2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application

Characterization and Implication of Edge WebAssembly Runtimes

Zhen Wang, Jianda Wang, Zhendong Wang, Yang Hu
ECE Department, The University of Texas at Dallas, Richardson, TX 75080 USA
Email: {zhen.wang2, jxw174930, zhendong.wang, yang.hu4}@utdallas.edu

***Abstract*—WebAssembly, an emerging bytecode format, which is initially developed for partially replacing JavaScript and speeding up browser applications, has been extended to the server-side due to its speed and security promise. It has been considered as a promising alternative to the widely deployed container technique for isolating lightweight applications. To run WebAssmebly from the server-side, aside from the NodeJS runtime, several WebAssembly native runtimes have been proposed. We characterize major WebAssembly runtimes through extensive applications and metrics. Our results show that different runtimes fit different application scenarios. Based on that, a framework for reducing the startup latency of WebAssembly service while keeping maximum performance is provided. To identify the root causes of the performance gap, the analysis of emerging Cranelift compiler against LLVM in detail is reported. In addition, this paper gives revealing suggestions and architectural proposals for designing an efficient WebAssembly runtime. Our work provides insights on both WebAssembly runtime enhancement and WebAssembly-based cloud service exploitation.**

***Index Terms*—WebAssembly, cloud VM, FaaS, performance evaluation**

I. Introduction

Cloud services are becoming ubiquitous, but they normally fail to provide a real-time response due to inherent service latency and network delay caused by congestion or distance. In hopes of meeting the quality and time pressures of many latency- and safety-critical services such as autonomous driving tasks which involve massive data processing, edge-cloud is becoming an active research topic where servers closer to the end-user as known as the edge servers are responsible for the preprocessing and thus reduces the cloud side pressure [1]–[4]. However, such a paradigm cannot eliminate the inherent latency and brings new challenges. On the one hand, either for the cloud or the edge server, creating an isolated service consumes non-trivial time, on the other hand, an edge server tends to have limited hardware resources available. As a result, co-locating multi-functions in a speedy and resources-friendly while secure way is crucial.

WebAssembly, also known as Wasm, is a size- and load-time-efficient bytecode format that is originally designed for the web. It makes running code written in multiple languages in browsers at near-native speed possible by compiling the code into WebAssembly. Meanwhile, there is growing interest in pushing WebAssembly to the edge side because of the sandbox execution environment and the portability of WebAssembly. With WebAssembly, it is possible to securely run a untrusted service, regardless of underlying ISA and OS in a multi-tenant edge node while not overwhelming it.

Before WebAssembly, container, a more lightweight and scalable isolation solution than the traditional virtual machine (VM), has changed the face of today's service computing. Considerable public and private cloud services such as Amazon ECS and Google Cloud Functions rely on containers as their underlying execution sandboxes. The success of the emerging cloud paradigm, Function as a Service (FaaS) or serverless is largely attributed to the container. Unsurprisingly, there has been effort on pushing containers to the edge platforms, because a safely-isolated and multi-tenant execution environment is critical for the edge service provider. Meanwhile, there is a fast-growing WebAssembly ecosystem that makes WebAssembly a more competent alternative for the edge. In summary, WebAssembly has the following benefits over containers.

❶ Fast response is the key that motivates the edge research community, but a container-based edge serverless platform fails to provide real-time response. The main reason is the high cold start latency which remains challenging in the foreseeable future. Such latency hinders us from exploiting the numerous benefits brought by the function as a service (FaaS) paradigm. A WebAssembly-based solution incurs less startup latency because of its small code size and the streaming compiling further reduces the latency [5]. ❷ A container is a process that utilizes several Linux features to create an isolated function execution environment. Also, container images are executables that have OS and library incorporated. A container-based service thus incurs non-trivial memory and CPU overhead. This is prohibitive for resource-constrained edge devices. For example, the state-of-the-art embedded AI computing device, Jetson TX2 is equipped with only 8 GB of memory [6]. Previous work has validated the feasibility of executing a WebAssembly service within a nano-process that consumes fewer resources without compromising security [7]. ❸ WebAssembly possesses "compile once run anywhere" feature that container does not support, because WebAssembly code is not related with the computer hardware it runs atop of, and the ongoing WebAssembly System Interface (WASI) project provides a portable interface between the WebAssembly program and the OS kernel. This endows WebAssembly with advantages on a heterogeneous platform.

Currently, companies such as Cloudflare and Fastly are building WebAssembly-based cloud service platforms using

978-1-6654-9457-1/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCC-DSS-SmartCity-DependSys53884.2021.00037

2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application

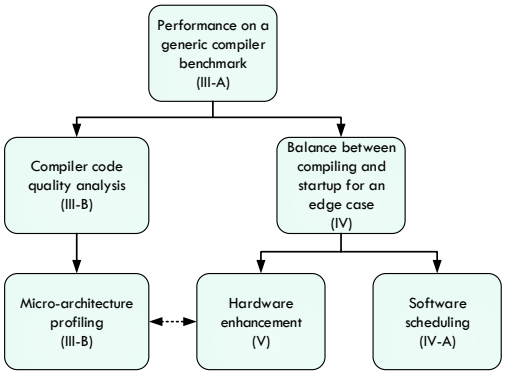

Fig. 1: Paper structure

different WebAssembly runtimes [8], [9]. Their cloud services claim to be able to spin up within milliseconds. Similar work from academia is reported in [5], [7], [10].

To run WebAssembly in the edge or cloud environment, many novel WebAssembly execution environments as known as runtimes have been launched. A WebAssembly runtime, a.k.a. WebAssembly virtual machine (VM) is responsible for compiling WebAssembly binary into a WebAssembly module and then instantiate the module to run the binary. However, there is no previous work explored the implication of those WebAssembly runtimes. We take the first step by systematically examining the native WebAssembly runtimes and analyzing the implication from application scenario, compiler backend and hardware architecture level.

In this paper, we start from explaining the necessity of developing WebAssembly runtimes other than NodeJS (section II-B & II-C). Next, as Fig. 1 outlines, we characterize the four major server-side WebAssembly runtimes a.k.a. WASI-native runtimes along with NodeJS (section III-A) on a widely-used compiler benchmark, PolyBench, which is used in numerous WebAssembly works. The WASI-native runtimes include Wasmtime, Wasmer, Lucet and WAVM. To explain the characterization result, the subsequent code quality analysis on runtimes' backend compilers is reported in section III-B. Micro-architecture profiling also shows agreement with the above compilation analysis. Discussions on the tradeoff between compilation and startup latency follows in section IV. While PolyBench provides a way to investigate the WebAssembly runtime compiler performance, it does not incorporate all the application cases that a cloud/edge server would host. In section IV-A, we propose a scheduling framework for handling execution requests in edge application scenarios based on the insights obtained from the tradeoff discussion in section IV. Section V provides architecture extension on enhancing the runtimes which does not breach the fast startup principle.

To summarize, our contributions are as follows:

- We systemically characterize and analyze major edge WebAssembly runtimes for the first time.
- We identify the root cause of performance gap between different WebAssembly runtimes through backend compiler and hardware implication analysis. The results will benefit the emerging WebAssembly-centric compiler backend, Cranelift.
- We provide advice and an empirical framework for the potential WebAssembly cloud provider to decide the best runtime(s) choice that can keep a balance between the speed and startup latency by taking application scenario and computation size into consideration.
- We suggest architecture support in the CPU that can improve performance while not compromising the startup latency.

II. Background

In this section, we will explain these concerns: (1) What are the benefits of WebAssembly? (2) How do we run WebAssembly at server side using NodeJS? (3) What is WebAssembly System Interface (WASI), the interface between WebAssembly and kernel, and how does WASI propel the development of new WebAssembly runtimes?

A. WebAssembly and its features

WebAssembly is a bytecode format similar to Java bytecode, but its polyglot and security features are what Java does not possess. Apart from C/C++, many other programming languages such as Rust can be seamlessly converted into WebAssembly. It was initially devised to improve web application performance by reusing C/C++ code while improving the security. A number of previous works have shown the performance advantages of WebAssembly over JavaScript counterpart [11], [12], while it is still slower than native code. There is a surge of interest of extending WebAssembly application scenarios to the server especially the edge server side because it meets performance, portability, and security at the same time.

- **Security:** The security of WebAssembly is ensured by memory safety and control flow integrity. As the Fig. 2 shows, the memory and function call are out of the control of the WebAssembly instructions. A Memory access in WebAssembly can only happen in a bound-checking linear memory using the offset from the base of the linear memory. While the host can manipulate this linear memory area at will, WebAssembly module has no information to the host or other WebAssembly modules. Meanwhile, a WebAssembly module cannot directly call a function but call by the index to a function table which lives outside linear memory. As a result, no matter how a hijacker change the contents inside the WebAssembly memory, they cannot execute external malicious code. Reflected in Fig. 2, the hexadecimal values translated from the indexes are what the untrusted third-party WebAssembly code cannot know and manipulate. Also, a function call is subject to a type signature check at runtime to ensure control flow integrity.
- **Portability:** WebAssembly code is not related with hardware architecture and the same WebAssembly code can run across heterogeneous OS and hardware. Multiple programming languages such as C/C++, Rust, and Go can be converted into WebAssembly. While LLVM IR

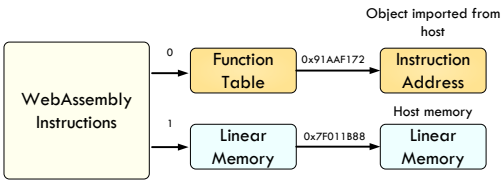

Fig. 2: Safety mechanism of WebAssembly bytecode

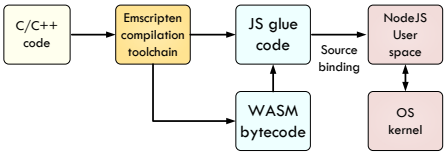

Fig. 3: Workflow of reusing C/C++ code by porting them to WebAssembly and using WebAssembly inside NodeJS

bytecode can represent constructs and semantics of many languages, it is not portable because the representation is different under different architecture.

- **Speed:** WebAssembly is an intermediary representation (IR) which is compact. As a result, it requires less time to download, decode. Meanwhile, since WebAssmebly is closer to machine code, the optimization and compiling takes less time as well.

B. Emscripten and NodeJS

NodeJS is a popular server-side framework built upon V8 engine. It not only supports running JavaScript but also WebAssembly. Emscripten is a complete tool chain used to produce both WebAssembly module and the JavaScript (JS) glue code to instantiate WebAssembly module so that we can execute WebAssembly inside NodeJS or browser. The workflow is depicted in Fig. 3. The JS glue code is also responsible for invoking JS version libc for system calls. This is how WebAssembly interface with operating system before the WebAssembly system interface (WASI) being proposed: the NodeJS host takes the job.

C. WASI and WebAssembly native runtimes

The emergence of edge WebAssembly runtimes (aka WASI-native runtimes) cannot leave WASI, the interface between WebAssembly module and OS kernel.

To a WebAssembly module, WASI, is a set of callable functions that can be imported by an index. While the functions can be implemented in a variety of ways as long as it wraps system calls (such as file, console) and provides an interface, WASI-libc is the common implementation that is supported by Wasmtime, and Lucet among others. The workflow of porting code to WebAssembly and running it inside WASI-native runtimes is shown in Fig. 4: the runtime exposes its WASI-libc implementation for WebAssembly module to call instead of relying on NodeJS API.

To build a solid foundation for the server-side WebAssembly, an emulation based on the NodeJS cannot serve as the de facto standard. From the comparison, it is clear that NodeJS-based WebAssembly application carries baggage associated

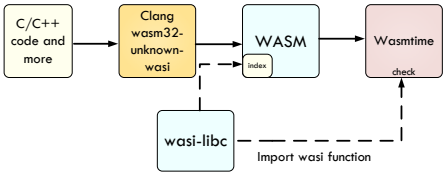

Fig. 4: Workflow of porting application into WebAssembly and run it inside a WASI-native runtime such as Wasmtime

with JavaScript. WASI-native runtimes have at least the following benefits over NodeJS: (1) they supports multiple host languages such as C, Python, or Rust while NodeJS only supports JavaScript. (2) Emulating kernel interface (POSIX) inside NodeJS can introduce performance cost. For example, for NodeJS runtime, Emscripten is normally used for creating WebAssembly, and thus A WebAssembly module runs on NodeJS has no direct access physical file system: it has to use either virtual file system of NodeJS (NODEFS) that is mapped to the physical one or JS glue code as an intermediary passing data back and forth. This limit not only complicates the file access but also slows down the file access.

III. WebAssembly Edge Runtimes Characterization

We start from characterizing the major edge WebAssembly runtimes. Wasmtime and Lucet runtimes are officially supported by Bytecode Alliance, an open source community dedicated to forging the ecosystem of WebAssembly. We also include another two typical runtimes WAVM and Wasmer because of the extensive attention they have received from the community. Wasmtime is the most documented and serves as the foundation. Lucet adopts AOT compilation and claims low-latency. WAVM uses LLVM and claims near-native performance. Wasmer has the the greatest number of Github starts.

The hardware we use includes an Intel machine equipped with a i7-8700 CPU, 16 GB RAM, 32K L1d cache, 32K L1i cache, 256K L2 cache, and 12288K L3 cache; and a NVIDIA Jetson TX2 runs ARMv8 aarch64 CPU and 8GB RAM. The OS we use is Ubuntu 18.04. We use vtune and perf as our performance analyzing tool. While vtune gives us more intuitive results, perf enables more flexibility. Version of NodeJS, Wasmtime, Lucet, WAVM and Wasmer are 12.9.1, 0.19.0, 0.7.0-dev, 0.0.0-prerelease, and 1.0.0-alpha5 respectively. It should be noted that in our experiments, we aim at investigating different WebAssembly runtimes, so we choose the same optimization level when generate WebAssembly from C code. We thus ensure the fairness: every WebAssembly runtime interprets the same WebAssembly.

A. PolyBench testing

To have a general look, we examine the runtimes over PolyBench. PolyBench is a popular compiler benchmark which is designed to measure the effect of polyhedral loop optimizations. It involves many computation-intensive scientific kernels such as image processing and data mining. It is a widely-used benchmark among multiple WebAssembly papers [10], [11].

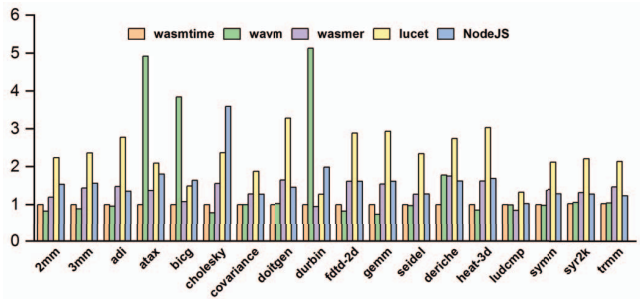

Fig. 5: The normalized overall run time comparison between Wasmtime, WAVM, Wasmer, Lucet, and NodeJS where time of Wasmtime is set to 1

The normalized execution time on top of Wasmtime, WAVM, Wasmer, Lucet and NodeJS is shown in Fig. 5. First of all, apart from the drawbacks discussed in section II, among all the applications in the figure, there is no one that NodeJS shows the best performance, so introducing the novel runtimes also brings the performance benefits.

Among the four edge WebAssembly runtimes, we can find the performance of WAVM (the second bar) is on par with Wasmtime (the first bar), and actually has a marginal speedup over Wasmtime on most applications. This observation holds for most of the PolyBench applications that are `2mm`, `3mm`, `adi`,`cholesky`, `covariance`, `doitgen`, `fdtd-2d`, `gemm`, `seidel`, `heat-3d`, `ludcmp`, `symm`, `syr2k`, and `trmm`. The speedup is especially obvious for applications such as `cholesky` and `gemm`. The speedup comes from WAVM using LLVM as its machine code generator, which is a well-developed compiler project. However, among them, there are three applications that WAVM shows the slowest performance. This is caused by the high startup latency of WAVM as we will discuss in IV.

Surprisingly, Lucet has the worst performance across PolyBench though it claims to be extremely fast [9]. Lucet adopts an ahead-of-time (AOT) compiler, lucetc, built on top of the Cranelift code generator which translates target-independent intermediate representation (IR) into executable machine code. As a result, it should be faster than other JIT style WebAssembly runtime. As we will show in IV, Lucet does has speed benefits in terms of short-lived application. As for Wasmer, its default compiler backend is also Cranelift, but it does not perform as well as Wasmtime, so we do not focus on it in this paper. Although Lucet, Wasmer, and Wasmtime all adopt Cranelift as their backend, there are obvious performance gaps between them. This can be caused by `FuncEnvironment` that each compiler plugs into the Cranelift backend [13]. Those `FuncEnvironment` allows Cranelift to generate different code sequences and operation types, but different `FuncEnvironment` is set according to the WebAssembly targeting application scenarios.

B. Performance analysis

To investigate the code size and performance gap between different runtimes, it is necessary to analyze the native code generated by them. Due to the space limit, it is impractical to analyze the machine instructions from each of the four runtimes. Using the same matrix multiplication WebAssembly binary which contains tight for loops (Listing (1)), the partial corresponding machine code generated from WAVM (Listing (2)) and Wasmtime (Listing (3)) is reported in Fig. 6. The reason that we choose matrix multiplication is because similar matrix processing that involves nested loops and arrays are major operations in PolyBench.

We choose WAVM and Wasmtime because they represent the major WebAssembly compilers that are LLVM and Cranelift, respectively. While Lucet and Wasmer also use Cranelift as the default backend WebAssembly compiler, it is shown from the above characterization that they do not represent the best performance of Cranelift. While Cranelift is a promising WebAssembly compiler framework written in Rust, it is new and lacks further optimizations. The increased code size as our following analysis shows, is caused by register abuse, indirect memory address, and inappropriate instruction choice.

*1) **register Abuse**:* From Listing (3) we find Cranelift does not utilize the existing registers which do not need to be preserved according to Cranelift's calling convention, instead it abuses unnecessary registers and cause unnecessary data movement between registers. Take the first loop part among multiple cases as an example, at line 16, `esi` register is used to fetch the value of `c` from the stack, and this part will choose to jump or not according to the `cmp` result of `c` and `m`. However, instead of directly using `esi` register, it moves `esi`'s value into `eax`. Lastly, to use `eax`, it keeps `eax`'s original value into `esi` for future restoring after the comparison. By contrast, LLVM code (line 15-17, Listing (2)) takes the task in a concise way: it fetches the value of `c` from the stack using `ebx` register and it directly addresses `m` from the stack in `cmp` instruction rather than wasting another register to fetch `m`. The value of `m` is fetched in every for loop iteration, so keeping `m` in a register is a waste.

As explained above, there are useless register swaps in Cranelift code. The assumption that it must preserve `esi`'s value according to the calling convention also does not hold because shortly `esi` is used for temporary store. Even though it may believe `esi` cannot involve `cmp` operation for a special reason, there are plenty non-volatile registers available for stack fetch.

The inferior machine code of Wasmtime is caused by Cranelift's expansion based machine instruction generator. It expands every Cranelift intermediate representation (IR) instruction until it finds the sequence of machine instructions to represent this IR. Such expansion-based legalization fails to generate efficient code because it lacks a global view.

*2) **Indirect memory address**:* Cranelift adopts inefficient memory address. Example can be found at line 64, instead of directly addressing the memory for the add instruction, that is, `add <mem>,<reg>`, it uses a register to read and write back, and thus brings unneeded instructions.

```c
int main(){
int m = 17, n = 17, p = 17, q = 17;
int first[m][n], second[p][q], mul[m][q];
for(int c = 0 ; c < m ; c++ ){
    for ( int d = 0 ; d < q ; d++ ){
        for ( int k = 0 ; k < p ; k++ ){
          mul[c][d] += first[c][k]*second[k][d];
        }
      }
    }
return 0;
}
```

Listing (1) matrix multiplication C code

```
...
;##allocate 4*m*n bytes on stack for first[m][n]
mov  edi,DWORD PTR [rsi+rcx*1+0x38]   ;edi = m
mov  r8d,DWORD PTR [rsi+rcx*1+0x34]   ;r8d = n
mov  DWORD PTR [rsi+rcx*1+0x28],esi
mov  DWORD PTR [rsi+rcx*1+0x24],edi
imul edi,r8d
lea  edi,[rdi*4+0xf]
and  edi,0xfffffff0
neg  edi                              ; edi = -4*m*n
lea  r9d,[rdx+rdi*1]
add  r9d,0xfffffffc0;first[m][n]'s stack end addr
...
;##1st loop for(int c = 0 ; c < m ; c++ )
mov  ebx,DWORD PTR [rdi+rcx*1+0xc]   ;ebx = c
cmp  ebx,DWORD PTR [rdi+rcx*1+0x38]
jge  253 <functionDef2+0x183> ; jump if c >= m
...
;##mul[c][d] += first[c][k]*second[k][d];
mov  ebx,DWORD PTR [rdi+rcx*1+0xc]
imul ebx,r8d                          ;ebx = c*n
lea  ebx,[r9+rbx*4]
mov  edx,DWORD PTR [rdi+rcx*1+0x4]
 ;edx = offset, 4(cn+k), in first[][]
lea  edx,[rbx+rdx*4]
 ;edx = first[c][k]
mov  edx,DWORD PTR [rcx+rdx*1]
mov  ebx,DWORD PTR [rdi+rcx*1+0x4]   ;ebx = k
imul ebx,r10d                         ;ebx = k*q
lea  ebx,[r11+rbx*4]
mov  esi,DWORD PTR [rdi+rcx*1+0x8]   ;esi = d
 ;esi = offset, 4(kq+d), in second[][]
lea  esi,[rbx+rsi*4]
 ;edx=first[c][k]*second[k][d]
imul edx,DWORD PTR [rcx+rsi*1]

mov  esi,DWORD PTR [rdi+rcx*1+0xc] ;esi = c
imul esi,r14d                      ;esi = c*q
lea  esi,[r15+rsi*4]
mov  ebx,DWORD PTR [rdi+rcx*1+0x8]   ;ebx = d
lea  esi,[rsi+rbx*4]
  ;esi= offset,4(cq+d), in mul[c][d]
  ;mult[c][d] += first[c][k]*second[k][d]
add  DWORD PTR [rcx+rsi*1],edx
```

Listing (2) partial X86-64 code from WAVM runtime (LLVM)

```
...
;##allocate 4*m*n bytes on stack for first[m][n]
rex mov esi,DWORD PTR [rbx+rdx*1+0x38] ; esi = m
rex mov edi,DWORD PTR [rbx+rdx*1+0x34] ; edi = n
rex mov DWORD PTR [rbx+rdx*1+0x28],ecx
mov    r8d,esi                         ; r8d = esi
imul   r8d,edi                         ; r8d = m*n
shl    r8d,0x2         ; r8d = r8d<<2, int 4 bytes
add    r8d,0xf
and    r8d,0xfffffff0
mov    r9d,ecx
sub    r9d,r8d      ; first[m][n]'s stack end addr
...
;##1st loop for(int c = 0 ; c < m ; c++ )
rex mov edx,ecx
rex mov esi,DWORD PTR [rbx+rdx*1+0xc]    ; esi = c
mov    r12d,DWORD PTR [rbx+rdx*1+0x38]  ; r12d = m
mov    r14d,eax               ;keep eax value, 0
rex mov eax,esi               ; eax = esi = c
mov    esi,r14d               ; esi = r14d = 0;
cmp    eax,r12d
setl   al               ; if c < m, al = 1
movzx  eax,al       ; extend higher bits with 0
and    eax,0x1
mov    r12d,eax         ; if c < m, r12d = eax = 1
rex mov eax,esi             ;restore eax value, 0
mov    esi,r12d  ; if c < m, esi = r12d =  eax = 1
rex test esi,esi              ;jmp if esi == 0
je     2f0 <_wasm_function_3+0x227>
...
;##mul[c][d] += first[c][k]*second[k][d];
rex mov esi,DWORD PTR [rbx+rdx*1+0xc]
imul   esi,edi
shl    esi,0x2                        ;esi = c*n
mov    r12d,r9d
add    r12d,esi
rex mov esi,DWORD PTR [rbx+rdx*1+0x4]    ; esi = k
shl    esi,0x2
add    r12d,esi                       ;r12d = 4(cn+k)
 ;esi = r12d = offset, 4(cn+k), in first[][]
mov    esi,r12d
rex mov esi,DWORD PTR [rbx+rsi*1]  ;esi = first[c][k]

mov    r12d,DWORD PTR [rbx+rdx*1+0x4]   ; r12d = k
imul   r12d,r8d                       ; r12d = k * q
shl    r12d,0x2
mov    r14d,r11d
add    r14d,r12d
 ;r12d = r14d = offset, 4(kq+d), in second[][]
mov    r12d,r14d
mov    r12d,DWORD PTR [rbx+r12*1];r12d = second[k][d]
imul   esi,r12d     ;esi = first[c][k] * second[k][d]

mov    r12d,DWORD PTR [rbx+rdx*1+0xc]
imul   r12d,r10d                      ; r12d =  c * q
shl    r12d,0x2
mov    r14d,r13d
add    r14d,r12d
mov    r12d,DWORD PTR [rbx+rdx*1+0x8]    ;r12d = d
shl    r12d,0x2
add    r14d,r12d
 ;r12d = r14d = offset,4(cq+d), in mul[c][d]
mov    r12d,r14d
mov    r14d,DWORD PTR [rbx+r12*1]  ;r14d = mult[c][d]
add    r14d,esi                   ;mul[c][d]+=result
mov    DWORD PTR [rbx+r12*1],r14d ;write back stack
```

Listing (3) partial X86-64 code from Wasmtime runtime (Cranelift)

Fig. 6: X86 Machine code generated by WAVM & Wasmtime from the same WebAssembly matrix multiplication bytecode

*3) **Inappropriate instruction choice**:* Cranelift also does not generate appropriate instructions as can be seen from multiple places at Listing (3). To take a jump, instructions start from line 22 to line 29 are involved, while LLVM only requires a `jge`, jump-if-greater-or-equal, instruction (line 17, Listing (2)). Cranelift uses jump-if-equal which is also good and should also be able to finish the task in one instruction, but it uses `eax` for `cmp` (line 21, Listing (3)), while decides to jump or not according to another register esi (line 28, Listing (3)). Moreover, line 24, uses `and eax, 0x1` on the result of `setl` which is already 0 or 1, so this `and` instruction does not have any effects at all.

Another type of inappropriate instruction choice is a meaningless `REX` prefix on many 32-bit instructions that do not use any of `r8-r15` registers. There are already several cases, such as line 26, `rex mov eax,esi` in such a short snippet Listing (3). This `REX` prefix makes this `mov` instruction one byte larger in size and thus results in higher i-cache footprint. It will exacerbate the front-end bottleneck and limit performance. While i-cache footprint cannot be dynamically measured to the best of our knowledge, the larger every instruction is in size, the less amount of instructions can be kept in I-cache and the following instruction fetch and decode phase would be hurt.

From the above analysis, it is clear that Wasmtime instructions are more redundant and have lower instruction locality than WAVM. To verify it, Perf provides events `ICACHE_64B.IFTAG_HIT` and `ICACHE_64B.IFTAG_MISS` to count the L1 instruction cache hit and miss number, respectively. However, It must be noted that there is also a micro-operation (uOps) cache (DSB) which acts as a "L0" I-cache that is not counted in the above two events. DSB (Decoded Stream Buffer), stores uOps that have already been decoded, avoiding many of the penalties of the legacy decode pipeline as known as MITE (Micro-instruction Translation Engine). To catch DSB data, we need `IDQ.MITE_UOPS` and `IDQ.DSB_UOPS` events.

Instruction decoder inside CPU breaks the instructions with variable length into equal length uOps. We randomly pick several PolyBench applications and report their DSB miss and L1 I-cache miss in table I. WAVM has considerably lower DSB miss rate than Wasmtime throughout the applications. For applications such as heat-3D, both runtimes shows low DSB miss rate. We do not consider the L1 I-cache miss rate because most instructions are from DSB, as the last column of the table I shows: DSB reference number (miss + hit) is much higher than L1 I-cache. Normally, for RISC, one instruction is processed into one micro operation, even with micro/macro operation fusion, the ratio still remains between 0.9 and 1.1.

To conclude, we first analyze the runtime backend performance from instruction level and then take the micro-architecture characterization and identify the front-end bottleneck of Wasmtime coming from instruction cache. The characterization shows agreement with the code analysis.

C. ARM case

At the time of writing, Lucet and WAVM do not support ARM aarch64 architecture. We are therefore only allowed to test Wasmtime and Wamser on a NVIDIA Jetson TX2 which is regarded as a powerful edge device. For clarity, it is not reported in the figure. The result still fits the pattern reported in Fig. 5 that is Wasmtime has marginal advantage over Wasmer especially for short-lived applications. Because Wasmer supports LLVM backend as well, we are able to compare the LLVM and Cranelift on ARM. LLVM still generates machine code of better quality, but the performance gap between the two compiler backends is smaller on ARM than on X86. An obvious difference is LLVM cannot exploit the direct memory address, because on ARM, memory contents need to be loaded into registers first before performing any logical or arithmetic operation. Therefore, ARM is more Cranelift-friendly in this respect. Also, the main goal of edge WebAssembly runtime is to provide lightweight service isolation at the edge server side. IoT or ARM device which has inferior performance than X86 server is currently not feasible to exploit.

IV. Startup latency

The question naturally arises: should Wasmtime optimizes its backend compiler? While WAVM has the best performance on most PolyBench WebAssembly applications where tens or even hundreds of seconds are consumed, there are also cases such as `atax` in Fig. 5 where WAVM shows the worst performance. This is because applications such as `atax` are short-lived and thus the startup latency cannot be ignored. This is especially important to the edge because the major goal of the edge is to assist the cloud and thus involves less computation-intensive tasks. Polybench cannot reflect this application feature.

Also, edge scenario tends to have stringent latency requirement. Without being able to provide fast startup, the benefits brought by fast edge network transmission will wane. In the Function-as-a-Service (FaaS) realm, considerable work has tried to tackle the startup latency issue because it is currently the most urgent one. [14], [15]. Here we refer to the WebAssembly application startup latency as the time spent on process bootstrapping and module compiling. Future instantiation will not suffer from this cold startup latency when the process keeps alive and the compiled module is preserved and shared. It has been shown that WebAssembly is an effective alternative to reduce the startup latency in a serverless prototype [5]. It is necessary to investigate how the emerging WebAssembly runtimes affects startup latency.

To show the effects of cold start on the overall run time, we break the run time into startup and execution parts in Fig. 7. The applications we select from PolyBench are `atax`, `bicg`, `durbin`, and `trisolv` because WAVM does not show good performance on these four applications as Fig. 7 shows. Fig. 7 (a) shows when there is only one invocation which is cold, because for WAVM, its higher startup latency dominates the overall time, the performance of WAVM is on average $3\times$ slower than Wasmtime. Keeping the process alive,

TABLE I: Micro-architecture events on instruction cache

Application	Runtime	DSB Misses	DSB Hits	DSB Miss Rate	L1 I-Cache Misses	L1 I-Cache Hits	$\frac{DSB\ Ref}{L1\ I-Cache\ Ref}$
2mm	WAVM	1.64E+09	4.44E+10	3.56%	3.53E+07	4.92E+09	9.297848921
	Wasmtime	3.98E+10	4.18E+10	48.74%	3.19E+06	6.92E+09	11.78195449
3mm	WAVM	1.69E+09	7.04E+10	2.35%	3.99E+07	9.14E+09	7.849410328
	Wasmtime	9.49E+10	3.51E+10	73.03%	5.55E+06	1.13E+10	11.51041366
symm	WAVM	1.59E+09	2.59E+10	5.80%	3.37E+07	3.41E+07	404.7020417
	Wasmtime	4.10E+10	2.80E+09	93.60%	1.69E+06	3.76E+09	11.62256781
heat-3D	WAVM	3.43E+09	1.83E+11	1.84%	3.90E+07	1.59E+10	11.73076489
	Wasmtime	3.02E+09	2.92E+11	1.02%	3.75E+06	2.02E+10	14.58986501
seidel	WAVM	1.81E+09	1.55E+11	1.15%	3.95E+07	1.47E+10	10.63568714
	Wasmtime	2.47E+11	1.23E+10	95.26%	5.67E+06	2.02E+10	12.82836407
gemm	WAVM	1.51E+09	3.71E+10	3.92%	4.33E+07	4.90E+09	7.817842999
	Wasmtime	3.03E+08	6.66E+10	0.45%	1.73E+06	5.56E+09	12.03330069

the following invocations will not experience cold start but just instantiate an existing compiled WebAssembly Module. This pattern is reported in 7 (b) where we sequentially call the applications for 50 times. In this scenario, the execution part dominates the whole time and WAVM outperforms Wasmtime. Because there is only one cold start and the following 49 invocations use the WebAssembly Module.

The above-characterized PolyBench still fails to unbiasedly reflect the advantages of Lucet and Wasmer because those applications are all computationally heavy for them, so we examine four runtime across different sized random matrix operation, and report the result in Fig. 8. This time Lucet shows the smallest inherent startup latency and thus the best performance when the input size is small, e.g., 30×30, but as the matrix grows, the time drastically increases, and this proves its poor performance on large scale application handling, while WAVM behaves oppositely. Wasmtime has the most balanced performance among the runtimes.

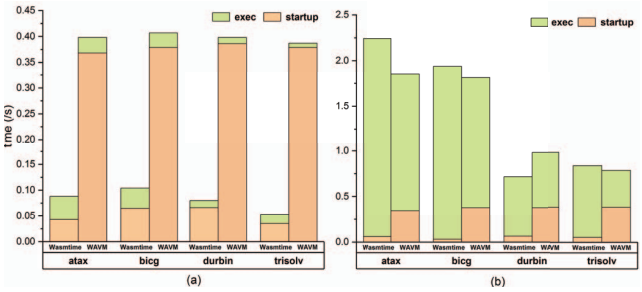

Fig. 7: Startup and execution time breakdown under different invocation pattern. (a) one clod invocation, (b) fifty sequential invocations

A. A set of typical edge workload & runtime scheduling

Numerous papers have shown the necessity of offloading cloud workload to the edge for saving network bandwidth and reducing latency [16], but for computationally heavy applications, the communication latency improvement may not

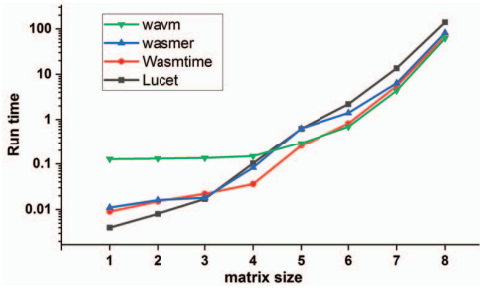

Fig. 8: Total run time on different-size 2D matrix multiplication over four WASI-native WebAssembly runtimes. From the left to the right, matrix size increases from 30×30 to 2000×2000

be as significant as the computation time benefits brought by the remote but powerful cloud servers. Typically, if the time required by an application is more than one second, offloading the application to the cloud instead of the edge is more reasonable because the internet round trip time is within this range while we can ignore the 5G network latency between the user and an edge node.

Instead of replacing containers or virtual machines, WebAssembly aims for enabling edge servers to respond to requests from end users promptly and safely. PolyBench is more of a cloud-side benchmark which is ideal for compilation performance comparison. Therefore, we propose a typical edge benchmark that consists of tasks common in the autonomous driving scenario. They are (1) edge detection: a Canny edge detector that uses a multi-stage algorithm to detect a wide range of edges in an image; (2) vehicle plate detection: isolate the license plate part using morphological image processing algorithms; (3) attitude Estimation: fast complementary filter for car attitude estimation using a set of local data collected from MARG (Magnetic, Angular Rate, and Gravity) sensors; (4) Image classification: MNIST digit classification using LeNet. (5) Decompressing: a 1MB zip file decompressing via

Huffman method. As can be found from the Fig. 9, no runtime can always maintain the best performance across five real-world applications.

Deciding the best runtime is therefore difficult. However, according to our targeting scenario, it is reasonable to assume that we know what applications the WebAssembly runtime will serve. For example, for an autonomous driving service provider, it is their responsibility to decide which application(s) should be dispatched to a server or a base station. We, therefore, are allowed to implement inexpensive offline profiling as Fig. 9 does.

The scheduling diagram is shown in Fig. 10 where we choose Rust language as the host (embedder) because Rust has good WebAssembly support. The host parses the request and uses a pre-loaded profiler to decide the best embedded runtime. Currently, there are Wasmtime, Wasmer, and Lucet runtimes that are published as Rust crates available. The embedded runtimes are exposed to the Rust host and provide API such as `Module::new` for developers to call. We are therefore able to compile the WebAssembly binary into the module and then instantiate the module with the imports imported, and finally run it from the host environment. However, Lucet is special because it uses the ahead-of-time compilation so it can be directly instantiated without the compilation step. We monitor the total run time and use the results to dynamically update the profiler because of the correlation between physical time and profiling. This empirical scheduler can effectively reduce the cold start latency. We leave a more comprehensive scheduler that combines invocation patterns with run time as our future work, but the basic idea is the Rust host allows us to store the read-only compiled artifact that can be safely shared, and thus the future instantiations will not require bootstrap, so the prewarming should be as lucrative as possible (choosing the best machine code generator) but also in a resources-and-time-allowed fashion.

in Table II, we report the scheduler performance on the five edge applications we described above. The result is the average of speedup percentage of hybrid mode per application because short-lived applications should have the same weight on the result as the long-lived applications. The speedup ranges from $1.5\times$ to $16.9\times$ which is striking.

TABLE II: The speedup of using hybrid mode compared to the single runtime mode on the edge workload set

	WAVM only	Wasmtime only	Lucet only	Wasmer only
Hybrid runtimes	$16.9\times$	$2.8\times$	$1.5\times$	$3.8\times$

V. Architecture implication

The challenge arisen in III-B is an inefficient runtime instruction generator, while from IV we find the tradeoff between the instructions and startup latency brought by the compiler improvement (e.g., Wasmtime vs WAVM). As a result, optimizing the runtime backend from a software compiler level may breach the fast startup principle of an edge WebAssembly runtime. However, inspired by the front-end bottleneck identification in III-B, we investigate how does hardware optimization improve the system while not comprising the startup latency.

TABLE III: Branch misprediction rate (%) under different branch predictors

Application	Runtime	2-bit	Bi-mode	Tournament
2mm	WAVM	22.15	6.91	2.73
	Wasmtime	16.47	5.40	1.92
symm	WAVM	24.83	6.23	3.28
	Wasmtime	18.31	6.54	4.03

To observe the hardware extension effects, we use gem5 simulator which relies on a pre-built custom disk image with the applications installed to boot. We characterize the I-cache and D-cache performance over the cache size change and report the result in Fig. 11. Both the I-cache and D-cache miss rates drop when the cache size increases. However, I-cache size has a more obvious effect on performance than D-cache. Also, two rumtimes show a similar D-cache miss rate across the different D-cache sizes. We, therefore, argue that data cache size is less important for Wasmtime, but instruction cache can be exploited. The Wasmtime applications show a much higher I-cache miss rate than WAVM applications as Fig. 11 (a) shows, but it is no longer a major concern when I-cache size is large. Table III shows branch misprediction rates using different predictors. The predictors include a simple 2-bit local predictor, a bi-mode predictor, and a tournament predictor. We set branch target buffer 2048 entries, size of global and choice predictor both 2Kb for the bi-mode, and size of local, global and choice predictor 1Kb, 4Kb, 4Kb respectively for the tournament. We claim that unlike cache, branch predictor leaves less leeway for optimization because the complicated predictor such as Tournament has already shown good results. We expect other architecture implications such as cache associativity and block size can also be exploited.

To conclude, though optimizing the backend compiler may harm the performance for short-lived applications, hardware enhancement provides new opportunities. To exploit the potential of each runtime, specialized hardware configurations are needed.

VI. Related work

Previous work on server-side WebAssembly focuses on its security and feasibility.

Safety. To investigate the safety of using WebAssembly as the lightweight isolation, Lehmann et al. analyze to what extent are vulnerabilities exploitable in WebAssembly binaries, and how does this compare to native code [17]. Extensions to WebAssembly that enables higher security and subsequent performance trade-offs are discussed in [18], [19].

Serverless prototype. Hall et al. show the potential of WebAssembly to the edge by comparing it with containers, but their design brings security and flexibility concerns [20]. Shillaker and Pietzuch implement a WebAssembly-based new

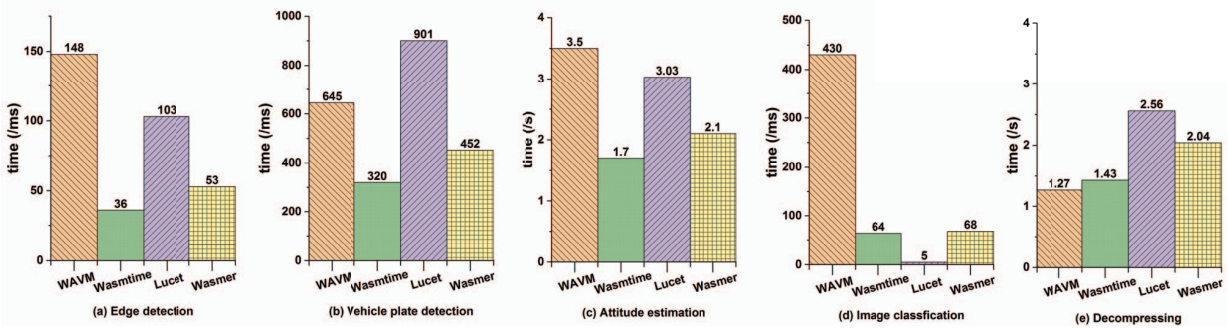

Fig. 9: Performance of the WASI-native WebAssembly runtimes on a typical edge workload set

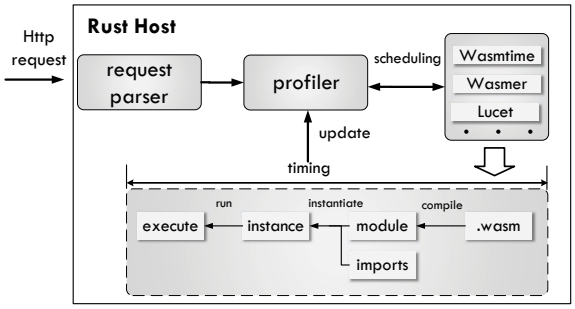

Fig. 10: A hybrid execution scheduler for tackling the cold start latency

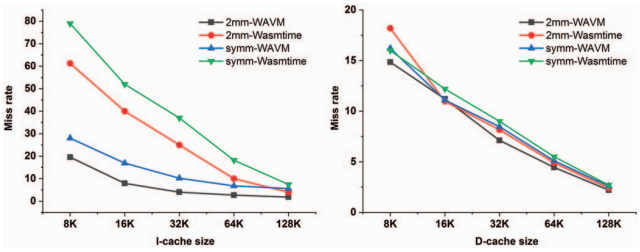

Fig. 11: I-cache and D-cache miss rate under different cache sizes

isolation abstraction for high-performance serverless computing [7]. Their work all show WebAssembly-based cloud service has advantages on serving latency.

Characterization. Jangda et al. compare the performance of WebAssembly running on browser over native C code [12]. Kent et al. investigate the performance of WebAssembly compilation in V8 [21]. Radhakrishnan et al. characterize and analyze the JAVA runtimes [22]. However, this paper is the first work on systematically examining the cloud WebAssembly runtimes and analyzing the implication from application scenarios, compiler backend, and hardware architecture level.

VII. Conclusion and Future Work

In this paper, a systemic characterization of major edge WebAssembly runtimes is presented. We discuss the runtime performance under different application scenarios. We analyze the root cause of different runtimes behavior. Based on the characterization and analysis, practical guidance for selecting the runtime as per target applications is given and the hardware implication on balancing the execution and startup is discussed. We leave a hardware and software co-optimized framework for efficiently handling edge WebAssembly function service as our future work. We also plan to further the study on WebAssembly performance and backend compiler.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grants CCF-1822985 and CCF-1943490 (CAREER). Yang Hu is the corresponding author.

References

[1] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[2] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, Xin Peng, Wenli Zheng, and Minyi Guo. Qos-aware and resource efficient microservice deployment in cloud-edge continuum. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 932–941. IEEE, 2021.

[3] Xiaowen Chu, Hongbo Jiang, Bo Li, Dan Wang, and Wei Wang. Advances in mobile, edge and cloud computing. *Mobile Networks and Applications*, pages 1–3, 2020.

[4] Lu Zhang, Chao Li, Pengyu Wang, Yunxin Liu, Yang Hu, Quan Chen, and Minyi Guo. Characterizing and orchestrating nfv-ready servers for efficient edge data processing. In *Proceedings of the International Symposium on Quality of Service*, pages 1–10, 2019.

[5] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 225–236, 2019.

[6] Zhendong Wang, Zhen Wang, Cong Liu, and Yang Hu. Understanding and tackling the hidden memory latency for edge-based heterogeneous platform. In *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[7] Simon Shillaker and Peter Pietzuch. Faasm: lightweight isolation for efficient stateful serverless computing. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 419–433, 2020.

[8] Latent AI. Exploring webassembly ai services on cloudflare workers. https://blog.cloudflare.com/exploring-webassembly-ai-services-on-cloudflare-workers/, 2020.

[9] Fastly. Experiment, innovate, and step into future of the edge. https://www.fastlylabs.com/, 2020.

[10] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. ewasm: Practical software fault isolation for reliable embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3492–3505, 2020.

[11] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[12] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not so fast: analyzing the performance of webassembly vs. native code. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 107–120, 2019.

[13] Rust. Environment affecting on a single webassembly function translation. https://docs.rs/cranelift-wasm/0.24.0/cranelift_wasm/trait. FuncEnvironment.html, 2020.

[14] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

[15] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.

[16] Zheng Dong, Yan Lu, Guangmo Tong, Yuanchao Shu, Shuai Wang, and Weisong Shi. Watchdog: Real-time vehicle tracking on geo-distributed edge nodes. *arXiv preprint arXiv:2002.04597*, 2020.

[17] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 217–234, 2020.

[18] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2019.

[19] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. Swivel: Hardening webassembly against spectre. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[20] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys–efficient in-process isolation for risc-v and x86. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1677–1694, 2020.

[21] Tobias Nießen, Michael Dawson, Panos Patros, and Kenneth B Kent. Insights into webassembly: compilation performance and shared code caching in node. js. In *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*, pages 163–172, 2020.

[22] Ramesh Radhakrishnan, Narayanan Vijaykrishnan, Lizy Kurian John, Anand Sivasubramaniam, Juan Rubio, and Jyotsna Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on computers*, 50(2):131–146, 2001.