Improving C/C++ Open Source Software Discoverability by Utilizing Rust and Node.js Ecosystems

Kyriakos-Ioannis D. Kyriakou[1], Nikolaos D. Tselikas[1(✉)], and Georgia M. Kapitsaki[2]

[1] Communication Networks and Applications Laboratory, Department of Informatics and Telecommunications, University of Peloponnese, End of Karaiskaki Street, 22 100 Tripolis, Greece
{kyriakou,ntsel}@uop.gr

[2] Department of Computer Science, University of Cyprus, 75 Kallipoleos Street, P.O. Box 20537, 1678 Nicosia, Cyprus
gkapi@cs.ucy.ac.cy

Abstract. Discovering Open Source Software (OSS) components efficiently is not always an easy task. Node.js is a popular JavaScript runtime environment, whereas Rust is widely used for system programming, and both can be utilized for OSS discovery purposes. In this work, we examine whether Rust and Node.js can be used, along with their respective tooling and package repositories, in order to achieve improved discoverability of existing OSS implemented in C/C++. The paper describes how the capabilities of Rust in C/C++ interoperability can be combined with novel compilation techniques of low-level code to asm.js and WebAssembly, in order to harness JavaScript's popularity as the medium to publicize hard to discover C/C++ OSS. A proposed incremental methodology is presented and the main, as well as the collateral, effects of enforcing the proposed methodology in a proof-of-concept situation are examined. Our findings indicate potential increase in discoverability, code quality, portability, along with viable performance degradation of portable binaries, demonstrating 8.7 times slower execution compared to machine code, in a worst-case scenario.

Keywords: Free open source software · Software discoverability
Software performance evaluation · Software convergence
Software interoperability · C/C++ · WebAssembly · Node.js · Rust

1 Introduction

Node.js [1] is an open source JavaScript (JS) runtime built around V8 [2], the JS engine used in Chromium, the base for Google's Web browser. It has gained massive adoption by developers and organizations around the world, because of its ease of development, as well as the efficient, event-driven and non-blocking input/output (I/O) model. According to the results of the 2017 annual survey conducted by Stack Overflow, JS

© IFIP International Federation for Information Processing 2018
Published by Springer International Publishing AG 2018. All Rights Reserved
I. Stamelos et al. (Eds.): OSS 2018, IFIP AICT 525, pp. 181–192, 2018.
https://doi.org/10.1007/978-3-319-92375-8_15

has been declared as the most popular programming language for the fifth consecutive time [3]. The overall shifts in popularity throughout the years the survey has been conducted, displays both JS and Node.js as technologies with the greatest gain in traction among all popular choices; thus, representing the ubiquity of JS as a programming language of choice, in both server and client infrastructures. JS as part of the Web standard specifications, has enabled the production of complex applications that require no installation or upgrades, enable real-time communications, provide access to device-specific hardware and foster portability and accessibility, due to the prevalence of Web browsers.

One key advantage that can be attributed to the success of JS based systems is Node.js' package manager, `npm` [4], which houses the largest distribution of open source libraries in the world, counting over 570,000 individual modules [5]. Publicly available modules provide ease in the discovery of building blocks, that enable rapid prototyping of systems with minimal effort, through code reuse maximization [6].

Although Node.js applications are most commonly written in pure JS, the underlying interoperability with foreign compiled code is abstracted. According to Node.js' announcement for version 8.0.0 of the platform, 30% of all modules rely indirectly on native modules [7]. Node.js developers can provide their own bindings to C/C++ libraries, in order to extend the platform's capabilities and optimize performance, but in doing so, new challenges related to the application's integrity arise. Furthermore, re-purposing such modules to the Web platform has been largely impossible, due to the fact that the only recognized programming language in such environments is JS. Recent advancements have made compilation of lower-level languages to JS, or the WebAssembly portable binary format, possible. Whereas these novel technologies enable components to be written in C/C++, the long-studied challenges related to memory-safety and language misuse, propagate towards the higher layer, where the presumed safe JS code executes. In addition, the absence of a modules' system and of a common mechanism to package, document, discover and distribute libraries written in C/C++ hinders the discovery of existing libraries.

Rust is a systems programming language designed to prevent common C/C++ pitfalls, while at the same time it incorporates contemporary development methods, encouraging collaboration in OSS. Having the above as starting points, we observed an opportunity to study whether Rust's capabilities of providing safe interfaces to existing libraries, can be combined with the proliferation of the npm ecosystem, in order to enhance the discoverability and reuse potential of open source projects, and evaluate the side effects of such a coupling. This work describes the process we have followed, in order to perform the above investigation, and how this approach can positively affect discoverability, as well as quality aspects.

The rest of the paper is structured as follows. Section 2 introduces the background of this work describing the current state of JavaScript and C/C++ OSS, along with open challenges. The main contribution of our work with its architecture are presented in Sect. 3, and evaluated and discussed in Sect. 4. Finally, Sect. 5 concludes the paper outlining directions of future work.

2 Background and Motivation

Translation of programs written in C/C++ for the Web has been a recent topic of interest in various fields of research. Compilation of audio tools has been examined by Letz et al. and Zbyszyński et al. [8, 9], whereas a distributed evolutionary algorithm has been investigated by Leclerc et al. [10]. Furthermore, the potential of using Rust instead of other systems programming languages is another emerging recent topic. Rust has been shown to produce efficient code for the implementation of garbage collectors reducing at the same time the programmer error surface [11, 12]. Combination of both is possible, and we were motivated to examine the application of these technologies in junction with modern development trends in OSS, in order to improve the state of C/C++ software discoverability. The rest of this section provides the background information in order to justify our thought process, as well as the technologies chosen.

2.1 JavaScript in Open Source Projects

JS is one of the most popular programming languages intended to facilitate interaction with the user in web applications [3]. More recently, it has been employed in server-side infrastructure for distributed services, in cross-platform applications targeting stationary and mobile users alike, but has also been used as a compilation target for a plethora of programming languages [13]. In our previous work, we have examined the aforementioned aspects of JS by implementing a full-featured high-performance cross-platform social application and distributed service, with only OSS components, and evaluated the benefits and implications of our choices [14–16]. This endeavor was only made possible due to the wide spectrum of focused OSS modules, easily accessible via the npm repository. The word “module” is used to describe building blocks, usually performing a single task, leading to composable, instead of monolithic, design patterns. According to Modulecounts [5], a service monitoring language for module repositories, npm averages 697 new modules/day, followed by Packagist (PHP) with 136 modules/day and Maven Central (Java) with 100 modules/day. It is noteworthy, that npm module submissions follow an exponential growth curve, clearly outpacing all the other repositories. Furthermore, TF Bissyandé et al. have studied 100,000 OSS projects hosted on GitHub [17]. Their findings exhibited that JS was the programming language that appeared the most frequently in multi-language projects. The Node.js platform is such a multi-language project, where its components are written in both JS and C/C++.

Node.js Architecture. The “Applications/Modules” space is where all JS project files reside. They may make either direct, or indirect use of precompiled foreign code. Additionally, required external dependencies, which are reused in the application’s logic, belong in this space as well. Although JS has positive aspects, Node.js has to rely upon compiled code to perform I/O operations [18]. The runtime’s garbage collector (GC) abstracts the, potentially error prone, manual dynamic memory management, but there is no thread-safety mechanism present when performing I/O, making memory related faults, and race conditions possible [19]. Furthermore, failure points may be present in the underlying foreign compiled code, propagating to the higher levels, and

leading to unexpected behaviors and faults. Node.js utilizes C/C++ libraries internally, in order to provide access to the operating system resources. Such libraries provide efficient solutions to all I/O related operations included as core functionalities. For instance, the libuv project provides event-loop and asynchronous I/O access [20]. Other examples of libraries used internally by Node.js are c-ares, zlib, and OpenSSL. The "C/C++ Bindings" space is where the functionality of such libraries is exposed via the core JS Application Programming Interface (API). Some examples in this space are the `os`, `fs`, `net` and `http` modules.

Addons in Node.js refer to libraries and their corresponding bindings, which are not included in the core modules [21]. They are usually written in C/C++, in order to extend Node.js' functionality, or provide performance gains, when a JS implementation is found to be lacking. For instance, μWebSockets is a popular WebSocket protocol implementation for Node.js, which out-performs all known pure JS implementations [22]. Moreover, libxmljs provides bindings to the popular XML parsing C library, libxml, fulfilling Node.js lack of XML support [23].

Although addons have capabilities to extend Node.js, they require knowledge of how the V8 engine works and their implementations must be fine-tuned, in order to avoid locking the main thread JS executes, duplication of memory allocations, data races, memory faults, etc. Another method of interfacing shared objects with Node.js is via the `ffi` module. It involves no elaborate setup, at the cost of highly reduced performance on high Input/Output systems. We theorize that writing and publishing Node.js bindings, may pose an opportunity for undiscoverable C/C++ OSS to receive exposure and be collaboratively improved, due to the massive reuse potential in Node.js projects. Unfortunately, their implementation is connected to a performance-productivity trade-off.

2.2 Challenges in C/C++ OSS

OSS implemented in the popular systems programming languages C and C++ predates the proliferation of cloud computing, which enabled OSS to flourish. There is evidence of the inherent inflexible codebase componentization in the amount of build systems available. Some well-known examples are CMake [24], qmake [25], SCons [26], and GYP [27], with the latter being used by Node.js. Legacy codebases that are not using such systems gradually degrade in maintainability, as observed by Dayani-Fard et al. [28]. In contrast, most prevalent programming languages in OSS implement some form of enforced conventions, as well as a queryable directory or repository, containing the corresponding metadata for every published project, e.g. via mvn for Java [29], npm for JS [4], gem for Ruby [30], pip for Python [31], etc. Those enforced conventions serve as guidelines to interact with code repositories, document, license, test, build, distribute, etc., features which may exist for C/C++ in the form of various third-party tools, but are incapable of providing the cohesion needed across OSS. The lack of advocated methodology in C/C++ OSS is more apparent in legacy projects, with many of them accessible only through manual pursuit via web search-engines. Downloading arbitrary dependencies by-hand, extracting, copying-and-pasting, figuring out the right compiler flags, are not uncommon practices.

Finally, inconsistent dynamic memory management is common in even mainstream utilities. For example, the GNU tool `ls`, has been known to leak memory, and is considered as a non-issue by its maintainers [32]. Although in its intended use, the operating system would handle the leak, using the library unknowingly of that issue in a persistent system, e.g. a server, would pose a threat to robustness. Moreover, lack of C/C++ programming experience may cause integer overflow/underflow leading to "undefined behavior". Type safety is not guaranteed by C/C++, and although programs may not exhibit type errors, undefined behaviors are incorporated in the standard specification, leading compilers to produce unspecified results and also to allow the program to do practically anything. A simple example of iterator invalidation, leading to undefined behavior is demonstrated in the following listing.

```
std::vector v;
v.push_back(MyObject);
for (auto x : v) {
  v.clear();
  x->whatever(); // results in undefined behavior
}
```

If the contents of a container that is being iterated over are destroyed, the program is led into undefined behavior. This is an example of a perfectly valid code from a C++ compiler's perspective, capable of halting the project using it. Such cases of undefined behavior have already been investigated in depth [33]. Memory safety is set at risk by null-pointer dereferences (`NULL` in C and `nullptr` in C++) that cause programs to crash, dangling pointers allowing access to heap allocated resources that have not lived as long as they had to, and buffer overruns allowing the program to access elements before the start or beyond the end of an array [34]. Malicious software has been taking advantage of the way C and C++ programs handle memory and exploiting bugs in the code. Hence, OSS discovered in the wild may propagate unwanted effects to derivative projects.

2.3 Rust: A Young Contender in OSS

Rust was created in order to address the challenges presented in Sect. 2.2. It follows the C++ philosophy of zero-cost abstractions and takes a step further, by incorporating memory-safety and data-race free concurrency without the need for a GC [35]. This is accomplished by statically tracking ownership and lifetimes of all variables and their references. The ownership system enables Rust to automatically deallocate and run destructors on all values immediately, when they go out of scope and prevents values from being accessed after they are destroyed. Rust applies some established techniques from academia, e.g. `enums` as algebraic data types, common in the ML family of languages, and `traits`, which enable polymorphism similar to Haskell's type classes. Similarly to JS, both procedural and functional paradigms are used, as examined by Poss [36].

Rust is available on GitHub, where all parts of the compiler and tooling are accessible for contributions [37]. The integrated command line application `cargo` serves as a complete project management tool. It is capable of instantiating new projects, building for various architectures, managing dependencies, testing, producing documentation, and more. Furthermore, `cargo` is responsible for enforcing the practices that enable OSS to be discoverable and maintainable. Interaction with C APIs is free of overheads, and the binaries produced can be called from C with no setup. As Rust utilizes LLVM for machine code emission, we were triggered to examine the possibility of utilizing this system for bridging the gap between C/C++ codebases and modern OSS development practices.

2.4 Compilation to JavaScript and WebAssembly

A strict subset of the JS programming language was designed as a compilation target, in order to allow for translation of programs written in other languages. It became known as asm.js, and the Emscripten compiler was created by Alon Zakai in 2011. This language is statically compiled and has shown near native performance [38]. One of the key benefits to its adoption was that even if a Web browser had not implemented optimizations for the subset, it could still run on every JS interpreter. That is one reason it is still in use as a fallback from the newer WebAssembly specification [39]. WebAssembly is a portable size and load-time efficient format, suitable for compilation to the Web, implemented in all major Web browsers, but still under the process of standardization via the W3C WebAssembly Working Group. It features language and platform independence, safe execution, and has shown promising performance gains of up to 5.89 times when replacing JS components with Rust code in real-use parsing scenarios [40]. Currently not all features have been finalized and garbage collection, threads, SIMD, etc. are in progress. It is relevant to both JS and C/C++/Rust OSS, because the combination of those technologies may solve the portability issue in the dissemination of multi-language OSS.

3 Exposing C/C++ OSS via Rust and Node.js

In this section the independent processes that realize our proposed methodology are presented and discussed. Each process is incremental and not mandatory, but adds to a project's exposure. The high-level architecture of our proposal is presented in Fig. 1, followed by a proof-of-concept example.

3.1 Using Rust to Package and Publish on crates.io

The processes of taking the source files of a C/C++ project and producing a package for publishing are depicted in the upper-half part of Fig. 1. By issuing the `cargo new` command, followed by the name of the package to be published, a local git repository is initiated, and the Cargo.toml file holding the project's metadata and dependencies is produced. By adding the `cc`, and `bindgen` packages to the dependencies, the project is now capable of generating bindings statically via header files, compiling the source

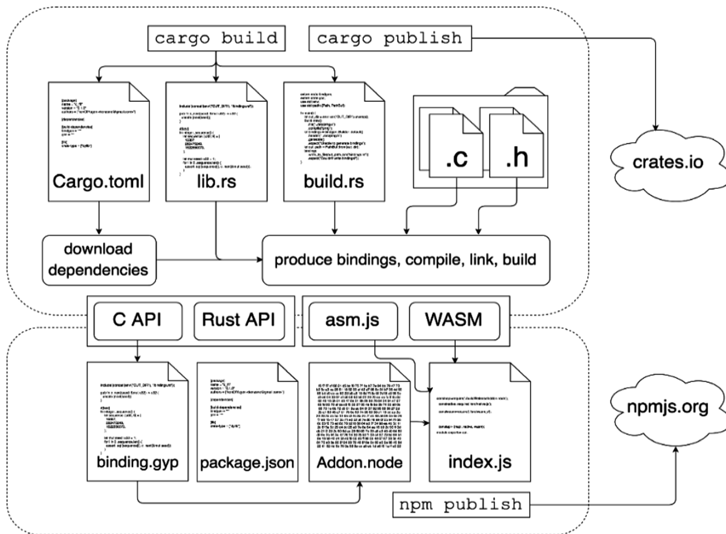

Fig. 1. The proposed high-level architecture

files and linking them automatically. By creating a build.rs file, where the build parameters are specified, the process is complete. The produced bindings can be included in lib.rs, where a new Rust API may be written. By issuing the `cargo publish` command, the package is uploaded to crates.io, the main Rust packages repository, and can be discovered by querying it. As C/C++ and Rust can all emit asm.js and WebAssembly files, portable executables may be built for reuse by JS environments.

3.2 Using Node.js to Package and Publish on npm

The second part of the process is depicted in the lower-half part of Fig. 1. Node.js uses a package.json file, to hold the project's dependencies and metadata. It can be generated by issuing the `npm init` command. The library can be used directly by utilizing the `ffi` module, but in order to create the more efficient Addon, the `nan` module or the official N-API dependencies can be used. A C/C++ bridge must be created to convert from C types to V8 types and provide a JS API. By instructing GYP via the binding.gyp file to use the bridge and link to the libraries produced by Rust, the Addon is created. Finally, by creating an index.js file which exports the Addon, the asm.js and WebAssembly, the process is complete. The package can be published on the npmjs.org repository by using the `npm publish` command.

3.3 The Hypothetical Park-Miller-Carta PRNG Case

For our proof of concept scenario, we hypothesize that a researcher is seeking an efficient Pseudo-Random Number Generator (PRNG) for a system's prototype. They are instructed by their colleague to use the Park-Miller-Carta PRNG.

We searched on GitHub for "Park Miller Carta PRNG", and at the time this paper was written, 1 result came up and it was a package implemented in JS. By searching the Web, we came across a page dedicated to the algorithm, including documented sources in assembly, C and C++ [41]. We proceeded to perform first step of the proposed process, and created a local Rust project repository with `cargo`, where we included the C/C++ source files.

The process of generating bindings automatically was successful and we then created a safe interface for the library in Rust. The first observation was that when interfacing with foreign code, the `unsafe` notation is constantly used. Its purpose is to mark the calls to foreign functions, raw pointer dereferences, access to global mutable variables, as well as inline assembly, the parts of the code the Rust compiler cannot provide guarantees for. In the case of this particular PRNG library, the seed-state is held in a global mutable variable. By writing tests to verify that the produced interface is implemented correctly, the second observation was that the foreign code is not thread-safe. Rust runs tests in parallel and, by adding the flag `--test-threads=1`, to force consecutive execution, the tests pass. As the C/C++ implementation would serve its intended use in 16bit microcontrollers, the original code was retained in a module named `ffi_unsafe`, and documentation was written along with inline tests-as-examples, to be part of the project's documentation. In this particular case, the library's focus is rather narrow; hence, it was trivial to replace the unsafe blocks by altering the way the state is stored and accessed. The product was an idiomatic Rust API, including a C compatible API, that have been documented and tested. The package was then pushed on GitHub and published on crates.io.

By utilizing the new C API, we compiled the library via emscripten to asm.js and WebAssembly exposed from a high-level idiomatic API in the index.js file, in order to be accessible to systems compatible with Node.js modules. The `ffi` module was used to dynamically link directly to the shared object produced by Rust. At this point, we setup a stress test to examine the potential performance degradation in each approach.

All versions were initialized with the seed 1. A for-loop cycled through each next random integer invocation for 10 million times, in order to warm-up Node.js and enable all the potential of Just-In-Time-Compilation. Then the `benchmark` library performed the evaluation of each version. For Rust, the included library Bencher was instructed to measure the execution time for 10 million integer generations, and was averaged over 10 repetitions. Table 1 contains the results in mean number of executions per second, along with the mean divergences recorded. The software and hardware specifications were the following: macOS 10.13, MacbookPro 2.3 GHz Intel Core i5, 4 GB 1333 MHz DDR3, node v8.9.1, rustc 1.25.0-nightly, clang 4.0.0, emcc 1.37.29.

Table 1. Mean executions per second and mean divergence

	Rust	ffi	WebAssembly	asm.js
Executions/second	219,159,372	180,622	25,214,517	7,969,921
Divergence	±4.89%	±1.8%	±1.78%	±1.05%

Thereafter, the ffi, WebAssembly and asm.js version, may be made available on npm for direct distribution and use in all JS environments by creating a package.json file and filling in the module's metadata.

4 Results and Discussion

By following the proposed methodology on the hypothetical, albeit pragmatic scenario, the following observations were made. The code and information related to this study are available on [42].

4.1 Discoverability Improvement

According to the popular Web metrics provider Alexa, the website hosting the examined Park-Miller-Carta PRNG implementation receives 1 page-view per day on average, with an unknown amount of those views resulting in downloads. After publishing the investigated asm.js/WebAssembly implementations derived from the type-checked code, as well as the Node.js `ffi` version, as a package on npm they received 161 downloads in a period of about two and a half months, which translates to about 2.15 downloads per day. Furthermore, the proof of concept package that was published on crates.io averaged 0.7 additional downloads per day, during the same period. The reported metrics suggest that re-packaged OSS according to our proposed methodology can improve the state of C/C++ codebase distribution and discovery, by multiplying the exposure to multiple repositories and providing a high-level API for easier engagement. In addition, larger codebases can be modularized into more manageable and maintainable components, and by publishing each one focused component, exposure improvements can be realized collectively.

4.2 Code Quality Control

By interfacing the foreign C/C++ code with Rust, an undocumented thread-safety weakness was discovered. An experienced programmer may have been able to realize this fragility by going through the code, but in more complex scenarios, and especially in the plane of OSS collaborations, code reviews alone cannot warrant the code's correctness. Wrapping error-prone code in Rust's unsafe blocks and documenting them, is a reasonable method to minimize the debugging surface. In addition, the process of improving the quality of the code can be performed incrementally and largely unfocused codebases can benefit from the concept of smaller components in the form of modules.

4.3 Performance Degradation

The scenario was deliberately chosen, in order to stress the performance of function invocation during context-switching interoperability and determine the overhead. Rust is able to call into C/C++ libraries without the associated overhead interpreted languages impose. Hence, calling the original PRNG library via the safe interface and auto-generated bindings, exhibited the same performance as the corrected Rust version.

Node.js-C shared object interoperability was examined via the ffi module. This choice was made in order to serve as a fair comparison against the asm.js and WebAssembly versions, as the cost is about equal in terms of programming time. Our measurements indicate that both asm.js and WebAssembly have superior performance, by a factor of 44 and 140 times respectively. Due to the fact that the time the program spends performing actual calculations, is much shorter than the time it spends switching contexts, the modules were operating at their weakest possible scenario. The V8 engine does not implement full optimizations for the asm.js subset, as it would have performed similarly to the WebAssembly otherwise. Still the minimum observed overhead by WebAssembly, while operating in a biased scenario against it, was found to impose about 8.7 times slower execution, compared to the standalone native library. The trade-off in performance vs productivity ratio appears to be improved by WebAssembly vs the more common ffi approach, as a single codebase can produce, at least, good-enough solutions for use in the most wide-spread platform, the Web.

4.4 Code Portability

The aspect of portability can be greatly improved by the proposed methodology. Asm.js is capable of executing in all JS interpreters, and with practically every system incorporating a Web browser, the coverage gains are immeasurable. WebAssembly, while still in its infancy, exhibits the same trait for current Web browsers and Node.js, but shows more future potential, with the on-progress features and its standardization. Either technology was found to be capable of realizing portable libraries from C/C++/ Rust codebases. Finally, the Emscripten compiler is capable of bridging the gap of code targeting the machine and the Web standards, and is expected be even more prominent in the future.

5 Conclusions and Future Work

In this paper, we proposed a methodology for converting existing C/C++ OSS to packages via Rust and Node.js, and publicizing them on the crates.io and npm repositories. This procedure takes into account the state of Node.js and C/C++ complexities, as well as the novel compilation to WebAssembly. Our evidence based on a realistic scenario suggests that discoverability, code quality and portability are improved, as well as the performance when compared to same cost time-wise existing alternative, all beneficial aspects to OSS. We plan to conduct a larger-scale study, and produce tooling to automate the process further. WebAssembly is still in a minimum-viable-product state, once it matures and the JS engines are optimized further, we plan to conduct research on the planned features, such as threading, which will enable more intensive C/C++ libraries to be converted through our proposed methodology.

References

1. Node.js. https://nodejs.org. Accessed 18 Jan 2018
2. V8 Repository. https://chromium.googlesource.com/v8/v8.git. Accessed 18 Jan 2018
3. Stack Overflow Survey 2017. https://insights.stackoverflow.com/survey/2017. Accessed 18 Jan 2018
4. npm. https://www.npmjs.com/. Accessed 18 Jan 2018
5. Modulecounts. http://www.modulecounts.com/. Accessed 18 Jan 2018
6. Sojer, M., Henkel, J.: Code reuse in open source software development: quantitative evidence, drivers, and impediments (2010)
7. Node.js 8: Big improvements for the debugging and native module ecosystem. https://medium.com/the-node-js-collection/node-js-8-big-improvements-for-the-debugging-and-native-module-ecosystem. Accessed 18 Jan 2018
8. Letz, S., Denoux, S., Orlarey, Y., Fober, D.: Faust audio DSP language in the Web. In: Proceedings of the Linux Audio Conference (LAC-15), Mainz, Germany, April 2015
9. Zbyszyński, M., Grierson, M., Fedden, L., Yee-King, M.: Write once run anywhere revisited: machine learning and audio tools in the browser with C++ and emscripten (2017)
10. Leclerc, G., Auerbach, J.E., Iacca, G., Floreano, D.: The seamless peer and cloud evolution framework. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016, pp. 821–828. ACM, July 2016
11. Lin, Y., Blackburn, S.M., Hosking, A.L., Norrish, M.: Rust as a language for high performance GC implementation. In: Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management, pp. 89–98. ACM, June 2016
12. Blanco-Cuaresma, S., Bolmont, E.: What can the programming language Rust do for astrophysics? Proc. Int. Astron. Union **12**(S325), 341–344 (2016)
13. List of Languages that compile to JS. https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js. Accessed 18 Jan 2018
14. Chaniotis, I.K., Kyriakou, K.-I.D., Tselikas, N.D.: Proximity: a real-time, location aware social web application built with Node.js and AngularJS. In: Daniel, F., Papadopoulos, G.A., Thiran, P. (eds.) MobiWIS 2013. LNCS, vol. 8093, pp. 292–295. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40276-0_23
15. Chaniotis, I.K., Kyriakou, K.I.D., Tselikas, N.D.: Is Node.js a viable option for building modern web applications? A performance evaluation study. Computing **97**(10), 1023–1044 (2015)
16. Kyriakou, K.I.D., Chaniotis, I.K., Tselikas, N.D.: The GPM meta-transcompiler: harmonizing JavaScript-oriented Web development with the upcoming ECMAScript 6 “Harmony” specification. In: 12th Annual IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas, NV, USA, pp. 176–181 (2015)
17. Bissyandé, T.F., Thung, F., Lo, D., Jiang, L., Réveillere, L.: Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In: 2013 IEEE 37th Annual Computer Software and Applications Conference (COMPSAC), pp. 303–312. IEEE, July 2013
18. Crockford, D.: JavaScript: The Good Parts: The Good Parts. O’Reilly Media, Inc., Sebastopol (2008)
19. Daloze, B., Marr, S., Bonetta, D., Mössenböck, H.: Efficient and thread-safe objects for dynamically-typed languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 642–659 (2016)
20. libuv. http://libuv.org/. Accessed 18 Jan 2018

21. Addons Node.js. https://nodejs.org/api/addons.html. Accessed 18 Jan 2018
22. µWebSockets. https://github.com/uNetworking/uWebSockets. Accessed 18 Jan 2018
23. libxml. https://github.com/libxmljs/libxmljs. Accessed 18 Jan 2018
24. CMake. https://cmake.org/. Accessed 18 Jan 2018
25. Qmake. http://doc.qt.io/qt-5/qmake-manual.html. Accessed 18 Jan 2018
26. SCons. http://scons.org/. Accessed 18 Jan 2018
27. GYP. https://gyp.gsrc.io/. Accessed 18 Jan 2018
28. Dayani-Fard, H., Yu, Y., Mylopoulos, J., Andritsos, P.: Improving the build architecture of legacy C/C++ software systems. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 96–110. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31984-9_8
29. Maven. https://maven.apache.org/. Accessed 18 Jan 2018
30. Ruby Gems. https://rubygems.org/. Accessed 18 Jan 2018
31. Pip Python. https://pypi.python.org/pypi/pip. Accessed 18 Jan 2018
32. bug#8755: "ls -l" leaks memory. https://lists.gnu.org/archive/html/bug-coreutils/2011-05/msg00062.html. Accessed 18 Jan 2018
33. Dietz, W., Li, P., Regehr, J., Adve, V.: Understanding integer overflow in C/C++. ACM Trans. Softw. Eng. Methodol. (TOSEM), **25**(1), article 2 (2015)
34. Tselikis, G.S., Tselikas, N.D.: C: From Theory to Practice, 2nd edn. CRC Press, Boca Raton (2017)
35. Stroustrup, B.: Abstraction and the C++ machine model. In: Wu, Z., Chen, C., Guo, M., Bu, J. (eds.) ICESS 2004. LNCS, vol. 3605, pp. 1–13. Springer, Heidelberg (2005). https://doi.org/10.1007/11535409_1
36. Poss, R.: Rust for functional programmers. arXiv preprint arXiv:1407.5670 (2014)
37. The Rust Programming Language. https://github.com/rust-lang. Accessed 18 Jan 2018
38. Zakai, A.: Emscripten: an LLVM-to-JavaScript compiler. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, pp. 301–312. ACM, October 2011
39. Rossberg, A.: WebAssembly: high speed at low cost for everyone. In: ML16: Proceedings of the 2016 ACM SIGPLAN Workshop on ML (2016)
40. Oxidizing Source Maps with Rust and WebAssembly. https://hacks.mozilla.org/2018/01/oxidizing-source-maps-with-rust-and-webassembly/. Accessed 26 Jan 2018
41. Park-Miller-Carta Pseudo-Random Number Generator. http://www.firstpr.com.au/dsp/rand31/. Accessed 18 Jan 2018
42. rust_node_wasm. https://github.com/kenOfYugen/rust_node_wasm.git. Accessed 26 Jan 2018