DOPPIO: Breaking the Browser Language Barrier

John Vilk Emery D. Berger

School of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{jvilk,emery}@cs.umass.edu

Abstract

Web browsers have become a *de facto* universal operating system, and JavaScript its instruction set. Unfortunately, running other languages in the browser is not generally possible. Translation to JavaScript is not enough because browsers are a hostile environment for other languages. Previous approaches are either non-portable or require extensive modifications for programs to work in a browser.

This paper presents DOPPIO, a JavaScript-based runtime system that makes it possible to run unaltered applications written in general-purpose languages directly inside the browser. DOPPIO provides a wide range of runtime services, including a file system that enables local and external (cloud-based) storage, an unmanaged heap, sockets, blocking I/O, and multiple threads. We demonstrate DOPPIO's usefulness with two case studies: we extend Emscripten with DOPPIO, letting it run an unmodified C++ application in the browser with full functionality, and present DOPPIOJVM, an interpreter that runs unmodified JVM programs directly in the browser. While substantially slower than a native JVM (between 24× and 42× slower on CPU-intensive benchmarks in Google Chrome), DOPPIOJVM makes it feasible to directly reuse existing, non compute-intensive code.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.4.1 [*Operating Systems*]: Process Management – Multitasking; D.4.2 [*Operating Systems*]: Storage Management – Secondary Storage; D.4.4 [*Operating Systems*]: Communications Management – Network Communication

General Terms Languages, Design

Keywords Web, Browsers, Programming Languages, JVM, Operating Systems, JavaScript

1. Introduction

Web browsers have become an increasingly attractive platform for application developers. Browsers make it comparatively easy to deliver cross-platform applications, because they are effectively ubiquitous. Practically all computing platforms—from desktops and tablets to mobile phones—ship with web browsers. Browsers are also getting faster. Most now incorporate optimizing just-in-time compilers for JavaScript, and expose features like access to the GPU through WebGL and high-speed video chat via WebRTC [10, 14]. This combination of features makes it possible for browsers to host the kind of richly interactive applications that used to be restricted to native environments.

In effect, web browsers have become a *de facto* universal computing platform: its operating system is the browser environment, and its sole "instruction set" is JavaScript. However, running languages other than JavaScript in the browser is not generally possible.

There are numerous reasons why browser support for programming languages other than JavaScript would be desirable. JavaScript is a dynamically-typed, prototype-based language whose design contains numerous pitfalls for programmers. Problems with JavaScript have led language implementors to design new languages for the browser that overcome JavaScript's shortcomings, but these solutions all require that programmers learn a new language. Programmers who prefer to program in other paradigms (e.g., functional, object-oriented) currently must abandon these or build hacks onto JavaScript to accomodate their needs. There is also a vast body of well-debugged, existing code written in general-purpose programming languages. Making it possible to reuse this code, rather than requiring that it all be re-written in JavaScript, would speed application development and reduce the risk of introducing errors.

Translation, interpretation, or compilation of languages to JavaScript is necessary but not sufficient. Browsers lack many key abstractions that existing programming languages expect, impose significant limitations, and vary widely in their support for and compliance with standards:

- **Single-threaded Execution**: JavaScript is a single-threaded event-driven programming language with no support for interrupts. Events either execute to completion, or until they are killed by the browser's watchdog thread because they took too long to finish.
- **Asynchronous-only APIs**: Browsers provide web applications with a rich set of functionality, but emerging APIs are exclusively asynchronous. Due to the limitations of JavaScript, it is not possible to create synchronous APIs from asynchronous APIs.
- **Missing OS Services**: Browsers do not provide applications with access to a file system abstraction. Instead, they offer a panoply of limited persistent storage mechanisms, making it difficult to manage large amounts of persistent data. Browsers also lack other OS services such as sockets.
- **Browser Diversity**: Users access the web from 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 to address any of the issues above without excluding a large portion of the web audience.

PLDI '14, June 09–11 2014, Edinburgh, United Kingdom

http://dx.doi.org/10.1145/2594291.2594293

Category	Feature	DOPPIO(JVM) *JVM*	GWT *Java*	Emscripten *LLVM IR*	ASM.js	IL2JS *MSIL*	WeScheme *Racket*
OS SERVICES	File system (browser-based) (§5.1)	✓		*			
	Unmanaged heap (§5.2)	✓		*	†		
	Sockets (§5.3)	✓		✓			
EXECUTION SUPPORT	Automatic event segmentation (§4.1)	✓					✓
	Synchronous API support (§4.2)	✓					✓
	Multithreading support (§4.3)	✓					✓
	Works entirely in the browser	✓					
LANGUAGE SERVICES	Exceptions (§6.6)	✓	✓	✓		✓	✓
	Reflection	✓					

Table 1. Feature comparison of systems that execute existing code inside the browser. The asterisk and dagger indicate limitations that prevent execution on browsers used by over half of the web population today: "∗" denotes a feature that requires a (non-default) backwards-compatibility flag in order to work in those browsers, while a "†" indicates that the feature will not work for them [3]. DOPPIO and the DOPPIOJVM implement all of these features in a cross-platform approach, letting it run unmodified programs in the vast majority of browsers.

Although previous work aims at supporting other languages than JavaScript in the browser, these all fall short. Conventional programming languages and their standard libraries expect the relatively rich execution environment that modern operating systems provide. The fact that browsers lack standard operating systems features like threads, file systems, and blocking I/O means that these projects cannot run existing programs without substantial modifications (§2.1).

This paper identifies and describes how to resolve the impedance mismatch between the browser and the native environment that conventional programming languages expect. We present DOPPIO, a runtime system that makes it possible to execute unmodified applications written in conventional programming languages inside the browser. Its execution environment overcomes the limitations of the JavaScript single-threaded event-driven runtime model (§3) by providing language implementations with emulated threads that support suspending and resuming execution to enable blocking I/O and multithreading in the source language (§4). To support standard library and language features, DOPPIO provides common operating system abstractions including a Unix-based file system abstraction (providing local and cloud-based storage), network sockets, and an unmanaged heap for dynamic memory allocation (§5). All of this support serves as an abstraction layer over the many differences between browsers, letting code run unmodified across Google Chrome, Firefox, Safari, Opera, and Internet Explorer.

We demonstrate the feasibility of using DOPPIO through two case studies. We present the DOPPIOJVM, a prototype yet robust implementation of a Java Virtual Machine interpreter on top of DOPPIO that can run complex unmodified JVM programs in the browser without plugin support (§6). We show that the combination of DOPPIO and the DOPPIOJVM makes it possible to run full, unmodified JVM applications inside a wide range of browsers (§7.1). While substantially slower than a native JVM (between 24× and 42× slower on CPU-intensive benchmarks in Google Chrome), DOPPIOJVM provides acceptable performance for non compute-intensive tasks, and has already been integrated into an educational website that interactively teaches students how to program in Java [21]. We also demonstrate DOPPIO's generality by augmenting Emscripten [26] with DOPPIO and present a case study of porting a C++ game to the browser (§7.2).

Finally, based on the insights learned from implementing DOPPIO and DOPPIOJVM, we propose several extensions to browsers that would greatly simplify and speed support for executing conventional languages (§8).

Contributions

The contributions of this paper are the following:

1. We identify the *execution support* and *operating system abstractions* that conventional programming languages and their runtime libraries require, yet are not present in browsers.
2. We describe how to emulate these resources in the browser on top of JavaScript, and implement them in a runtime system called DOPPIO.
3. As a proof-of-concept, we port the Java Virtual Machine to the browser using DOPPIO, allowing multiple languages and unmodified programs written in those languages to function completely in the browser.
4. We extend the Emscripten system with DOPPIO, making it possible to run a broader range of C/C++ applications inside the browser without modification.
5. We propose several unintrusive browser extensions that would greatly simplify supporting other programming languages inside the browser.

2. Related Work

While DOPPIO is the first runtime system and DOPPIOJVM the first language implementation to allow unmodified code written in a conventional programming language to execute across browsers, previous projects have (partially) implemented existing languages in the browser. Table 1 presents an overview of the features implemented by some well-known projects; only DOPPIO and the DOPPIOJVM implement all of the features required to run unaltered programs.

2.1 Conventional Languages

One of the most prominent and earliest implementations of conventional languages inside the browser is Google Web Toolkit (GWT), a source-to-source compiler from Java to JavaScript [11]. 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, in addition to the usual difficulties of statically compiling Java. With GWT, 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 only a limited subset of the classes available in the Java Class Library, including essential Java data structures, interfaces, and exceptions [8].

Mozilla Research's Emscripten project lets developers compile applications from the LLVM Intermediate Representation to JavaScript so they can run in the browser [26]. 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 convert the program into finite-duration events to prevent blocking browser events (see Section 3 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—that is, the browser will freeze—unless the application is completely rewritten in an event-driven manner to conform to the browser environment. Finally, Emscripten does not emulate synchronous source language functions like the file system API in terms of the asynchronous APIs available in the browser, which prevents applications from operating on files or updating the display with the expected semantics. Alon Zakai, the lead developer of Emscripten, specifies that "JavaScript main loops must be written in an asynchronous way: A callback for each frame", and that "if you do want [synchronous display updates] in a game you port, you'd need to refactor them to be asynchronous" [25].

Mozilla Research's ASM.js project provides language implementations with a stripped-down subset of JavaScript that, when coupled with explicit browser support, removes garbage collection overhead and allows the program to be compiled ahead-of-time (rather than just-in-time) [4]. To accomplish this, ASM.js applications do not use JavaScript objects at all; instead, they manipulate binary structs on its emulated unmanaged heap. As it is a subset of JavaScript, ASM.js applications are still restricted by JavaScript's single-threaded event-driven runtime model (see Section 3). Thus, applications ported to ASM.js face the same runtime-related issues as those ported to JavaScript.

Fournet et al. describe a verified compiler that compiles an ML-like language called F* to JavaScript [6]. The project used the λ_{JS} JavaScript semantics to formally verify the correctness of the compiler's transformations [12]. 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 [17]. This project can compile arbitrary .NET programs into JavaScript, but these programs cannot take advantage of operating system features such as the file system, 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 convert programs into a series of finite-duration events.

Yoo et al. describe WeScheme, a hybrid system that makes it possible to run Racket code in the browser [24]. WeScheme comprises a compiler server 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. WeScheme does not emulate operating system services such as the file system or the unmanaged heap, and lacks support for many Racket language features, including reflection and certain primitive functions.

The Native Client (NaCl), Portable Native Client (PNaCl), and Xax projects let web sites execute sandboxed native code in an efficient manner [2, 5, 23]. NaCl and Xax applications are distributed in machine code form, and are not portable across architectures; the web page must provide a precompiled version of the software for each architecture. PNaCl overcomes this limitation via an architecture-independent bitcode format. However, PNaCl does not support C++ exception handling, dynamic linking, or the most commonly used implementation of the standard library – glibc [9]. All three solutions completely circumvent the JavaScript engine, and thus require explicit browser support to function. These systems provide limited interoperability with JavaScript; as a result, programs running in these systems typically operate as black boxes on web pages, much like Java applets. Unlike these systems, DOPPIO can execute unmodified programs in any modern web browser by taking advantage of its existing JavaScript engine.

By contrast with the systems above, DOPPIO provides a complete platform that makes it possible to run unaltered applications written in conventional programming languages across browsers. DOPPIO ensures that the web page remains responsive regardless of the length of any computation, supports multithreaded applications, and implements the full range of required runtime and operating system abstractions, including synchronous I/O and a file system. The DOPPIOJVM supports running arbitrary, unmodified JVM programs, and supports access to common operating system features through DOPPIO.

2.2 New Languages

Several new languages have been proposed for the browser. Google has created Dart, a language that can be compiled to JavaScript or executed on a custom VM [7]. A number of so-called *transpilers* like CoffeeScript provide a convenient layer of syntactic sugar over JavaScript; CoffeeScript's motto is "it's just JavaScript" [13]. TypeScript is a typed superset of JavaScript from Microsoft that lets programmers annotate JavaScript programs with types, classes, and interfaces [19]. The TypeScript compiler performs type checking before performing a direct translation into JavaScript. DOPPIO itself is written in TypeScript.

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.

2.3 OS Approaches

In recent years, a number of operating systems have appeared that use a modified browser as the exclusive platform for applications. FirefoxOS is a Firefox-based operating system for mobile devices that only supports JavaScript and HTML based applications. ChromeOS is a Google Chrome-based operating system that takes the same approach as FirefoxOS, but adds support for Native Client applications. Both expose additional APIs to access OS-specific components; applications tailored to these environments can use them for additional functionality. As all applications for these platforms must be written in JavaScript or compiled to Native Client, they either suffer from the execution problems outlined in Section 3, or are non-portable across browsers.

The Illinois Browser Operating System (IBOS) tightly couples the browser with the operating system to safely sandbox web pages from native applications and to enable the development of new browser security policies [20]. Rather than providing a path for bringing existing applications to execute in the browser as JavaScript applications, IBOS lets existing applications run

in a native sandbox that exposes a UNIX compatibility layer. In other words, applications are effectively virtualized inside a native environment.

3. Background: Browser Execution Model

This section provides detailed background on the browser environment and JavaScript, focusing on their characteristics and idiosyncrasies that make it impossible to directly execute applications written in conventional programming languages inside the browser. The following sections describe how DOPPIO overcomes these limitations.

3.1 The Execution Model

The JavaScript execution model in the browser is similar to standard GUI application development: JavaScript programs are single-threaded and completely event driven. That is, JavaScript programs execute as a sequence of finite-duration events that block UI interactions. Popular GUI toolkits for other languages, such as Swing, Windows Forms, and Windows Presentation Foundation (WPF), operate in a similar fashion; any computation performed in response to an event blocks all UI repainting and interaction.

Unfortunately, many applications written in conventional languages do not fit this model; that is, they do not decompose naturally into finite chunks of computation, or they rely on multiple simultaneous threads of execution. Even when an application *does* decompose into finite chunks of computation, there is still a problem: the running time of these finite chunks must be limited depending on the browser and the performance of the platform running it. Browsers stop scripts that make it unresponsive to user input for too long (e.g., 5 seconds) due to long-running events. There are no mechanisms for saving execution state to the heap or for performing meaningful stack introspection. As a result, long-running tasks cannot "pause" themselves for later execution (i.e., block) unless they do not rely on stack state or if the programmer manually performs "stack ripping" [1] to convert the application into continuation-passing style. These issues raise significant barriers to bringing existing applications into the web environment, which typically expect a more traditional execution environment.

3.2 Asynchronicity

Most input and output functionality in the browser environment can only be accessed through asynchronous APIs. An asynchronous function receives a callback function as an argument, which it will later invoke with the requested information. Due to JavaScript's execution model, callback invocation occurs as an event; the event will not execute until the JavaScript thread has finished processing all events that occur before it. The JavaScript application cannot block waiting for the event to return, and it cannot introspect on waiting events to process an event at an earlier time. The application must stop executing—that is, complete all processing—in order to let the JavaScript thread process waiting events.

As a result, it is impossible to emulate a synchronous function call using an asynchronous function call. Any functionality available in the browser solely through asynchronous means can never be emulated through synchronous functions. This limitation severely restricts the class of applications that can be brought into the web environment with minimal changes.

As a concrete example of how serious this restriction is, consider the following C++ application. This example does not map cleanly into the browser because it relies on synchronous keyboard input, whereas the browser only exposes asynchronous keyboard events:

```
#include <iostream>
using namespace std;
int main () {
  char name[256];
  cout << "Please enter your name: ";
  cin.getline (name,256);
  cout << "Your name is " << name << endl;
  return 0;
}
```

To port an application like this to JavaScript, in addition to changing the requisite library calls, the application would need to be broken up into separate events that can be assigned to callbacks:

```
function main() {
  var t = document.getElementById('terminal');
  t.innerHTML += "Please enter your name: ";
  var name = "";
  t.onkeypress = function(e) { // Enter key
    if (e.charCode === 13) {
      t.innerHTML += "<br />Your name is " + name + "
          <br />";
      t.onkeypress = null;
    } else {
      var c = String.fromCharCode(e.charCode);
      name += c;
      t.innerHTML += c;
    }
  };
}
```

This case is reasonably straightforward to port, but this type of transformation can become unmanageably complex when blocking is invoked deep within the program. The program must then somehow postpone execution to free up the JavaScript thread until after the callback terminates.

Unfortunately, many browser features, including binary file downloads, are restricted to asynchronous APIs. As a result, it becomes difficult to port applications into the browser that expect to use these features synchronously.

3.3 WebWorkers and their Limitations

One apparent solution to this issue is a browser feature known as WebWorkers. WebWorkers let browser applications offload computation to a separate thread of execution. Unlike threads in other languages, WebWorkers do not share any memory with the JavaScript thread that spawned them. Instead, the only way the JavaScript thread and WebWorkers can communicate is via an asynchronous two-way communication channel that allows either thread to a send a message to the other. These messages are processed using a registered callback.

WebWorker execution proceeds much like the main JavaScript thread. However, WebWorkers have no direct access to user input or to elements on the web page, so event execution does not block user input or GUI repainting. As a result, WebWorkers are well suited for long-running tasks.

Unfortunately, WebWorkers do not solve the problems described above. If a script executing in a WebWorker relies on mid-execution input, it must receive that information from the main JavaScript thread through its asynchronous message-passing interface. WebWorkers also do not enable true shared-memory multithreading in the browser, as there is no shared state among workers and the main JavaScript thread.

4. The DOPPIO Execution Environment

As Section 3 explains, it is not possible to perform a direct translation of arbitrary code into JavaScript for execution in the web browser because of issues with the event-driven browser execution model and the semantics of asynchronous JavaScript APIs. The program must either be extensively modified to deal with the differing semantics, or it must execute in a different execution environment that emulates the source language semantics that it expects.

DOPPIO takes the latter approach. In this section, we explain how the DOPPIO's entirely JavaScript-based execution environment

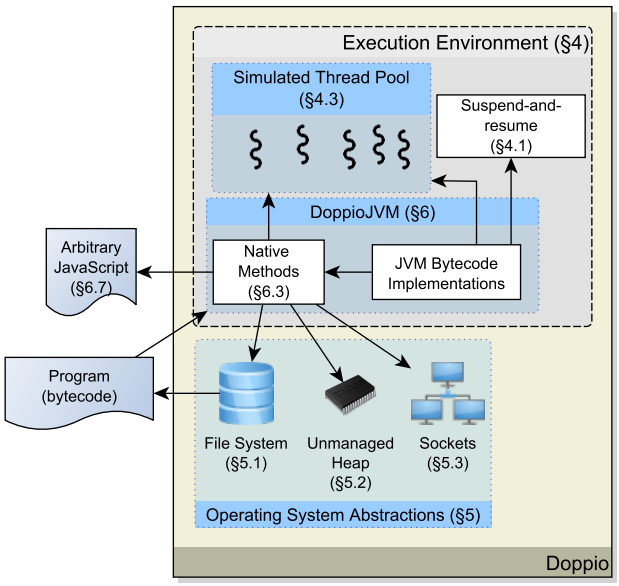

Figure 1. The DOPPIO runtime system makes it possible for existing programs to execute in an unmodified browser via its virtualized execution environment and operating system abstractions. This diagram displays the various components of DOPPIO at a high level, and illustrates how language implementations, such as DOPPIOJVM, rely on them.

automatically segments existing programs into finite-duration events to prevent them from making the browser unresponsive to user input. We next describe how we use this mechanism both to emulate synchronous APIs in the *source* language in terms of asynchronous JavaScript APIs, and to implement multithreading.

4.1 Automatic Event Segmentation

To cope with the browser's execution model, DOPPIO must break up the execution of existing programs into finite-duration events. To perform this task, DOPPIO's execution environment contains a mechanism called *suspend-and-resume* that allows an executing program to suspend itself to the heap to be resumed later. With this mechanism, a program executing in this environment can periodically suspend itself to let other events in the browser event queue like user input execute before it resumes.

Because this mechanism is not natively available in JavaScript, languages implemented using DOPPIO must satisfy two properties:

The call stack must be explicitly stored in JavaScript objects. JavaScript lacks comprehensive introspection APIs and has no mechanism for saving stack state. As a result, programs executing in DOPPIO can only reliably use the JavaScript stack for transient state that will not be needed for program resumption.

The program must be augmented to periodically check if it should suspend. JavaScript lacks preemption: once an event starts executing, it will continue executing until it completes or is killed by the browser. As a result, a language implemented using DOPPIO must call the execution environment periodically to check if it should suspend execution to free up the JavaScript thread.

Both of these transformations can be performed automatically by the language implementation. Section 6.1 describes how DOPPIOJVM implements these features for the JVM.

With an explicit call stack representation in hand, the DOPPIO execution environment can suspend a program for later resumption. To do so, it first creates an anonymous function—the *resumption callback*— that captures the call stack in a closure and that contains the logic needed to resume the program. It then passes the function to an asynchronous browser mechanism that will invoke it later. Various browsers provide different mechanisms that DOPPIO can exploit for this task; we describe these in Section 4.4. Finally, it notifies the language implementation that it should halt execution, with a promise that it will handle resuming it from that point later.

An alternative to this approach is to use ECMAScript 6 generators, which can be used to effectively "pause" a JavaScript function mid-execution with the `yield` statement. This functionality could be used to implement suspend-and-resume by yielding up the call stack. ECMAScript 6 is still in the drafting process, and the proposed generator functionality has only recently been implemented in Firefox and Chrome. As a result, DOPPIO does not use this strategy.

To prevent applications from executing for too long, DOPPIO uses a simple counter to determine when an application needs to suspend. Each suspend check initiated by the language implementation decrements the counter by 1. When the counter reaches 0, DOPPIO determines how long it took for the counter to tick to 0. It then updates a cumulative moving average representing how often the program checks whether or not it should suspend. This new value, along with a preconfigured time slice duration, is then used to set the new counter value.

4.2 Simulating Blocking with Asynchronous APIs

As stated earlier, it is not possible to emulate a synchronous JavaScript API in terms of an asynchronous JavaScript API. However, it *is* possible to emulate a synchronous API in the *source language* in terms of an asynchronous JavaScript API.

To accomplish this, the DOPPIO execution environment provides a variation on the suspend-and-resume functionality described in Section 4.1. When it wishes to invoke an asynchronous JavaScript function, the language implementation must craft a callback function that encapsulates the logic for migrating the data provided through the asynchronous API into items that the language can understand. DOPPIO wraps this callback in a variation of the resumption callback, and then calls the asynchronous API with the modified callback function.

When the browser triggers the resumption callback, it resumes program execution and forwards the data from the asynchronous call to the callback provided by the language implementation. The program executing in DOPPIO resumes as if it had just received data synchronously from a regular function call in its language.

4.3 Multithreading Support

DOPPIO implements multithreading by exploiting the fact that programs executing in DOPPIO maintain an explicit representation of their stack. Since JavaScript lacks a mechanism for preempting execution, multithreading is necessarily cooperative from the JavaScript point of view. However, as language implementations must voluntarily specify valid context switches to DOPPIO, the semantics of multithreading may be preemptive in the source language (as in the Java Virtual Machine).

DOPPIO provides language implementations with a mechanism for switching threads, which is a simple variation of the suspend-and-resume functionality described in Section 4.1. DOPPIO maintains a "thread pool" – essentially an array of call stacks. When the language implementation determines that it is time for a context switch, DOPPIO saves the call stack of the currently running thread into this array, and chooses another thread to resume. Language implementations can provide a scheduling function that determines which thread to resume. By default, DOPPIO resumes an arbitrary thread from the thread pool that is marked as 'ready'.

4.4 Browser Mechanisms for Quick Resumptions

To efficiently implement the suspend-and-resume mechanism described in Section 4.1, DOPPIO needs an asynchronous browser API that is able to insert the resumption callback into the JavaScript event queue as quickly as possible. However, most browsers lack an explicit mechanism for this use case. Below, we describe the options available to DOPPIO; it uses the best choice available in the browser executing it.

`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 the specified delay. However, even if the specified delay is 0, its specification dictates a minimum delay of 4ms, which would result in unacceptable performance [22].

`sendMessage` is a mechanism for sending string-based messages to other open browser windows or tabs. The JavaScript application must register a global event handler to process these messages. This function is a better option for DOPPIO, as it places a message event on the back of the JavaScript event queue immediately. In most browsers, DOPPIO uses this mechanism to implement suspend-and-resume. Since it uses string-based messages, the DOPPIO execution environment generates unique string IDs for each resumption callback, and maintains a map from IDs to callbacks. When DOPPIO receives a message from itself through this interface, it calls the relevant resumption function through the map.

Unfortunately, `sendMessage` is synchronous in Internet Explorer 8; messages sent through `sendMessage` immediately trigger the message handler. For IE8, DOPPIO uses `setTimeout` instead.

`setImmediate` is a mechanism for immediately placing an event to the back of the JavaScript event queue with no delay. This mechanism is ideal for DOPPIO, as it has the exact semantics required to implement suspend-and-resume. As this time, Internet Explorer 10 is the only browser that implements this API, although efforts are underway to make it a standard [18].

5. DOPPIO Operating System Services

The web browser lacks a number of core operating system features that existing programs depend on, such as the file system, access to unmanaged memory, and network sockets. As a result, these abstractions must be implemented in terms of the resources available in the browser so that arbitrary programs can run in the web environment. This section outlines how DOPPIO implements these abstractions.

5.1 File System

Many existing programs depend on the presence of a file system to persist state, but browsers do not provide such a facility. Instead, they provide a hodgepodge of persistent storage mechanisms with different storage formats, restrictions, compatibility across browsers, and intended use cases. Furthermore, many do not expose synchronous interfaces, making it impossible to implement a blocking file system on top of them. Table 2 illustrates the properties and compatibility of a subset of these mechanisms.

However, by combining the execution environment outlined in Section 4 with a unified asynchronous file-based storage abstraction, DOPPIO can provide existing programs with the synchronous file system semantics they expect, with high compatibility across browsers. This approach requires three primary components: (1) a mechanism for manipulating and interpreting binary file data, (2) an implementation of this unified file system API, and (3) a mechanism for defining different "file system" backends for each persistent storage solution, including cloud storage. Figure 2 displays an overview

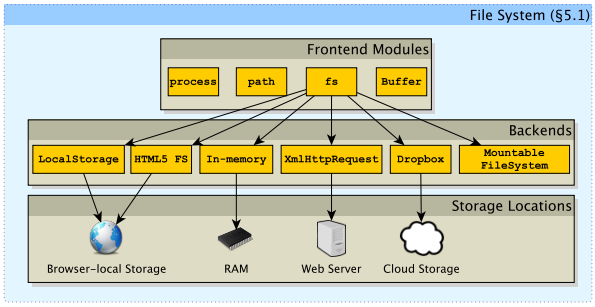

Figure 2. An overview of the DOPPIO file system. Much like an actual operating system, DOPPIO's architecture decouples the frontend interface that programs interact with from the backend implementation that is responsible for interfacing with a particular type of persistent storage.

of the DOPPIO file system. We describe these three components below.

Binary Data in the Browser. Because it is a high-level language, JavaScript does not offer extensive support for manipulating binary data. Some browsers contain an API for natively downloading and manipulating binary data, called "Typed Arrays". Others lack this functionality, and can only download binary data as a JavaScript string. All browsers lack a mechanism for converting between JavaScript strings and binary data, which is required to make use of many of the string-based persistent storage mechanisms in the browser.

To address these deficiencies and inconsistencies, DOPPIO's file system implements the Node JS `Buffer` module in the browser. `Buffer` provides applications with a comprehensive mechanism for manipulating a binary buffer of data. It allows applications to read and write unsigned integers, signed integers, and floating-point numbers of various sizes. It also contains a mechanism for reading and writing binary string data in various formats (ASCII, UTF-8, UTF-16, UCS-2, BASE64, and HEX).

DOPPIO's implementation of `Buffer` can either be backed by typed arrays if the browser has support for them, or by a regular JavaScript array of numbers. The string conversion functionality present in the `Buffer` class serves double-duty as a centralized mechanism that any file system backend can use to read from and write to string-based persistent storage mechanisms. In light of this fact, our `Buffer` implementation supports a special "binary string" format that efficiently packs 2 bytes of data into each JavaScript UTF-16 character; this functionality is only available in browsers that do not perform validity checks on JavaScript strings, as a number of 2 byte sequences are not valid UTF-16 characters. For other browsers, this string format reverts to storing a single byte per character.

Unified File System API. To provide language implementations with a familiar and consistent file system API, DOPPIO emulates the Node JS file system module, `fs`, inside the browser. `fs` is a light JavaScript wrapper around Unix file system calls, like `open` and `stat`. As a result, most languages' file system APIs map cleanly onto its functionality.

DOPPIO also emulates two other Node modules that are closely related to the file system module: `path` and `process`. `path` contains useful path string manipulation functions. `process` encapsulates miscellaneous environment features; DOPPIO only implements the functionality required to emulate changing the current working directory, which programs may rely upon to resolve relative file paths.

	Name	Storage Format	Synchronous	Maximum Size	Compatibility[3]
STANDARDIZED	Cookies	String key/value pairs	✓	4KB	Over 99%
	`localStorage`	String key/value pairs	✓	5MB	$\sim 90\%$
	`IndexedDB`	Object database		User-specified	$< 50\%$
DEFUNCT	`userBehavior`	String key/value pairs	✓	1MB	$< 40\%$
	Web SQL	SQL database		User-specified	$< 25\%$
	`FileSystem`	Binary blobs		User-specified	$< 20\%$

Table 2. Comparison of persistent storage mechanisms available in the browser. Note that this is only a partial listing – we do not include popular storage options enabled through native plugins, such as Flash or Silverlight; nor do we list the numerous cloud storage options. *Synchronous* describes whether or not the mechanism exposes a synchronous interface in the main JavaScript thread. *Compatibility* illustrates the mechanism's compatibility across the current desktop browser market.

The original Node `fs` module exposes two variants of its API: a synchronous and an asynchronous version. Since we are unable to provide a synchronous JavaScript interface for many persistent storage mechanisms, our emulated `fs` module only guarantees the availability of the asynchronous interface for any given backend. Language implementations can combine our asynchronous `fs` module with the synchronous source language API support outlined in Section 4.2 to provide existing programs with the synchronous file system API they normally expect. We describe this process for the JVM in Section 6.3, where we discuss the implementation of DOPPIOJVM.

To better take advantage of JavaScript's strengths, the `fs` API deviates from the Unix standard in one important way: DOPPIO's file descriptors are actual objects. This approach simplifies the implementation of separate backends, which we discuss next.

Backend API. A backend for the file system API only needs to implement nine methods that correspond to standard Unix file system commands: `rename`, `stat`, `open`, `unlink`, `rmdir`, `mkdir`, `readdir`, `close`, `sync`. A backend can optionally also support `chmod`, `chown`, `utimes`, `link`, `symlink`, and `readlink`. The unified file system API handles standardizing arguments to these methods, throwing syntax errors when appropriate, and simulating redundant API functions in terms of these core functions; this service dramatically reduces the amount of logic that each file system needs to implement.

The DOPPIO file system provides backends with a number of useful utility classes: an index that any backend can use to cache directory listings and files, a standard *file* implementation that loads the entire file into memory and implements *sync-on-close* semantics, and the standard `Buffer` module for manipulating binary file data. However, a file system is not required to use any of these utilities; each has complete freedom to implement the internal data structures in any way so long as it consistently implements the backend API.

Via the utility classes, a file system needs to implement just nine methods to provide a new file system backend with full-featured read/write functionality, NFS-style *sync-on-close* semantics, and files that are completely loaded into memory before they can be operated on. This approach makes it possible to quickly build new file system backends.

Unlike Unix, DOPPIO uses objects to represent file descriptors. In addition to being a natural design decision for an object-oriented language, these objects let separate file system backends share core file manipulation logic, which determines the syncing and prefetching strategy for the file system.

Using this backend API, we have implemented backends for five separate file storage mechanisms, which can be seen in Figure 2. Two are backed by different browser-local storage mechanisms (described in Table 2), one provides temporary in-memory storage, one offers read-only access to files served by the web server, and one provides access to Dropbox cloud storage.

Mounting File Systems. DOPPIO's emulated `fs` module is only responsible for interacting with a single root file system. However, a number of systems may want to mount multiple file systems in a Unix-style directory tree. This would provide programs with a convenient mechanism for transferring files to different backends, or for implementing an in-memory temporary file system that emulates `/tmp`.

To facilitate this use case, DOPPIO provides a standard `MountableFileSystem` that handles performing operations across different file system backends. This file system simply uses the standard backend API to facilitate these interactions; as a result, it will be compatible with any new file systems that are implemented in the future, including cloud storage backends.

5.2 Unmanaged Heap

Programs use the unmanaged heap either to perform unsafe memory operations (in managed languages), or as the source of dynamically allocated memory (in unmanaged languages).

DOPPIO emulates the unmanaged heap using a straightforward first-fit memory allocator that operates on JavaScript arrays. 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. When an 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; we chose little endian in order to be consistent with DOPPIO's alternative Typed Array heap implementation, which necessarily uses little endian. When the data is later retrieved, DOPPIO decodes it back into its original form.

Due to the encoding/decoding process, data stored to and read from DOPPIO's heap are actually copied; updates must be kept in sync according to the language's semantics.

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 efficiently convert between numeric types.

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.

5.3 TCP Sockets

For security reasons, browsers do not provide JavaScript applications with direct access to network sockets. Instead, modern browsers provide a feature called WebSockets that enable JavaScript applications to make *outgoing* full-duplex TCP connections with WebSocket servers. For security reasons, JavaScript applications cannot accept *incoming* WebSocket connections.

Newly-opened WebSockets perform a standardized handshake that "promote" an HTTP connection to the WebSocket server

to a WebSocket connection. Once the handshake completes, the JavaScript application can send and receive WebSocket messages, which are encapsulated in WebSocket data frames.

Existing socket-based servers and clients expect a standard TCP handshake and the ability to define custom application-layer data frame formats. As a result, they will not be able to send or receive WebSocket connections out-of-the box.

Resolving this problem requires a solution for *clients* running in the browser that make outgoing socket connections, and *servers* running on native hardware that expect incoming socket connections. DOPPIO resolves the client side of the issue by emulating a Unix socket API in terms of WebSocket functionality. The freely-available Websockify program provides a solution for the server end of the problem; it wraps unmodified programs, and translates incoming WebSocket connections into normal TCP connections [16]. In addition, Websockify provides a JavaScript library that proxies WebSocket connections through a Flash applet in older browsers that lack WebSocket support. DOPPIO uses this library to supply programs with socket support in a wide variety of browsers.

6. DOPPIOJVM

To demonstrate DOPPIO's suitability as a full-featured operating environment for executing unaltered applications written in conventional programming languages, we built DOPPIOJVM. DOPPIOJVM is a robust prototype Java Virtual Machine (JVM) interpreter that operates entirely in JavaScript. DOPPIOJVM implements all 201 bytecode instructions specified in the second edition of the Java Virtual Machine Specification [15], supports multithreaded programs, runs multiple languages that run on top of the JVM, and implements many of the complex mechanisms and native functionality that JVM programs expect. This level of compatibility would not have been possible without the support provided by the DOPPIO execution environment and operating system abstractions. This section describes a number of DOPPIOJVM's key features, and how they rely on support provided by DOPPIO.

6.1 Segmented Execution

Due to the JavaScript execution model, DOPPIOJVM must execute as finite-duration events to prevent the browser from stopping its execution. DOPPIOJVM uses DOPPIO's suspend-and-resume functionality to achieve this. However, it must satisfy the requirements outlined in Section 4.1 before it can use this mechanism.

DOPPIOJVM contains a straightforward JavaScript representation of the JVM call stack. The JVM Specification states that each stack frame contains an operand stack, and an array of local variables. JavaScript arrays are unbounded, and support `push` and `pop` operations; thus, DOPPIOJVM's stack frame is a JavaScript object that contains an array for the operand stack, an array for the local variables, and a reference to the method that the stack frame belongs to. The call stack is simply an array of these stack frame objects. A positive side effect of explicitly representing the call stack in this manner is that DOPPIOJVM trivially supports the Java Class Library reflection APIs for stack introspection.

To ensure that it suspends in a timely fashion, DOPPIOJVM checks at each function call boundary whether it should suspend. This is not a perfect solution, as it is possible in theory to execute an extremely long-running loop that makes no method calls. This concern does not arise in practice; however, it would be possible to instrument loop back edges to perform the same checks.

6.2 Multithreading

DOPPIOJVM uses DOPPIO's "thread pool" to emulate multiple JVM threads. DOPPIOJVM checks for waiting threads at fixed context switch points, such as JVM monitor checks, atomic operations, and any other form of lock-checking.

The current implementation does not prevent the starvation that can occur if a running thread never reaches one of these context switch points. That said, DOPPIOJVM supports a wide range of complex multithreaded programs, some of which we evaluate in Section 7. We plan to switch to a more general mechanism, such as switching threads each time the JVM invokes the DOPPIO suspend-and-resume mechanism.

6.3 Native Methods

The Java Class Library exposes JVM interfaces to a wide variety of native functionality, such as the file system, unsafe memory operations, and network connections. These methods cannot be implemented using JVM bytecodes, and are marked as "native".

DOPPIOJVM implements a wide variety of these native methods directly in JavaScript. The methods corresponding to the file system API use the DOPPIO file system, the methods corresponding to unsafe memory operations use the DOPPIO heap, and the methods corresponding to network connections use DOPPIO sockets. When a native method needs to use an asynchronous browser API, DOPPIOJVM uses the suspend-and-resume mechanism in the manner described in Section 4.2 to "pause" execution until the browser triggers the resumption callback. In this way, the native methods retain their JVM-level synchronous semantics.

Occasionally, JVM programs define and invoke their own native methods written in C, C++, or assembly through the Java Native Interface (JNI). In order to run in DOPPIOJVM, these native methods will need to be reimplemented in JavaScript and registered with DOPPIOJVM in the same manner as Java Class Library native methods.

6.4 Class Loading

When a bytecode instruction references a class for the first time, the JVM invokes a complex dynamic class loading process to resolve the reference to a class definition. This process is specified in Chapter 5 of the JVM specification [15].

However, the class loading mechanism described in the specification assumes the presence of a file system. To resolve a class reference, the class loader is required to check the folders and `JAR` archive files specified on the class path for the class's representative `class` file. In addition, decoding these class file definitions requires functionality that can convert the binary representations of various numeric formats and a standard string format into JavaScript numbers and strings. Neither of these features are available in standard browser environments.

The DOPPIOJVM class loader uses the DOPPIO file system and its `Buffer` module to appropriately download and parse JVM class files. In particular, DOPPIOJVM uses a file system backend that is backed by dynamic file downloads to make the entire Java Class Library available in the browser. When the class loader opens a class file for reading, the file system backend launches an asynchronous download request for the particular file to load it into memory before passing it to the class loader for further execution. This design prevents DOPPIOJVM from loading unneeded class files into memory or browser-local persistent storage before execution.

6.5 Unsafe Memory Operations

The `sun.misc.Unsafe` API lets the JVM perform unsafe memory operations via access to an unmanaged heap. The OpenJDK Java Class Library requires this API, which it uses to determine the underlying system's endianness at startup. DOPPIOJVM uses DOPPIO's unmanaged heap implementation to provide this functionality to JVM programs via the same API.

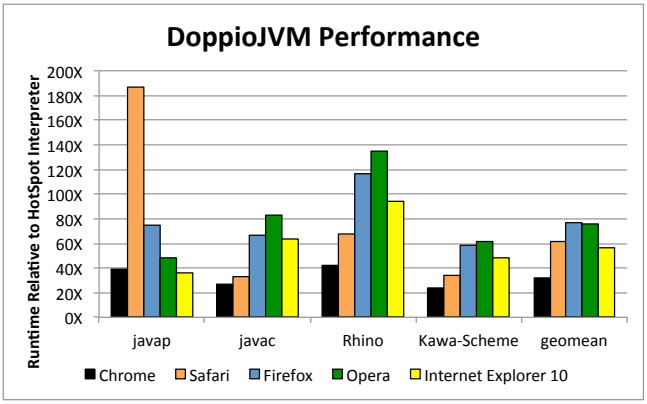

Figure 3. DOPPIOJVM's performance on our benchmark applications relative to the HotSpot JVM interpreter bundled with Java 6. DOPPIOJVM runs between 24× and 42× slower (geometric mean: 32×) than the HotSpot interpreter in Google Chrome. Note that `javap`'s poor performance in Safari is due to a browser bug; we discuss this further in Section 7.

6.6 Exceptions

The JVM is natively aware of exceptions and exception-handling logic. However, because DOPPIOJVM uses DOPPIO to execute as finite-length events, it cannot use JavaScript's native exception mechanisms to emulate JVM exceptions.

Instead, DOPPIOJVM emulates JVM exception handling semantics by iterating through its virtual stack representation until it finds a stack frame with an applicable exception handler, or until it empties the stack and exits with an error.

6.7 JVM Objects and Arrays

DOPPIOJVM maps JVM objects to JavaScript objects, where each object contains a reference to its class and a dictionary that contains all of its fields keyed on their names. JVM arrays are a special type of JVM object; these are mapped to a JavaScript object that contains an array of values and a reference to the special array class that the JVM constructs according to the array's component type. DOPPIOJVM takes full advantage of the JavaScript garbage collector, which automatically collects JVM objects when they fall out of scope.

6.8 Interoperability with JavaScript

While DOPPIOJVM is capable of executing programs entirely written in a JVM language, it can also interoperate with the JavaScript environment. DOPPIOJVM exposes an `eval` method that lets JVM programs execute snippets of JavaScript. This method returns a JVM `String`, which contains the return value of the operation coerced into string form. DOPPIOJVM also makes it possible for a JavaScript program to invoke the JVM much as one would invoke Java on the command line via an API: the programmer specifies the classpath, main class, and arguments, and optionally, custom functions to redirect standard input and output.

7. Evaluation

7.1 Case Study 1: DOPPIOJVM

We evaluate DOPPIOJVM's completeness and performance on a set of real and unmodified complex JVM programs across a wide variety of browsers. We compare DOPPIOJVM's performance to Oracle's HotSpot JVM interpreter provided with OpenJDK, which is able to run JVM programs in the browser using an applet plugin. While Section 2 describes a variety of systems that bring existing programming languages into the browser, these systems are unable to run our benchmarks, so we are unable to compare their performance to DOPPIOJVM.

Our benchmarks and their respective workloads are as follows: `javap` (4KLOC) is the Java disassembler. We run `javap` on the compiled class files of `javac`, which comprises 491 class files. We use the version of `javap` and the class files of `javac` that ship with OpenJDK 6. `javac` (44KLOC) 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` (57KLOC) 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` (121KLOC) 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 10.8.4 with a 4-core 2GHz Intel Core i7 processor and 8GB of 1333 MHz DDR3 RAM. We evaluate DOPPIOJVM in Chrome 28.0, Firefox 22.0, Safari 6.0.5, Opera 12.16, and Internet Explorer 10, with Internet Explorer 10 running in a Windows 8 virtual machine using the Parallels 8 software.

DOPPIOJVM is able to successfully execute all of these applications to completion; we did not need to make any modifications to these applications. Figure 3 presents execution times across various browsers versus Oracle's HotSpot interpreter. DOPPIOJVM achieves its highest performance on Chrome: compared to the HotSpot interpreter, DOPPIOJVM runs between 24× and 42× slower (geometric mean: 32×). This performance degradation is explained by two facts: first, DOPPIOJVM 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.

As Figure 3 shows, Chrome performs better than other browsers across most of the benchmarks we examine. However, it would be problematic to draw any conclusions about Chrome's superiority with respect to other browsers, as we used Chrome as the development platform for DOPPIOJVM. As a result, we may have inadvertently made design decisions that benefited Chrome over other browsers.

While running the `javap` benchmark, we discovered a bug in Safari that causes significant performance degradation. Safari does not properly garbage collect typed arrays; they remain in memory until the browser closes. DOPPIO's file system implementation makes heavy use of typed arrays in browsers that support them to efficiently represent binary data. This detail poses a problem for the `javap` benchmark in this browser, as it manipulates a considerable number of files. As a result, Safari's memory footprint grows to over 6GB over the course of each `javap` benchmark run, causing the OS to page memory to disk and degrade DOPPIOJVM's performance. We have reported this issue to Apple.[1]

Microbenchmarks: To better understand the performance of DOPPIOJVM in isolation from DOPPIO's operating system abstractions, we evaluate DOPPIOJVM on two microbenchmarks: DeltaBlue: a one-way constraint solver, and pidigits: a program that calculates the digits of pi. We run 100 iterations of the DeltaBlue benchmark, and we instruct pidigits to calculate the first 200 digits of pi. These single-threaded benchmarks will spend most of their execution time inside DOPPIOJVM's interpreter loop, as they do not interact with any of DOPPIO's operating system abstractions. However, as we discuss in Section 4.1, DOPPIOJVM *must* periodically suspend-and-resume to remain responsive in the browser environment; thus, benchmark runtimes will include time spent suspended.

[1] See `https://bugs.webkit.org/show_bug.cgi?id=119049`

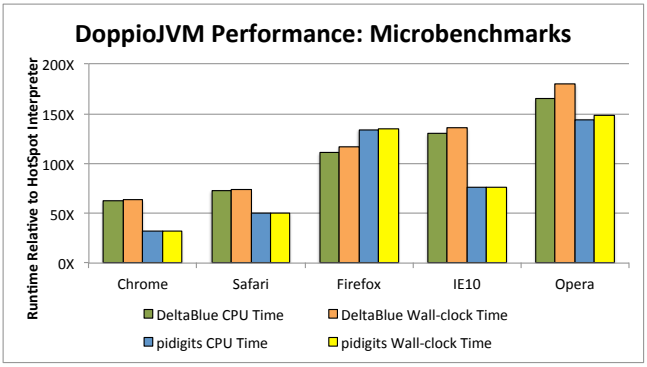

Figure 4. DOPPIOJVM performance on microbenchmarks relative to the HotSpot interpreter. *CPU Time* measures the amount of time that DOPPIOJVM actually spends executing the benchmark, while *Wall-clock Time* measures overall benchmark duration.

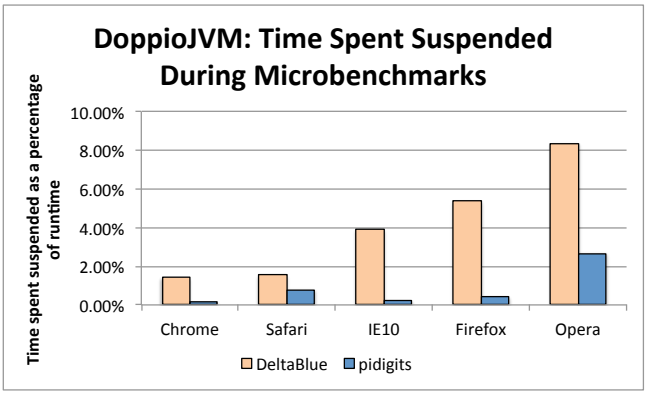

Figure 5. DOPPIOJVM suspension time on microbenchmarks as a percentage of total runtime. As described in Section 4.1, DOPPIOJVM must periodically suspend to remain responsive in the browser. DOPPIOJVM is suspended for less than 2% of execution time in Google Chrome and Safari, suggesting that DOPPIO's threading facilities are not a significant performance bottleneck.

For these microbenchmarks, we augment DOPPIO's suspend-and-resume mechanism to track the amount of time that DOPPIOJVM spends in a suspended state. Like in previous benchmarks, we compare against the HotSpot interpreter to measure the performance gap between a native JVM interpreter and DOPPIOJVM.

Figure 4 displays DOPPIOJVM's performance across different browsers relative to the HotSpot interpreter. *CPU Time* represents actual execution time, disregarding suspension time, whereas *wall-clock time* includes suspension time. Figure 5 displays suspension duration as a percentage of benchmark runtime. Due to DOPPIO's fast suspend-and-resume mechanism (described in Section 4.4), DOPPIOJVM spends less than 2% of runtime suspended in Google Chrome and Safari for DeltaBlue, and less than 1% for pidigits. This result suggests that DOPPIO's threading facilities are not a significant performance bottleneck for languages implemented using DOPPIO.

7.2 Case Study 2: DOPPIO and C++

To further demonstrate DOPPIO's utility and generality, we combined DOPPIO with Emscripten, extending its ability to port C++ applications to the browser. As a case study, we used it to to run the C++ game *Me and My Shadow* in the browser. The Emscripten developers previously ported the core of this game to the web, but the port was incomplete: because Emscripten does not support synchronous dynamic file loading and does not back files to a persistent storage mechanism, the Emscripten demo needs to load all of the games assets into memory prior to execution and does not support game saving.

We modified Emscripten to use the DOPPIO file system, which is able to download the static game assets synchronously as the game requires them, and back the game's configuration folder to `localStorage`. We did not need to modify the game in order to do this; we took the same source code that the Emscripten developers used to make their demo, compiled it with our augmented version of Emscripten, and configured the DOPPIO file system to mount the game's resources and the browser's persistent storage at appropriate folders in the file system hierarchy. The resulting demo does not preload any files, and is able to write to the file system to save game progress and settings.

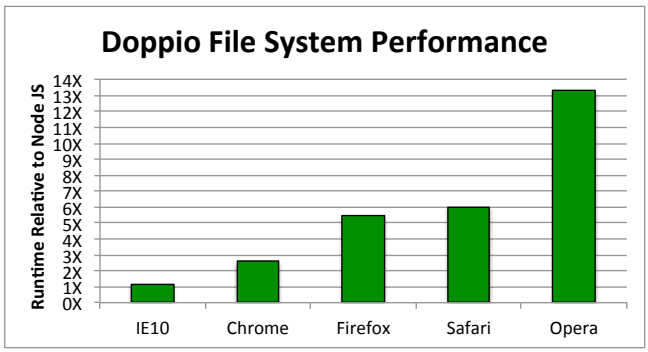

Figure 6. DOPPIO file system performance on recorded file system calls from DOPPIOJVM's `javac` benchmark relative to Node JS running on top of the native OS file system. The DOPPIO file system has nearly identical performance to the native file system in Internet Explorer 10, and is only 2.5× slower in Google Chrome.

7.3 DOPPIO File System

We evaluate the DOPPIO file system on recorded file system calls from DOPPIOJVM's `javac` benchmark. This benchmark performs 3185 file system operations, touches 1560 unique files, reads over 10.5 megabytes of data, and writes 97 kilobytes of data back to disk. Much of this activity is due to the JVM classloader, which pulls in many individual JVM class files as the program references them. We compare our performance against Node JS running on top of the native file system because it implements the same API as DOPPIO's file system. The results from this comparison should indicate how well DOPPIO's file system compares to native file system performance on this particular workload.

Figure 6 displays the results of this benchmark. DOPPIO's file system performance is only 18% slower than native performance in Internet Explorer 10, and about 2.5× slower in Google Chrome.

8. Discussion

Based on insights gleaned while implementing DOPPIO and the DOPPIOJVM, we believe that browsers could add several features that would make it far easier and more efficient for browsers to support conventional languages. These features are limited in scope, are fairly circumscribed in terms of implementation, and we expect they would have little impact on JavaScript programmers or users, while making it far easier to run other languages. By contrast, consider adding multiple threads of execution to JavaScript: while this would ease porting multithreaded applications, it would likely

lead to shared-memory related concurrency errors inside JavaScript applications.

Synchronous message-passing API. A synchronous message-passing API for WebWorkers would allow WebWorkers to subscribe to and periodically check for events through the main JavaScript thread without requiring them to yield the JavaScript thread for event processing. This feature would make 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 periodically poll for a response.

Stack introspection. A sufficiently complete stack introspection mechanism would allow programs to persist their state on the JavaScript heap as objects. Language implementations could then use this feature to implement multithreading and automatic event segmentation without needing to explicitly store the stack state themselves.

Numeric support. Direct support for 64-bit integers would enable languages to efficiently represent a broader range of numeric types in the browser. The DOPPIOJVM 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.

9. Conclusion

While web browsers have become ubiquitous and so are an attractive target for application developers, they support just one programming language—JavaScript—and offer an idiosyncratic execution environment that lacks many of the features that most programs require, including file systems, blocking I/O, and multiple threads. They also are incredibly diverse, further complicating the task of programming web-based applications.

This paper presents DOPPIO, a runtime system for the browser that breaks the browser language barrier. DOPPIO addresses the challenges needed to execute programs written in general-purpose languages inside the browser by providing key system services and runtime support that abstracts away the many differences across browsers. Using DOPPIO, we built DOPPIOJVM, a proof-of-concept complete implementation of a Java Virtual Machine in JavaScript. DOPPIOJVM makes it possible for the first time to run unmodified, off-the-shelf applications written in a conventional programming language directly inside the browser. DOPPIOJVM is already deployed as the compilation and execution engine for the educational website `CodeMoo.com`, which teaches students how to program in Java [21]. We further demonstrate DOPPIO's utility by combining it with Emscripten, extending its ability to port C++ applications to the browser. DOPPIO is available for download at `http://www.doppiojvm.org/`.

Acknowledgements

The authors would like to thank CJ Carey and Jez Ng for their invaluable contributions to DOPPIO and DOPPIOJVM. We also thank Google for funding two Google Summer of Code students to work on DOPPIO, and the students themselves: Giles Lavelle, who implemented Dropbox cloud storage for DOPPIO's file system, and Braden McDorman, who implemented DOPPIO's TCP socket support. We also thank Daniel Jimenez, Yannis Smaragdakis, Kathryn S. McKinley, Arjun Guha, Shriram Krishnamurthi, and the anonymous reviewers 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] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 355–366. ACM, 2011.

[3] P. Bright. Internet Explorer 10 share doubles again on the back of Windows 7. `http://goo.gl/hLYPa5`.

[4] David Herman and Luke Wagner and Alon Zakai. asm.js. `http://asmjs.org/spec/latest/`.

[5] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In R. Draves and R. van Renesse, editors, *OSDI*, pages 339–354. USENIX Association, 2008.

[6] 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.

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

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

[9] Google. NaCl and PNaCl. `https://developers.google.com/native-client/pnacl-preview/nacl-and-pnacl`.

[10] Google. WebRTC. `http://www.webrtc.org/`.

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

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

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

[14] Khronos Group. WebGL - OpenGL ES 2.0 for the Web. `http://www.khronos.org/webgl/`.

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

[16] J. Martin. kanaka/websockify. `https://github.com/kanaka/websockify`.

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

[18] Microsoft Corporation. setImmediate API. `http://ie.microsoft.com/testdrive/Performance/setImmediateSorting/Default.html`.

[19] Microsoft Corporation. Welcome to TypeScript. `http://www.typescriptlang.org`.

[20] R. Sasse, S. T. King, J. Meseguer, and S. Tang. Ibos: A correct-by-construction modular browser. In C. S. Pasareanu and G. Salaün, editors, *FACS*, volume 7684 of *Lecture Notes in Computer Science*, pages 224–241. Springer, 2012.

[21] University of Illinois. Code Moo – A playful way to learn programming. `http://www.codemoo.com/index2.html`.

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

[23] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, 2010.

[24] D. Yoo, E. Schanzer, S. Krishnamurthi, and K. Fisler. Wescheme: the browser is your programming environment. In G. Rößling, T. L. Naps, and C. Spannagel, editors, *ITiCSE*, pages 163–167. ACM, 2011.

[25] A. Zakai. Porting "Me & My Shadow" to the Web. `http://mzl.la/17Mujzr`.

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