Halophile: Comparing PNacl to Other Web Technologies

Sable Technical Report No. 2015-01

Lei Lopez

Last updated: April 2015

Contents

Abstract

Most modern web applications are written in JavaScript. However, the demand for web applications that require more numerically-intensive calculations, such as 3D gaming or photo-editing, has increased. This has also increased the demand for code that runs near native speeds. PNaCl is a toolchain that allows native C/C++ code to be run in the browser. This paper provides a comparison of the performance of PNaCl to native code and JavaScript. Using a benchmark suite that covers a representative set of numerical computations, it is shown on average, that the performance PNaCl is within 9% of native C code.

1 Introduction

The web browser has become one of the most popular interfaces to interact with software, due to its ubiquity, ease of access, and updatability. Web applications can easily be used by the millions of people who already have access to browsers. These web applications have come a long way from simple scripts to change HTML elements. Today, web browsers must be able to handle everything from personalizing text on a webpage to computationally-intensive applications such as video-editors and physics simulators.

There have been many types of web technologies used to support more sophisticated web applications, such as Java applets [5], Adobe Flash [6], and Microsoft Silverlight. However, these all require the user to download the supporting technology which can deter many potential users.

Other approaches aim to enhance JavaScript performance, such as development of more complex virtual machines and the widespread usage of JIT compilers over interpreters [18].

Another approach has been to compile native code to JavaScript that can be run in the browser. For example, Emscripten [2] is a compiler that translates LLVM bytecode to asm.js [1] a subset of JavaScript that is better suited to JIT optimization. This allows applications that are written in C/C++ to be run in the browser.

A study of the performance of the above approaches for computationally-intensive numerical applications was published by Khan et al. [20] Missing from this study, however, is another approach to run native code in the browser: Google's Portable Native Client (PNaCl) [4] PNaCl is a toolchain that compiles C/C++ code to an intermediate representation that is stored on the server and is compiled to native code when the application is loaded in the browser.

This paper presents two main contributions: (1) The addition of Halophile, a new set of benchmark implementations to Ostrich, a benchmark suite that implements a representative set of typical numerical computing algorithms. (2) An inspection of the performance of PNaCl compared to native code and JavaScript, showing that PNaCl can run code at speeds close to native C code.

We will continue as follows: Section 2 gives an overview of the main technologies studied and tools used, including PNaCl and Ostrich. Section 3 describes the methodology and machines used for this performance study. Section 4 covers the performance results. Section 5 offers related work in the area of running native code in the browser. Finally, we finish with conclusions and future work in Section 6.

2 Background

This section provides the essential background to the technologies and benchmark suite referred to in this paper. We'll begin with an overview of PNaCl and move on to the Ostrich benchmark suite.

2.1 PNaCl

Portable Native Client is an extension of Native Client [9], which is a secure sandbox that allows native code to be run in the browser. Many applications have already been ported to Native Client, such as the Unity Game Engine [3].

The main difference between NaCl and PNaCl is that NaCl compiles C/C++ code based on the user's architecture, whereas PNaCl improves on this by compiling to an intermediate representation that is architecture independent. This means it only takes one compilation to allow the application to run on different machines.

This intermediate representation is a portable executable (or pexe) that can be saved on a server with other common web application file types, such as HTML, CSS, or JavaScript files. At runtime, when the browser is loading the application, it sees the pexe file and translates it to native code (called a nexe) on the user's machine, which can then be used by the application. The nexe is the same format as what would be generated through the NaCl toolchain and is run in the Native Client sandbox within the browser.

Since PNaCl aims to be a secure technology, the Native Client sandbox enforces various contraints on the code that it runs, such as only allowing certain API calls. In addition, some static analysis is done on the code via the NaCl validator to constrain permitted code to certain safe patterns.

Currently, the pexe to nexe translator and NaCl sandbox are only available in Google Chrome [8].

2.2 Ostrich

Ostrich is a benchmark suite that was created by members of Sable lab to test the performance of web technologies in the field of numerical computing [20]. It was inspired by a team at Berkeley working off Colella's work in identifying common patterns of numerical computation. Colella originally described seven patterns, and named them dwarfs [10]. Another team of researchers have subsequently added six more patterns, bringing the total to 13 [12].

An example of a dwarf is the Graph Traversal dwarf, which covers applications that traverse many objects, sometimes inspecting their contents, but not doing a lot of computation. The benchmark that implements this is the breadth-first search algorithm, also known as `bfs`.

Ostrich previously had implementations for 12 of the 13 dwarfs. Each benchmark was implemented in C, JavaScript, OpenCL, and WebCL. Some of these benchmarks were taken from other benchmark suites, namely from Rodinia [15] and OpenDwarfs [16], but the rest were first implemented in Ostrich.

The Ostrich results showed that JavaScript performance was within a factor of two compared to native C code.

Table I shows all the dwarfs and their corresponding benchmarks in Ostrich.

Dwarf	Benchmark
Branch and Bound	*nqueens*
Combinational Logic	*crc*
Dense Linear Algebra	*lud*
Dynamic Programming	*nw*
Graph Traversal	*bfs*
Graphical Models	*hmm*
MapReduce	*page-rank*
N-Body Methods	*lavamd*
Sparse Linear Algebra	*spmv*
Spectral Methods	*fft*
Structured Grid	*srad*
Unstructured Grid	*back-prop*

Table I: Dwarfs and their Ostrich implementation.

2.2.1 PNaCl Integration

A major task in Halophile was to adjust the Ostrich benchmarks to properly integrate with the PNaCl toolchain. The first order of business was to update the mostly C benchmarks to compile with a C++ compiler, as this is what is used the the PNaCl toolchain. This toolchain also forced us to fix many warnings due to its stricter build rules. We also took the time to update the benchmarks so they all interface with PNaCl uniformly.

After adjusting the benchmarks, the next step was to write PNaCl modules, which provided the connection between the benchmarks and the browser.

Once we were able to get the benchmarks to compile and run, we made sure to check the correctness of the benchmarks. This was especially important after the necessary code cleanup in the first step of PNaCl integration. Finally, the infrastructure to automatically run the Ostrich suite was updated to include PNaCl.

3 Methodology

This section dicusses pertinent research questions, the experimental setup for collecting data, and finally, how the measurements were taken.

3.1 Research Questions

Halophile focuses on the performance of PNaCl, which is supposed to offer a faster alternative for developers to write their web applications in, versus writing them in JavaScript. Thus, this brings us to our first question:

(*RQ1*) **Does PNaCl offer a performance improvement over hand-written JavaScript?**

PNaCl also claims performance near native code speed, despite its constraints, which leads to the second question:

(*RQ2*) **Is the performance of PNaCl competitive with C?**

3.2 Experimental Setup

The machine used for Halophile was a typical lab desktop, as is described in Table II.

Cheetah	
Processor	Intel Core i7 @3.2GHz x12
OS	64-bit Ubuntu 12.04
Physical Memory	16 GiB
Browser	Chrome 41
PNaCl API	Pepper 39
gcc	4.6.4

Table II: Specifications of lab machine.

All the tests were performed using the Ostrich benchmark suite (specifically the C and JavaScript implementations) that was updated to support PNaCl.

3.3 Measurements

Since Halophile is an investigation of how PNaCl compares to JavaScript and native code for typical numerical computations, both a JavaScript and a C performance baseline were necessary. Each benchmark for each technology was executed four times, with a script that collected only the execution time the core algorithm took to run and averaged them. The geometric mean of PNaCl against JavaScript or C was calculated from those collected times.

4 Results

This section discusses the results gathered from Halophile, which we use to answer the two research stated earlier. We begin by presenting a comparison of PNaCl versus JavaScript. This is followed by the comparison of PNaCl versus C.

4.1 PNaCl vs JavaScript

We compared PNaCl to JavaScript to ensure that using PNaCl really did offer a performance boost over hand-written JavaScript.

Figure 1 shows that for nine of the twelve benchmarks, PNaCl offers a performance boost over JavaScript. In some cases, such as for *fft* and *nw*, PNaCl offers more than 3x the speedup. Due to the lack of profiling tools for PNaCl, the reason for this is yet to be determined. *lavamd*, the only benchmark that did not offer a speedup, executes faster in JavaScript due to a faster (but less precise) exponentiation function that is used in Google Chrome [7]. The geometric mean shows that PNaCl, on average, offers a respectable 1.5x performance boost over JavaScript code. This provides an answer to our first research question, that PNaCl does offer a performance improvement over hand-written JavaScript.

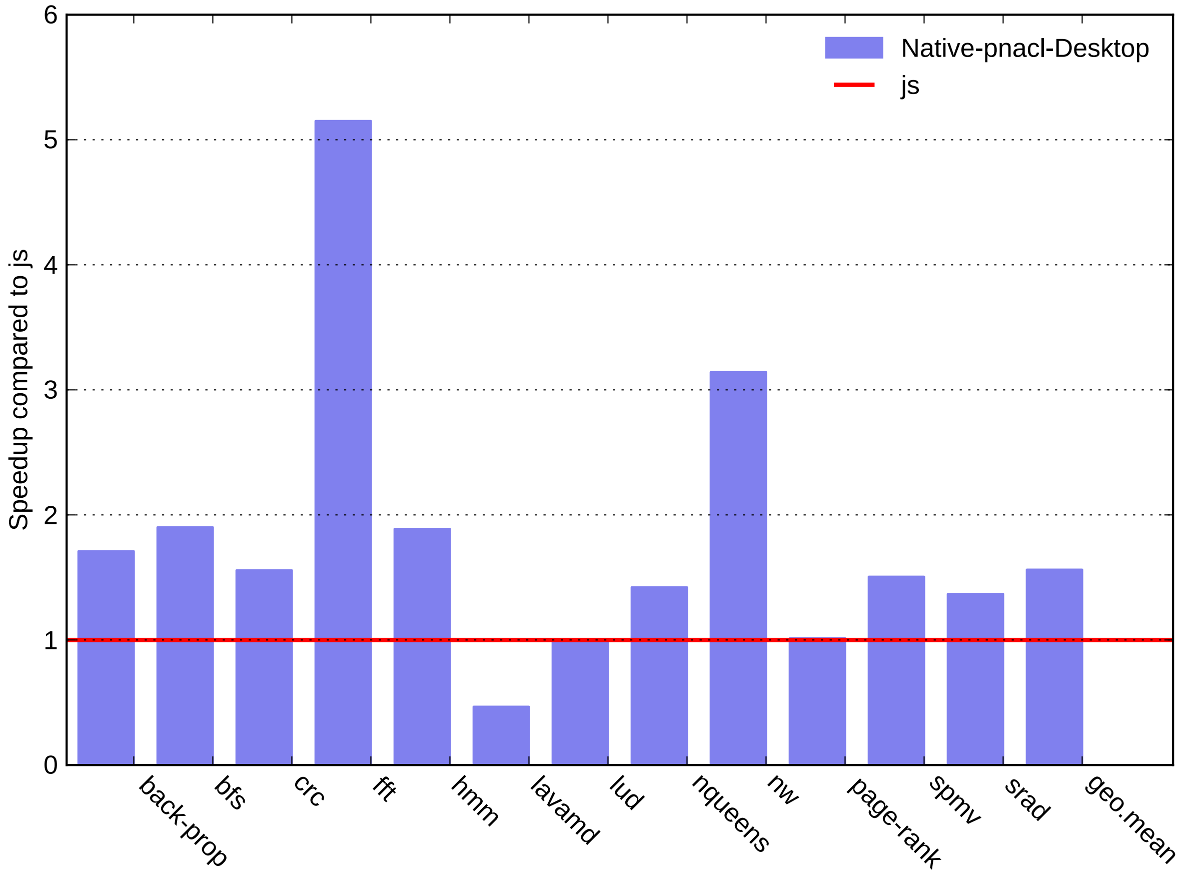

Figure 1: PNaCl ratios compared to JavaScript

4.2 PNaCl vs C

The comparison between PNaCl and C is of much greater interest, because the main goal of PNaCl is to offer near native code performance.

Examining the Figure 2, we can see that for most benchmarks, PNaCl is within 1.2x the speed of native code, or better. Surprisingly, there are four benchmarks that are faster than native code. However, like the PNaCl vs JavaScript experiment, the reasons for this are yet to be determined. The same goes for the three benchmarks that are around 1.5x slower than native code. All in all, PNaCl is competitive with C, with a geometric mean slowdown of around 9%. This answers our second research question: Is the performance of PNaCl competitive with C.

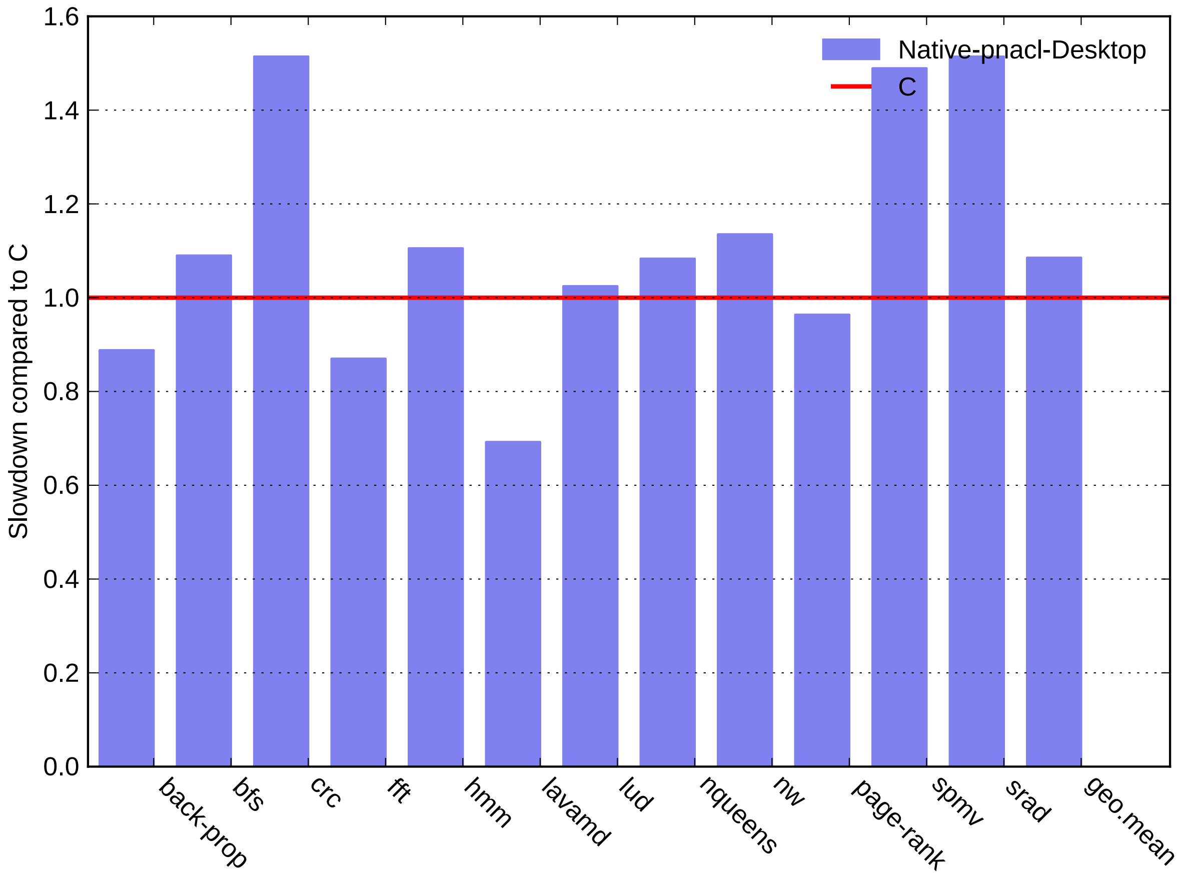

Figure 2: PNaCl ratios compared to C

4.3 Summary

This section has shown that C/C++ code compiled via PNaCl does offer a significant improvement in performance over JavaScript code. The comparison against C shows that PNaCl application can also run as fast as promised, around 9% the speed of native code.

5 Related Work

PNaCl is a technology that aims to run C/C++ code securely in the browser, not only offering alternatives to writing web applications in JavaScript, but also enabling legacy applications to distributed via the web. Halophile shows that PNaCl performs well for numerical applications.

To show that uses of C++ for numerical applications exist, we can look to the fields of biology [24], physics [19] [17], and math [11] for a few examples. Machine code emitted by PNaCl is run in a sandbox to ensure security. Other efforts in this area include MiniBox [21] for x86 Native Code and Robusta [26] for Java applications.

An alternate way of compiling native code to be run in the browser is Emscripten [29]. The usage of Emscripten in the fields of music [13] and games [23] interest in allowing legacy applications to be run in the browser. Another article [14] explores the differences between native and web applications, touching on the importance of performance. PNaCl also lines up with Google's plans concerning Chrome OS, as dicussed by Wright [28].

As mentioned in the introduction, one of the approaches to allowing computationally-intensive applications to run in the browser is to enhance JavaScript performance. This approach is used by Martinsen et al. [22] where they use thread-level speculation to allow JavaScript to make use of multicore processors. At the programmer-level, Souders [27] encourages best practices for programming in JavaScript with performance in mind. In terms of benchmarking JavaScript, this paper [25] explores the performance of real-web applications, comparing them to commonly used benchmark suites.

6 Conclusions and Future Work

This paper has presented Halophile, a PNaCl-friendly addition to Ostrich and confirmed the high performance of PNaCl for numerical computing benchmarks.

Using the infrastructure of Ostrich, we were able to successfully update the C/C++ implemenations to compile with PNaCl. With the improved set of benchmarks, we were able to show the average performance gain of around 50% when using PNaCl over JavaScript (*RQ1*), and that PNaCl is competitive with native C code (*RQ2*).

One of the next steps for Halophile would be to find a way to profile the PNaCl code. This way, we could understand where the performance gains and losses occur and analyze the code further.

Since PNaCl and asm.js offer two different ways to run native code in the browser, a more in-depth comparison of the two could yield interesting results in their strengths and weaknesses. Another path to be taken would be a look into PNaCl-level optimizations, that would occur before the native code would be generated. Similar to the paper [20] that inspired this project, we would also like to explore the ways that PNaCl supports concurrency, to see if gains could be made there.

References

[1] asm.js. `http://asmjs.org`.

[2] Emscripten - Mozilla — MDN. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Emscripten`.

[3] External Resource Directory. `http://www.chromium.org/nativeclient/reference/external-resource-directory/`.

[4] Introduction to Portable Native Client. `http://www.chromium.org/nativeclient/pnacl/introduction-to-portable-native-client/`.

[5] Lesson: Java applets. `https://docs.oracle.com/javase/tutorial/deployment/applet/`.

[6] Microsoft Silverlight. `http://www.microsoft.com/silverlight`.

[7] Profile your web application with v8's internal profiler. `https://developers.google.com/v8/profiler\_example/`.

[8] Technical overview. `https://developer.chrome.com/native-client/overview/`.

[9] Welcome to Native Client. `https://developer.chrome.com/native-client/`.

[10] Defining software requirements for scientific computing, 2004.

[11] K. Ahnert and M. Mulansky. Odeint - solving ordinary differential equations in C++. *CoRR*, abs/1110.3397, 2011.

[12] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[13] M. Borins. From Faust to web audio: Compiling Faust to JavaScript using Emscripten. In *Linux Audio Conference, Karlsruhe, Germany*, 2014.

[14] A. Charland and B. Leroux. Mobile application development: Web vs. native. *Commun. ACM*, 54(5):49–53, May 2011.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[16] W.-c. Feng, H. Lin, T. Scogland, and J. Zhang. OpenCL and the 13 Dwarfs: A work in progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 291–294, New York, NY, USA, 2012. ACM.

[17] G. Furnish. Container-free numerical algorithms in C++. *Computers in Physics*, 12(3):258–266, 1998.

[18] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

[19] H. Jasak, A. Jemcov, and Z. Tukovic. Openfoam: A C++ library for complex physics simulations. 2013.

[20] F. Khan, V. Foley-bourgon, S. Kathrotia, E. Lavoie, and L. Hendren. Using JavaScript and webCL for numerical computations: A comparative study of native and web technologies.

[21] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference*, 2014.

[22] J. Martinsen, H. Grahn, and A. Isberg. Using speculation to enhance javascript performance in web applications. *Internet Computing, IEEE*, 17(2):10–19, March 2013.

[23] C. McAnlis, P. Lubbers, B. Jones, D. Tebbs, A. Manzur, S. Bennett, F. dErfurth, B. Garcia, S. Lin, I. Popelyshev, J. Gauci, J. Howard, I. Ballantyne, J. Freeman, T. Kihira, T. Smith, D. Olmstead, J. McCutchan, C. Austin, and A. Pagella. HTML5 games in C++ with Emscripten. In *HTML5 Game Development Insights*, pages 283–298. Apress, 2014.

[24] G. R. Mirams, C. J. Arthurs, M. O. Bernabeu, R. Bordas, J. Cooper, A. Corrias, Y. Davit, S.-J. Dunn, A. G. Fletcher, D. G. Harvey, M. E. Marsh, J. M. Osborne, P. Pathmanathan, J. Pitt-Francis, J. Southern, N. Zemzemi, and D. J. Gavaghan. Chaste: An Open Source C++ Library for Computational Physiology and Biology. *PLoS Computational Biology*, 9:2970, Mar. 2013.

[25] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[26] J. Siefers, G. Tan, and G. Morrisett. Robusta: Taming the native beast of the JVM. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 201–211, New York, NY, USA, 2010. ACM.

[27] S. Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, Dec. 2008.

[28] A. Wright. Ready for a web OS? *Commun. ACM*, 52(12):16–17, Dec. 2009.

[29] A. Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, New York, NY, USA, 2011. ACM.