BROWSIX: Bringing Unix to the Browser

Bobby Powers, John Vilk, Emery D. Berger
University of Massachusetts, Amherst

Abstract

While standard operating systems like Unix make it relatively simple to build complex applications, web browsers lack the features that make this possible. In this paper, we present BROWSIX, a JavaScript-only framework that brings the essence of Unix to the browser. BROWSIX makes core Unix features available to web applications (including pipes, processes, signals, sockets, and a shared file system) and extends JavaScript runtimes for C, C++, Go, and Node.js programs so they can run in a Unix-like environment within the browser. BROWSIX also provides a POSIX-like shell that makes it easy to compose applications together for parallel data processing via pipes. We illustrate BROWSIX's capabilities by converting a client-server application to run entirely in the browser and developing a serverless LaTeX editor that executes PDFLaTeX and BibTeX in the browser. Creating these applications required less than 50 lines of glue code, demonstrating how easily BROWSIX can be used to build sophisticated web applications from existing parts without modification.

1 Introduction

Web browsers make it straightforward to build user interfaces, but they can be difficult to use as a platform to build sophisticated applications. Code must generally be written from scratch or heavily modified; compiling existing code or libraries to JavaScript is not enough because these applications depend on standard OS APIs, which browsers do not support. Many web applications are thus divided between a browser front-end and a server backend. On the server, applications run on a traditional operating system where they can take advantage of familiar OS abstractions and run a wide variety of off-the-shelf libraries and applications.

As an example, the website MemeGenerator.net lets users create *memes* consisting of images with (hopefully) humorous overlaid text, and performs all image processing server-side. To create a meme, users select a base image and can see a live preview of the resulting image as they type. Each keystroke launches an HTTP request to a server, which runs an image editing program that reads in the base image from the file system and produces a modified image with the desired text. Once complete, the server sends the image in an HTTP response.

Moving meme generation into the browser would reduce MemeGenerator's server costs and reduce latency when the network is overloaded or unreliable, but doing so presents a significant engineering challenge. Browsers lack traditional POSIX sockets, so the client and server code would need to be rewritten to communicate through other means. Browsers also lack processes, so if the server spawns a subprocess to produce memes in parallel, it must be overhauled to use shared-nothing WebWorkers, requiring both sides to be rewritten to use asynchronous message passing for communication. Finally, browsers lack a file system, so the image editing program would need to use other means for IO. Each of these modifications would be nontrivial, and would involve far more engineering effort than simply writing a webserver that spawns ImageMagick processes to handle image generation.

To overcome these limitations, we introduce BROWSIX, a framework that brings Unix abstractions to the browser. BROWSIX is written entirely in JavaScript and requires no plugins, letting it run in modern web browsers including Google Chrome, Firefox, and Microsoft Edge. BROWSIX exposes a wide array of operating system services that applications expect, letting it run unmodified Unix applications (compiled to JavaScript) directly in the browser. BROWSIX does *not* attempt to achieve full coverage of all of POSIX (nor does any standard OS [2]), but instead aims to provide enough coverage to enable porting a wide range of existing applications to the browser:

- **Processes:** BROWSIX implements the POSIX process API (including `fork` and `wait4`) on top of WebWorkers, letting applications run in parallel and spawn subprocesses.
- **Signals:** BROWSIX implements a subset of the POSIX signals API, including `kill` and signal handlers, letting processes communicate with each other in an asynchronous manner.

- **Shared Filesystem:** BROWSIX implements a shared filesystem within the browser, letting processes share state through the FS.

- **Pipes:** BROWSIX exposes the standard pipe API including `pipe2`, making it simple for developers to compose processes into pipelines.

- **Sockets:** Finally, BROWSIX supports TCP socket-based servers and clients, making it possible to run server applications like databases and HTTP servers together with their clients in the browser.

With BROWSIX in hand, we can now directly bring a MemeGenerator-like meme server into the browser with no code modifications (Figure 1). As before, the UI and server communicate via HTTP requests, except now BROWSIX routes the requests to the server process running in the browser. The server uses BROWSIX's shared file system to read in source images, as in a traditional Unix environment. The result is an in-browser application composed of existing parts, with virtually no engineering effort required.

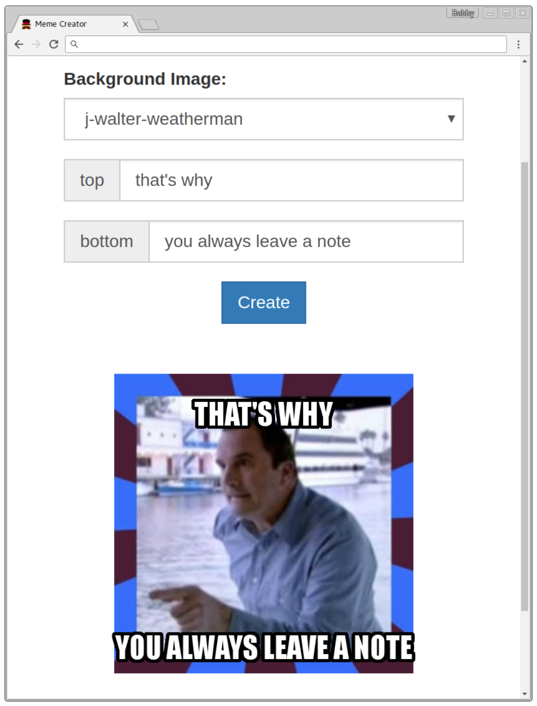
We demonstrate the utility of BROWSIX with two case studies. Using BROWSIX, we extend the meme generator described above to dynamically choose between the cloud-based server or the in-browser server, depending on the client's perceived performance and battery life. We use BROWSIX to build a serverless LaTeX editor on top of PDFLaTeX and BibTeX, demonstrating how easily BROWSIX can be used to build sophisticated applications from existing parts without modifying any code.

Contributions

The contributions of this paper are the following:

- We show how to emulate key Unix abstractions and services in the browser on top of existing web APIs and the challenges of the browser environment it must overcome to do so. We implement these in BROWSIX, a JavaScript-only framework that runs on all modern browsers (§3). BROWSIX includes a shell and terminal to make its features easily accessible to developers (§4).

- We extend the JavaScript runtimes of Emscripten (a JavaScript backend for C/C++ compilation), GopherJS (a Go to JavaScript compiler), and Node.js with BROWSIX support, letting unmodified C, C++, Go, and Node.js programs execute and interoperate with one another within the browser as BROWSIX processes (§5).



j-walter-weatherman

top: that's why

bottom: you always leave a note

Figure 1: A meme generator built using BROWSIX. All server-side functionality was moved into the browser without modifying any code.

- We demonstrate BROWSIX's utility by building a client-server web application and a LaTeX editor out of off-the-shelf components without modification. We characterize BROWSIX's performance with case studies and with microbenchmarks (§6).

- Based on our experience writing BROWSIX, we discuss current browser limitations and propose solutions (§7).

2 BROWSIX Overview

To give an overview of BROWSIX's features, we walk through the process of using BROWSIX to turn a web application into a serverless application without the need for any modifications to the server or client. Note that serverless applications are so called because they lack a backing *application server*; the application's static assets (such as HTML, JavaScript and image files) are still served over HTTP from some web server.

Meme Generator: Our running example is the meme generator described in Section 1. Figure 1 displays a screenshot of the meme generator running as a web application. To generate a meme, the user selects a base image and writes text, and the application issues an HTTP request to the server. The server produces a new image with the text overlaid and embedded into the image. Users can then download the image and share it on social media. Figure 2a presents the application's architecture, which is deliberately similar to the longest running meme generator service on the internet, MemeGenerator.net, which also generates its memes server-side; we wrote our own meme generator because MemeGenerator's source code is not publicly available.

The client is an HTML5 web application, and the server is written in Go. The server reads base images and font files from the filesystem, and uses an off-the-shelf third-party Go library for image manipulation to produce memes [5]. The server also uses Go's built-in `http` module to run its web server.

2.1 Converting to BROWSIX

Converting web applications to use BROWSIX generally consists of the same three step process: (1) compile the code to JavaScript, (2) stage any files required by the application for placement in the in-browser filesystem, and (3) add setup code to the core HTML file to initiate BROWSIX and launch the server. Figure 2b shows a diagram of the meme generator application running with BROWSIX.

Compiling to JavaScript: To run the server in BROWSIX, we compile the server to a single JavaScript file using GopherJS, a Go to JavaScript compiler [9]. To produce JavaScript code, instead of invoking `go build`, the developer simply invokes `gopherjs build`. GopherJS contains a runtime library with JavaScript versions of some of the native methods required by Go's standard library. To connect these to BROWSIX, we extended the GopherJS runtime system with native methods for sockets, processes, and the file system that use BROWSIX (Section 5 describes these changes in detail, which we also applied to Emscripten). As a result, the server runs in BROWSIX without any code modifications, as all of the Go libraries it uses have the native method support that they require to function.

Staging the Filesystem: The meme service requires a set of images and font files to function properly. BROWSIX's file system extends Doppio's file system known as BrowserFS; this in-browser file system supports files stored in cloud storage, browser-local storage, traditional HTTP servers, and more [10]. As we have a small, fixed number of static files, we use the HTTP-backed file system, which requires a file index to support directory listings. We generate this file index using a command line tool that ships with BrowserFS, and then place all of the files on the HTTP server that will serve static web application assets.

BROWSIX Setup Code: The only change needed to the web application is to modify the HTML to load and initialize the BROWSIX JavaScript library. To load the library, we add a `script` tag referring to `browsix.js`; to initialize it, we add another `script` tag with inline JavaScript that calls BROWSIX's `Boot` function with parameters describing the filesystem, along with a callback to invoke when the kernel is ready. In the callback, we instruct the kernel to run the `meme-service.js` program in a new BROWSIX process, which will launch a WebWorker to run the server in parallel with the rest of the application.

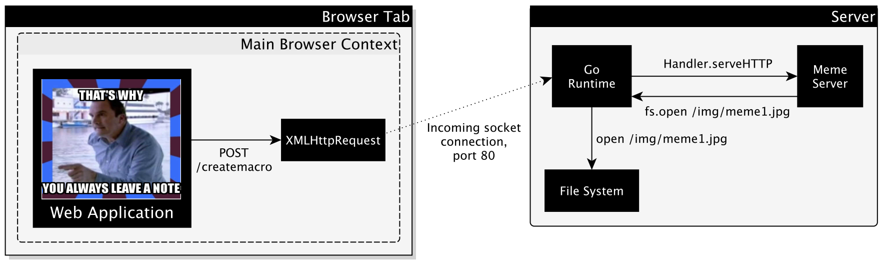
2.2 Execution with BROWSIX

To illustrate how both sides of the application use BROWSIX (client and server), we walk through a simple meme request. Figure 2b displays some of the interactions we describe below.

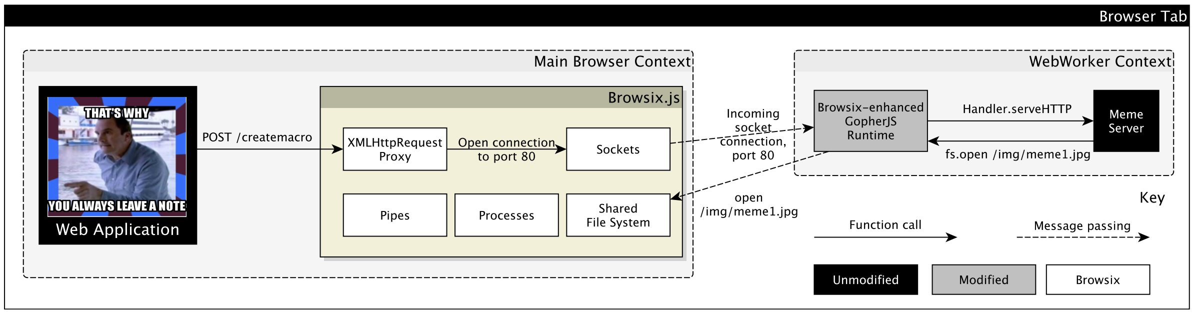
When the user presses the button to generate a meme, the web application issues an HTTP request to the server using the browser's `XMLHttpRequest` interface. BROWSIX's initialization routine installs a modified version of `XmlHttpRequest` that proxies HTTP requests through BROWSIX; this is possible because web browsers expose these interfaces as mutable global variables. This `XMLHttpRequest` interposition is a convenience facility that makes it easy for existing client apps to work with a BROWSIX-hosted server app.

If a server is running in BROWSIX on the target port, BROWSIX proxies the request to the server running in the browser. This process happens *transparently* to the application, which believes it is communicating with a server over the network.

HTTP is implemented over TCP, so the meme server expects HTTP requests to arrive over a TCP connection. BROWSIX translates the HTTP request into the traditional series of BSD TCP socket operations: `socket`, `connect`, `writes`, `reads`, and `close`. Each operation results in the completion of a system call from the perspective of the meme server, which sends a message to the WebWorker running the server. These system call responses are processed by a BROWSIX-specific `syscall` module in the GopherJS runtime library, which translates



(a) Meme generator running without BROWSIX



(b) Meme generator running with BROWSIX.

Figure 2: System diagram of the meme generator application with and without BROWSIX, demonstrating how the client and server interact with one another. With BROWSIX, the server runs in the browser without code modifications.

these messages into the form expected by the standard Go libraries. GopherJS manages necessary stack ripping and restoration [1], while the standard Go libraries handle parsing the request and triggering the meme server's request handler function.

To produce the meme, the server needs to read in the desired image and font files from the file system. For each, it issues calls to read directory contents and open files using Go's standard library, which call into the platform-specific `syscall` module. In Go, this module provides a uniform interface to OS and architecture-dependent functionality – we provide a BROWSIX-specific implementation that sends system calls to our shared BROWSIX kernel over WebWorker messages. Some system calls, such as `getdents64` for reading directory entries, pass structured data between the kernel and processes. All marshaling and unmarshaling of data from byte arrays to the JavaScript objects used by the kernel and BrowserFS happens kernel-side. In the case of `open(2)`, the BROWSIX kernel responds with an integer file descriptor. The application uses this file descriptor in subsequent `read` system calls, which trigger additional messages that relay the contents of the file from the kernel to process.

Once the server has the files it needs, it uses a third-party Go library for graphics manipulation and TrueType text blitting to produce the meme. Then, it sends the resulting image over the BROWSIX TCP socket in an HTTP response. Finally, BROWSIX's HTTP request proxy parses the response, and translates it into `XMLHttpRequest` events that the application expects.

This overview demonstrates how straightforward BROWSIX makes it to port existing components – designed to work in a Unix environment – and execute them seamlessly inside a web browser. The next section provides details of how BROWSIX provides these Unix-like abstractions in the browser environment.

3 The BROWSIX Framework

The core of BROWSIX is a kernel that controls access to shared Unix services. Unix services, including the shared file system, pipes, sockets, and task structures, live inside the kernel, which runs in the main browser thread. Processes run separately and in parallel inside WebWorkers, and access BROWSIX kernel services through the standard system call interface. BROWSIX and all of its runtime services are implemented in JavaScript and TypeScript, a typed variant of JavaScript that compiles to pure

Class	System calls
Process Management	`fork`, `spawn`, `pipe2`, `wait4`, `exit`
Process Metadata	`chdir`, `getcwd`, `getpid`
Sockets	`socket`, `bind`, `getsockname`, `listen`, `accept`, `connect`
Directory IO	`readdir`, `getdents`, `rmdir`, `mkdir`
File IO	`open`, `close`, `unlink`, `llseek`, `pread`, `pwrite`
File Metadata	`access`, `fstat`, `lstat`, `stat`, `readlink`, `utimes`

Figure 3: A representative list of the system calls implemented by the BROWSIX kernel. `fork` requires additional support from the language runtime, and is currently only supported for C and C++ programs.

JavaScript. Figure 8 provides a breakdown of each of BROWSIX's components.

3.1 Kernel

The kernel lives in the main JavaScript context alongside the web application, and acts as the intermediary between processes and loosely coupled Unix services. Processes issue system calls (or *downcalls*) into the kernel to access shared resources, and the kernel relays these requests to the appropriate service. When the service responds to the system call, it relays the response to the process. The kernel is also responsible for relaying signals to processes, which we describe further in Section 3.2. Figure 3 presents a partial list of the key system calls that the kernel currently supports.

In a departure from modern Unix systems, BROWSIX does not support multiple users. A traditional kernel would, for example, use user identities to check permissions on certain system calls or for access to files. Instead, BROWSIX leverages and relies on the browser's built-in sandbox and security features, such as the same origin policy. In other words, a BROWSIX application enjoys the same level of protection and security as any other web application.

3.2 Processes

BROWSIX uses *WebWorkers* as the foundation for emulating Unix processes. However, WebWorkers differ substantially from processes, and BROWSIX must provide a significant amount of functionality to bridge this gap.

In Unix, processes execute in isolated virtual address spaces, run in parallel with one another when the system has multiple CPU cores, and can interact with system resources and other processes via system calls. However, the web browser does not expose a process API to web applications. Instead, web applications can spawn a WebWorker that runs a JavaScript file in parallel with the application.

A WebWorker has access to only a subset of browser interfaces, which notably excludes the Document Object Model (DOM), runs in a separate execution context, and can only communicate with the main browser context via asynchronous message passing. WebWorkers are not aware of one another, cannot share memory with one another, and can only exchange messages with the main browser context that created them (see Section 3.6 for a discussion). Most browsers do not support spawning subworkers from workers, called *nested workers*, and have not added support for them since they were first proposed in 2009. Thus, if a WebWorker needs to perform a task in parallel, it must delegate the request to the main browser thread, and proxy all messages to that worker through the main browser thread. Perhaps unsurprisingly, the limitations and complexity of WebWorkers have hindered their adoption in web applications.

By contrast, BROWSIX implements Unix processes on top of Web Workers, giving developers a familiar and full-featured abstraction for parallel processing in the browser. Each BROWSIX process has an associated task structure that lives in the kernel that contains its process ID, parent's process ID, Web Worker object, current working directory, and map of open file descriptors. Processes have access to the system calls in Figure 3, and invoke them by sending a message with the system call name and arguments to the kernel. As a result, processes can share state via the file system, send signals to one another, spawn sub-processes to perform tasks in parallel, and connect processes together using pipes. Below, we describe how BROWSIX maps familiar POSIX interfaces onto Web Workers.

`spawn`: BROWSIX supports `spawn` (`posix_spawn` in POSIX), which constructs a new process from a specified executable on the file system. `spawn` is the primary process creation primitive used in modern programming environments such as Go and Node.js, as `fork` is unsuitable for general use in a multithreaded process. `spawn` lets a process specify an executable to run, the arguments to pass to that executable, the new process's working directory, and the resources that the subprocess should inherit (such as file descriptors). In BROWSIX, an executable is a

JavaScript file in its file system. When an process invokes `spawn`, BROWSIX creates a new task structure with the specified resources and working directory, and creates a new Web Worker that runs the target JavaScript file.

There are two technical challenges to to implementing `spawn`. First, the WebWorker constructor takes a URL to a JavaScript file as its first argument. Files in BROWSIX's file system may not correspond to files on a web server. For example, they might be dynamically produced by other BROWSIX processes. To get around this restriction, BROWSIX generates a JavaScript *Blob* object that contains the data in the file, obtains a dynamically-created *URL* for the blob from the browser's window object, and passes that URL as a parameter to the Web Worker constructor. All modern web browsers now support constructing Workers from blob URLs.

The second challenge is that there is no way to pass data to a Worker on startup apart from sending a message. As processes access state like the arguments vector and environment map synchronously, we require that BROWSIX-enabled runtimes delay execution of a process's `main()` function until after the worker has received an "init" signal containing the process's arguments and environment.

`fork`: The `fork` call creates a new process that is a copy of the current address space and thread. Web Workers do not expose a cloning API, and JavaScript lacks the needed reflection primitives required to serialize a context's entire state into a snapshot. Thus, BROWSIX only supports `fork` when a process is able to completely enumerate and serialize its own state. Section 5 describes how we extend Emscripten to provide `fork` support for C/C++ programs compiled to JavaScript.

`wait4`: The `wait4` system call is a potentially blocking system call that blocks until a child can be reaped. It returns immediately if a child has already died, or the `WNOHANG` option is specified. Waiting requires that the kernel not immediately free task structures, and required us to implement the zombie task state for children that have not yet been waited on. The C library used by Emscripten, musl, uses the `wait4` system call to implement the C library functions `wait`, `wait3`, and `waitpid`.

`exit`: BROWSIX-enhanced runtimes are required to explicitly issue an `exit` syscall when they are done executing, as a parent has no other way of being notified that a Worker has finished. This is due to the event-based nature of JavaScript environments – even if there are no pending events in the Worker's queue, the main JavaScript context could, from the perspective of the browser, send the Worker a message at any time.

`getpid`, `getppid`, `getcwd`, `chdir`: These four system calls operate on the data in current process's task structure, which lives in the BROWSIX kernel. `getpid` returns the process's ID, `getppid` returns the parent process's ID, `getcwd` returns the process's working directory, and `chdir` changes the process's working directory.

3.3 Pipes

BROWSIX pipes are implemented as in-memory buffers with read-side wait queues. If there is no data to be read when a process issues a `read` system call, the callback encapsulating the system call response is enqueued and will not be invoked until data is written to the pipe. For simplicity of implementation, writes to pipes do not currently block, which is a departure from POSIX semantics. We enforce blocking in order to prevent resource exhaustion in the kernel because it limits the writer's ability to execute. Otherwise, because of the asynchronous nature of message passing system calls, the kernel would not be able to limit resource consumption caused by messages from a malicious or wayward process.

3.4 Sockets

BROWSIX implements a subset of the BSD/POSIX socket API, with support for `SOCK_STREAM` (TCP) sockets for communicating between BROWSIX processes. These sockets enable servers that `bind`, `listen` and then `accept` new connections on a socket, along with clients that `connect` to a socket server. Sockets are sequenced, reliable, bi-directional streams. BROWSIX uses a pair of pipes, one per direction, to implement sockets.

3.5 Shared File System

BROWSIX extends BrowserFS's file system, part of Doppio [10]. BrowserFS includes support for multiple mounted filesystems in a single hierarchical directory structure. BrowserFS provides multiple file system backend implementations, such as in-memory, zip file, XMLHttpRequest, Dropbox, and an overlay filesystem. BrowserFS provides a unified, encapsulated interface to all of these backends, so the rest of the kernel does not need to know which filesystem is in use.

BROWSIX system calls that operate on paths, like `open` and `stat`, are implemented as method calls on the kernel's BrowserFS instance. When a system call takes a file descriptor as an argument, the kernel looks up the descriptor in the tasks's file hashmap, and invokes the appropriate methods on that file object. The file descriptor table is inherited by child processes, and each object (whether it is a file, directory, pipe or socket) is managed with reference counting.

3.6 Browser-Imposed Limitations

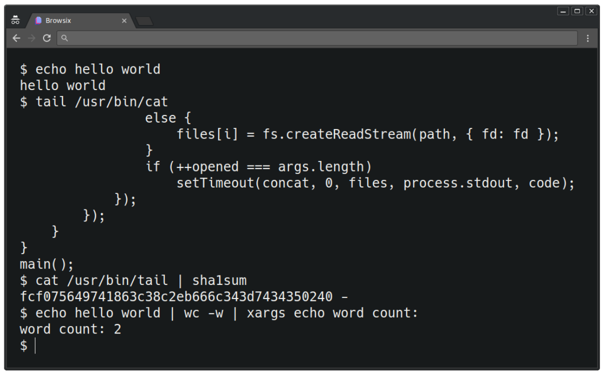
The browser environment currently imposes certain limitations that prevent BROWSIX from implementing a broader range of Unix functionality; in particular, they do not provide a facility for shared memory. The development editions of certain browsers do provide support for `SharedArrayBuffers`, which enable shared access to a segment of memory, along with atomic and synchronization primitives [7]. However, this support is not yet widespread; as of this writing, it is only supported by development editions of Firefox and Chrome and the specification is currently in draft status. Until this feature is implemented by standard browsers, it is not practical to provide support for shared memory across processes or to provide multi-threading that takes advantage of multiple cores.

4 The BROWSIX Shell

To make it easy for developers to interact with BROWSIX, we implement an in-browser Unix shell modeled on `sh`. Like other standard shells, the BROWSIX shell allows combining arbitrary processes into pipelines, as well as backgrounding processes.

BROWSIX also includes a terminal and a variety of utilities written in JavaScript for Node.js. The terminal, as shown in Figure 4, features line buffering – when the user hits enter, the current line is sent to a shell command, `sh`, running in a regular BROWSIX process, which parses the command, creates pipes, launches processes, and waits for processes to complete.

The shell includes the following commands: `cat`, `cp`, `curl`, `echo`, `exec`, `grep`, `head`, `ls`, `mkdir`, `rm`, `rmdir`, `sh`, `sha1sum`, `sort`, `stat`, `tail`, `tee`, `touch`, `wc`, and `xargs`. These were initially developed and tested on Linux and MacOS X desktop systems under Node.js 4.2.1, and run equivalently and without modification in BROWSIX.



```
$ echo hello world
hello world
$ tail /usr/bin/cat
                else {
                    files[i] = fs.createReadStream(path, { fd: fd });
                }
                if (++opened === args.length)
                    setTimeout(concat, 0, files, process.stdout, code);
            });
        });
    }
}
main();
$ cat /usr/bin/tail | sha1sum
fcf075649741863c38c2eb666c343d7434350240 -
$ echo hello world | wc -w | xargs echo word count:
word count: 2
$ |
```

Figure 4: A browser-based terminal that provides a development environment on top of BROWSIX, giving easy access to the BROWSIX shell (§4).

```
kernel.system(
    'pdflatex example.tex',
    function(pid, code) {
        if (code === 0) {
            console.log('success');
        } else {
            console.log('failed');
        }
    }, logStdout, logStderr);
```

Figure 5: Creating a BROWSIX process from JavaScript.

5 Runtime Support

Applications access BROWSIX system calls indirectly through their runtime systems. This section describes the runtime support we have built for the browser environment, GopherJS, Emscripten, and Node.js so they can use BROWSIX without code modifications.

5.1 Browser Environment Extensions

Web applications run alongside the BROWSIX kernel in the main browser context, and have access to BROWSIX features through several global APIs. BROWSIX exposes new APIs for process creation, file access, and socket notifications, and extends the existing XMLHttpRequest interface to transparently proxy HTTP requests to BROWSIX processes.

File access acts as expected, and allows the client to manipulate the filesystem, invoke a utility or pipeline of utilities, and read state from the filesystem after programs have finished executing. Figure 5 shows how client applications invoke BROWSIX processes and react when processes exit through an API similar to C's `system`.

Socket notifications let applications register a callback to be invoked when a process has started listening on a particular port. These notifications let web applications launch a server as a process and appropriately delay communicating with the server until it is listening for messages. Web applications do not need to resort to polling or ad hoc waiting.

BROWSIX patches the global XMLHttpRequest API to transparently proxy HTTP 1.1 requests to BROWSIX processes as if they were remote servers. This API handles connecting a BROWSIX socket to the server, serializing the HTTP request to a byte array, sending the byte array to the BROWSIX process, processing the (potentially chunked) HTTP response, and generating the expected web events. When no process is listening on the destination port, BROWSIX forwards the request to the remote server as normal.

5.2 Common Services

BROWSIX provides a small `syscall` layer as a JavaScript module that runs in a Web Worker. This layer provides a concrete, typed API over the browser's message passing primitives. Language runtimes use this module from their standard libraries to communicate with the shared kernel. Methods provided by the `syscall` layer take the same arguments as Linux syscalls of the same name, along with an additional argument: a callback function. This callback is executed when the `syscall` module receives a message response from the kernel. Unlike a traditional single-threaded process, a BROWSIX process can have multiple outstanding system calls, which enables runtimes like GopherJS to implement user-space threads on top of a single Web Worker execution context.

Signals are sent over the same message passing interface as system calls. The common `syscall` module provides a way to register signal handlers for the standard Unix signals, such as `SIGCHLD`.

5.3 Runtime-specific Integration

For many programming languages, existing language runtimes targeted for the browser must bridge the impedance mismatch between synchronous APIs and threads, present on Unix-like systems, and the asynchronous, single-threaded world of the browser. Systems like Doppio, clojurejs, and GopherJS all employ different approaches. Rather than attempting to unify thread handling across implementations, BROWSIX only requires that implementations expose a way to save and restore a thread of execution (the stack and program counter (PC)).

```
function sys_getdents64(cb, trap, fd, dirp, len) {
    var done = function (err, buf) {
        if (!err)
            dirp.set(buf);
        cb([err ? -1 : buf.byteLength, 0, err ? err :
              0]);
    };
    syscall_1.syscall.getdents(fd, len, done);
}
```

Figure 6: Implementing the `getdents64` syscall in GopherJS.

Go: Go is a systems language developed at Google designed for readability, concurrency, and efficiency. To run Go programs under BROWSIX, we extended the existing GopherJS compiler and runtime to support issuing and waiting for system calls under BROWSIX. GopherJS already provides full support for Go language features like goroutines (lightweight threads), channels (communication primitives), and delayed functions.

We extended the GopherJS runtime with support for BROWSIX through modifications to the runtime. The main integration points are a BROWSIX-specific implementation of the `syscall.RawSyscall` function (which handles syscalls in Go), along with overrides of several Go runtime functions.

We wrote a replacement for `RawSyscall` in Go. It allocates a Go channel object, and this function invokes the BROWSIX JavaScript syscall library, passing the system call number, arguments, and a callback to invoke. `RawSyscall` then performs a blocking read on the Go channel, which suspends the current thread of execution until the callback is invoked. When the system call response is received from the BROWSIX kernel, GopherJS's existing runtime takes care of re-winding the stack and continuing execution. The syscall library indexes into a system call table by syscall number, and invokes a function specific to that system call to marshalling data to and from the BROWSIX kernel. Adding support for any new system call is a matter of writing a small handler function and registering it in the system call table; an example is shown in Figure 6

BROWSIX replaces a number of runtime functions, but the most important are `syscall.forkAndExecInChild` and `net.Listen`. The former is overridden to directly invoke BROWSIX's `spawn` system call, and the latter to provide access to BROWSIX socket services. Additional integration points include an explicit call to the `exit` syscall when the main function exits, and waiting until the process's arguments and environment have been received before starting `main()` (see §3.2).

```
__syscall220: function(which, varargs) {
  return EmterpreterAsync.handle(function(resume) {
    var fd = SYSCALLS.get(), dirp = SYSCALLS.get(),
        count = SYSCALLS.get();
    var done = function(err, buf) {
      if (!err)
        HEAPU8.subarray(dirp, dirp+buf.byteLength).
             set(buf);
      resume(function() {
        return err ? err : buf.byteLength;
      });
    };
    SYSCALLS.browsix.syscall.getdents(fd, count,
         done);
  });
},
```

Figure 7: Implementing the BROWSIX `getdents64` syscall in Emscripten.

Component	Lines of Code (LoC)
Kernel (§3)	2,058
BrowserFS modifications	40
Shared syscall module (§5.2)	421
Emscripten integration* (§5.3) *(C/C++ support)*	1,115
GopherJS integration* (§5.3) *(Go support)*	724
Node.js integration (§5.3)	1,957
TOTAL	**6,315**

Figure 8: BROWSIX components. * indicates these components are written in JavaScript, while the rest of the components are written in TypeScript.

C and C++: We also extend Emscripten, Mozilla Research's LLVM-based C and C++ compiler that targets JavaScript, with support for BROWSIX. This work requires use of Emscripten's interpreter mode (named the "Emterpreter") to saving and restore the C stack. While BROWSIX support requires functions that may be on the stack under a system call to be interpreted (so that the stack can be replayed when the system call completes), Emscripten can selectively compile other parts of an application, such as computational kernels that do not issue system calls down to `asm.js`, which will be JIT-compiled and run as native JavaScript by the browser.

As with GopherJS, Emscripten provides a clear integration point at the level of system calls. Emscripten provides implementations for a number of system calls, but is restricted to performing in-memory operations that do not block. We replace all Emscripten syscall implementations with ones that call into the BROWSIX kernel, such as in Figure 7. In the case of `getdents` and `stat`, padding was added to C structure definitions to match the layout expected by the BROWSIX kernel.

When a process calls `fork`, the runtime sends a copy of the global memory array, which includes the C stack and heap, along with the current program counter (PC) to the kernel. After the kernel launches a new Web Worker, it transfers this copy of global memory and PC to to the new Worker as part of the initialization message. When the BROWSIX-augmented Emscripten runtime receives the initialization message, if a memory array and PC are present the runtime swaps them in, and invokes the Emterpreter to continue from where `fork` was invoked.

Node.js: Node.js (a.k.a. "Node") is a platform for building servers and command line tools with JavaScript, implemented in C, C++ and JavaScript, on top of the v8 JavaScript engine. Node.js APIs are JavaScript modules that can be loaded into the current browser context by invoking the `require` built-in function. These high-level APIs are implemented in platform-agnostic JavaScript, and call into lower-level C++ bindings, which in turn invoke operating system interfaces like filesystem IO, TCP sockets, and child process management. Node.js embraces the asynchronous, callback-oriented nature of JavaScript – most Node APIs that invoke system calls take a callback parameter that is invoked when results are ready.

To run servers and utilities written for Node.js under BROWSIX, we provide a `browser-node` executable that packages Node's high-level APIs with pure-JavaScript replacements for Node's C++ bindings that invoke BROWSIX system calls as a single file that runs in a BROWSIX process. BROWSIX also replaces several other native modules, like the module for parsing and generating HTTP responses and requests, with pure JavaScript implementations. Node executables can be invoked directly, such as `node server.js`, or will be invoked indirectly by the kernel if node is specified as the interpreter in the shebang line of a text file marked as executable.

6 Evaluation

This evaluation answers the following questions:

1. Does bringing Unix abstractions into the browser enable compelling use cases?

2. Is the performance impact of running programs under BROWSIX acceptable?

6.1 Case Studies

We evaluate the applicability and advantages of bringing Unix abstractions into the browser with two case studies. First, using the meme generator from the overview (§2), we modify the web application to perform local BROWSIX-based or remote requests to a service running in the cloud based on current network and device characteristics. We then build programs from texlive-2015 for BROWSIX, and use it in a statically-hosted LaTeXeditor.

6.1.1 Dynamically Switching Between Cloud and In-Browser Execution

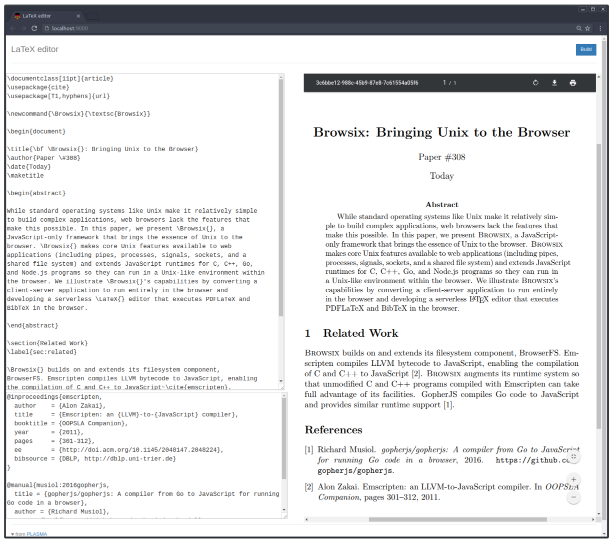
Starting with our in-browser meme server from the overview (§2), we modify the client-side of the application to dynamically route requests to either the in-BROWSIX server or a remote server running in the cloud. The policy that determines whether to route locally or remotely is an arbitrary JavaScript function. In our implementation, it takes into account network connectivity and browser version. If a network request fails due to connectivity issues, or if the browser identifies itself as a desktop browser (a proxy for being a powerful platform relative to mobile devices), the application switches to using the in-BROWSIX server instead of the remote server.

This modified MemeGenerator app works even if internet connectivity is disabled after the page has loaded, letting the application work in disconnected contexts. The code required to implement this policy and dynamic behavior amounted to less than 30 lines of JavaScript.

6.1.2 An In-Browser LaTeX Editor

We next demonstrate how BROWSIX lets us construct a web application that provides a serverless LaTeX editing environment with minimal effort. We use a BROWSIX-enhanced Emscripten toolchain to implement an in-browser LaTeX editor. We start by compiling pdflatex and bibtex from texlive-2015 to BROWSIX programs. Next, we populate a directory with a number of LaTeX data files, including fonts, class, style, and the `pdftex.map` font mapping, and produce an `index.json` directory index for BrowserFS. For the client, we built a simple split-pane user interface, with two text-boxes on the left-hand side, one for the LaTeX file and one for the bibliography, and a "build" button along with a preview of the PDF on the right-hand side, as seen in Figure 9.

When the user clicks the "build" button, the JavaScript application code writes the current contents of the text fields to the BROWSIX file system. It then executes four commands in series, `pdflatex main; bibtex main; pdflatex main; pdflatex main`, which results in either the creation of a PDF file or an error. If creating the PDF was successful, the editor reads the contents off the filesystem, creates a `Blob` object corresponding to the contents, and updates the preview to point to the newly generated PDF. The user has the option of downloading the generated PDF at any time. The application's logic, including starting and interacting with BROWSIX and handling user interaction were implemented in 79 lines of JavaScript in under an hour.



```
\documentclass[11pt]{article}
\usepackage{cite}
\usepackage[T1,hyphens]{url}

\newcommand{\Browsix}{\textsc{Browsix}}

\begin{document}

\title{\bf \Browsix{}: Bringing Unix to the Browser}
\author{Paper \#308}
\date{Today}
\maketitle

\begin{abstract}

While standard operating systems like Unix make it relatively simple
to build complex applications, web browsers lack the features that
make this possible. In this paper, we present \Browsix{}, a
JavaScript-only framework that brings the essence of Unix to the
browser. \Browsix{} makes core Unix features available to web
applications (including pipes, processes, signals, sockets, and a
shared file system) and extends JavaScript runtimes for C, C++, Go,
and Node.js programs so they can run in a Unix-like environment within
the browser. We illustrate \Browsix{}'s capabilities by converting a
client-server application to run entirely in the browser and
developing a serverless \LaTeX{} editor that executes PDFLaTeX and
BibTeX in the browser.

\end{abstract}

\section{Related Work}
\label{sec:related}

\Browsix{} builds on and extends its filesystem component,
BrowserFS. Emscripten compiles LLVM bytecode to JavaScript, enabling
the compilation of C and C++ to JavaScript~\cite{emscripten}.
```

```
@inproceedings{emscripten,
  author    = {Alon Zakai},
  title     = {Emscripten: an {LLVM}-to-{JavaScript} compiler},
  booktitle = {OOPSLA Companion},
  year      = {2011},
  pages     = {301-312},
  ee        = {http://doi.acm.org/10.1145/2048147.2048224},
  bibsource = {DBLP, http://dblp.uni-trier.de}
}

@manual{musiol:2016gopherjs,
  title = {gopherjs/gopherjs: A compiler from Go to JavaScript for running
Go code in a browser},
  author = {Richard Musiol},
```

Browsix: Bringing Unix to the Browser

Paper #308

Today

Abstract

While standard operating systems like Unix make it relatively simple to build complex applications, web browsers lack the features that make this possible. In this paper, we present BROWSIX, a JavaScript-only framework that brings the essence of Unix to the browser. BROWSIX makes core Unix features available to web applications (including pipes, processes, signals, sockets, and a shared file system) and extends JavaScript runtimes for C, C++, Go, and Node.js programs so they can run in a Unix-like environment within the browser. We illustrate BROWSIX's capabilities by converting a client-server application to run entirely in the browser and developing a serverless LaTeX editor that executes PDFLaTeX and BibTeX in the browser.

1 Related Work

BROWSIX builds on and extends its filesystem component, BrowserFS. Emscripten compiles LLVM bytecode to JavaScript, enabling the compilation of C and C++ to JavaScript [2]. BROWSIX augments its runtime system so that unmodified C and C++ programs compiled with Emscripten can take full advantage of its facilities. GopherJS compiles Go code to JavaScript and provides similar runtime support [1].

References

[1] Richard Musiol. *gopherjs/gopherjs: A compiler from Go to JavaScr for running Go code in a browser*, 2016. `https://github.co gopherjs/gopherjs`.

[2] Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *OOPSL Companion*, pages 301–312, 2011.

Figure 9: An LaTeX editor built using BROWSIX. `pdflatex` and `bibtex` are run under BROWSIX, and the resulting PDF displayed with Chrome's built-in PDF viewer. This application runs entirely in the browser.

Summary: BROWSIX makes it trivial to execute native applications within the browser, enabling the rapid development of sophisticated web applications that can easily migrate server code into the browser, and that harness the functionality of large bodies of existing code bases.

6.2 Performance

We evaluate the performance overhead of BROWSIX on our case studies. All experiments were performed on a Thinkpad X1 Carbon with an Intel i7-5600U CPU and 8 GB of RAM, running Linux 4.5.3.

BROWSIX Terminal and Utilities: We test the performance of the BROWSIX terminal versus native and

Command	Native	Node.js	BROWSIX
`sha1sum`	0.002s	0.067s	0.189s
`ls`	0.001s	0.044s	0.108s

Figure 10: Execution time of utilities under BROWSIX, compared to the same utilities run under Node.js, and the native GNU/Linux utilities. `sha1sum` is run on `usr/bin/node`, and `ls` is run on `/usr/bin`. Running in JavaScript (with Node.js and BROWSIX) imposes most overhead; running in the BROWSIX environment adds roughly another $3\times$ overhead.

Node.js performance to tease apart the source of overheads. Figure 10 presents the results of running the same JavaScript utility is run both under BROWSIX, and on Linux under Node.js, and compared to the execution time of the corresponding utility written in C. Most of the overhead is due to JavaScript (the basis of Node.js and BROWSIX); running in the BROWSIX environment imposes roughly a $3\times$ overhead. Nonetheless, this performance (completion in under 200 milliseconds) is low enough that it should be generally acceptable to users.

Meme Generator: We compare the performance of meme generation when run in-browser to running as a native Go application on Linux (run in a local server). The in-BROWSIX takes approximately two seconds to generate a meme when running in the browser, versus 200 ms when running server-side. This inefficiency is primarily due to missing 64-bit integer primitives when numerical code is compiled to JavaScript with GopherJS; we expect future browsers to support native access to 64-bit integers, which we believe will lead the two versions to deliver roughly the same performance.

LaTeX Editor: While the LaTeX editor represents a sophisticated application, its performance overhead is currently excessive. For example, for the one page example shown in Figure 9, the the `pdflatex; bibtex; pdflatex; pdflatex;` sequence takes 400 milliseconds under Linux (as native executables), and 82 seconds under BROWSIX. We note that we have spent no time optimizing this application, which is currently entirely interpreted by the Emterpreter. By identifying and recompiling core CPU-bound functions that do not invoke system calls to `asm.js` (a compiler flag to Emscripten), it is likely that the gap between native and in-browser performance could be closed considerably. We leave this as future work.

Summary: BROWSIX's performance is primarily limited by the performance of underlying browser primitives (notably, the lack of native 64-bit longs). While some applications perform substantially slower, it can provide a usable level of performance for certain applications.

7 Discussion

The process of implementing BROWSIX has highlighted opportunities for improvement in the implementation and specification of Web Workers. We outline a number of optimizations and natural extensions that are generally useful, and would extend BROWSIX's reach.

Worker Priority Control: The parent of a Web Worker has no way to lower the priority of a created worker. As workers are implemented on top of OS threads, this concept maps cleanly onto OS-level priorities/niceness. Providing this facility would let web applications prevent a low-priority CPU-intensive worker from interfering with the main browser thread.

`postMessage()` Backpressure: Traditional operating systems attempt to prevent individual processes from affecting system stability in a number of ways. One of these is providing backpressure, wherein the process attempting to write to a pipe or socket is suspended (the system call blocks) until the other end of the pipe reads the data, or it can fit into a fixed size buffer. This approach prevents unbounded resource allocation in the kernel. In the browser, the `postMessage()` function can be called from a process an unbounded number of times and will eventually cause the browser to run out of allocatable memory.

Message Passing Performance: Message passing is three orders of magnitude slower than traditional system calls in the two browsers we evaluate, Chrome and Firefox. A more efficient message passing implementation would improve the performance of BROWSIX's system calls and other inter-process communication.

8 Related Work

In-Browser Execution Environments: BROWSIX significantly extends past efforts to bring traditional APIs and general-purpose languages to the browser; Table 1 provides a comparison. Doppio's focus is providing single-process POSIX abstractions [10]. BROWSIX builds on and

		File system	Socket clients	Socket servers	Processes	Pipes	Signals
ENVIRONMENTS	BROWSIX	✓	✓	✓	✓	✓	✓
	DOPPIO [10]	†	†				
	WebAssembly						
LANGUAGE RUNTIMES	Emscripten (C/C++)	†	†	†			
	GopherJS (Go)						
	BROWSIX + Emscripten	✓	✓	✓	✓	✓	✓
	BROWSIX + GopherJS	✓	✓	✓	✓	✓	✓

Table 1: Feature comparison of JavaScript execution environments and language runtimes for programs compiled to JavaScript. † indicates that the feature is only accessible by a single running process. BROWSIX provides multi-process support for all of its features.

extends its filesystem component, BrowserFS, to support multiple processes. Emscripten compiles LLVM bytecode to JavaScript, enabling the compilation of C and C++ to JavaScript [11]; as Section 5 describes, BROWSIX augments its runtime system so that unmodified C and C++ programs compiled with Emscripten can take full advantage of its facilities. BROWSIX provides similar runtime support for Go programs through GopherJS [9].

Kernel Design and OS Interfaces: BROWSIX most closely resembles the structure of Barrelfish, a many-core, heterogenous OS [3]. Like Barrelfish, BROWSIX implements inter-domain system calls as asynchronous messages on top of channel primitives and requires user-level thread scheduling. In addition, BROWSIX mirrors the per-core, shared-nothing structure of a multikernel because its processes do not use inter-domain communication for tasks such as memory allocation and timers.

Migration from Server to Browser: One of our case studies migrates a meme generation server into the browser, and uses a policy to decide if meme requests should be handled locally or in the cloud. In keeping with industry best-practices, this server is state-free to avoid the complications that arise with state management [8], but there are cases where a stateful server is required. BROWSIX could take advantage of existing research on local/remote state management. Tango introduced flip-flop relocation, where an Android application runs locally as well as on the cloud, with the leader dynamically switching between the two instances [6]. In addition, BROWSIX could adopt a more sophisticated migration policy; systems like CloneCloud employ static analysis and dynamic profiling to decide how to partition computation between local and remote systems [4].

9 Conclusion

This paper introduces BROWSIX, a framework that brings the essence of Unix to the browser. BROWSIX makes processes, pipes, sockets, a shared file system, and a shell available to web applications on top of existing browser APIs. In addition, BROWSIX provides runtime support for a variety of languages and systems that can compile to or are written in JavaScript, including C, C++, Go, and Node.js. BROWSIX makes it almost trivial to build complex web applications from components written in a variety of languages without modifying any code, and promises to significantly reduce the effort required to build highly sophisticated web applications. BROWSIX is open source, and is freely available at `github.com/plasma-umass/browsix`.

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] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2016.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[4] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 301–314, Apr. 2011.

[5] M. Fogleman. *fogleman/gg: Go Graphics - 2D rendering in Go with a simple API*, 2016. `https://github.com/fogleman/gg`.

[6] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 137–150. ACM, 2015.

[7] L. T. Hansen and J. Fairbank. *ECMAScript Shared Memory and Atomics*, 2016. `https://tc39.github.io/ecmascript_sharedmem/shmem.html`.

[8] T. Mauro. *Adopting Microservices at Netflix: Lessons for Architectural Design*, 2015. `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`.

[9] R. Musiol. *gopherjs/gopherjs: A compiler from Go to JavaScript for running Go code in a browser*, 2016. `https://github.com/gopherjs/gopherjs`.

[10] J. Vilk and E. D. Berger. DOPPIO: Breaking the browser language barrier. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014)*, pages 508–518. ACM, 2014.

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