The DOPPIO JVM: Building a Complete Language in the Browser

John Vilk CJ Carey Jez Ng† Emery D. Berger

School of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003

†Dept. of Computer Science
Amherst College
Amherst, MA 01002

{jvilk,ccarey,emery}@cs.umass.edu, jezreel@gmail.com

Abstract

Web browsers are rapidly overtaking native environments as the central focus of application development. Developers are building increasingly rich and responsive applications that run entirely in the browser. However, web applications must be written in JavaScript, the only language available in all browsers. Developers thus cannot reuse existing code in other languages; rewriting this code in JavaScript can be onerous and error-prone.

An attractive alternative is to enable the execution of standard programming languages within the browser. This approach would let programmers use familiar languages and reuse the large body of existing code. However, targeting JavaScript and the browser environment presents significant challenges to language implementors: the vagaries of JavaScript and idiosyncrasies of the browser environment, such as asynchronous-only APIs; a single thread of execution; a lack of standard runtime and OS services; and the diversity of browser implementations.

This paper analyzes the challenges that anyone building a complete language implementation in the browser must face, and shows how to overcome them. We present the design and implementation of DOPPIO, the first complete implementation of the Java Virtual Machine in JavaScript that runs entirely in the browser. We describe how DOPPIO provides the full range of JVM functionality in the browser environment, including language implementation, runtime, and system services. While complete, DOPPIO remains slow: across a set of complex JVM applications, DOPPIO runs between 11× and 60× slower than Oracle's HotSpot JVM interpreter (geometric mean: 29×).

UMass CS Technical Report

1. Introduction

Recent dramatic improvements in JavaScript performance have made it possible to write a wide range applications that run entirely inside the browser. Examples of these include full office suites [7, 16], instant messengers [4], and photo editors [6]. Browsers are an attractive implementation substrate because web applications can run unchanged across desktop, mobile, and tablet platforms.

However, these applications must be written specifically in JavaScript and tailored to the browser environment. Developers cannot take advantage of existing bodies of well-tested code in languages like Java, and must instead rewrite this code in JavaScript. Porting applications to the browser is a difficult process, and presents many challenges:

- **Asynchronous-only APIs**: Core browser APIs are exclusively asynchronous: they must be called with a callback function that the browser invokes when the requested information is available. Developers must therefore rewrite every use of blocking APIs in continuation-passing style.
- **Single-threaded Execution**: JavaScript only supports one thread of execution, complicating support for multiple threads and making it difficult to run long-running code. Browsers cannot repaint the webpage or handle input events until the JavaScript program yields control back to the browser (or when the browser prompts the user to terminate the application).
- **Lack of Operating System Abstractions**: The browser environment lacks many standard OS features that programming languages take for granted, including a filesystem, unmanaged memory access, and standard I/O.
- **Browser Diversity**: Users access the web a wide range of browser platforms, operating systems, and devices. Each combination may have unique performance characteristics, differing support for JavaScript and Document Object Model (DOM) features, and outright bugs. This diversity makes it difficult for developers to ensure that their applications work properly across browsers.

Category	Feature	DOPPIO *JVM*	GWT *Java*	Emscripten *LLVM IR*	IL2JS *MSIL*	WeScheme *Racket*
SYSTEM SERVICES	Filesystem (browser-based) (§4.1)	✓		✓		
	Unmanaged heap (§4.2)	✓		✓		
	Standard input (§4.3)	✓	*	†	*	✓
RUNTIME SYSTEM	Simulates synch APIs using asynch APIs (§5.1)	✓				✓
	Multithreading support (§5.2)	✓				✓
	Responsive long-running apps (§5.3)	✓	*	*	*	✓
LANGUAGE SEMANTICS	Preserves numeric type semantics (§6.4)	✓	✓			✓
JVM FEATURES	Bytecode support (§7.1)	✓		-	-	-
	Dynamic class loading (§7.2)	✓		-	-	-
	Reflection support (§7.3)	✓		-	-	-

Table 1. Feature comparison of language implementations in the browser. A "*" indicates that the feature may require extensive program modifications. "†" means that the feature is incomplete. Unlike previous language implementations, DOPPIO implements all of these features, enabling it to run unmodified programs.

Contributions

This paper analyzes the challenges that anyone building a complete language implementation in the browser must face, and shows how to overcome them. It does so in the context of DOPPIO, the first complete implementation of the Java Virtual Machine in JavaScript that runs entirely inside the browser. DOPPIO is an interpreter that implements the entire set of JVM bytecodes specified in the second edition of the Java Virtual Machine Specification [14]. DOPPIO also implements the set of "native methods" in the Java Class Library (JCL); these are methods that cannot be expressed in bytecode, as they interface with operating system and platform-specific features. Because it faithfully implements the JVM and the native portion of the JCL, DOPPIO can execute unmodified JVM-based programs. Figure 1 compares the architecture of a native JVM implementation to DOPPIO, focusing on system and runtime support.

We describe how DOPPIO is able to provide the full range of JVM functionality in the browser environment, including language implementation, runtime, system services, and managing differences across browser implementations. While DOPPIO cannot compete with a state-of-the-art JVM in terms of performance—it is an untuned interpreter running on top of JavaScript, and degrades performance by 11×-60× over the HotSpot interpreter—it serves as a proof-of-concept that demonstrates the feasibility of implementing complete programming languages inside the browser.

The remainder of this paper is organized as follows: Section 2 presents related work, especially focusing on previous implementations of existing programming languages in the browser. Section 3 provides background information on key features of JavaScript and the browser for readers unfamiliar with this domain. The following sections describe in detail how to implement a language in the browser: Section 4 explains how to implement core operating system features, Section 5 describes runtime system support, and Section 6 focuses on support for common programming language features. Section 7 describes DOPPIO's implementation of JVM-specific functionality like class loaders and reflection. Section 8 empirically evaluates DOPPIO across a range of applications and browsers, demonstrating its functionality and compatibility. Section 9 outlines key modifications to browsers that would dramatically aid the language implementor in terms of simplicity and performance. Finally, Section 10 concludes with directions for future work.

2. Related Work

While DOPPIO is the first comprehensive language implementation in the browser, previous projects have partially implemented existing languages in the browser. Table 1 presents an overview of the features implemented by some well-known projects. DOPPIO is the first system to support all of the features required to run unmodified programs together with full support across a wide range of browsers.

One of the first and most notable language implementations in the browser is the Google Web Toolkit, or GWT, a source-to-source compiler from Java to JavaScript [10]. The goal of GWT is to let web developers write AJAX web applications using a restricted subset of Java. GWT developers can write small widgets and page components in Java which GWT compiles directly to JavaScript. However, GWT does not support compiling *arbitrary* Java programs to JavaScript. Using GWT imposes a number of limitations: widgets must be coded carefully to avoid long-running functions that may make the web page unresponsive; programs can only be single-threaded; and most Java libraries are unavailable. GWT has its own class library that is modeled after the APIs available in the web browser. This class library emulates a limited

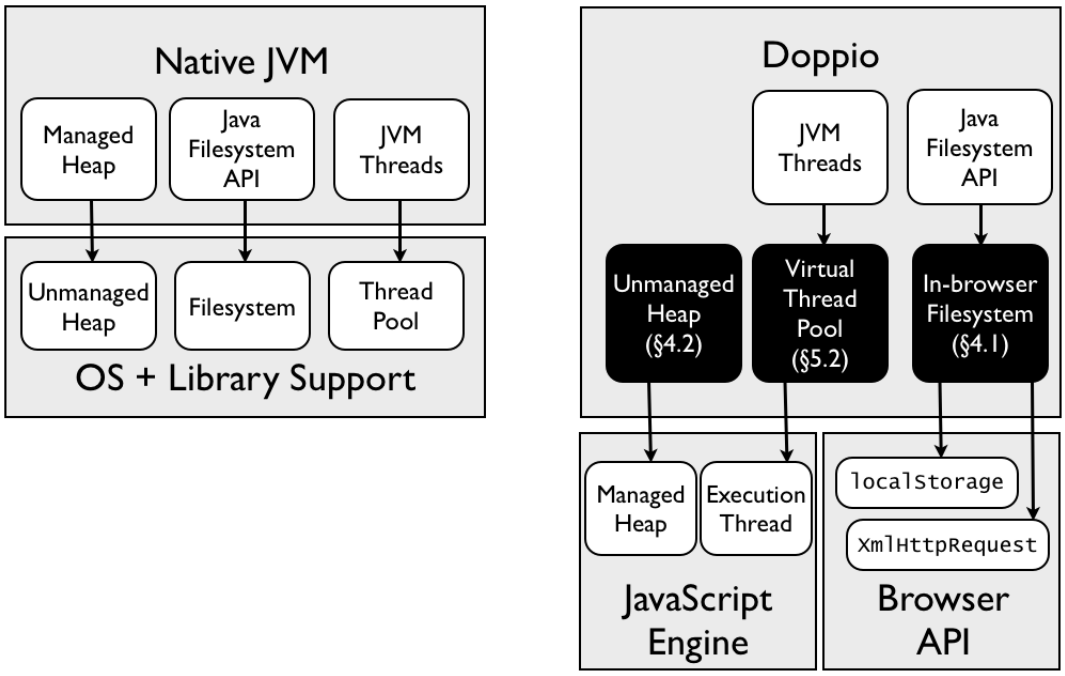

Figure 1. Architectural diagram of runtime and system services used by a native JVM vs. DOPPIO's implemention of a JVM inside the browser. Many components in the desktop environment that are present in system libraries and the operating system are not available in the browser environment: the black boxes indicate DOPPIO-specific components that provide needed functionality.

subset of the classes available in the Java Class Library: essential Java data structures, interfaces, and exceptions [8]. Unlike GWT, DOPPIO runs arbitrary JVM programs in the browser that can be written in any JVM language. DOPPIO ensures that the web page remains responsive, supports multithreaded programs, and implements the runtime and operating system abstractions that the full Java Class Library depends upon.

Mozilla Research's Emscripten project lets developers compile applications from the LLVM Intermediate Representation to JavaScript so they can run in the browser [22]. Emscripten primarily supports C and C++ applications, though in principle it can support any code compiled into LLVM's IR. Emscripten emulates a number of core operating system services such as the heap and the file system, and provides partial graphics and audio support. However, long-running applications freeze the webpage because Emscripten does not automatically yield the JavaScript thread periodically to process browser events (see Section 3.1 for details). Emscripten also does not support multithreaded applications, so each application "thread" must run to termination before other program code can be executed; yielding to other "threads" is not possible. As a result, program event handlers for mouse and keyboard events will not fire unless the application is completely rewritten in an event-driven manner to conform to the browser environment. Finally, Emscripten does not emulate synchronous functions like standard input in terms of the asynchronous APIs available in the browser, which prevents applications from accepting standard input using webpage elements. By contrast, DOPPIO supports long-running applications without freezing the webpage, multithreaded applications, is responsive to browser events, and emulates synchronous functions in terms of asynchronous APIs in the browser.

Fournet et al. describe a verified compiler that compiles an ML-like language called F* to JavaScript [3]. The project used the λ_{JS} JavaScript semantics to formally verify the correctness of the compiler's transformations [11]. However, as this compiler is for a new ML-like language and not for an existing language, it cannot be used to compile and run existing programs in the browser. Furthermore, this compiler does not provide support for any operating system abstractions.

Microsoft's IL2JS compiles .NET Common Intermediate Language (CIL) into JavaScript [15]. This project can compile arbitrary .NET programs into JavaScript, but these programs cannot take advantage of operating system features such as the filesystem, the unmanaged heap, or standard input and output, since IL2JS does not implement any of the native methods in the .NET Base Class Library (BCL). As with other systems described above, any long-running programs compiled with IL2JS will freeze the browser, since IL2JS does not automatically yield the JavaScript thread. IL2JS also does not support multithreading code, and does not support unsigned integers or 64-bit integers; these numeric types are

simply translated into 64-bit doubles. DOPPIO supports running arbitrary unmodified multithreaded JVM programs with full JVM numeric type semantics.

In an unpublished PhD thesis developed concurrently with this work, Yoo et al. describe WeScheme, a hybrid system that makes it possible to run Racket code in the browser [21]. WeScheme comprises a compiler server that is responsible for compiling Racket code into JavaScript, and a JavaScript-based runtime system that copes with many of the drawbacks of the browser environment that DOPPIO overcomes. However, because WeScheme is intended for web programming, it does not emulate operating system services such as the file system or the unmanaged heap. It also does not support all Racket language features, such as reflection and certain primitive functions. By contrast, DOPPIO is written completely in JavaScript and requires no compiler backend, provides JVM programs with operating system services, and implements the full JVM (including reflection functionality) with the goal of providing support for all JVM programs.

Other language implementations exist for the browser. A notable example is Google's Dart language, which can be compiled to JavaScript or executed on a custom VM [5]. In addition, there are a number of so-called *transpilers* like CoffeeScript that provide a convenient layer of syntactic sugar over JavaScript; CoffeeScript's motto is "it's just JavaScript" [13]. DOPPIO itself is written in CoffeeScript.

These languages let developers write web applications using an alternative syntax to JavaScript, and compile directly to JavaScript in a straightforward manner. As a result, these languages face many of the same challenges as JavaScript for application development.

3. Background: JavaScript and the DOM

This section describes the key aspects of the browser environment that are important to understand in order to appreciate the challenges involved when implementing a language in the browser. These include both the JavaScript language and the API exposed to it in the browser, called the Document Object Model (or DOM). The rest of this paper assumes this background knowledge; readers who are intimately familiar with JavaScript web development can skip this section.

3.1 JavaScript Background

JavaScript is a dynamically typed language with prototype-based inheritance and a single thread of execution. It has a number of idiosyncratic properties that are obstacles to implementing more traditional languages.

Prototype-based Inheritance

Because objects in JavaScript are prototype-based, traditional class hierarchies do not map cleanly onto JavaScript objects. Section 6.1 discusses how to represent class-based objects in JavaScript.

Single Threaded Execution and the Event Queue

Due to its origins as an event-driven language, JavaScript has a single thread of execution that is responsible for executing events in a global event queue.[1] When an event arrives in the queue, the JavaScript thread dequeues the event, and executes the event handler to completion. The thread will not handle the next event in the queue until the previous event finishes. It is important to understand that the event queue is not a structure exposed directly to JavaScript code; it is controlled by the JavaScript runtime to queue up events triggered by asynchronous Document Object Model methods. Section 3.2 discusses these methods in more detail.

Intensive JavaScript applications can therefore cause the web page to "lock up" as new user input events are not processed until the current event handler finishes executing. In addition, many browser engines will wait until the JavaScript thread is free to repaint or reflow the webpage in response to page element changes. If an event handler runs for too long, the browser will ask the user if it should stop the script. Language implementations and runtimes are likely to trigger these warnings on nontrivial programs; Section 5.3 discusses how DOPPIO resolves these issues.

Limited Range of Data Types

There are just five core data types guaranteed to be present in all JavaScript implementations: objects, numbers, strings, arrays, and functions. This limited range of data types presents several challenges.

First, none of the core data types are convenient for representing binary data. Newer browsers support binary blobs and typed arrays, but these types are not available in all browsers. Section 4.1 discusses this problem further.

Second, JavaScript stores all numbers as 64-bit floating point numbers. As a result, JavaScript numbers cannot represent integers with more than 52 bits of precision, so 64-bit integers must be emulated. Perhaps surprisingly, a number's value is converted into a 32-bit signed integer for any bit-level operations, so `2147483648.2|0` produces `-2147483648`. Thus, a JavaScript number can have the semantics of a 64-bit floating point number at one point in time, and a 32-bit signed integer at another. Section 6.4 describes strategies for coping with these quirks in the context of implementing a language's numeric types.

Global Scope Name Clashes

JavaScript is a lexically scoped language with a single global scope and function scope. The global scope is shared among all scripts loaded on the current web page, which can result in variable name clashes between an application and any third party libraries it loads. Redeclaring a variable is not a runtime error in JavaScript, so only the version of the variable

[1] Modern browsers have a feature called WebWorkers that enable a limited form of parallel processing in JavaScript; Section 3.2 discusses this feature further.

defined last will be visible. While there are module design patterns that are commonly used in the JavaScript community, there is no official built-in module system to prevent libraries from trampling on other objects in the global scope. As a result, language implementors must be cognizant of the global variables that third party libraries define.

3.2 Document Object Model Background

The Document Object Model (DOM) is the API that the browser exposes to JavaScript applications. It contains functions for a growing list of tasks including webpage element manipulation, file downloads, mouse and keyboard events, and much more. All of these functions are either rooted in the global scope or are defined in the prototype of objects received through the DOM.

Asynchronous-only APIs

Many DOM functions are asynchronous: they consume some arguments and a callback function, which the browser invokes when the requested input is ready. This event registration mechanism makes it simple to create dynamic webpages that react to simple mouse and keyboard events.

For example, a developer can capture keystrokes on the webpage and pop up a window with the key pressed. Unfortunately, if the application is processing something else when the user presses a key, the event handler will not trigger until the processing has completed; it will be stuck waiting in the JavaScript event queue.

These asynchronous API functions are why existing browser language implementations cannot run applications that require some form of input mid-execution. These applications must be rewritten in a completely event-driven manner. Furthermore, some language features, such as standard input, are impossible to properly implement in the browser in a synchronous fashion.[2] Section 5.1 discusses how DOPPIO overcomes this limitation.

Parallel Processing with WebWorkers

In newer browsers, the browser API provides a mechanism called WebWorkers that lets JavaScript applications run multiple scripts in parallel. WebWorkers do not share memory: they run in isolation with individual event queues, and asynchronously pass string messages back and forth with whomever created them. They also do not have access to most of the Document Object Model, so they cannot mutate or access any webpage elements or input devices. Any interaction of WebWorkers with the webpage or input devices must be proxied through the main JavaScript thread. Furthermore, WebWorkers do not address any of the issues mentioned thus far as they still need to deal with asynchronous APIs, such as the API that they use to receive messages, and the JavaScript threading model. Therefore, it is not possible to directly map Java threads to WebWorkers.

Despite their limitations, WebWorkers present an opportunity to take advantage of multiple cores for parallel processing. In particular, they can offload work from the main JavaScript thread to improve application responsiveness, which is a boon for language implementations. Section 5.3 discusses how WebWorkers can be harnessed for a responsive language implementation.

Browser Diversity

Browser diversity presents a challenge to both language implementors and web developers. Browsers differ in both *supported features* and *correctness*, which have important implications for complex web applications.

Some features may not be available in all browsers. In the volatile browser environment, browsers are adding new JavaScript and DOM features at a rapid pace, but many are not implemented in all browsers. In order to take advantage of new features without sacrificing compatibility, a JavaScript application must employ *feature detection* to determine when it has to fall back to simulating the new feature in terms of core JavaScript and DOM functionality. Feature detection tests for the existence of a function object in the global scope, or in a prototype of a class of object defined in the global scope. It is analogous to detecting processor features in a native application. For example, the following code tests for the presence of the WebWorker API:

```
if (window.Worker) {
  // The WebWorker constructor exists, so the
  // application can use WebWorkers.
} else {
  // The browser does not support WebWorkers.
}
```

Feature detection can make JavaScript applications difficult to maintain, as developers must test the normal and the fallback paths through the code to ensure that the application has decent backwards compatibility. There are a number of "polyfills" that define a missing feature in terms of existing features, but many features either cannot be emulated perfectly or are expensive to emulate. A language implementor must be wary of the runtime cost of any polyfills used in a language implementation, since languages contain functionality that may be invoked millions of times in one execution. The runtime in one browser may be completely different in another due to polyfills with different runtime complexity than the feature they emulate.

Some browsers incorrectly implement features. While a browser may have all of the features that an application needs, their implementation may deviate from the specification or have known bugs. An infamous example of incorrect feature implementation is browser support for DOM events, which is widely inconsistent [17]. Some of these bugs are maintained for backward compatibility, as many commonly

[2] There is actually one way: the application can use `window.prompt` to pop up a window requesting a single line of input, which is how Emscripten handles standard input. This approach is generally unsatisfactory.

used JavaScript libraries, such as jQuery, perform browser detection to work around alternative behavior [18].

4. System Services

The web browser lacks a number of core operating system features that modern programming languages depend on, such as the file system, access to unmanaged memory, and standard input and output. As a result, most language implementations in JavaScript will need to implement a number of these operating system abstractions in terms of the resources available in the browser. This section outlines how DOPPIO implements these abstractions.

4.1 File System

Most programming languages depend on the presence of a file system, but browsers do not provide such a facility. DOPPIO simulates a file system using the limited storage features available in the browser. It combines a number of separate features into a single comprehensive Unix-style filesystem. We first describe these storage mechanisms individually before diving into file system design.

The primary browser-based storage mechanism is known as `localStorage`, a key-value store that allows a web application to store up to 5MB of data with the browser. `localStorage` only accepts string keys and string data. DOPPIO stores files into `localStorage` as JSON strings keyed on the file's absolute path. It represents all file data in string form for cross-browser support, which is a relatively space-efficient way to encode arbitrary file data in JSON. For convenience, DOPPIO currently only updates files stored into `localStorage` when the application closes them.

Many languages, including Java, need to dynamically load classes or object files during execution. To enable this, DOPPIO provides a facility for read-only access to files on the webserver that hosts the web application itself. DOPPIO uses a pre-generated JSON listing of files stored at a known URL to determine what files can be retrieved in this manner. These files are downloaded through the `XMLHttpRequest` API, which can asynchronously fetch files using a HTTP request. There are a number of challenges associated with using this API for binary data, which we illustrate in a brief subsection below. DOPPIO primarily uses this filesystem as a method for lazily retrieving Java classes, although it preloads a core set of Java Class Library classes at startup to avoid startup delays caused by the latency involved in each download.

DOPPIO unifies read-only and updatable files into a single virtual filesystem. This filesystem comprises a Unix-style directory tree with different file providers mounted into different directories. Read-only files on the webserver are mounted into one location in the file system, with all other locations mapped into `localStorage` for permanent storage. File providers are responsible for representing their portion of the directory tree, which allows for considerable flexibility in their implementation. Once loaded from a file provider, each file is represented as a JavaScript object that contains the file's data in string form, access permissions, and modification time. JVM programs use the standard `java.io.*` interfaces to interact with the filesystem and individual files, so these details are invisible to the program.

Operating on Binary Data

For convenience and space-efficiency, DOPPIO uses strings to represent binary data. For technical reasons related to JavaScript's string implementation, this form is somewhat tricky to use for byte manipulation.

JavaScript provides a standard `charCodeAt(i)` function that retrieves the Unicode character code of the character that begins at byte `i`. Since Unicode characters can be multiple bytes long, this function can return values up to 2^{16}. However, since `i` specifies a byte offset and not a character offset, JavaScript applications can use bit operations to mask the top bytes of the value returned from `charCodeAt(i)` to get the `i`th byte value.

`XMLHttpRequest` Challenges

Web browsers provide the `XMLHttpRequest` object for transferring data between a client and a server. This object is central to AJAX applications but has inconsistent behavior among browsers [20]. As a result of these inconsistencies, DOPPIO contains three separate methods for using this API to download binary data.

On most browsers, a JavaScript program can fetch a binary file using `XMLHttpRequest` with a user-defined MIME type and retrieve its contents in string form by reading the object's `text` field once the download completes.

However, all versions of Internet Explorer truncate the `text` field at the first NULL character regardless of the MIME-type specified, which is problematic for binary data. DOPPIO works around this limitation by embedding a Visual Basic method into the webpage that, when called, fetches a field on the `XMLHttpRequest` object that Internet Explorer only exposes to Visual Basic code, encodes it as a JavaScript string, and returns it to the JavaScript code.

The ARM variant of Internet Explorer 10 has this NULL character issue and lacks Visual Basic support. Fortunately, Internet Explorer 10 has typed array support which allows the browser to download files as binary blobs. To avoid supporting two different internal representations of data, DOPPIO converts these binary blobs into strings.

4.2 Unmanaged Heap

Many languages expose unsafe memory operations in some manner, such as through APIs for explicit memory allocation and deallocation. In the JVM, programs can take advantage of the nonstandard `sun.misc.Unsafe` API to perform these types of tasks. This functionality requires an unmanaged heap, which is a feature missing in the browser environment. The

unmanaged heap must be simulated in terms of the features available in the browser.

DOPPIO emulates the unmanaged heap using a straightforward first-fit memory allocator that operates on JavaScript arrays. JVM applications rarely use the `Unsafe` API, so we do not expect this allocator to be a performance bottleneck.

Each element in the array is a 32-bit signed integer, which represents 32-bits of data. This approach is convenient because JavaScript only supports bit operations on signed 32-bit integers (see Section 3.1 for more information). When a JVM application calls an API method to write data to the unmanaged heap, DOPPIO converts the data into 32-bit chunks and stores it into the array in little endian format. When the data is later retrieved, DOPPIO decodes it back into its original form. This process can be a complex procedure in JavaScript for data such as floating point numbers; Section 6.4 describes the encoding and decoding process for various numeric types.

Due to the encoding/decoding process, data stored to and read from DOPPIO's heap are actually copied; updates to the data must be kept in sync with the heap copy according to the language's semantics. In the JVM, all unmanaged heap accesses and modifications are explicitly performed through the `sun.misc.Unsafe` interface, so these copy semantics work perfectly for DOPPIO.

Unmanaged Heap using Typed Arrays

Modern browsers support typed arrays that operate on a fixed-size `ArrayBuffer` object. The data in the `ArrayBuffer` can be interpreted as an array of various signed, unsigned, and floating point data types by initializing a typed array of the appropriate type with the `ArrayBuffer`. As a result, DOPPIO can use typed arrays to convert between numeric types in the heap as in the following example:

```
var buff = new ArrayBuffer(4);
var uint32 = new Uint32Array(buff);
uint32[0] = 0x000000FF;
var uint8 = new Uint8Array(buff);
// Prints out 255.
console.log(uint8[0]);
```

Note that typed arrays are little endian; this detail is not configurable. DOPPIO uses `ArrayBuffer` objects for its heap when available to take advantage of these simple numeric conversions.

4.3 Standard Input / Output

Many programs use standard input and output streams to interact with the user. DOPPIO's frontend uses a mock terminal interface in the browser for standard input and output.

Because reading from standard input is a blocking function in the JVM, DOPPIO uses the technique described in Section 5.1 to simulate the blocking functionality in terms of the non-blocking input events exposed by the browser. When a JVM application reads from standard input using the standard `java.io.InputStreamReader` on `System.in`, DOPPIO registers an event handler for keyboard events on the terminal, and yields execution until the requested keyboard event occurs.

Standard output can be synchronous, but the output may not be visible until the application yields the JavaScript thread. This problem is caused by a common optimization. Since repainting and reflowing the webpage is an expensive operation, most browsers wait until the JavaScript thread is not in use to process DOM changes. Some will perform certain changes while JavaScript is executing if enough changes accumulate, but this behavior cannot be depended on. DOPPIO solves this problem by yielding the JavaScript thread each time the JVM application prints to the console to allow the browser to repaint the user interface.

5. Runtime System

Due to JavaScript's threading model (see Section 3.1), implementing a language in the browser also requires extensive runtime support. Many issues stem from JavaScript's inability to switch between execution contexts without destroying the JavaScript stack. We outline how to overcome this issue to enable a language implementation to simulate blocking language support functions in terms of asynchronous browser functions, support multithreaded programs, and ensure that programs remain responsive to user input.

5.1 Simulating Blocking Functions with Asynchronous APIs

Many of the APIs available in the browser environment are asynchronous (see Section 3.2). As a result, most language implementers will encounter synchronous language features, such as standard input, that must be emulated using the asynchronous APIs in the browser.

Simulating synchronous functions with asynchronous APIs requires a form of stack ripping [1]. Stack ripping involves saving the current thread's state into a form that can be resumed later. Due to the JavaScript threading model, this can only be accomplished through explicitly saving the stack and other needed state into a data structure; there is no way to "sleep" or yield execution in any other way for further processing without destroying the stack (Section 3.1 explains JavaScript threading further).

As DOPPIO is an interpreter for a stack-based language, it already has an explicit representation of the stack, which Section 6.2 describes further. DOPPIO uses this object to implement stack ripping, as the callback passed to an asynchronous function can use the object to resume the interpreter. When the JavaScript engine invokes the callback, DOPPIO processes any relevant event data into a form understandable by the JVM, and uses the saved state to resume execution.

One important side effect of stack ripping in this environment is that a language implementation cannot directly use the JavaScript call stack as the language's call stack as the JavaScript stack must be destroyed each time the program

calls an asynchronous function. In addition, there is no reflection API in JavaScript that would make it possible to save the JavaScript stack for restoration later. The JavaScript stack can only be used directly as the language's stack in cases where the language can guarantee that a method call will not use an asynchronous method, but using this optimization is risky for potentially long-running methods (see Section 5.3).

5.2 Multithreading Support

Multithreading is a nontrivial feature to implement in an environment where there is no language support for threads or stack introspection. In JavaScript, the language implementation must be able to switch between multiple different contexts to simulate multiple concurrently running threads. Contextual information for each thread (namely, the call stack) must be explicitly saved and resumed in some manner to simulate switching among different threads.

DOPPIO manages a virtual threadpool that contains each JVM thread's contextual information, and runs only one JVM thread at a time. DOPPIO views thread yielding as a special case of emulating a synchronous function (`java.lang.Thread.yield()`) in an asynchronous manner. When the running thread needs to yield, DOPPIO saves its state using the mechanism described in Section 5.1 and notifies the runtime to potentially switch to another thread.

Using WebWorkers for Multithreading

As described in Section 3.2, WebWorkers enable JavaScript applications to perform a limited form of parallel processing. However, they are not a suitable platform for multithreading support because they do not share memory with any other JavaScript contexts executing in parallel.

Simulating shared memory via message passing would be prohibitively expensive because the message passing interface is asynchronous and a WebWorker follows the same threading model as the main JavaScript thread (which is described in Section 3.1). Each memory access would invoke the stack ripping procedure outlined in Section 5.1.

5.3 Responsive Execution

In a normal environment, input events such as mouse clicks and key presses can be processed as they occur. This functionality is not possible in the browser due to the JavaScript threading model (see Section 3.1). While JavaScript code is executing, the browser will cease to respond to any user input. The user will not be able to scroll the webpage, click on buttons, type characters into forms, or perform any sort of interaction with the webpage. The browser has no control over a webpage's JavaScript code, so its response to this issue is to present the user with a dialog box after a browser-independent delay that allows the user to kill the JavaScript application. This delay is not standardized and is typically time-based, although older browsers that only have a JavaScript interpreter may prompt after a certain number of JavaScript statements.[3]

DOPPIO yields the JavaScript thread periodically to ensure that the web page and application are responsive to user input. At the moment, it uses a timer-based mechanism that works fine in practice, even on JavaScript interpreters with a statement-based threshold for prompting the user to kill the application. Configuring the timeout delay involves a tradeoff between *throughput* and *responsiveness*; yielding more often will ensure that an application receives events as quickly as possible, but will degrade the rate at which language statements are processed.

Using Webworkers for Responsive Execution

If the browser supports WebWorkers, then the language implementation can run inside of the WebWorker to avoid freezing the web page and triggering a browser prompt to kill the application. WebWorkers execute independently from the main JavaScript thread, and thus will not block the main JavaScript thread from receiving and processing events.

However, the tradeoff between throughput and *application* responsiveness remains. WebWorkers do not have access to most of the Document Object Model, and must use the main JavaScript thread as a proxy for many events. These events then get added to the WebWorker's event queue as message passing events, and are blocked until the WebWorker yields the JavaScript thread and processes them. The result is that a user will be able to interact with a non-frozen page while the WebWorker works in the background, but will need to wait until the WebWorker acts upon the input event to notice that the application responded to the event.

If an application does not make use of any DOM functionality that involves receiving messages at arbitrary times, then it does not need to periodically yield the WebWorker thread for responsiveness. Furthermore, if the program does occasionally require this type of DOM input events, the language implementation can dynamically toggle periodic yielding to maximize execution speed while the application is not subscribed to unpredictable DOM events.

Quick JavaScript Yields

Section 5.3 describes that yielding the JavaScript thread is the key to having a responsive language implementation. However, there are multiple mechanisms that a language implementation can use to trigger re-execution, and some are more expensive than others. This section describes a collection of mechanisms that are suitable for a language implementation.

`setTimeout` is commonly-used for delaying a function's execution by a certain number of milliseconds. `setTimeout` is implemented by delaying the placement of the callback event to the back of the JavaScript event queue by at least

[3] In particular, Internet Explorer 8 and below only have JavaScript interpreters and use this metric.

the specified delay. However, even if the specified delay is 0, its specification dictates a minimum delay of 4ms, which significantly degrades the performance of an application that frequently uses it [19].

`Window.sendMessage` is a better option, as it immediately places a message event to the back of the JavaScript event queue, allowing events that occur before it to execute. These messages are string-based, so the application must have a mechanism for associating callback functions with string messages. The application can register an event handler for `sendMessage`, intercept message strings, and use them to trigger the appropriate callback function.

Unfortunately, `Window.sendMessage` is synchronous in Internet Explorer 8; messages sent through `sendMessage` immediately trigger the message handler. In this particular browser, there is an ugly fix: Dynamically insert a `script` element into the HTML document with a callback registered with its `onreadystatechange` event. This callback should remove the `script` element and run the desired callback.

6. Programming Language Implementation

A number of unique characteristics of the browser environment make it difficult to perform a straightforward language port. Any project that ports a language to the browser will need to grapple with implementing many common language features that are not normally problematic in other environments.

In this section, we explain how DOPPIO implements the generic language features of the JVM in the browser; Section 7 describes the implementation of JVM-specific features like class loaders and reflection.

6.1 Objects

Since JavaScript objects are prototype-based, an inheritance-based language's objects cannot be straightforwardly mapped directly onto JavaScript objects. A JavaScript object is essentially a hash table with string keys for fields that has a few built-in properties, such as the `__proto__` field that specifies the object's prototype.

DOPPIO stores each JVM object as a JavaScript object that contains a reference back to its class and a nested object that contains all of its fields keyed on their field name. These field names are qualified with the class that defined them, since Java classes can have fields with the same name as fields declared earlier in the inheritance hierarchy. While not optimal for performance, this arrangement makes it simple to write "native" JavaScript code that interacts with JVM objects, since fields can be accessed by name. It also improves debuggability, as the Java fields are human-readable from within JavaScript debuggers. DOPPIO stashes additional information into certain types of objects, such as `java.io.File`, which DOPPIO uses to link the JVM object with regular JavaScript objects. Any method references, static field accesses, or casting checks are handled through the class reference stashed inside the object.

By mapping Java objects onto JavaScript objects, DOPPIO leverages the existing support for garbage collection within JavaScript. One downside to this approach is that it does not allow the language to use weak references, as there are no weak references in JavaScript. Note that weak references and soft references are only space optimizations, and do not impact Java semantics.

6.2 Call Stack

A language implementation cannot directly use the JavaScript call stack as the language's call stack due to asynchronous operations; this issue is discussed further in Section 5.1. As a result, the language's call stack must be explicitly represented in enough detail to pause and resume execution.

DOPPIO explicitly represents the JVM call stack of a thread as an array of stack frames. Each JVM stack frame contains an array of local variables and, because the JVM is a stack-based machine, a stack for intermediate values. In DOPPIO, each stack frame is represented as an object that contains an array to store the local variables, and another array to represent the stack. As arrays are unbounded, elements can be pushed and popped off the end of an array to simulate a stack; this usage is intended, since JavaScript arrays have `push` and `pop` functions.

6.3 Interpreter Loop

For a language interpreter in the browser such as DOPPIO, most of the execution time is spent in the interpreter loop. As a result, changes to the loop, instruction representation, and JavaScript code associated with an instruction can strongly affect program runtime.

In DOPPIO, each JVM bytecode instruction in a program is represented as a JavaScript object that contains a method for executing the instruction, the length of the instruction in bytes (used to advance the program counter), and the operands to the instruction. Implementing each instruction as an object makes it simple to dynamically rewrite individual instruction's execution methods to avoid expensive runtime checks that must occur the first time it executes. For example, a number of Java bytecode instructions must ensure that the class that they reference is correctly loaded and resolved in the Java class loader before executing, which is a potentially expensive operation. Rewriting the execution method so that future instruction executions skip these checks dramatically improves the interpreter's performance.

6.4 Numeric Types

JavaScript numbers have unique semantics that can make it challenging to simulate some numeric types in the web browser. Numbers in JavaScript are a hybrid of 64-bit floating point and 32-bit signed integer semantics. That is, numbers act as 64-bit floating point values unless they are operated on using bit operations. As a result, JavaScript can naturally

express 32-bit signed integers and 64-bit floating point numbers with ease. However, a language like Java has other data types, such as `long` which is a 64-bit signed integer, that are not easily represented using JavaScript numbers.

DOPPIO uses a single JavaScript number to represent integer types with less precision than 32-bits. In the JVM, these are `boolean`, `byte`, `char`, and `short`. After each operation, DOPPIO checks if an overflow or underflow event has occurred, and correctly emulates the event when it does. These numeric types can generally take advantage of JavaScript's bit-level operations, since the language implementation can mask off the unused top bits after an operation. However, care must be taken to appropriately handle the signed bit for signed types. The JVM only contains opcodes for performing bit-level operations on 32-bit and 64-bit signed quantities, so this issue does not surface in DOPPIO.

It is much more difficult to represent integer types with greater precision than a 32-bit signed integer in JavaScript. Integers with up to 52 bits of precision can be represented within the mantissa of a 64-bit floating point number, although any bit operations must be simulated. Any integer types that require more precision must be simulated using multiple numbers. For 64-bit signed integer operations, DOPPIO uses an accurate `long` library that ships as a part of Google's Closure Tools [9].

32-bit floating point numbers can be simulated using the available 64-bit floating point numbers, but, as with the other numeric types, edge cases need to be monitored. Any operation must be checked to see if results in a quantity that is too large or too small to represent according to the IEEE-754 standard [12]. Operations that result in too-small quantities should result in 0, and those that result in too-large quantities should result in positive or negative infinity according to the quantity's sign.

DOPPIO uses JavaScript's `Number.POSITIVE_INFINITY` and `Number.NEGATIVE_INFINITY` for the infinity quantities; this ensures that future operations with the produced value are processed correctly for special cases that result in `NaN` or infinity. One downside of representing 32-bit floating point numbers as 64-bit floating point numbers is that 32-bit floating point operations will have extra bits of accuracy. We have not discovered any correctness issues with this arrangement in practice, although note that it is possible to expensively simulate floating point arithmetic using a 32-bit bitfield if further correctness is desired.

One technical challenge to floating point number representation is representing various values of `NaN`. Unlike other special quantities such as positive or negative infinity, `NaN` can take on a range of bit values. Embedded programmers sometimes take advantage of this range and use special `NaN` values as sentinels. However, the bit value of `NaN` produced through various operations in JavaScript is nonstandard and undiscoverable in JavaScript engines without typed arrays. DOPPIO uses the `NaN` value available through JavaScript's `Number.NaN`, and makes no guarantee concerning the bit representation of the value.

Reinterpreting as Other Numeric Types

Many languages allow numeric types to be reinterpreted as other numeric types using the bit value of the number. Numeric reinterpretation is required for reading numeric constants from binary files, reading and writing to the JavaScript heap proposed in Section 4.2, and, in the case of Java, formatting floating point numbers as strings. As JavaScript does not provide any language features for reintepreting bits, DOPPIO performs these reinterpretations using regular arithmetic operations.

If the JavaScript engine has typed arrays, DOPPIO leverages this functionality to perform most of these reinterpretations natively. Section 4.2 outlines this process further.

6.5 Exceptions

In the Java programming language, methods can throw exceptions to propagate errors back to a calling function for appropriate handling. This functionality is translated into an `athrow` bytecode instruction in the JVM. The JVM handles locating the appropriate catch block to handle the exception in the methods on the call stack.

DOPPIO implements JVM exceptions in a straightforward manner by using its explicit representation of the JVM call stack (see Section 6.2). When a method throws an exception, DOPPIO pops stack frames off of its call stack representation until it finds a method with the appropriate catch block, which it resumes at the appropriate program counter location to handle the exception.

7. JVM-Specific Concerns

The Java Virtual Machine is a stack-based architecture that comprises over 200 bytecode instructions. The JVM mixes standard low-level instructions like adding values with high-level instructions that directly encodes semantics of Java objects and classes. It also depends on a complex class loading mechanism for dynamically loading in Java class definitions at runtime. This section describes these mechanisms, which are all fully implemented in DOPPIO.

7.1 Bytecode Instructions

The JVM operates on programs represented through a comprehensive set of bytecode instructions. These 200+ bytecode instructions encode many Java programming language features, such as class definitions, exceptions, objects, interfaces, arrays, multidimensional arrays, and monitors. There are also a number of bytecode instructions for miscellaneous control flow operations, ranging from simple jumps to lookup switches. Many bytecode instructions implicitly invoke a class loading process, which Section 7.2 describes further.

DOPPIO implements all 201 bytecode instructions specified in the second edition of the Java Virtual Machine Specification, except for the `breakpoint` opcode which is only

used for debugging purposes [14]. DOPPIO also does not yet implement the `invokedynamic` opcode introduced in Java 7.

7.2 Class Loading

When a bytecode instruction references a class for the first time, the JVM invokes a complex class loading process to resolve the reference to a class definition. This process is specified in Chapter 5 of the JVM specification [14]. DOPPIO uses the process defined in the specification to invoke the appropriate class loader to resolve any class references to concrete class definitions.

The JVM itself must provide a single class loader, called the "bootstrap class loader". This class loader is responsible for resolving references to classes in the Java Class Library and in the user-specified classpath. The classpath contains a colon-delimited list of directories that the bootstrap class loader uses to search for `.class` files. DOPPIO uses its in-browser filesystem to implement this file-searching logic. Most Java programs rely solely on this class loader, as it provides enough functionality for most applications.

The JVM also allows programs to specify custom class loaders, which can either dynamically construct classes in memory or load class files from the file system like the bootstrap class loader. Custom class loaders are commonly employed to implement new languages on top of the JVM. Some large JVM programs also use custom class loaders to implement a module design pattern, with each module using its own class loader to resolve class references. DOPPIO supports custom class loaders, which lets it run programs written in JVM languages other than Java in the browser.

7.3 Reflection

The JVM also provides programs with a rich set of reflection APIs for introspecting upon classes, interfaces, methods, fields, constructors, the JVM call stack, and more. These APIs are provided as a part of the Java Class Library in the `java.lang.reflect` package.

DOPPIO supports the full suite of JVM reflection operations by implementing the native methods associated with reflection in the Java Class Library. When a program uses a reflection operation that invokes one of these native methods, such as checking the fields on an object, DOPPIO creates a JVM object with the requested information using information from its internal data structures. For certain JVM objects produced through reflection, such as `java.lang.Class` objects, DOPPIO stashes a reference to its internal representation of the reflected item into the object. DOPPIO then uses this reference if the program performs further reflection on the object.

8. Evaluation

We evaluate DOPPIO on a set of real and unmodified complex JVM programs across a wide variety of browsers. These benchmarks, and their respective workloads, are as follows:

- `javap` is the Java disassembler. We run `javap` on the compiled class files of `javac`, which comprises 478 class files. We use the version of `javap` and the class files of `javac` that ship with OpenJDK 6.
- `javac` is the Java compiler. We run `javac` on the 19 source files of `javap`. We use the version of `javac` that comes bundled with OpenJDK 6, and the source of `javap` from the same bundle.
- `Rhino` is an implementation of the JavaScript language on the JVM. We run `Rhino` 1.7 on the `recursive` and `binary-trees` programs from the SunSpider 0.9.1 benchmark suite.
- `Kawa-Scheme` is an implementation of the Scheme language on the JVM. We evaluate `Kawa-Scheme` 1.13 on the `nqueens` algorithm with input 8.

Our benchmark computer is a Mac Mini running OS X 10.8.3 with a 4-core 2GHz Intel Core i7 processor and 8GB of 1333 MHz DDR3 RAM. We evaluate DOPPIO in Chrome 26.0, Firefox 19.0.2, Safari 6.0.3, Opera 12.14, and Internet Explorer 10, with Internet Explorer 10 running in a Windows 8 virtual machine using the Parallels 8 software.

Results

Figure 2 presents execution times across various browsers versus Oracle's HotSpot interpreter; we compare against an interpreter to isolate the cost of running inside the browser. DOPPIO achieves its highest performance on Chrome: compared to the HotSpot interpreter, DOPPIO runs between 11× and 60× slower (geometric mean: 29×). This performance degradation is explained by two facts: first, DOPPIO is largely untuned; second, it pays the price for executing on top of JavaScript and inside the browser. By contrast, the HotSpot interpreter is a highly-tuned native executable. While Chrome performs uniformly much better across the benchmarks, we note that we used Chrome as the development platform for DOPPIO. As a result, we may have made design decisions that inadvertently benefited Chrome over other browsers.

9. Discussion

Implementing a language in the browser is a challenging endeavor that could be made much easier with a few new browser features:

Message-passing API. A synchronous message-receiving API for WebWorkers would allow WebWorkers to subscribe to and periodically check for DOM events through the main JavaScript thread without requiring them to yield the JavaScript thread. This makes it trivial to implement synchronous language functionality in terms of asynchronous browser functionality, as a WebWorker could use the main JavaScript thread to perform the asynchronous operation and poll for a response. Language implementations will no longer need to resort to stack ripping for single threaded programs

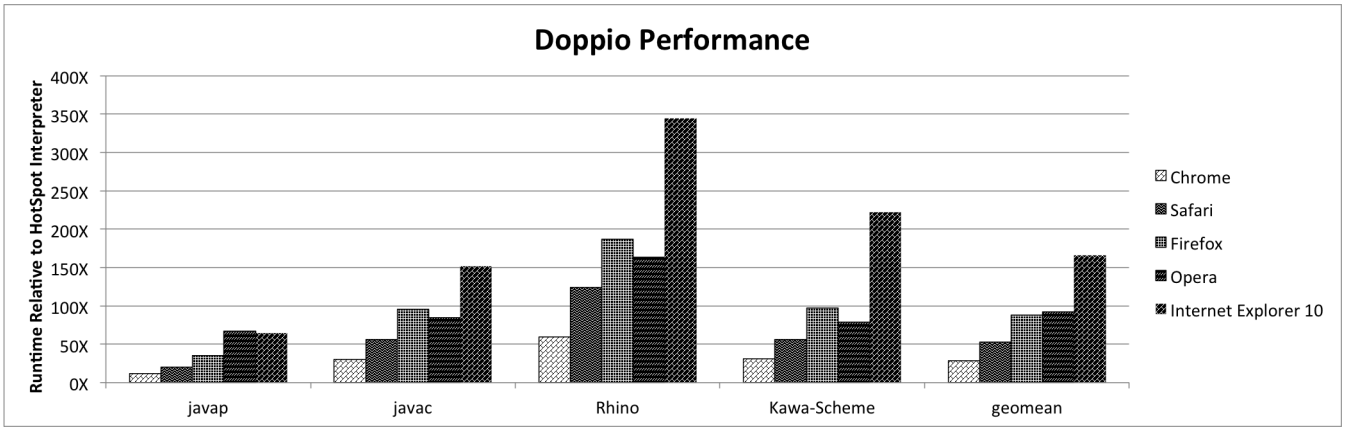

Figure 2. DOPPIO's performance on our benchmark applications relative to the HotSpot JVM interpreter bundled with Java 6. DOPPIO runs between 11× and 60× slower (geometric mean: 29×) than the HotSpot interpreter in Google Chrome.

in the browser, as these programs will no longer have a need to yield the JavaScript thread from within a WebWorker.

Numeric support. Direct support for 64-bit integers would enable languages to efficiently represent a broader range of numeric types in the browser. DOPPIO uses a comprehensive software implementation of 64-bit integers to bring the `long` data type into the browser, but it is extremely slow when compared to normal numeric operations in JavaScript.

10. Conclusion

This paper presents DOPPIO, the first complete implementation of a JVM in JavaScript and the browser. It outlines the challenges required to fully implement any complete language in the browser, including system services, runtime support, and implementation of a wide range of programming language features.

Our near-term plans for DOPPIO include adding additional features such as AWT and Swing support for graphical applications, and cloud storage support for its filesystem. We also plan to investigate ways to speed the interpreter, such as by using superinstructions [2] and potentially exploring lightweight JIT compilation techniques to generate code optimized for particular browsers.

DOPPIO is available for download at `https://github.com/plasma-umass/doppio`.

11. Acknowledgements

The authors would like to thank Jonathan Leahey his contributions to DOPPIO. We also thank Arjun Guha and Shriram Krishnamurthi for their feedback, which greatly improved this paper.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[2] K. Casey, M. A. Ertl, and D. Gregg. Optimizations for a java interpreter using instruction set enhancement. *Department of Computer Science, University of Dublin, Trinity College, Tech. Rep. TCD-CS-2005-61*, 2005.

[3] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *POPL*, pages 371–384, 2013.

[4] Google. About chat in Gmail - Google Chat help. `http://support.google.com/chat/bin/answer.py?hl=en&answer=161934`.

[5] Google. Dart: Structured web apps. `http://www.dartlang.org/`.

[6] Google. Edit your photos - Google+ help. `http://support.google.com/plus/answer/1053729?hl=en&topic=1257351&ctx=topic`.

[7] Google. Google drive. `https://www.google.com/intl/en_US/drive/start/index.html`.

[8] Google. JRE Emulation Reference - Google Web Toolkit - Google Developers. `https://developers.google.com/web-toolkit/doc/latest/RefJreEmulation`.

[9] Google. long.js (Closure Library API Documentation). `http://closure-library.googlecode.com/svn/docs/closure_goog_math_long.js.html`.

[10] Google Web Toolkit Community. Google web toolkit. `https://developers.google.com/web-toolkit/`.

[11] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP*, pages 126–150, 2010.

[12] IEEE. IEEE Standard for Floating Point Arithmetic. `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4610935`.

[13] Jeremy Ashkenas. CoffeeScript. `http://coffeescript.org/`.

[14] T. Lindholm and F. Yellin. *The Java virtual machine specification*. Java series. Addison-Wesley, 1999.

[15] Microsoft Corporation. IL2JS - an intermediate language to JavaScript compiler. `https://github.com/Reactive-Extensions/IL2JS`.

[16] Microsoft Corporation. What is Office 365 - Office.com. http://office.microsoft.com/en-us/business/what-is-office-365-FX102997580.aspx.

[17] Peter-Paul Koch. Event compatibility tables. http://www.quirksmode.org/dom/events/index.html.

[18] The jQuery Foundation. jQuery. http://jquery.com/.

[19] W3C Working Group. 6. Web application APIs. http://www.w3.org/TR/html5/webappapis.html#timers.

[20] W3C Working Group. XmlHttpRequest. http://www.w3.org/TR/XMLHttpRequest/.

[21] D. Yoo. *Building Web Based Programming Environments for Functional Programming*. PhD thesis, Worcester Polytechnic Institute, February 2012.

[22] A. Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *OOPSLA Companion*, pages 301–312, 2011.