Emscripten vs Native Client: A Case Study

Aaron Dodson

Geoffrey Douglas

Professor Chandra Krintz

CS 263: Programming Language Implementations

Abstract

Although they have only been around for a relatively short period of time, web applications account for a large percentage of the work done today by computer programmers and software engineers. Having an application that runs inside of an Web browser allows an application developer to distribute his or her work widely across the world. Web applications have evolved to provide a wide range of services, from online banking, to social networking, cloud storage, and games. It is no wonder that a huge effort has been and continues to be made to make web applications easier to create. These efforts have largely taken two forms, the most common of which is making development more accessible and abstracting away details to make it quicker and easier to develop new applications. At the same time, however, tools are being created that allow pre-existing code to be migrated to the web. Two such pieces of technology that have been developed with this goal in mind are Google's Native Client (NaCl) and Emscripten. Each technology delivers the same promise: being able to compile and run code written in for local systems on the web. However, these technologies accomplish this task in very different manners. In this paper, we describe how each technology achieves its aims and compare their relative effectiveness.

Introduction

Google’s Native Client (henceforth known as NaCl) and Emscripten, developed by the Mozilla Foundation, both aim to provide a way to run native code written in C or C++ on the web, in the interest of improving portability, performance, and allowing existing legacy codebases to be migrated to the web. Despite their similar purpose, the two projects take dramatically different approaches: NaCl is implemented as a sandboxed browser plugin that directly executes compiled native code on the user's machine, whereas Emscripten cross-compiles LLVM bitcode, usually generated from C or C++ source code, into JavaScript, which is executed by the web browser. Each approach has its own merits and drawbacks: NaCl provides better performance, but at the cost of reimplementing a sandbox model (and associated potential security issues) and requiring support to be explicitly added by browser developers, whereas Emscripten incurs a performance penalty but allows code to be run in all major browsers.

Native Client: An Overview

Google's Native Client, as has been noted, centers around a browser plugin, which as of this writing is supported only by Chrome. The plugin executes compiled C or C++ code directly on the user's machine within a sandboxed environment to minimize the security risk inherent in running untrusted code. To further address this problem, the technology is supported by default only for applications distributed through the Chrome Web Store. Additionally, the Native Client distribution includes the Pepper SDK which provides additional features, including libraries, runtime support, and facilities for communicating between native code and JavaScript running in the browser.

Since NaCl executes native code, there must be native code present for it to execute. When NaCl modules are compiled, binaries for all desired architectures (x86, x86-64, and ARM) are produced, and the browser plugin loads and executes the required architecture. As a result, systems with different architectures are unable to execute NaCl modules, even if the browser supports the technology.

A Native Client project consists of three main elements: the code to be compiled natively (C/C++ source files); the HTML/JavaScript/CSS files for setting up and controlling the web page; and a manifest file for specifying the various architectures supported by the project. The key to using the native compiled code within the web application is in two special classes that must be included in the source files: the Module class, which represents the compiled Native Client executable; and the Instance class, which represents an element on a web page. Every Native Client project has only one module, but that module can load as many instances as are specified by the programmer. Instances are created and added to a web page by including an `<embed>` tag in the HTML file(s). The `<embed>` tag specifies the name of the module and the name of the manifest file, so the web browser knows where to load the module from to create a new instance. A sample `<embed>` tag is shown below:

```
<embed name="nacl_module" id="pngcrush" width=0 height=0
src="pngcrush.nmf" type="application/x-nacl"/>
```

Note that the height and width of a Native Client instance can be 0, meaning that the instance itself has no physical appearance on the web page while still allowing for communication between JavaScript and native code. The embedded module can post messages back to an instance, which receives the message in the compiled code. The compiled code must have a method to handle the incoming message so that it may process the message, and optionally post its own message back to the JavaScript running in the browser. Similarly, the JavaScript should provide a function that handles incoming messages for processing. This communication process between the JavaScript and C/C++ code is asynchronous, meaning that when one posts a message to the other, it continues to do work, rather than waiting for a reply.

Emscripten: An Overview

Unlike Native Client, which requires a browser plugin to execute specially compiled modules, Emscripten takes the approach of providing a compiler targeting JavaScript. Rather than taking source files as input, it accepts LLVM bitcode, which can be produced from any language supported by the `clang` compiler. In practice, Emscripten too is mostly limited to C and C++, but can run other languages by cross-compiling their runtimes or interpreters that are themselves written in C/C++. This design decision makes it somewhat more flexible in terms of languages that can be ported to the web, and also allows code ported using Emscripten to run in any modern browser without needing to install plugins or other support software.

In addition to the cross compiler, Emscripten also provides JavaScript implementations of the C and C++ standard libraries, SDL (using WebGL for drawing), a filesystem API, and an API to facilitate calling into and out of external and cross-compiled JavaScript.

An Emscripten project does not require a manifest file or any special file layout; the distribution includes drop-in wrappers for `gcc`, `make`, and other standard tools, which can in theory simply be replaced in a standard Makefile to port projects. Output can be produced as either a standalone JavaScript file, or as an HTML page including the JavaScript along with text areas and canvas elements to display program output. The resulting JavaScript can also be run off the web in standalone JavaScript engines, such as node.js and Spidermonkey.

Communication between code ported using Emscripten and external JavaScript is comparatively straightforward, since ported functions are compiled to Javascript functions, which can freely interoperate with external code. Emscripten also provides an API to make calling these functions somewhat cleaner, including support for passing standard Javascript datatypes as arguments.

Porting PNGCrush with Emscripten

We opted to attempt to port PNGCrush with Emscripten before NaCl, largely as a result of the Emscripten project being somewhat better documented. After downloading and setting up Emscripten, the PNGCrush source was extracted. The compilation process required several steps: first, the PNGCrush source files were compiled to LLVM bitcode using `emcc`, the Emscripten compiler. This produced a single linked bitcode file, which was then compiled into JavaScript, again using `emcc`. To improve performance, this step was performed at optimization level `-O2`. However, after some trial and error, it was determined that runtime errors in the resulting JavaScript were the result of over-aggressive optimization by the Closure compiler, which is invoked by Emscripten at high optimization levels. To work around this problem, the `--closure 0` flag was added to the `emcc` invocation. At this point, the resulting program was able to be successfully executed by node.js. Unfortunately, since it was unable to read the input file, it simply printed usage instructions and exited.

To make the ported PNGCrush truly useful, it was necessary to allow it to read and write input and output files. This required the use of Emscripten's JavaScript filesystem API. This step too was hampered by scarce documentation, the author's lack of familiarity with JavaScript, and the bleeding-edge nature of some required web technologies. Since the goal was to integrate PNGCrush into a web application, file uploads were provided using the HTML 5 drag and drop APIs, and the corresponding JavaScript interface. These APIs provided the file contents as a base 64-encoded data URL, which had to be converted to a byte array to be used as input to a call to Emscripten's `FS.createDataFile` method. A similar call was used to create an empty output file for PNGCrush to write to. In addition to the API, Emscripten also provided compatible implementations of the C standard library functions `fopen`, `fread`, `fwrite` and `fclose`, so the original PNGCrush code did not require changes to use these shim files. With this framework in place, PNGCrush was able to read and write files.

Since PNGCrush was now being executed in a browser, as opposed to in node.js, it was necessary to invoke the `main()` method from other JavaScript code. Although Emscripten provided an API (`Module.ccall`) to call functions in the cross-compiled code, this function only supported basic JavaScript data types, which proved to be a problem for the `char *argv[]` expected by PNGCrush's `main` method. Instead, this data had to be manually constructed, included padding, and then passed to the JavaScript `main` function. Fortunately, this was sufficient to provide the basic functionality: PNGCrush was able to be invoked, read an input file, and write the compressed output to another file.

For the finishing touches, some communication between the cross-compiled PNGCrush code and the JavaScript that drives the web app interface was required. Emscripten provided a function, `emscripten_run_script`, which could be called from PNGCrush's C source code to evaluate JavaScript. However, this required the JavaScript to be compiled in, and generally proved undesirable. Instead, we took advantage of Emscripten's ability to compile in JavaScript libraries to allow more modular code. Again, this was poorly documented; the JavaScript to be included had to be wrapped into a call to `mergeInto(LibraryManager.library)`, saved in a file, and the file included in the final compilation step using the `--js-library` flag. This allowed JavaScript functions to be called from PNGCrush when compression had finished (notifying the web app to download the output file) and while it was ongoing (to update the progress bar).

Since communication between the original and external code is done entirely through JavaScript, updating the user interface and keeping it responsive proved challenging, since all processing was done on a single thread. Due to inter-file dependencies and the complicated nature of the generated JavaScript produced by Emscripten, running the PNGCrush task in a web worker background thread proved infeasible. Additionally, since web workers are not allowed to modify the DOM, updating the interface using them was also impossible. Ultimately, this issue was not resolved – the entire web app simply blocks while PNGCrush is running, and updates to the progress bar do not appear until it is finished.

Although a poor user experience, the web app was functional at this point, and the port can reasonably be deemed successful.

Porting PNGCrush with Native Client

Getting started with the Native Client port was somewhat easier, since some difficulties had already been confronted and resolved during the Emscripten port. After downloading and installing the NaCl SDK, the decision was made to base the port off of the `hello_nacl_io` example included in the SDK distribution, on the theory that dealing with file I/O was likely to be the most complex and involved aspect of the port. The example provided a good starting point, including a makefile and sample code that used the C standard library functions for interacting with files. Although most documentation about Native Client suggests that I/O is limited to an asynchronous model incompatible with the blocking C functions, a blocking wrapper was recently integrated into the standard SDK distribution that allows these functions to work normally, much like the Emscripten port. The example included a module that handled message sending and receiving, as well as corresponding JavaScript code to set up the environment and send and receive messages from the web app. After modifying the makefile to include the PNGCrush source files, the various NaCl modules were successfully built for all three architectures, and did not require code changes.

Although the Native Client Pepper SDK provides its own file system, much like Emscripten, the new blocking model is based on the HTML 5 filesystem API. This proved rather easy to integrate into the pre-existing web app: the base 64-encoded data URL containing the uploaded image was converted into a Blob, and the filesystem API was used to write it to local storage. This file was then accessible to PNGCrush through `fopen` and and `fread` at a fixed path, and the output file written by PNGCrush was accessible to the web app through the same HTML 5 filesystem API, where it was read back as a blob and downloaded to the user's system.

Where Emscripten required complicated arguments to be manually constructed in order to call PNGCrush's `main` method and begin execution, this was comparatively simple in the Native Client port: a call to main was simply included in the C++ Native Client module file, with the appropriate arguments.

Surprisingly, interacting between the native code and web app JavaScript was easier with Native

Client, where communication is managed by message passing across language and runtime boundaries, than in Emscripten, where the entire system was JavaScript. A few calls were made in the main loop in the PNGCrush source code and at the end of the `main` function to different methods in the Native Client module, which passed messages asynchronously to the web app JavaScript. This did not require including additional files or any special syntax.

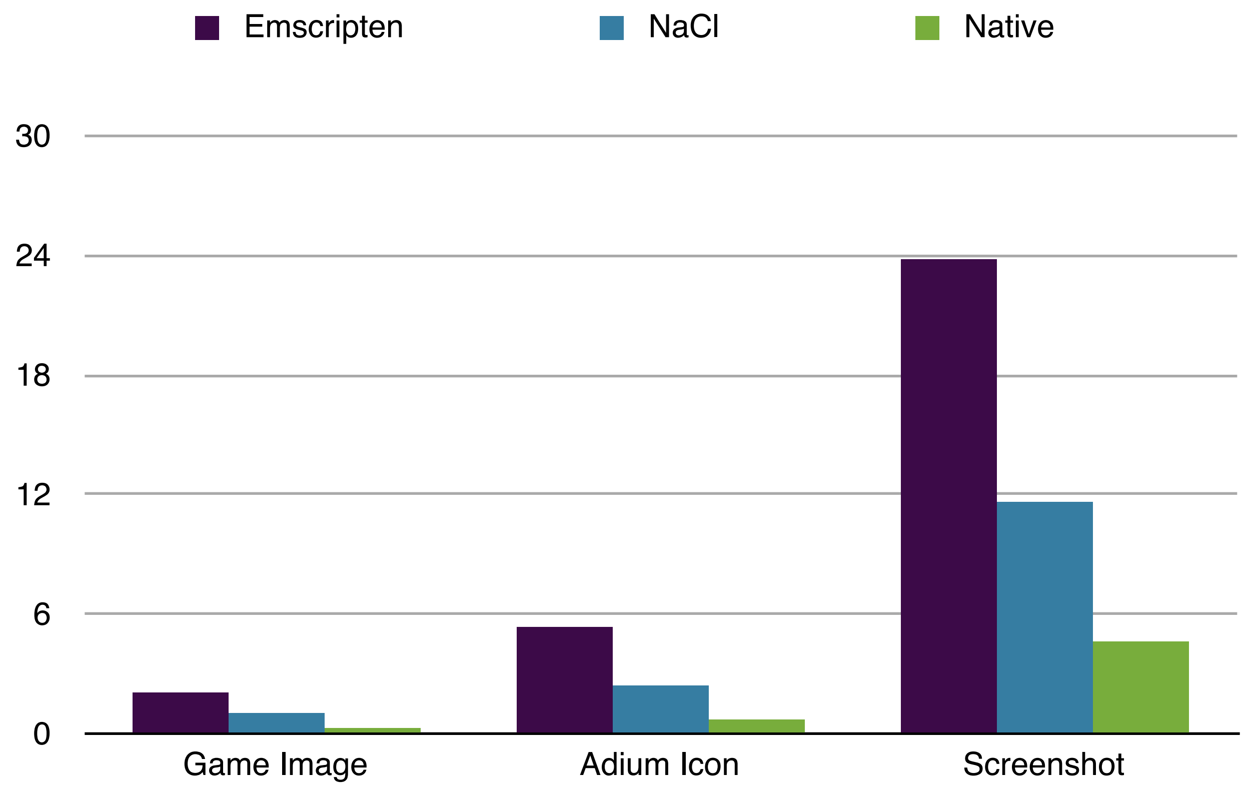
Conveniently, since communication is asynchronous, the blocking problems encountered in the Emscripten port were not present with Native Client: the code responsible for most processing and computation was executed natively in a separate process by the Native Client sandbox, freeing up the JavaScript to receive messages and modify the interface, allowing the progress bar to be smoothly updated as the image was compressed in the background and keeping the entire web app responsive at all times. Once this was implemented, both ports reached feature parity.

Emscripten and Native Client Ports: A Postmortem

Overall, the primary complicating factor in both ports was a lack of documentation. Since the web app depended heavily on features from Emscripten, Native Client and HTML 5, all of which are under very rapid development, finding detailed information that was also up to date and accurate proved challenging throughout every step of the process. Once it was determined how to convert file representations, handle I/O and communicate between the ported code and supporting web app code, the ports proceeded smoothly – however, these three tasks were all difficult to complete.

Although PNGCrush is not a particularly complex code base, neither is it a trivial one – the distribution includes 27 source files comprised of over 30,000 lines of code focused on low-level operations on files, a domain that is not typically the strength of web-based programs. Despite this, both Emscripten and Native Client handled it with aplomb – the code required almost no changes to compile under either, and its semantics were correctly preserved, as evidenced by the ported program correctly reading and compressing PNG files with the same results as the original command line utility. As a result, it seems fair to state that both tools are effective, reasonably mature, and up to the task of porting a legacy codebase.

Unsurprisingly, performance was notably better with Native Client – although JavaScript engines have made immense gains in recent years, they are still outperformed by a comfortable margin by native code. Additionally, Native Client's asynchronous approach to communication allowed the UI to remain responsive, increasing perceived performance in addition to absolute performance. Testing the two web apps on three PNG files yielded the following results (execution times in seconds reported by PNGCrush):



For each test image, the Emscripten version took very close to twice the time of the Native Client version, which took between 2-3 times longer than the purely native command line version. Although these results may seem surprising, since the Native Client version is executing native code, the plugin's sandbox likely accounts for the additional overhead. Overall, the performance of both methods was very reasonable, although the Native Client implementation's responsive interface made it seem much more responsive in practice.

Conclusion

Ultimately, the choice between Native Client and Emscripten for porting native code to the web comes down to personal preference and the set of tradeoffs best suited to an individual project.

Emscripten allows code to be run in any browser, but Native Client’s performance benefits may be worthwhile for “in house” applications where requiring a specific browser and plugin is not a problem. Both systems are capable of porting code accurately, provide a basic interface for dealing with I/O, and have options for communicating with external code. Native Client’s asynchronous message passing model can give better responsiveness, but Emscripten’s pure JavaScript communication may be more familiar and easier to work with for developers familiar with web technologies.