Insights into WebAssembly: Compilation Performance and Shared Code Caching in Node.js

Tobias Nießen
Faculty of Computer Science
University of New Brunswick
tniessen@unb.ca

Michael Dawson
IBM Runtime Technologies
IBM Canada
Michael_Dawson@ca.ibm.com

Panos Patros
Software Engineering
University of Waikato
panos.patros@waikato.ac.nz

Kenneth B. Kent
Faculty of Computer Science
University of New Brunswick
ken@unb.ca

ABSTRACT

Alongside JavaScript, V8 and Node.js have become essential components of contemporary web and cloud applications. With the addition of WebAssembly to the web, developers finally have a fast platform for performance-critical code. However, this addition also introduces new challenges to client and server applications. New application architectures, such as serverless computing, require instantaneous performance without long startup times. In this paper, we investigate the performance of WebAssembly compilation in V8 and Node.js, and present the design and implementation of a multi-process shared code cache for Node.js applications. We demonstrate how such a cache can significantly increase application performance, and reduce application startup time, CPU usage, and memory footprint.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Software performance**; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

WebAssembly, compiler, code cache, Node.js, V8, JavaScript

ACM Reference Format:
Tobias Nießen, Michael Dawson, Panos Patros, and Kenneth B. Kent. 2020. Insights into WebAssembly: Compilation Performance and Shared Code Caching in Node.js. In *Proceedings of 30th Annual International Conference on Computer Science and Software Engineering (CASCON'20).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

WebAssembly is a new hardware abstraction that aims to be faster than interpreted languages without sacrificing portability or security. Conceptually, WebAssembly is a virtual machine and binary-code format specification for a stack machine with separate, linearly addressable memory. However, unlike many virtual machines for high-level languages, the WebAssembly instruction set is closely related to actual instruction sets of modern processors, since the initial WebAssembly specification does not contain high-level concepts such as objects or garbage collection [8]. Because of the similarity of the WebAssembly instruction set to physical processor instruction sets, many existing "low-level" languages can already be compiled to WebAssembly, including C, C++, and Rust.

WebAssembly also features an interesting combination of security properties. By design, WebAssembly can only interact with its host environment through an application-specific interface. There is no built-in concept of system calls, but they can be implemented through explicitly imported functions. This allows the host to monitor and restrict all interaction between WebAssembly code and the host environment. Another important aspect is the concept of *linear memory*: Each WebAssembly instance can access memory through *linear memory*, a consecutive virtual address range that always begins at address zero. The host environment needs to translate virtual memory addresses into physical addresses on the host system, and ensure that virtual addresses do not exceed the allowed address range. On modern hardware, this can be implemented using the Memory Management Unit (MMU) and hardware memory protection features, leading to minimal overhead while allowing direct memory access to *linear memory* and preventing access to other memory segments of the host system [8]. Combined, these properties allow running the WebAssembly code both with full access to the real system, and in a completely isolated sandbox, without any changes to the code itself.

These properties make WebAssembly an attractive platform for performance-critical portable code, especially in web applications. However, WebAssembly makes no inherent assumptions about its host environment, and can be embedded in other contexts such as Node.js, a framework built on top of the JavaScript engine V8. Not only does Node.js share many technological aspects, such as the programming language JavaScript, with web applications, but its performance and portability goals align well with those of WebAssembly. Sandboxing, platform-independence, and high performance are especially relevant in cloud-based application backends, the primary domain of Node.js.

However, one hindrance remains: Because WebAssembly code is tailored towards a conceptual machine, it can either be interpreted on the host machine, or first be compiled into code for the actual host architecture. As we will see below, interpretation leads to

CASCON'20, November 10–13, 2020, Toronto, Canada

inadequate performance. At the same time, compilation of large WebAssembly modules can lead to considerable delays during the application's startup phase.

Node.js is often used in "serverless computing:" Instead of keeping a number of servers running at all times, the provider allocates resources dynamically with a minimal scope. For example, Function-as-a-Service (FAAS), sometimes referred to as "serverless functions," is a deployment model in which the provider only allocates enough resources for a single function to be executed for each request, and no internal state is preserved between requests [2]. In this scenario, it is crucial for the function to be executed quickly in order to produce the desired response to an incoming request with minimal latency, making it unfeasible to compile large WebAssembly modules on each request.

In this paper, we analyze the performance of V8's WebAssembly compilers. Sections 2 and 3 provide an overview of background and related work. We analyze the performance of code generated by V8 in Section 4. In Section 5, we investigate the performance of V8's WebAssembly compilers themselves, and the performance benefits of caching compiled code. Finally, in Section 6, we present and evaluate the design and implementation of a multi-process shared code cache for WebAssembly code in Node.js applications.

2 BACKGROUND

With the addition of WebAssembly to Node.js, there are three kinds of code that are supported: JavaScript, native addons, and WebAssembly. While JavaScript code and WebAssembly are interpreted and/or compiled by V8 (see Section 4), native addons behave like shared libraries and allow embedding "native" code, e.g., compiled C or C++ code. Unlike WebAssembly, native addons have direct access to the underlying system and its resources (file systems, network interfaces, etc.). While this might be a desired or even required feature for some use cases, it might be a security risk in others [7]. Additionally, native addons usually need to be compiled on the target platform, while WebAssembly is portable and agnostic of the system architecture. Most existing research around WebAssembly focuses on performance comparisons between JavaScript, WebAssembly, and native code.

Haas et al. described the motivation behind WebAssembly, its design goals, code execution and validation [8]. The authors used the PolyBench/C benchmark [23] to compare the performance of WebAssembly to that of native code and asm.js [12], a subset of JavaScript designed to be used as a compilation target for C code, which could then be executed by a JavaScript runtime. They found that WebAssembly was 33.7% faster on average than asm.js, and that the execution time of WebAssembly was less than 150% of the native execution time for 20 out of 24 benchmarks. It is important to note that V8 only implemented the TurboFan compiler [25] at that time, and did not use the Liftoff compiler [10].

Herrera et al. used the Ostrich benchmark suite [16] to compare the performance of native code, WebAssembly, and JavaScript. The benchmark performs numerical computations that are deemed relevant for scientific computations, such as machine learning. While they also found WebAssembly in Node.js to be slower than native code, WebAssembly consistently outperformed JavaScript in all tested web browsers and Node.js [13, 14].

Malle et al. conducted experiments comparing WebAssembly to JavaScript, asm.js, and native code in the context of artificial intelligence algorithms [18]. The results are in line with the results reported by Haas et al. [8] and Herrera et al. [13, 14], and again show that WebAssembly is faster than JavaScript, but slower than native code. The authors suggest that future additions to WebAssembly such as SIMD instructions will likely reduce the difference between WebAssembly and native code.

Hall et al. investigated WebAssembly as a platform for serverless applications. They came to the conclusion that the security properties of WebAssembly allow isolation similar to virtualization via containers, and that, while WebAssembly generally did not outperform native code, containers often took longer to start than the respective WebAssembly implementations [9].

Matsuo et al. suggested using WebAssembly in browser-based volunteer computing. They found WebAssembly outperformed JavaScript for long-running tasks, but the overhead of compiling and optimizing WebAssembly before being able to run it caused performance gains to disappear for tasks with short durations [19].

Jangda et al. exposed performance flaws of WebAssembly implementations in web browsers [15]. They found that, on average, WebAssembly code in the V8-based web browser Chrome is 55% slower than native code. While comparing code generated by V8 to native code generated by a C compiler, the authors observed that V8 produces more instructions. This leads to more CPU cycles required to execute the code and more cache misses in the processor's L1 instruction cache. Code generated by V8 also suffers from increased register pressure due to sub-optimal register allocations and the fact that V8 reserves some registers for memory management. Finally, the WebAssembly specification mandates certain safety checks at runtime, which also incur a performance cost. However, despite these problems, the authors also showed that WebAssembly was 54% faster than asm.js in the same browser.

The multitude of publications highlighting the performance benefits of WebAssembly over JavaScript has inspired efforts to simplify the integration of WebAssembly into existing JavaScript applications. For example, Reiser et al. proposed a cross-compilation method from JavaScript to WebAssembly that resulted in significant speedups of computationally intensive algorithms [24].

3 RELATED WORK

The idea of caching compiled code beyond single processes is not new, and has been implemented for other languages.

Bhattacharya et al. discussed improvements for the shared class cache (SCC) used by the J9 Java virtual machine. While its primary purpose is to reduce memory usage by sharing required class files, the SCC also contains compiled code, reducing application startup times significantly [3, 11].

Patros et al. invented a mechanism to reuse compilation results for Platform as a Service (PaaS) clouds via Dynamically Compiled Artifact Sharing (DCAS), with a focus on the Java SCC [6, 21].

Park et al. proposed a method to reuse code generated by an optimizing JavaScript just-in-time (JIT) compiler, allowing ahead-of-time (AOT) compilation based on previous compilations of the same code. Their benchmarks demonstrated significant speedups in JavaScript application performance [20]. The authors also highlighted the need for such technologies due to the increasing code size of web applications.

Haas et al. discuss two ways to improve compilation and startup times for WebAssembly in web browsers [8]. According to their research, parallel compilation using multiple threads leads to compilation times that are 80% lower than those of single-threaded compilation. V8 already implements parallel compilation by assigning individual WebAssembly functions to separate compilation threads. The authors also suggest that developers cache compiled WebAssembly modules in browsers using client-side IndexedDB databases [1]. However, IndexedDB is not available in Node.js, and as of 2020, support for WebAssembly modules in IndexedDB databases has been removed from V8, making it impossible for developers to explicitly cache compiled WebAssembly modules. Instead, web browsers are encouraged to implement implicit code caching as part of streaming WebAssembly compilation [4], which is not available in Node.js.

4 COMPILATION AT RUNTIME

The compiler infrastructure within V8 has changed significantly in the last few years. Even for the relatively new WebAssembly language, V8 implements a complex combination of compilation procedures for WebAssembly. The basic components are a WebAssembly interpreter, the baseline compiler *Liftoff* [10], and the optimizing compiler *TurboFan*, that V8 also uses to compile JavaScript [25, 26].

Since JavaScript code itself does not contain static type information, it is difficult to compile it directly [20]. Due to this difficulty, V8 begins JavaScript execution using the *Ignition* interpreter, and only when the interpreter has identified “hot” code sections, the TurboFan compiler is used to optimize and compile these JavaScript functions using type information gathered by Ignition. WebAssembly, on the other hand, is not a high-level language, and not dynamically typed, and it is, therefore, not necessary to collect dynamic type information before compiling WebAssembly code [10].

The TurboFan compiler optimizes and compiles WebAssembly through a complicated pipeline that first decodes WebAssembly function bodies and constructs graph representations. These graph representations are in *Static Single Assignment form (SSA)* and use the “Sea of Nodes” concept introduced by Click in his dissertation [5]. TurboFan then applies optimizations to the SSA, selects appropriate instructions for the target architecture, allocates CPU registers, and finally generates code.

The Liftoff compiler, on the other hand, was designed to be fast at the cost of generating less optimized code. Even though it is newer than the TurboFan compiler, it is not meant as a replacement, but as the initial compilation stage to quickly produce a usable module. Like TurboFan, Liftoff begins by decoding WebAssembly function bodies, but then immediately begins code generation in a single pass, without constructing an SSA graph representation or optimizing the code [10].

4.1 WebAssembly JavaScript Interface

From an application developer’s perspective, JavaScript applications can compile WebAssembly modules in two ways. The first is to call the constructor of the `WebAssembly.Module` class, which will synchronously compile the code, meaning that it will block the calling thread for the duration of the compilation. The second is the asynchronous function `WebAssembly.compile`, which will not block the calling thread.

By default, `WebAssembly.Module` uses Liftoff to compile the code, which is the faster compiler, and thus causes the smallest delay in the calling thread. V8 compiles the same module again, in a set of background threads, using the optimizing TurboFan compiler. When the optimized compilation result for a WebAssembly function is ready, the next invocation of the function uses the code produced by the TurboFan compiler instead of the output of Liftoff. This process is called “tiering up”, and is a tradeoff between startup time and code generation quality [10].

`WebAssembly.compile`, on the other hand, is an asynchronous function and, therefore, not as concerned with blocking the calling thread. Its default behavior is to use the TurboFan compiler, skipping the baseline compilation step. This causes the compilation to generally take longer than synchronous compilation would, but produces the optimized result directly.

4.2 Performance of generated code

In order to compare the performance of code generated by Liftoff to code generated by TurboFan, we compiled the *PolyBench/C 4.2* benchmarks [23] to WebAssembly with compiler optimization and Link Time Optimization (LTO) enabled. These benchmarks are scientific computing kernels and were already used by Haas et al. [8] and Jangda et al. [15] to compare the performance of WebAssembly to the performance of native code execution. Instead, we use the benchmarks to compare the performance of code generated by Liftoff to the performance of code generated by TurboFan.

We conducted all experiments on Ubuntu 19.04 running on an Intel® Core™ i7-8700 processor (base frequency 3.20GHz, turbo frequency 4.60GHz, 6 cores, 12 threads) with 32GB of memory (2666 MHz). We used Node.js v14.2.0, the most recent Node.js version at the time of writing, which is based on V8 version 8.1.307.31-node.33.

We compiled and ran each of the 30 PolyBench/C benchmarks one hundred times with only Liftoff enabled, and another one hundred times with only TurboFan enabled. We measured the time it took for the benchmarks to complete, which does not include their respective compilation times. Figure 1 shows the average speedup of the code generated by TurboFan with respect to the code generated by Liftoff for each benchmark, with error bars indicating the standard deviation. All benchmarks were faster when compiled with TurboFan, the average speedup across all benchmarks is 2.0, and the maximum speedup is 3.2.

We also ran all benchmarks using V8’s WebAssembly interpreter. On average, the PolyBench/C benchmarks were 247 times slower when interpreted than when compiled using TurboFan, and 115 times slower than when compiled using Liftoff. Sixteen of the 30 benchmarks were at least 200 times faster when compiled with TurboFan than when interpreted. While interpretation allows running WebAssembly code without prior compilation, its code execution is too slow for use in real applications.

We can conclude that the optimized code generated by TurboFan is indeed significantly faster than code generated by the baseline compiler Liftoff, and that the code produced by both compilers is

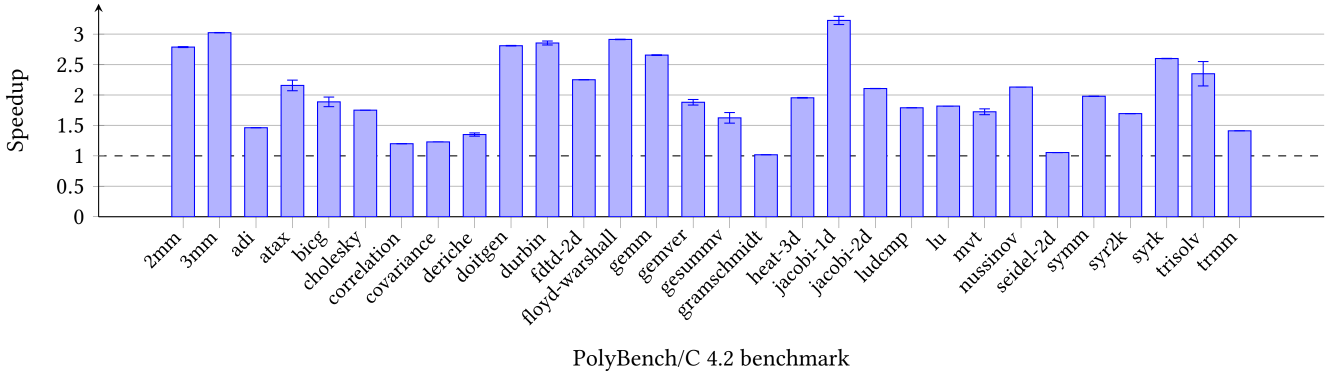

Figure 1: Speedup of code generated by TurboFan with respect to Liftoff

much faster than V8's WebAssembly interpreter. However, PolyBench/C is not a good basis for testing the performance of the compilers themselves, since the benchmarks are small and the WebAssembly modules are structurally very similar. In the following, we compare compilation times as well as CPU and memory footprint of both compilers.

5 CODE CACHING

Due to portability and security concerns, WebAssembly was designed to be compiled to the target architecture's instruction set at runtime. However, when running code from a trusted source on a single architecture, or untrusted code within a container or sandbox, these concerns become less relevant. Especially in scenarios where a Node.js application is expected to be initialized quickly, for example, when used as a command-line tool, as a desktop application, or in serverless computing, performance might be a more crucial factor. Here, using WebAssembly modules by first compiling them can cause visible delays.

5.1 Code extraction and insertion

Prior to designing a shared code cache, we need to find a way to efficiently retrieve compiled code from V8, and later inject the same code in a different V8 process.

While current versions of V8 provide such features for streaming WebAssembly compilation, no usable interface exists for Node.js, which only supports non-streaming WebAssembly compilation. However, V8 has internal functions that allow serializing compiled WebAssembly modules into byte sequences, and deserializing byte sequences into compiled WebAssembly modules. We developed an add-on for Node.js that exposes these internal V8 features to Node.js applications: `serialize` returns a JavaScript `ArrayBuffer` based on a given `WebAssembly.Module`, and `deserialize` creates a `WebAssembly.Module` based on the WebAssembly module bytes (referred to as "wire bytes" within V8) and the byte sequence generated by the `serialize` function.

This pair of functions is sufficient to extract code from a compiled module, store it in a cache entry, and later use the cache entry to obtain a usable module. This data flow is depicted in Figure 2.

V8 allows selectively disabling Liftoff and TurboFan. If a process is started with only Liftoff enabled, V8 prevents inserting code generated by TurboFan (and vice versa). A proper cache lookup therefore requires knowledge about the current process's V8 configuration. To achieve this, our Node.js add-on allows applications to check relevant V8 flags.

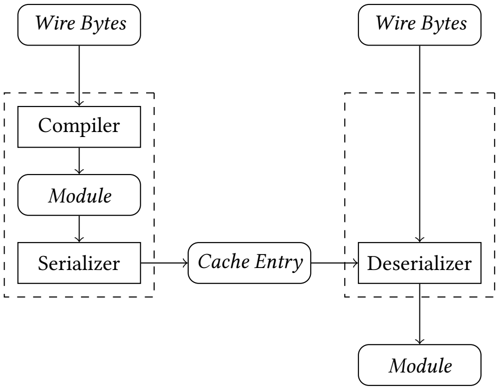

Figure 2: WebAssembly cache data flow: Cache entry creation (left) and cache entry retrieval (right)

In order to create realistic benchmarks, we extracted 115 WebAssembly modules from existing JavaScript applications, with module sizes ranging from as little as 1068 bytes to 37.3 MiB. It would be difficult to run the code represented by the WebAssembly modules in the way intended by their creators, since each module performs application-specific tasks and has certain requirements towards its host environment. However, our experiment is focused on compiling WebAssembly modules, which does not require running the compiled modules.

The approach Park et al. [20] used to cache compiled JavaScript code used cache entries that were much larger than the original JavaScript files. Similarly, we observe that serialized compiled WebAssembly modules are often considerably larger than the original WebAssembly files. Figure 3 shows the ratio of the serialized size to the original size based on the WebAssembly modules we extracted from existing applications, depending on which compiler was used by V8. In the case of JavaScript, better performance was achieved by caching optimized code in addition to intermediate bytecode, effectively increasing the size of cache entries [20]. For WebAssembly,

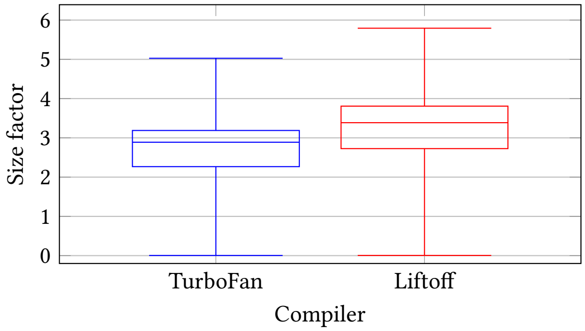

Figure 3: Ratio of serialized compilation result size to WebAssembly module size

it is sufficient to store the optimized compilation result (left side), which is significantly smaller than the result of the non-optimizing compiler Liftoff (right side), but still up to five times larger than the original WebAssembly module.

5.2 Compiler performance

Most importantly, we need to compare the performance of both compilers to the previously mentioned deserialization method. The hypothesis is that deserializing cached code uses less resources than compiling a WebAssembly module.

In order to test the hypothesis, we measured the real elapsed time it takes to obtain a compiled, usable module from the WebAssembly module bytes ("wire bytes"). To achieve comparable results, we disabled tiering up (see Section 4.1), only enabled one compiler at a time, and used the synchronous `WebAssembly.Module` constructor for Liftoff and TurboFan. Additionally, we took the same measurements for the previously described deserialization method. In this case, the module had already been compiled and optimized by the TurboFan compiler, and the serialized compiled module is available in memory (in addition to the wire bytes).

Each measurement is taken in a separate process. For each such process, we also record the CPU time of the process, that is, the total duration that threads of the process were scheduled on any of the CPU cores in user or system mode, and the peak physical memory usage through the `VmHWM` statistic provided by the Linux kernel. These metrics are equally if not more important than the elapsed real time required to compile a module. Even under the simplified assumption that CPU time and memory are the only resource constraints of a process, both of these resources are finite, and must be shared among all processes on the same system. While a high CPU time to real elapsed time ratio is an indication of well-designed parallelism, it also means that few concurrent instances of the same process might already use all available CPU time, and any additional instances could cause the performance of all processes to degrade. For cloud applications, it is realistic to assume that more than one process will be active on the same hardware at a time.

We recorded each measurement for each of the 115 WebAssembly modules 100 times. The mean values of elapsed real time, CPU time, and memory usage for each module are depicted in Figures 4, 5, and 6, respectively.

As shown in Figure 4, all three methods generally take longer for larger modules than for smaller ones. For legibility, Figures 4, 5, and 6 do not include errors bars. Instead, Figures 7, 8, and 9 show the significance of improvements. For two variables with mean values μ_1, μ_2 and standard deviations σ_1, σ_2, we define the *significance* of the change as $(\mu_1 - \mu_2)/(\sigma_1 + \sigma_2)$. By convention, we call the difference *statistically significant* if the significance is at least one.

By this definition, Liftoff was significantly faster than TurboFan for 111 modules (96.5%), used significantly less CPU time for 98 modules (85.2%), and had a significantly smaller memory footprint for 110 modules (95.7%).

When comparing deserialization to compilation using TurboFan, we measured statistically significant compilation time improvements for 112 modules (97.4%), CPU time improvements for 102 modules (88.7%), and memory usage improvements for 101 modules (87.8%). For 111 modules (96.5%), the speedup was at least 2, and for 77 modules (67.0%), the speedup was at least 20. Similarly, for 98 modules (85.2%), the CPU time was reduced by at least 50%, and for 54 modules (47.0%), it was reduced by at least 90%.

Compared to compilation using Liftoff, we observed significant compilation time improvements for 99 modules (86.1%), significant CPU time improvements for 77 modules (67.0%), and significant memory usage improvements for 41 modules (35.7%). For 69 modules (60.0%), the speedup was at least 2. The CPU time was reduced by at least 50% for 64 modules (55.7%).

The only statistically significant regression is an increase in memory usage for 39 modules (33.9%) when compared to Liftoff, and for 6 modules (5.2%) when compared to TurboFan. In these cases, however, the difference is small (less than 20%, see Figure 6).

Since the deserializer is synchronous, it is consistent to compare it to synchronous WebAssembly compilation. However, we also repeated this experiment with asynchronous WebAssembly compilation, and found that asynchronous compilation was significantly slower than synchronous compilation for 85 modules (73.9%) in the case of TurboFan, and for 91 modules (79.1%) in the case of Liftoff. For both compilers, this results in even larger differences when compared to deserialization, and we therefore decided not to present these results in detail.

We also repeated the experiment with deserialization of code generated by Liftoff instead of optimized code produced TurboFan, which leads to a larger serialized format (see Section 5.1). We found that it also causes longer deserialization times, and no significant improvements of real elapsed time, CPU time, or memory usage.

5.3 Module identification

The client needs to be able to identify each WebAssembly module in order to look it up in a cache. Since the JavaScript WebAssembly API is agnostic to the source of the WebAssembly module code, a module can only be identified by its code, and no file path or URL is available. Traditional information-theoretic algorithms, such as CRC32C, and cryptographic hash functions, such as SHA-1, provide reliable and, in the case of hash functions, collision-resistant identification methods. In scenarios where a hash collision could result in a security problem, cryptographic hash functions must be used for identification. However, these functions generally run in

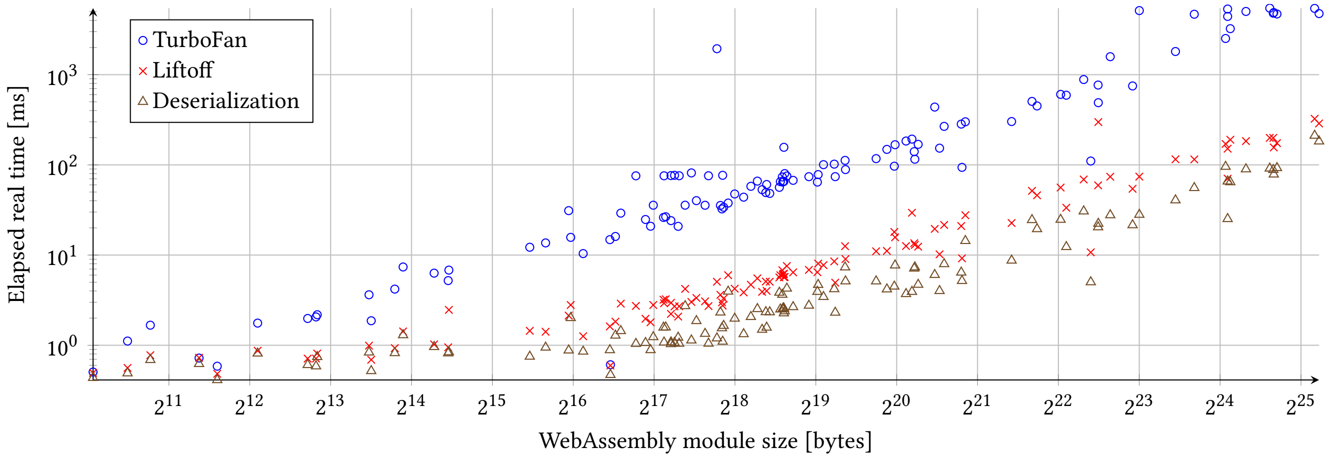

Figure 4: Compilation times by approach

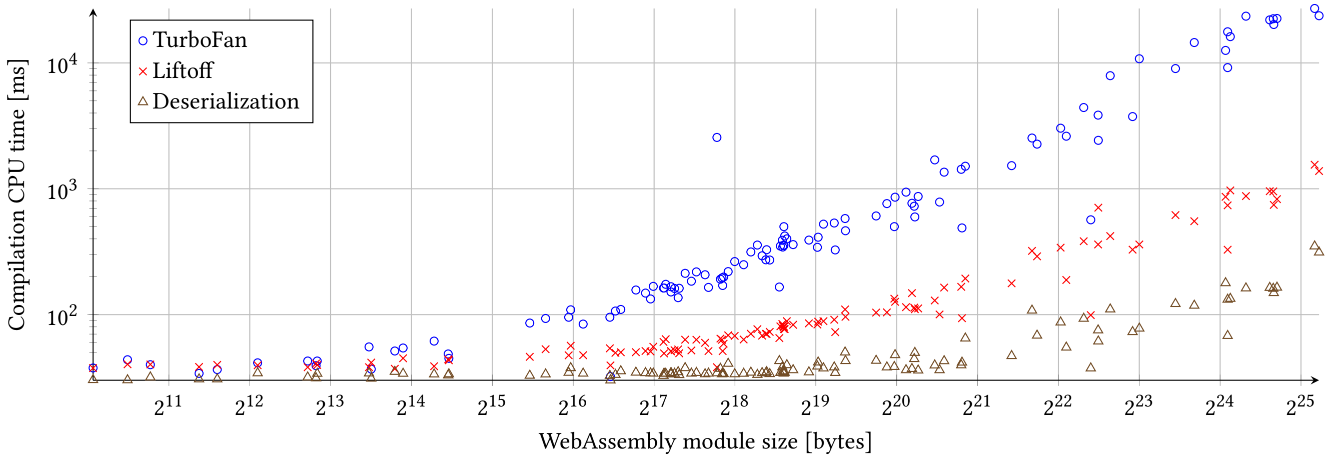

Figure 5: Compilation CPU times by approach

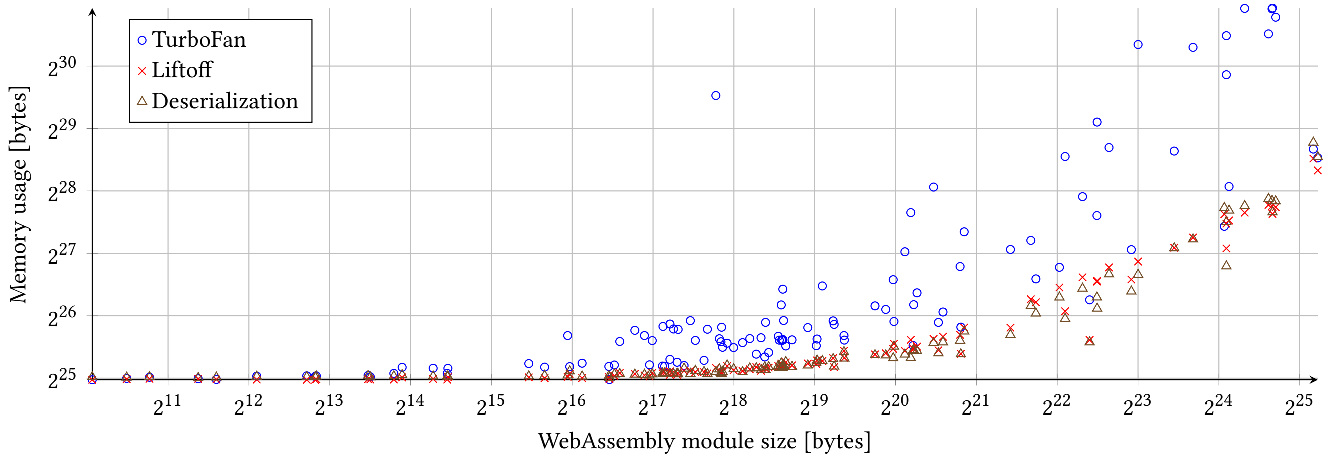

Figure 6: Compilation memory usage by approach

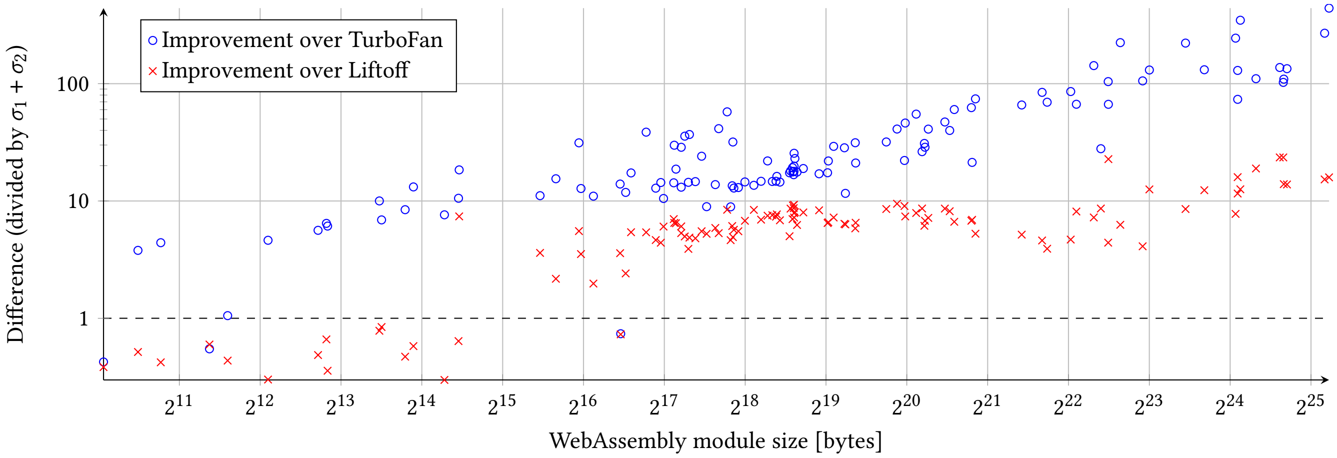

Figure 7: Significance of compilation time improvements

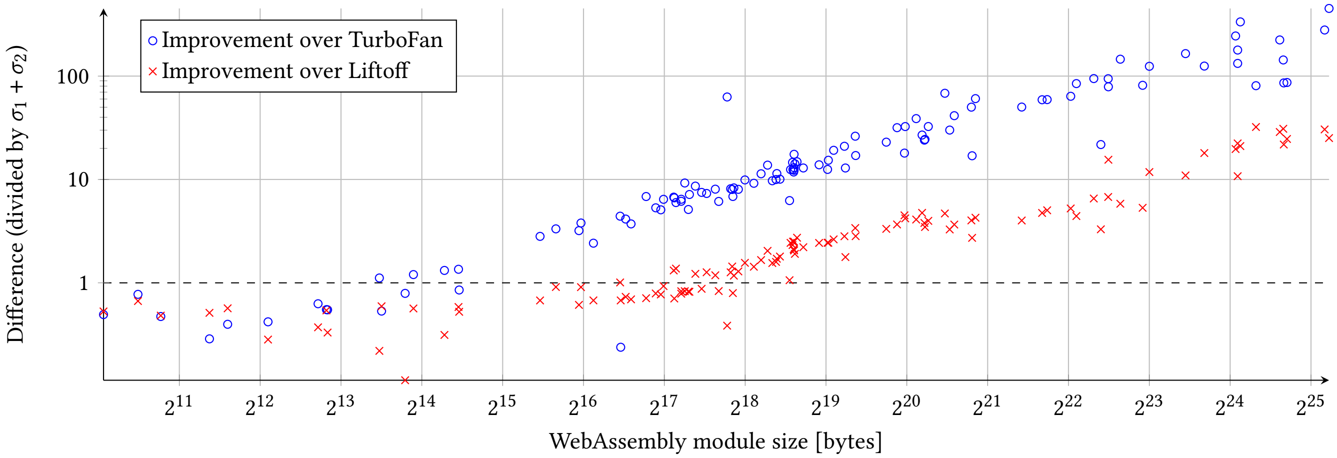

Figure 8: Significance of compilation CPU time improvements

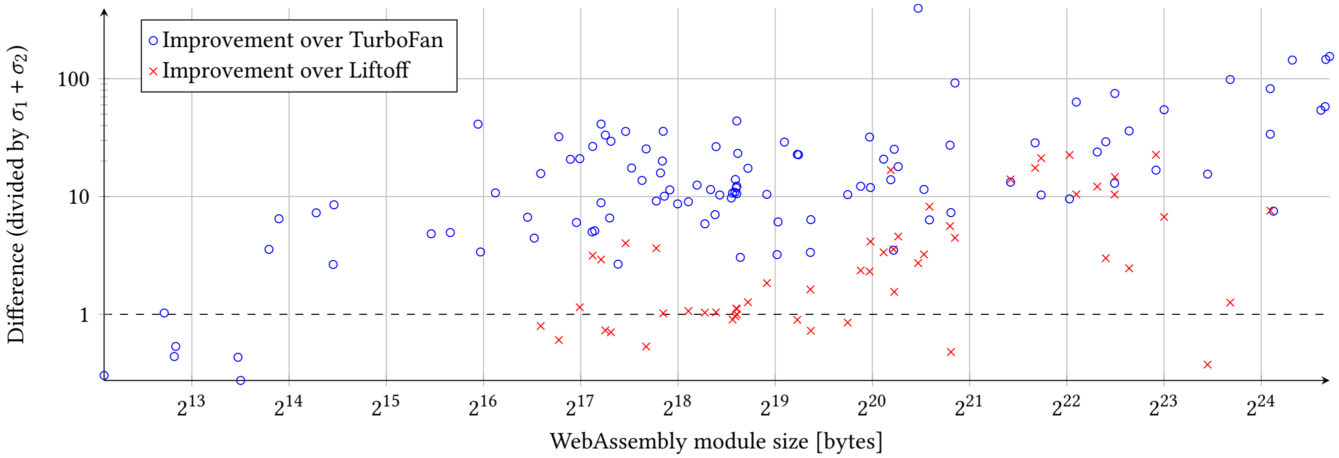

Figure 9: Significance of memory usage improvements

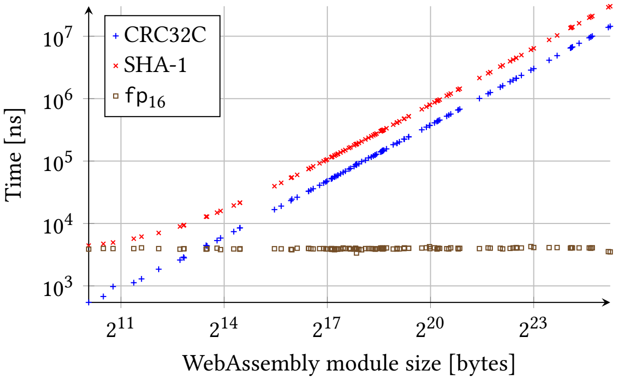

Figure 10: Performance comparison between fp16, CRC32C, and SHA-1 in Node.js

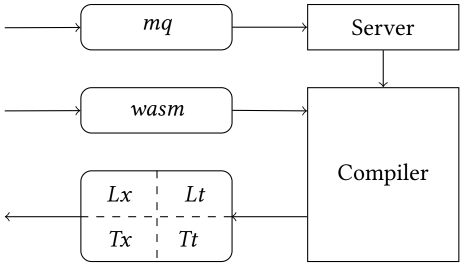

Figure 11: Shared cache server architecture

$\Theta(n)$, which is undesirable for large WebAssembly modules in situations where a collision would not result in a security problem, e.g., because access to the cache is restricted to a single client. For such cases, we define the function family $\mathtt{fp}_r$ for $r \geq 2$ as follows: Let $b_0, \ldots, b_{n-1}$ be the input byte sequence, with $b_i \in \{ 0, \ldots, 2^8 - 1 \}$ for $0 \leq i < n$. Let p be a linear congruential generator with $p(0) = b_{\lfloor n/2 \rfloor}$ and $p(i+1) = (a * p(i) + c) \bmod 2^{32}$. We chose $a = 1664525$ and $c = 1013904223$ as suggested by Knuth [17]. Let the result be the r byte vector $f_0, \ldots, f_{r-1}$ with $f_0 = \lfloor n/2^8 \rfloor \bmod 2^8$, $f_1 = n \bmod 2^8$, and $f_{2+i} = b_{p(i+1) \bmod n}$. In other words, the result consists of the length n modulo 2^{16} and the module bytes at $r-2$ pseudo-random locations. Each such function $\mathtt{fp}_r$ only requires 32-bit integer arithmetic, can be implemented efficiently in JavaScript, and runs in $O(1)$. While the resulting "fingerprint" is neither unique nor collision-resistant, it is sufficiently unlikely to collide with another module's fingerprint. Figure 10 shows a performance comparison between $\mathtt{fp}_{16}$, CRC32C, and SHA-1 in Node.js, based on running each function 100,000 times on each of the 115 WebAssembly modules. With a constant runtime of 4.0 μs, $\mathtt{fp}_{16}$ is significantly faster than other identification methods.

6 SHARED CODE CACHE

With the previously discussed cache creation and retrieval method from Section 5.1 and the performance benefits of code caching presented in Section 5.2, it is viable to construct a disk-based cache that can be populated and used by individual processes. However, this approach is troublesome for large application clusters: First, write access to the code cache should be controlled strictly to prevent malicious code injections, and it might be undesirable for all processes that use WebAssembly to have permission to write compiled code to the cache. Second, if a process uses Liftoff or tiering up to improve its own startup time (see Sections 4 and 5.2), it might not insert optimized code into the cache, but instead the output of the baseline compiler. Third, a disk-based cache might reduce expected speedups due to disk access times associated with potentially large cache entries (see Section 5.1), which each new process might copy from disk into memory.

6.1 Design and Implementation

To circumvent these problems, we designed and implemented a novel approach to share compiled WebAssembly code between Node.js processes. In the following, we will refer to the processes of one or more Node.js applications as *client processes*. In this context, a *V8 configuration* is a set of flags that affect V8's internal behavior. The cache implementation prevents loading incompatible cache entries, and potentially maintains multiple cache entries for the same WebAssembly module, but for different V8 configurations. For example, Figure 11 includes a matrix of four configurations *Lx*, *Lt*, *Tx*, and *Tt*, where the first letter indicates which is the fastest enabled compiler (**L**iftoff or **T**urboFan), and the second indicates whether the compiler is used e**x**clusively, or if **t**iering up is enabled.

Client processes compile WebAssembly modules through a modified WebAssembly JavaScript Interface, which is compatible with the one described in Section 4.1. The compilation procedure, given a module represented by bytes $b_0, \ldots, b_{n-1}$, computes the module identifier $\mathtt{fp}_{16}(b_0, \ldots, b_{n-1})$, and attempts to locate a cache entry in a shared memory segment based on the computed module identifier and the current V8 configuration. If such an entry exists, the compilaton procedure deserializes the cache entry to obtain a WebAssembly module instance without having to compile the module bytes. If no such entry exists, the client process copies $b_0, \ldots, b_{n-1}$ into a new shared memory segment (*wasm* in Figure 11), and sends a pointer to the new shared memory segment and the current V8 configuration to an existing message queue (*mq* in Figure 11). Finally, the client process falls back to V8's original compilation procedure, which, depending on the current configuration, uses Liftoff and/or TurboFan to compile the code.

A separate server process is responsible for creating the message queue *mq*. Upon receiving a pointer to a shared memory segment *wasm* along with a valid V8 configuration, the server process starts a new compiler process with parameters matching the received V8 configuration. The compiler process loads the module bytes $b_0, \ldots, b_{n-1}$ from the shared memory segment, unlinks the segment, and computes the module's fingerprint $\mathtt{fp}_{16}(b_0, \ldots, b_{n-1})$. After ensuring that no other compiler process is already compiling the same module with the same V8 configuration, the process compiles the module. If TurboFan has not been disabled in the given V8 configuration, the compiler process uses it to produce optimized code. Only if TurboFan has been disabled, the compiler process uses Liftoff, and therefore generates unoptimized code. Once compilation finishes, the compiler process serializes the compiled WebAssembly module, and writes the result to shared memory.

The modified WebAssembly JavaScript Interface was implemented in JavaScript, except for the deserialization logic, which was implemented in C++ due to the necessity to access internal V8 features, and communication with the message queue, which was implemented in C++ and uses POSIX functions.

The server process code is written in C using POSIX functions to access the message queue. The compiler processes execute JavaScript code, and serialization logic written in C++. The actual compilation procedures are an existing part of V8.

6.2 Evaluation

To evaluate our design and implementation, we consider two cases:

Cache miss: When a client process fails to locate a compiled WebAssembly module in the shared cache, it not only needs to compile the module itself, but also suffers from two additional performance impairments. First, the client process needs to copy the WebAssembly module bytes to a shared memory segment, and notify the server process about the cache miss. Depending on the module size, this can cause a short delay before compilation begins. Second, while the client process compiles the module itself, the server process will spawn a compiler process, which also compiles the module, effectively increasing the system load, and potentially increasing compilation times.

Cache hit: Upon successfully locating a compiled WebAssembly module in the shared cache, the client process benefits from two performance aspects. First, it does not need to compile the module, which, on average, improves the time until the module is available, and likely reduces CPU load and memory footprint (see Section 5.2). According to the model proposed by Patros et al. [22], this also reduces performance interference on co-located cloud tenants. Second, if not forbidden by the process's V8 configuration, the obtained compiled code is already optimized, which would not be the case with V8's default *tiering up* behavior, or when using Liftoff. As we have seen in Section 4.2, this can lead to improved execution times.

While we already know the impact of deserialization as compared to compilation based on Section 5.2, we used PolyBench/C (see Section 4.2) to create a set of artificial Node.js applications to evaluate the performance impact of the shared cache. We measured the real elapsed time it takes for each application to compile and then execute its associated PolyBench/C benchmark. For this experiment, we use the default V8 configuration, which enables both Liftoff and TurboFan, and uses tiering up (see Section 4), and ran each application 45 times.

Figure 12 shows the mean execution times for cache misses and cache hits, with error bars corresponding to the standard deviation. Since execution times between benchmarks vary tremendously, all execution times were divided by the same measurement taken without a cache in place. Similarly, the ratio between execution time and compilation time varies greatly, therefore, we do not display compilation and execution times in a stacked manner.

As expected, we see a performance regression for cache misses. It is worth noting that the benchmarks with the largest (by percentage) performance regressions such as jacobi-1d are particularly short-running, which means that the delay caused by copying the module bytes to shared memory has a larger (by percentage) impact on the total elapsed time.

We also observe performance improvements for almost all benchmarks when a cache entry is found. The average speedup is 1.8, and the maximum speedup is 3.0. We expect that different Node.js applications would see vastly different performance benefits, depending on the WebAssembly modules in use.

Finally, most operating systems allow protecting shared memory segments from unintended write access. It appears that such measures allow controlling read and write access to the shared code cache sufficiently to prevent malicious code injections, for example, by only giving write access to the compiler processes, and not to client processes.

7 FUTURE WORK

A future direction for a shared code cache could be an extension to a disk-based cache. While the system kernel might keep frequently accessed cache entries in memory up to a certain size, large cache entries might still have to be loaded from the disk, and could negatively affect the cache performance. A balanced strategy might be to only move modules from shared memory to disk when most of the available memory is in use, and to prioritize frequently accessed modules in memory.

While our shared cache implementation prevents duplicate compilation on the server side, it does not prevent duplicate work among client processes. The primary reason is that client processes benefit from the shorter compilation times of tiering up, whereas the server process is focused on producing optimized code at the cost of longer compilation times. A future implementation could reduce the amount of duplicate work between processes further.

It might also be worth considering data compression for cache entries. Park et al. compressed cache entries in their JavaScript code cache implementation, and successfully reduced the size of the code cache with only minimal performance sacrifices [20]. However, as long as cache entries are stored in shared memory, decompression would require copying the decompressed data to a new memory area on each invocation, which makes it unlikely to result in large performance improvements. A disk-based cache solution could potentially benefit from compression to reduce cache entry sizes and therefore disk access times.

8 CONCLUSION

As we have seen in Section 5.2, compiling WebAssembly modules at runtime can lead to a delay of multiple seconds during an application's startup phase, and can require vast amounts of CPU time and physical memory. While *Liftoff* is much faster than the optimizing compiler *TurboFan*, its generated code is significantly slower than the code produced by *TurboFan*, but still much faster than interpreting WebAssembly code without compiling it first.

We reduced module load times by caching compiled and optimized code for the target architecture, and observed large performance benefits for many WebAssembly modules. Finally, in Section 6, we extended the idea to a scalable multi-process shared code cache, which provides an efficient way to load WebAssembly modules in Node.js applications, without having to compile and optimize each module in each process. The smaller CPU and memory footprint can reduce interference on co-located cloud tenants, and, therefore, improve scalability [22].

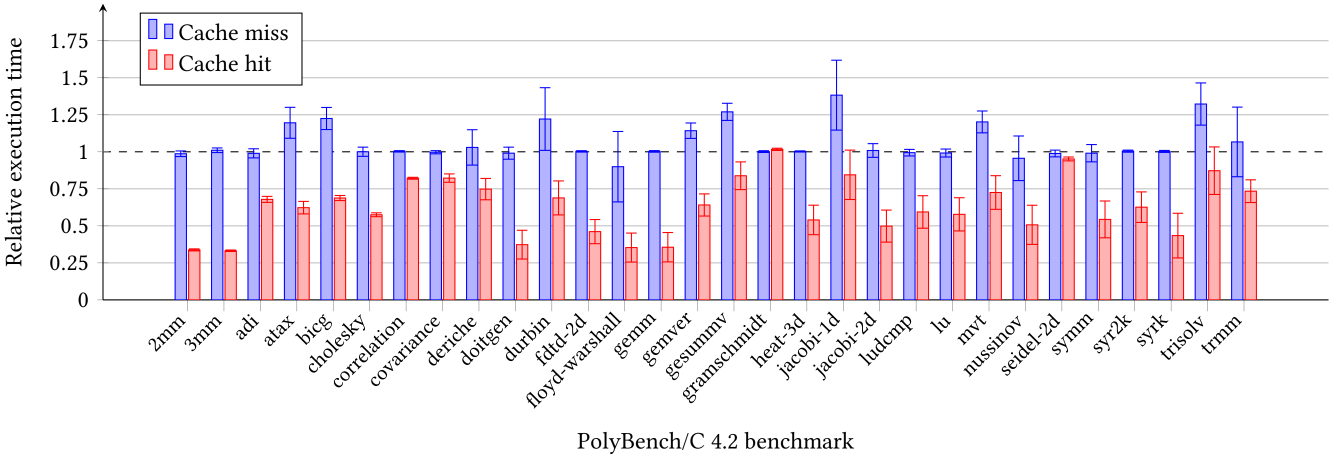

Figure 12: Shared cache performance impact on PolyBench/C benchmarks

While WebAssembly is still an emerging technology, we expect growing adoption over the next few years. These performance improvements and reduced startup times presented in this paper might allow widespread use of WebAssembly in serverless computing and other cloud configurations, without sacrificing the speed, portability, and security of WebAssembly.

9 ACKNOWLEDGMENTS

This research was conducted within the Centre for Advanced Studies — Atlantic, Faculty of Computer Science, University of New Brunswick. The authors are grateful for the colleagues and facilities of CAS Atlantic in supporting our research. The authors would like to acknowledge the funding support of the Natural Sciences and Engineering Research Council of Canada (NSERC), 501197-16. Furthermore, we would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

REFERENCES

[1] 2018. Indexed Database API 2.0. https://www.w3.org/TR/IndexedDB/

[2] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*, Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya (Eds.). Springer Singapore, Singapore, 1–20.

[3] Devarghya Bhattacharya, Kenneth B. Kent, Eric Aubanel, Daniel Heidinga, Peter Shipton, and Aleksandar Micic. 2017. Improving the performance of JVM startup using the shared class cache. In *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*. IEEE, Victoria, BC.

[4] Bill Budge. 2019. Code caching for WebAssembly developers. https://v8.dev/blog/wasm-code-caching

[5] Cliff Click and Keith D. Cooper. 1995. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17, 2 (March 1995), 181–196.

[6] Michael H. Dawson, Dayal D. Dilli, Kenneth B. Kent, Panagiotis Patros, and Peter D. Shipton. 2019. Dynamically compiled artifact sharing on PaaS clouds. Patent US 10,338,899 B2.

[7] Node.js Foundation. [n.d.]. C++ Addons. https://nodejs.org/api/addons.html

[8] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jf Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*. ACM Press, Barcelona, Spain, 185–200.

[9] Adam Hall and Umakishore Ramachandran. 2019. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation - IoTDI '19*. ACM Press, Montreal, Quebec, Canada, 225–236.

[10] Clemens Hammacher. 2018. Liftoff: a new baseline compiler for WebAssembly in V8. https://v8.dev/blog/liftoff

[11] Daniel Heidinga, Peter D. Shipton, Aleksandar Micic, Devarghya Bhattacharya, and Kenneth B. Kent. 2020. Enhancing Virtual Machine Performance Using Autonomics. Patent US 10,606,629 B2.

[12] David Herman, Luke Wagner, and Alon Zakai. 2014. asm.js. http://asmjs.org/

[13] David Herrera, Hanfeng Chen, Erick Lavoie, and Laurie Hendren. 2018. Numerical computing on the web: benchmarking for the future. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages - DLS 2018*. ACM Press, Boston, MA, USA, 88–100.

[14] David Herrera, Laurie Hendren, Hangfen Chen, and Erick Lavoie. 2018. *WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices.* Technical Report SABLE-TR-2018-2. Sable Research Group, School of Computer Science, McGill University, Montréal, Canada.

[15] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 107–120.

[16] Faiz Khan, Vincent Foley-Bourgon, Sujay Kathrotia, and Erick Lavoie. 2014. Ostrich Benchmark Suite. https://github.com/Sable/Ostrich

[17] Donald E. Knuth. 1981. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition.* Addison-Wesley.

[18] Bernd Malle, Nicola Giuliani, Peter Kieseberg, and Andreas Holzinger. 2018. The Need for Speed of AI Applications: Performance Comparison of Native vs. Browser-based Algorithm Implementations. *arXiv:1802.03707* (Feb. 2018).

[19] Hiroyuki Matsuo, Shinsuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. 2019. Madoop: Improving Browser-Based Volunteer Computing Based on Modern Web Technologies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Hangzhou, China, 634–638.

[20] Hyukwoo Park, Sungkook Kim, Jung-Geun Park, and Soo-Mook Moon. 2018. Reusing the Optimized Code for JavaScript Ahead-of-Time Compilation. *ACM Transactions on Architecture and Code Optimization* 15, 4 (Dec. 2018), 1–20.

[21] Panagiotis Patros, Dayal Dilli, Kenneth B. Kent, and Michael Dawson. 2017. Dynamically Compiled Artifact Sharing for Clouds. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Honolulu, HI, USA, 290–300.

[22] Panagiotis Patros, Stephen A. MacKay, Kenneth B. Kent, and Michael Dawson. 2016. Investigating Resource Interference and Scaling on Multitenant PaaS Clouds. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON '16)*. IBM Corp., USA, 166–177.

[23] Louis-Noel Pouchet and Tomofumi Yuki. 2016. PolyBench/C. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[24] Micha Reiser and Luc Bläser. 2017. Accelerate JavaScript applications by cross-compiling to WebAssembly. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages - VMIL 2017*. ACM Press, Vancouver, BC, Canada, 10–17.

[25] V8 Team. 2017. Launching Ignition and TurboFan. https://v8.dev/blog/launching-ignition-and-turbofan

[26] Seth Thompson. 2016. Experimental support for WebAssembly in V8. https://v8.dev/blog/webassembly-experimental