ramshaw [31]. Using Ramshaw’s algorithm, jump instructions are removed from an XMLVM<sub>JVM</sub> program, resulting in XMLVM<sub>FC</sub> as shown in Fig. 3. Note that the task of removing goto-statements is different from decompiling Java programs as described in [23] or [26]. In contrast to decompilation, goto elimination is not concerned with reverse engineering specific flow control statements such as for- or while-loops. Using our previous XMLVM<sub>JVM</sub> version of class Check, here is the result of this transformation:

```xml
1 <vm:code>
2  <jvm:iload index="0" />
3 </vm:code>
```
As can be seen, the goto-elimination algorithm introduces new XML-tags for flow control. Since these tags do not belong to the JVM instruction set, they are placed in their own XML-namespace. Note that in the example above, the `<jvm:ifeq>` from the original program has been replaced by two nested loops. Once gotos have been removed, it is relatively straightforward to map flow control statements to JavaScript instructions via stylesheets which will be described in the next section.

### 4.4. Code generation

Up to this point we have explained how to generate an XMLVM<sub>FC</sub> program from either a Java class file or a .NET executable. The next step consists in translating XMLVM<sub>FC</sub> to JavaScript. This translation can be done by an XSL stylesheet that maps the class structure as well as all JVM instructions one-to-one to the target language.

Since JavaScript is a prototype-based language that does not support object-oriented features known from Java or .NET, creating a class structure cannot be done with native language constructs alone. Fortunately, it is possible to implement some of these features on top of JavaScript. This has been done by some projects; one of which is the Qooxdoo library [30] that we make use of. Beside its wide range of user interface widgets, it contains an object-oriented programming simulator at its core. It provides functions to create classes by adding member variables and methods. The style of creating a class using Qooxdoo is similar to the JSON notation [5]. Embedded in braces, sections called `statics` and `members` contain the declarations of all static as well as member variables belonging to that class. The following JavaScript code shows the general template of a class defined using Qooxdoo:

```javascript
qx.Class.define("Derived", {
    extend: Base,
    statics: {
        staticMethod: function() { ... }
    },
    members: {
        field: 0,
        method: function() { ... }
    }
});
```

Qooxdoo does not support all required object-oriented programming features. One important feature missing is the possibility to overload methods. In languages like Java that enable overloading a method is well defined by its name and signature. The signature of a method is defined by the number of arguments, the arguments' types and their order. A common way to implement this in JavaScript is to use name mangling: The signature of a method is encoded as a string by concatenating the types of the arguments in the correct order and appending this to the original method name. Name mangling can be easily accomplished by an XSL stylesheet as the required information is contained in an XMLVM program. In order for this to work, the name mangling has to be applied to the declaration of a method as well as to every single invocation of that method.

The implementation of a method can be translated to JavaScript using XSL stylesheets as well. Since XMLVM is based on a stack-based machine, we simply mimic a stack machine in JavaScript with appropriate helper variables. An example helps to illustrate this approach. The JVM instruction `<jvm:irem>` introduced earlier pops off two values and pushes the remainder after division back on the stack. Here is the XSL template that creates JavaScript code for this instruction:

```xml
<xsl:template match="jvm:irem">
    <xsl:text>
        __op1 = __stack[--__sp];
        __op2 = __stack[--__sp];
        __stack[__sp++] = __op2 % __op1;
    </xsl:text>
</xsl:template>
```
We mimic the Java virtual machine via the variables __stack (for the stack), and __sp (for the stack pointer). Variables __op1 and __op2 are used as temporary variables needed by some JVM instructions. Those variables are declared for every method. Using stylesheets to translate JVM instructions to the target language works as long as there is a corresponding instruction.

In some instances this translation has to be done carefully in order to retain the semantics of the original instruction. For example, the JVM features an instruction for adding two integers, <jvm:iadd>. However, it is not correct to map addition to the + operator in JavaScript since integer values are treated differently in the JVM and JavaScript. The type int in Java is a 32-bit value. When adding two integers in Java, an overflow can occur in which case the most significant digits will simply be ignored. JavaScript on the other hand does not distinguish between int, float, or double. All integer values are represented as standard 8 byte IEEE floating-point values. This means that integers can only be accurately represented in the interval \[-2^{53} \ldots 2^{53}\]. As a consequence, if a Java program deliberately causes an overflow during the addition of two integers, simply adding those two values in JavaScript via the + operator would yield the wrong result. This difference has to be accounted for by using a wrapper function written in JavaScript that retains the original semantics of <jvm:iadd>.

The code below represents the JavaScript version of the class Check after applying all necessary stylesheets:

```javascript
// JavaScript generated by stylesheet
qx.Class.define("Check", {
  /* ... */
  statics: {
    _isOdd___int: function(__arg1) {
      var __locals = new Array(1);
      var __stack = new Array(2);
      var __sp = 0;
      var __op1;
      var __op2;
      __locals[0] = __arg1;
      __stack[___sp++] = __locals[0];
      __stack[___sp++] = 2;
      __op1 = __stack[--__sp];
      __op2 = __stack[--__sp];
      __stack[___sp++] = __op2 % __op1;
      label0: while (1) {
        label1: while (1) {
          __op1 = __stack[--__sp];
          if (__op1 == 0) break label1;
          __stack[___sp++] = 1;
          break label0;
        }
        __stack[___sp++] = 0;
        break label0;
      }
      return __stack[--__sp];
    }
  }
});
```

The JavaScript code above was generated automatically by applying an appropriate XSL stylesheet to the XMLVM\_FC version of class Check. As can be seen, there is a natural mapping from XMLVM\_FC to JavaScript. Lines 14–16 were generated by the XSL template for <jvm:irem>. The intention is not to generate readable code, but correct code that uses the semantics of the target language. It should also be obvious that the above JavaScript code will be less efficient than the original native application. Our assumption is that we do not migrate computationally-intensive applications to the browser. In Section 5.3 we provide an analysis of an interactive application that has been translated using XMLVM. It is also shown, why our approach does not affect the performance of such applications significantly.

5. Prototype implementation

In this section we give a short overview of our prototype implementation. Section 5.1 provides some implementation details of the XMLVM-based cross-compilation toolchain. Section 5.2 introduces two sample applications – one written in C# and one in Java – that were cross-compiled to web applications using XMLVM. Section 5.3 discusses some of our findings with regard to the runtime overhead of the resulting web applications.

5.1. Overview

Our prototype implementation consists of three parts: The cascaded cross-compilers as described in the previous section, a set of emulation libraries to mimic external libraries inside the browser and a packager that produces one self-contained JavaScript file that is ready to be deployed.
First there is the actual cross-compiler chain that takes a Java class file or a .NET executable and translates it to JavaScript as described in Section 4. The implementation of XMLVM is leveraging two Open Source libraries that allow to parse Java class files as well as .NET executables. For Java class files we use the BCEL (Byte Code Engineering Library) [6] and for .NET executables we use MBEL (Microsoft Byte code Engineering Library) [34]. The byte code translation process as well as the data flow analysis are done using XSL style sheets. We use Saxon [19] to process our style sheets to make use of new XSLT 2.0 features.

The second part of our implementation is the emulation library. In order to run migrated applications on another platform it is necessary to have such libraries, so method calls from an application to these external libraries lead to the expected behavior while running in the target environment. For example, the statement System.out.println("Hello World") could not execute inside the browser unless there would be a JavaScript version of class java.lang.System. Therefore XMLVM provides a set of JavaScript emulation libraries that imitate parts of the Java Runtime Environment. .NET runtime environment as well as other specific libraries. Apart from classes such as Java's System or Math it also includes an implementation of Java's AWT and Microsoft's WinForms GUI library. Applications written using these APIs can run inside the browser after being migrated. These client-side emulation libraries are implemented in JavaScript and use DOM manipulations to create GUI elements with the specified parameters. We leverage the aforementioned Qooxdoo library for those DOM manipulations [30].

An interesting side node concerns the emulation library for .NET. Just as for Java applications, XMLVM also includes compatibility libraries for .NET WinForms that include JavaScript versions of classes such as System.Windows.Forms.Button. Similar to the AWT compatibility classes, the implementation of the JavaScript version of these WinForms classes is also based on Qooxdoo. As explained earlier, .NET applications are first cross-compiled to XMLVM_{JVM} and then to JavaScript. However, it is possible to create a binary Java class file once the .NET program has been translated to XMLVM_{JVM}. This fact is shown in the bottom right corner of Fig. 3. For similar reasons, this class file is missing the .NET WinForms library when executed on a standard JVM. XMLVM includes a prototype implementation of such compatibility classes so that a .NET desktop application can effectively also be cross-compiled to a Java desktop application.

5.2. Sample applications

To demonstrate the .NET to web application toolchain, we have implemented a simple calculator in C# using the WinForms GUI library. Emphasis was not so much on the complexity of the application itself, but rather coverage of CLR concepts that are not natively supported in the JVM. Among those are the mapping of numerical operations as discussed in Section 4.2 as well as value types and delegates. Value types are stack-allocated objects whereas delegates resemble function pointers from C/C++. We use an Open Source implementation of .NET called Mono [25] to compile the C# calculator application. The resulting .NET executable is then translated to XMLVM_{CLR} containing a total of 727 CLR byte code instructions. After the data flow analysis, the resulting XMLVM_{JVM} contains 746 JVM byte code instructions. There is only a slight increase in the number of byte code instructions after cross-compiling the calculator application to the JVM. Based on the XMLVM_{JVM} version of the calculator application, the resulting JavaScript version of this application is 4648 lines of JavaScript code. The increase in code size can easily be explained by the fact that typically one stack-based byte code instruction translates to several push and pop operations in JavaScript.

Apart from the byte code level cross-Compilation, various helper classes are needed to run the calculator application. Those helper classes are written in both JavaScript and Java to allow cross-compilation to web as well as desktop applications. Since the JVM does not offer native support for features like delegates and value types, three helper classes have been introduced that mimic their behavior. Furthermore, 15 compatibility classes from the .NET namespace System.* were needed for the calculator application. This includes classes for the WinForms library and system classes such as System.Object or System.String. The WinForms wrapper classes for Java use AWT to render various UI element such as labels and input fields. The JavaScript version of those wrapper classes use Qooxdoo for the same task.

For the Java to web application scenario we have written a more complex application to demonstrate the feasibility of our approach. This application lists movies made in San Francisco [15]. For each movie, details such as release year and synopsis are shown. Additionally, locations where certain scenes of those movies were shot, are displayed as markers on a map of San Francisco. The data about movies and locations is stored in XML files on a server. Those files are fetched by the application using the HttpClient library [2] and the data then parsed using JDOM [18] in order to visualize for the user inside the user interface. To make these two APIs accessible inside the web browser, JavaScript emulation versions have been implemented.

For visualizing the different locations on a map, we make use of the Google Maps API. In order to expose the Google Maps API to a Java application, we have implemented a custom AWT widget that embeds Google Maps. This widget can be embedded in every AWT or Swing GUI application. The Java API of our custom Google Maps widget is inspired by the native JavaScript API as defined by Google. For the browser version, the implementation of this custom widget makes use of the native Google Maps API for JavaScript.

The AWT portion of SF-Movies consists of nine Java classes (not counting the aforementioned Google Maps proxies) with a total of 1684 lines of Java code. The resulting XMLVM of this Java code is 8649 lines of XML. This XML is then cross-compiled via an appropriate stylesheet to 12,008 lines of JavaScript code.

This cross-compiled JavaScript of SF-Movies itself is about 403 kB in size. Compared to the compiled Java code which is about 90 kB this is an increase in code size by a factor of 4. Another factor that comes into play is the missing JavaScript
Table 2
Comparison of related work.

<table>
<thead>
<tr>
<th>Business logic</th>
<th>Client-side execution platform</th>
<th>Cross-compilation</th>
<th>Library API</th>
<th>GUI updates</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP, JSP, etc.</td>
<td>Server</td>
<td>None</td>
<td>None</td>
<td>N/A</td>
<td>Page reload</td>
</tr>
<tr>
<td>Pure JS frameworks</td>
<td>Client</td>
<td>JS</td>
<td>None</td>
<td>Custom API</td>
<td>Partial updates</td>
</tr>
<tr>
<td>RAP, WebCream, SwingWeb</td>
<td>Server</td>
<td>JS</td>
<td>None</td>
<td>Standard API</td>
<td>Partial updates</td>
</tr>
<tr>
<td>WeirdX</td>
<td>Server</td>
<td>Java Applet</td>
<td>None</td>
<td>Standard API</td>
<td>Partial updates</td>
</tr>
<tr>
<td>Java2Script</td>
<td>Client</td>
<td>JS</td>
<td>1 to 1</td>
<td>Standard API</td>
<td>Partial updates</td>
</tr>
<tr>
<td>Qooxdoo web toolkit</td>
<td>Client</td>
<td>JS</td>
<td>1 to 1</td>
<td>Custom API</td>
<td>Partial updates</td>
</tr>
<tr>
<td>XML11</td>
<td>Client</td>
<td>JS</td>
<td>n to m</td>
<td>Standard API</td>
<td>Partial updates</td>
</tr>
</tbody>
</table>

emulation library that has to be loaded into the browser, too. Adding together the migrated application code as well as Qooxdoo and all emulation libraries results in a code size of 909 kB. This requires 9 times as much code to be transferred as the Java byte code, although the latter one benefits from a runtime library already being present inside a Java Runtime Environment. Fortunately, this generated JavaScript code is highly compressible. As all major browsers have built-in GZip support, the code size can be reduced to 23 kB for the application code itself and 132 kB for the complete code including the required libraries. In cases where GZip compression is not possible, special JavaScript compression algorithms can be applied. These take JavaScript code as input and produce again JavaScript that is semantically the same, but syntactically compressed. Two common actions taken by such compressors are stripping out whitespaces and shortening variable names. More advanced compressors, like the Java Script Compressor [35] which is based on the algorithms by Dean Edwards [10] additionally encode JavaScript by using a base64 encoding which can lower code size even more significantly. We integrated this JavaScript compressor as a optional element in our toolchain.

5.3. Analysis

While the code produced by our implementation shows only a minimal increase compared to the amount of original Java byte code, we have observed that network latencies play a more significant role than just code size. This led to the decision to develop a packager that compiles all code necessary to one big chunk that can be downloaded at once. XMLVM could mimic the class loader mechanism of a Java Virtual Machine on the client-side by only loading classes on demand. Depending on the use case, this could lower the total amount of code that needs to be transferred, as not all the classes might be required depending on which parts of the application the user needs. But tests have shown that latency will increase the total loading time significantly, if every class is loaded via a separate HTTP request. This increase is mainly due to the additional round-trip time for every single file. So putting everything in one file is the better solution for the file sizes produced by our approach.

A concern that might be raised by looking at the generated JavaScript code is whether simulating a stack machine results in code that performs well enough. While it is true that our approach generates some overhead, we found that interactive applications will not suffer noticeably. This shows when running a typical SF-Movies session with a profiler as an example. A session of about 3 min of constant use of the application showed that the functions with the most computing time were not related to the code produced by XMLVM. Instead, methods that deal with DOM manipulations that are used by our AWT emulation library showed to be the most computationally intense. The first function that is part of XML11 that needed the most computation time during that test session was actually a hand-written function that emulates String.replace(), because it contains regular expression processing. It was called almost 200 times and used only 36 ms in total. The first compiled method is found many entries below in the profiler’s list. When looking for migrated code, the function with the most processing time only took 11 ms to compute and is responsible for performing a search within the movie repository.

Although this test shows that XMLVM is able to produce code that runs fast for an interactive application, it would not be feasible to cross-compile computationally-intensive applications using our method. But it is questionable if those kind of applications should be cross-compiled to run inside a browser in the first place. In order to reduce the size of the generated code, we are considering to convert the stack-based XMLVM code to a register-based machine. This will be done as future work. Another measure to reduce code size that we have not yet added to our implementation is the removal of dead code. Code that is never reached and never called should not be transferred to the client.

6. Related work

Besides XML11 several other projects offer support for developers to write interactive web applications. These projects have different approaches and thus have different advantages and disadvantages when compared to one another. This sections lists these projects and rates them with respect to the criteria of our taxonomy introduced in Section 2. A summary of this comparison is given in Table 2.
Server-side technologies like PHP and JSP existed for a long time and are widely used for developing interactive web applications. The business logic is executed on the server exclusively. Typically this server-side code is called when a request is received. This code produces (X)HTML code that is then transferred to the client where it is rendered by the browser. No client-side execution takes place with this approach. As a consequence, a new request is sent to the server upon every user interaction. The server-side reacts by executing code that produces new (X)HTML which reflects the last user interaction. This process results in a low responsiveness of the application as the whole user interface needs to be updated after every user interaction. This lack of responsiveness is mainly caused by the fact that the UI becomes unusable from the moment a user causes a new request until the new (X)HTML is rendered by the browser. The increased amount of traffic and rendering time by the browser are additional factors that slow down the interactive session, as even parts of the UI that are static need to be re-transferred and re-rendered.

JavaScript is able to help realizing responsive web applications that lower the traffic and rendering time and stay responsive all the time. The obvious approach for writing such web applications is to use JavaScript directly. It is the native language amongst all browsers and can be executed by the built-in JavaScript interpreter. But browser differences make it difficult to write code that performs well and in the same way on every major platform. This is where pure JavaScript frameworks can support the development process by providing a custom JavaScript API that wraps around the common APIs present inside every browser. Different kinds of wrappers exist. Some JavaScript frameworks aim at cloning existing APIs that perform the same tasks as the original but behave the same across different browser platforms. One prominent example is PrototypeJS [1] that, amongst others, wraps the XMLHttpRequest API that is present in every major browser, but behaves quite differently even between different versions of the same browser. Developers need to learn this custom API first but are then able to write cross-browser JavaScript code, as long as all necessary APIs and browsers are covered by the JavaScript framework used.

Another benefit that JavaScript frameworks can offer are APIs that are not originally available within current JavaScript runtime environments. Such APIs include advanced graphical user interfaces (Qooxdoo [30], Yahoo UI [36]), special effects (Scriptaculous [11]) and remote procedure calls (AJAX RPC [24]). If the application is communicating with a server, this server-side code needs to be developed separately and communication needs to be implemented explicitly. This makes it difficult to test and debug all the code that belongs to an application as one unit. Such frameworks also do not shield from the difficulties introduced by JavaScript itself, which makes it impossible to use powerful tools like development environments and debuggers.

In order to avoid the issues introduced by JavaScript but still be able to take advantage of its features, several projects let developers write their application in another language that offers better support for tools and workflow. For example, several projects – commercial and Open Source – exist that aim at providing an easy migration path for Java applications to web applications. WebCream is a commercial product by a company called CreamTec [4]. They have specialized in providing AWT and Swing replacements that render the interface of the Java application inside of a web browser. Several Open Source projects follow the same idea of exposing Java desktop applications as web applications. One project is called WebOnSwing [28]. One feature offered by WebOnSwing are templates that allow us to change the look-and-feel of the application that is rendered inside the browser. Another project with similar features, but not quite as mature, is SwingWeb [20]. Another project called RAP (Rich AJAX Platform) [8] is part of the Eclipse framework. Similar to WebOnSwing and SwingWeb, RAP builds upon the SWT API. Processing happens on the server-side taking advantage of the OSGi framework. All of these approaches let developers use powerful tools for the Java programming language as well as well-established APIs for common libraries, thus leveraging the knowledge that is already present for Java desktop application developers. Rather than reloading the whole user interface after a user interaction, these approaches only update required parts of the page, thus also allowing the interface to stay responsive while a request is processed on the server-side.

A project that has similar qualities as the aforementioned Java-based solutions is WeirdX [17]. It consists of a Java applet that runs inside of a browser which connects to a running X11 server. It is thus able to display applications that are running on the server and render their GUI inside the X11 server. This presents an abstraction that is even higher than supporting GUI libraries like AWT to be displayed inside a browser. An X11 server is pixel based and a large number of graphical toolkits exist that can render their output on such a system. The disadvantage of this pixel-based approach is that it is not possible to leverage the power of HTML rendering inside a browser using native widgets. Instead, the image data of the rendered X11 server content needs to be sent to the client-side where they are displayed. This raises scalability issues on the server-side, as not only code execution but also rendering takes place on the server. In addition, a Java Virtual machine needs to be present to execute the WeirdX client-side code.

The major difference between approaches like WebCream, WebOnSwing, SwingWeb, RAP as well as WeirdX and the one introduced in this paper is that none of them supports code migration, which means that no client-side code is produced and executed. While the user interface rendered inside the browser is similar to that of a desktop application, every event such as clicking a button, requires an HTTP request to the remote server.

This is what code migration-based projects try to improve. One such project is the Open Source solution Java2Script [32]. It enables applications written in Java to be translated to JavaScript that can be loaded and run by a browser. Java2Script supports the use of basic Java runtime libraries and SWT for GUI development. Other development languages other than Java are not supported, as the project takes Java source code as the source for its translation process. Also JavaScript is the only output that can be generated, which differs from the approach we described in this paper. Communication with a server needs to be written explicitly by the developer by using a custom API provided by the toolkit.
A similar approach to Java2Script is the Google Web Toolkit (GWT) [13] that is also based on cross-compilation. GWT also translates from Java source to JavaScript which requires GWT to parse and support all features from this source code. Because of that the GWT only supports Java 1.4 at the moment and its compiler needs to adapt to every change within the source language in order to support it. Since XMLVM begins on byte code level, new features introduced in Java 5 (such as generics or annotations) are already supported (since generics are handled inside the JVM with existing byte code instructions). Beginning the translation process on the byte code level also makes other features such as the implicit middleware easier to implement, since XMLVM already offers all required information in easy to parse XML-markup.

Just like Java2Script, the GWT focuses on Java source code as the input and JavaScript as the output language, which also differentiates it from our approach. A common feature of both XML11 as well as the GWT is the ability to develop server-side as well as client-side code as one application, which enables the use of debuggers and development environments across the whole application. But the way the two projects handle this feature is different. While the GWT provides a custom RPC API that is used to make calls to server-side methods as well as a raw transport communication mechanism using simple HTTP requests, XML11 introduces the implicit middleware approach XMLIM. Application developers do not have to explicitly write a remote call as such, but instead make a normal call to a server-side method.

Another major difference between XML11 and the GWT that also affects application programmers has to do with the widgets used to build a user interface. GWT relies on either their own widget library or libraries contributed by third parties. That is, GWT is not leveraging existing GUI libraries such as AWT, Swing, or SWT. The benefit of our approach is that application developers can use their existing skill-set without having to learn the details of another GUI library. Our approach also makes it possible to debug the application using existing tools as if it were a desktop application.

Very similar to the GWT is the recently published Qooxdoo Web Toolkit [16]. It mimics the approach that the GWT is taking as it also translates Java source code to JavaScript. It also introduces its own APIs to handle RPC calls and building a GUI. The APIs are similar to the ones introduced by Qooxdoo’s JavaScript framework, thus trying to combine the advantages of Java with their JavaScript library. Again, this requires developers to learn a new API and not use existing Java libraries to develop their web application. Both GWT and QWT only support Java as the input language of their cross-compiler.

7. Conclusions and outlook

Building AJAX applications is a daunting task because of complexities of various technologies required to write end-to-end applications. In particular the JavaScript language itself as well as cross-browser portability issues make the development of AJAX applications difficult. A common solution in computer science is to create proper abstractions that abstract away from the complexities of the underlying system. In that sense, we view JavaScript as the assembler language of the web. XML11 allows a developer to implement an AJAX application in Java or any of the .NET programming languages that is then cross-compiled to portable JavaScript.

The prototype implementation of XML11 replaces Sun Microsystems AWT and Microsoft’s WinForm libraries. Future work will include other plugins to support Swing or SWT. More work is also planned on the cross-compiler. Since XMLVM is based on a stack machine model, the way the code generated during cross-compilations mimics a stack machine which creates unnecessarily verbose JavaScript code. Using a data flow analysis it will be possible to transform the stack-based machine to a register-based machine which would greatly optimize the generated JavaScript code. The foundation of this has been described in [7,33].

XMLVM as an abstraction of the JVM and .NET virtual machines also offers further interesting perspectives outside the domain of web applications. We are currently investigating an aspect weaver for heterogeneous byte code instructions. Using this approach it would be possible to weave an aspect written in Java into a .NET application and vice versa. XMLVM is particularly useful to represent and handle mixed byte code instruction sets. The prototype implementation of XML11 including XMLVM are available under an Open Source license from http://www.xml11.org/.

References