

Valent-Blocks: Scalable High-Performance Compilation of WebAssembly Bytecode For Embedded Systems

Fabian Scheidl
BMW Group AG
Munich, Germany
Technische Universität Wien
Vienna, Austria
fabian.scheidl@bmw.de

Abstract—In the field of emerging software architectures, there has been a dramatic push towards flexible and sandboxed software modules that allow systems to safely execute untrusted code in a guaranteed side-effect free manner. Latest developments have further given rise to portable and statically validatable representations of software in a bytecode format like WebAssembly. In order to ease the segue into the domain of embedded systems, this paper explores the feasibility of a novel and easily retargetable streaming quasi-singlepass on-target-compiler topology with concurrent bytecode validation. For this, a generalized compile-time virtual stack is employed which is logically partitioned into separately emittable blocks (named valent-blocks). This forms the foundation of a corresponding runtime for resource constrained systems that demonstrate the need for predictable, resource-efficient and fast sandboxed execution of hot-loaded software. This paper further benchmarks the resultant performance against current popular competing standalone WebAssembly runtimes.

Index Terms—retargetable compiler, internet of things, safety, sandbox, fault isolation, virtual machine, runtime, webassembly, performance, embedded, energy consumption, memory, resource constrained, sensor nodes

I. INTRODUCTION

In the field of embedded systems, software has been traditionally cross-compiled and linked for the respective architecture and monolithically flashed to the microcontroller. Due to the increasing complexity of systems those embedded devices find applications in, the necessity for the modularization and flexibilization of embedded software has emerged [1]. Modularization in this sense strives to enable flexible updates of certain software components, while at the same time encapsulating logical modules and thus keeping the remaining functionality untouched and working as intended. In order for this process to be guaranteed not to interfere with the rest of the system, strict process separation and well-defined interfaces are imperative.

Due to the inherent non-portability, non-flexibility and concomitant management and integration overhead of those solutions, lightweight virtualization options have been explored [2]. This development has gained momentum in the field of desktop browser systems which allow websites to locally execute untrusted code. Until more recently, websites could only load JavaScript code that would be executed by an interpreter or a just-in-time compiler. Current browser developments focus on technologies like WebAssembly, which represents a compact, portable and (generally) precompiled bytecode format of a virtual instruction set that can be

more efficiently compiled and executed than JavaScript. WebAssembly is further statically validatable in an efficient manner, thus drastically reduces the potential attack surface, and guarantees inherent sandboxing with strictly defined communication to the runtime [3]. The small instruction set and strict semantics of WebAssembly allow for both a minimal implementation and humble resource requirements.

This push in the direction of nanoproceses is similarly relevant for embedded systems, which often do not offer any hardware virtualization features, memory management units or might not run an operating system. Interpreting virtual machines, which often represent portable runtimes and offer effortless integration into existing systems, have been shown to be feasible for implementation and deployment on embedded systems [4]. However, interpreters are associated with a significant instruction dispatch overhead and have traditionally incurred a high rate of CPU branch mispredictions. (Due to improved branch prediction algorithms, modern desktop CPUs do not suffer from this problem anymore [5], unfortunately this does not generally apply to embedded systems) Even after optimizing the dispatch (e.g. by utilizing modern threading techniques), performance of interpreters is still orders of magnitudes slower than native execution [6], [7]. For embedded systems, this leads to significantly increased hardware costs and implies high power consumption and, in the case of battery-powered devices, decreased battery life.

The possibility of bringing on-target compilers to embedded devices has also been researched [8], [9]. Now, with the advent of WebAssembly, and with it an efficiently compilable, standardized and portable representation of software, this approach could ameliorate a number of problems of virtualization solutions for embedded systems.

II. CONTRIBUTION AND RELATED WORK

Based on the specification of WebAssembly [3] and Google's V8 baseline WebAssembly compiler Liftoff [10] we propose an extended minimal streaming quasi-singlepass compilation logic for WebAssembly that forgoes complex optimizations and keeps compile-time memory consumption to a minimum. For this, the syntactic restrictions of WebAssembly are leveraged to introduce the logical compilation-stage concept of valent-blocks.

V8 Liftoff already employs a compile-time stack for abstraction purposes, however we improve on that idea with

a more general representation of a compile-time operand stack that can record WebAssembly’s arithmetic operations as well, thereby essentially pushing pure expression trees from the instruction stream to the stack before generating code on-the-fly. Code is then only lazily emitted once certain trigger conditions are met.

Early pursuits of bringing WebAssembly to embedded systems have so far mostly focused on interpreting techniques [4], [11], [12]. Intel has released an open source standalone (interpreting or off-target ahead-of-time compiling) WebAssembly runtime called WebAssembly Micro Runtime (WAMR) for embedded devices [13].

Lightbeam is a minimal streaming singlepass compiler which is part of the wasmtime runtime and could technically be employed on embedded systems. Lightbeam uses an intermediate compilation step to Microwasm, a WebAssembly-compatible format that leverages the mechanisms of a pure stack machine without local function-scoped variables [14].

In contrast to this, valent-block compilation builds and consumes the stack-based intermediate representation concurrently, therefore minimizes memory consumption during compilation and is thus better suited for resource-constrained embedded devices.

III. VALENT-BLOCK COMPILATION

A. WebAssembly

WebAssembly (Wasm) is an open W3C standard which defines a stack-based minimal virtual instruction set for executable programs [3]. WebAssembly was originally conceived to provide fast execution speeds for computationally intensive tasks in Webrowsers and emerged from asm.js with contributions from Google’s Portable Native Client (PNaCl) [15]. WebAssembly further provides trivially mechanizable static validation, is strictly typed, prevents control-flow hijacking and offers inherent memory sandboxing and well-defined interfacing.

B. Goals and Limitations

The following sections shall provide a solution for on-target compilation of WebAssembly modules for embedded systems. This implicates that the compiler has to cope with existing restrictions and limitations of embedded systems. The proposed compiler shall further guarantee adequate sandboxing of the resulting executable program and make use of only minimal compiler logic.

Our assumption is that the structure and likeness of WebAssembly to commonly used instruction set architectures leaves most compiler optimizations redundant on the layer of WebAssembly, insofar as the upstream compiler toolchain (e.g. LLVM, a popular modular compiler infrastructure which can produce WebAssembly) already performs aggressive optimizations and most of those carry over downstream to on-target compilation activities. Fig. 1 shows the overarching toolchain involving multiple compilation stages. A program is translated to WebAssembly via a complex multipass optimizing compiler (e.g. LLVM) and is further optimized by an optional extra optimization pass for improved compilability via valent-block optimization. (See also Section III-D6) The WebAssembly bytecode is then transferred to the target platform (e.g. an embedded system),

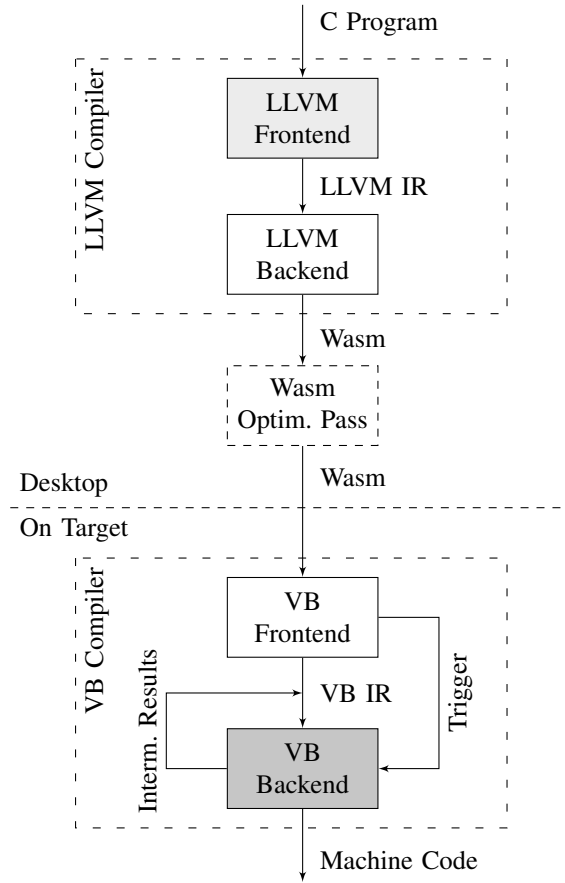


Fig. 1: Compiler toolchain flowchart

where it is compiled by the proposed compiler, which, analogously to the modular structure of LLVM, is split into a portable frontend and a platform-specific backend for easier retargetability. The valent-block intermediate representation (VB IR) is produced by the valent-block frontend and concurrently consumed by the backend. Backend machine code emission (and thus consumption of the IR) is triggered when the frontend encounters specific instructions. Intermediate results are returned onto the compile-time stack (IR), such that no additional memory has to be allocated during compilation.

A compiler topology involving only minimal compilation logic and limiting optimizations to only those which might have to be reverted in order for an intermediate representation like LLVM IR (Intermediate Representation) to be converted to WebAssembly, should be able to provide adequate performance while reducing the footprint of the compiler.

Conversely, this compilation logic does neither strive to produce size-optimized machine code nor highly efficient code for unoptimized input WebAssembly modules.

C. Non-Optimizing Compiler

Trivially, the constraints for employing an embedded on-target-compiler can be satisfied by utilizing a non optimizing singlepass compiler which sequentially translates WebAssembly instructions to the target architecture and stores all variables on an actual in-memory runtime stack. This way of compilation has the lowest possible requirements concern-

TABLE I: Comparison of resultant program footprint and execution time among different compilers for fib on x86-64

Benchmark	Non-optimizing Compiler	Valent Block	Clang
Execution time	9.03s	1.98s	1.82s
Footprint (ROM)	377B	142B	79B

ing RAM usage during compilation, however the resulting footprint and performance (and thus power consumption) are less than optimal. Table I shows a comparison of the execution time and program footprint among such a non-optimizing compiler, proposed valent-block compilation and LLVM’s Clang. (For more info see Section IV).

A major source of overhead when compiling WebAssembly modules can be traced back to redundant stack interaction. Register spills and loads (offloading CPU registers onto an in-memory stack) should be kept to a minimum in order to reduce the incurred overhead. This is especially true for constants that are pushed onto the stack only to be popped into a register in the next cycle.

D. Proposed Compiler Logic: Valent-Blocks

We base the following research and implementation on the logic that Google’s V8 Liftoff baseline compiler uses. V8 Liftoff is a singlepass compiler that strives to do sequential code generation soon after parsing each WebAssembly instruction. The compiler keeps track of a virtual compile-time WebAssembly stack that only exists during the compilation phase. For integer constants, for example, Liftoff only records the corresponding value in the virtual stack and does at that point not generate any code. When this constant is then used by a subsequent WebAssembly instruction, concrete machine code for that operation is emitted [10]. This avoids pushing the value onto an actual runtime in-memory stack only to pop it on the next instruction. We propose that performance of this general strategy can be further improved by classifying WebAssembly instructions based on various characteristics and generalizing the compile-time stack to include arithmetic instructions and thus effectively recording pure expression trees from the instruction stream. The compile-time stack can be partitioned into valent-blocks, which are then hierarchically and lazily emitted in a LIFO (Last In First Out) manner only when an instruction is encountered that leads to side-effects not affecting the WebAssembly runtime stack. Due to the fact that we not only have to parse the input stream but also have to traverse the virtual compile-time stack, we classify this method as being *quasi*-singlepass as opposed to a true singlepass-compiler.

1) *Classification of WebAssembly Instructions*: WebAssembly instructions exhibit certain characteristics which we will discuss in the following paragraphs. A classification according to those properties naturally leads to the idea of valent-block compilation.

a) *Valence of WebAssembly Instructions*: Classification can be done according to what we will call the valence or arity of a WebAssembly instruction. Valence in this context is an inherent feature of WebAssembly instructions

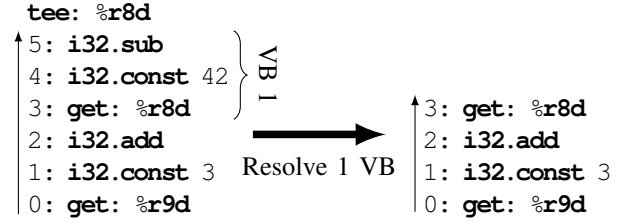


Fig. 2: Resolution of a valent-block

and corresponds to the number of stack elements a specific instruction consumes. For example: **i32.add** consumes two 32-bit integer values from the stack and returns a single 32-bit integer onto the stack. Thus we can classify the valence as two, whereas **i32.sqrt** exhibits a valence of one while only consuming a single stack element.

b) *Non-Stack Side-Effects*: WebAssembly instructions can be classed as leading to *non-stack* side-effects or as being side-effect free. Numeric instructions, conversions and reinterpretations only affect the WebAssembly stack and are thus inherently side-effect free. Side effects describe effects that transcend effects on the stack. For example, numeric instructions and the polymorphic instructions **drop** and **select** only affect variables that are present on the stack and are thus classified as (non-stack) side-effect free. Conversely, control flow instructions like **br**, **br_if**, **call** or those which modify the linear memory do exhibit non-stack side-effects by affecting linear memory or modifying the program counter.

2) *Valent-Blocks*: The preceding classifications lead to the concept of valent-blocks. Valent-blocks exist on the virtual compile-time stack and (logically) manifest as soon as side-effect afflicted instructions request their resolution. Each valent-block is composed of one or multiple WebAssembly instructions and represents an abstraction of live variables. Valent-blocks can be arbitrarily nested and stacked. (See also Fig. 4) At any point in time, a valent-block represents a single logical stack element once every contained instruction has been emitted and merged with the respective adjacent instructions. Subsequent WebAssembly operations logically affect the result of valent-blocks.

Fig. 2 shows a virtual compile-time stack with 6 elements. Once a side-effect afflicted instruction is encountered (**local.tee**, popping the top value into a variable and immediately pushing it back to the stack, in this case targeting a local variable currently living in register **%r8d**), the valence of the triggering instruction is determined. Due to the valence being 1 (**local.tee** consumes one element from the stack), the compiler resolves the uppermost valent-block, which in this case does not have any sub-valent-blocks and comprises three virtual stack elements. The corresponding machine code sequence is then emitted and the valent-block is popped from the stack. Due to the fact that **local.tee** sets and immediately gets the targeted local again, a **get** instruction for the targeted register is pushed back onto the virtual stack.

Due to the inherent laziness of this approach, resulting in delayed and on-demand resolution of valent-blocks, the compiler waits with emission of machine code until all potentially mergeable instructions have been processed.

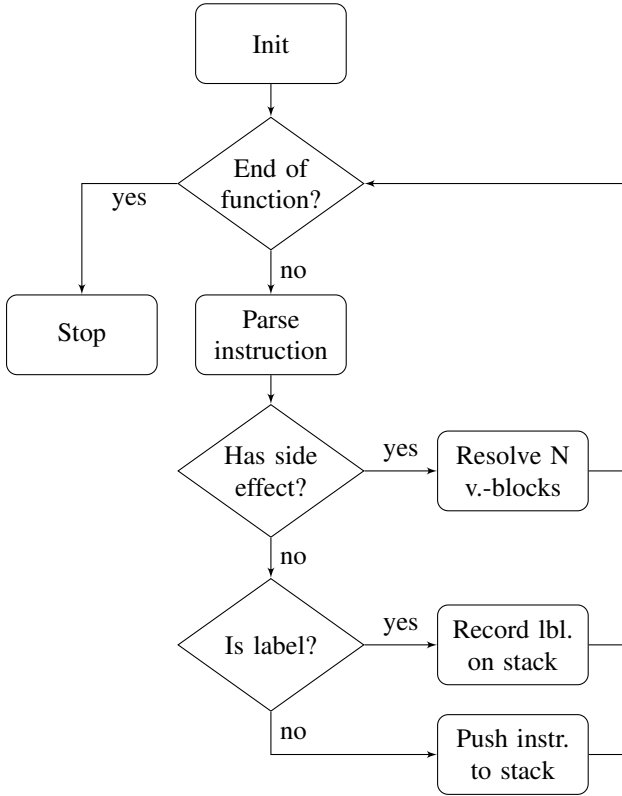


Fig. 3: Code generation process

3) *Code Generation*: Once a side-effect afflicted instruction is encountered, the respective number of the valent-blocks, matching the valence or arity of that instruction, is resolved. Upon resolution of valent-blocks, the corresponding machine code instructions are first saved to an intermediate storage in RAM and then written to flash memory. The general compilation process outlined in the next paragraphs is illustrated in Fig. 3. Backward-branch targets are effectively no-op. Their machine code offset is recorded as a reference on the compile-time stack for later branches targeting that label.

Valent-blocks can be resolved in a recursive fashion and are to be resolved from inner blocks (the smallest child) to outer blocks. The results of the emission of inner sub-blocks are either kept in a CPU register or in memory on the runtime stack. An abstract element representing the storage location of that result then replaces the valent-block on the virtual stack. Outmore blocks can then access that resulting abstraction as though it was a single native input element.

To better illustrate the process of resolving valent-blocks, Fig. 4 shows a sample WebAssembly module and its intermediate virtual stack states during compilation being compiled to x86-64 AT&T syntax assembly code.

4) *Concurrent Validation of WebAssembly Modules*: Since abstractions of WebAssembly instructions are recorded on a virtual stack for compilation, integrity of the WebAssembly module can be easily and concurrently validated. If a discrepancy (Type mismatch, stack underflow or a wrong number of return values) is observed, it is concluded that the module is malformed and compilation is then to be aborted.

5) *Optimizations*: We postulate that most of the optimizations that are carried out by an upstream compiler (like LLVM) translating a higher language program to WebAssembly persist until the WebAssembly stage. Valent-block compilation in essence applies peephole optimizations, merges instructions and thus minimizes register loads and spills.

6) *Register Allocation Strategy*: Due to the fact that we limit this compiler to a single pass over the input instruction stream, register allocation passes and live variable analyses are not possible. For this reason, we employ an optional offboard optimization pass (See Fig. 1) that analyzes the hotness and lifetime of the WebAssembly functions' local variables and reorders them in descending order of hotness. The on-target compiler can then allocate CPU registers to function locals (variables) in order. A predefined number of registers is kept unallocated as scratch registers. Any local function variables exceeding the number of available CPU registers are kept on the runtime stack. With LLVM performing register precoloring for local function variables, this strategy leads to impressive performance with virtually absent on-target overhead.

a) *Branches*: Besides function calls, WebAssembly allows branches in two ways: Forward branches to the end of a block and backward branches to the label of a loop. Backward branches however are relatively easy to resolve: The machine code offset of the occurrence of the label is recorded and kept in memory. Once a branch to that label is encountered, a branch to that address is emitted. The closing of open blocks and loops follows a guaranteed LIFO mechanism and can therefore be easily represented by a stack-like data structure.

Forward branches represent a greater inconvenience for compilers that do not keep the entirety of the emitted machine code in memory. In order to save flash write cycles on embedded devices, we strive to write flash in only a strict sequential manner. Therefore, once a machine code flash page is written, the address cannot be inserted into that page anymore.

Once a forward branch is detected, a dummy branch to the null pointer is emitted. If the corresponding branch label is parsed while the temporary (in-memory) machine code page still contains the instruction pointing to that label, we proceed analogously to backward branches and simply replace the corresponding null pointer with the address of the forward branch label. Otherwise, we record a forward branch ID in a table. Once this branch ID table exceeds an allotted portion of RAM for intermediate storage, the emitted machine code is interleaved with that branch table and written to flash. The branch is then emitted as one or multiple indirections, indexing the branch ID onto the interleaved branch table.

IV. BENCHMARKS

In order to be able to evaluate the performance (in this analysis we restrict this assessment to execution time) of the proposed logic, a number of highly computationally intensive calculations are carried out and compared to popular WebAssembly runtimes and native compilation.

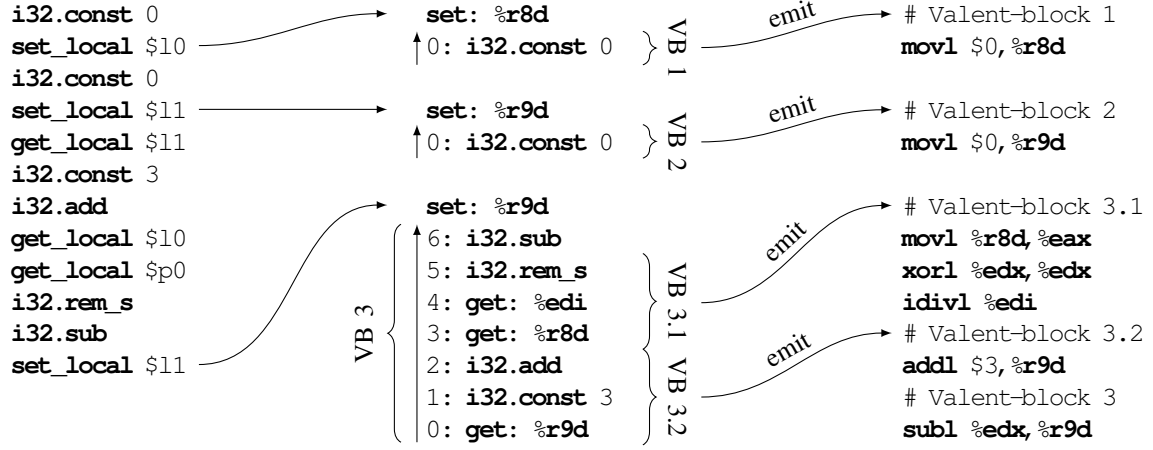


Fig. 4: A sample WebAssembly sequence (left), its corresponding virtual stack states during compilation (middle) and, for human readability, the equivalent AT&T syntax x86-64 assembly to the generated machine code (right)

A. Methodology

The WebAssembly modules were compiled from C code via Clang/LLVM (v8.0.1) to its 32-bit WebAssembly target. Those same programs were also directly compiled to target machine code using Clang and ARM GCC for baseline comparison. The x86-64 benchmarks were executed under macOS 10.14.5 on an Intel i7 8850H processor. In order to measure performance both on a superscalar CPU (like a modern x86 processor) and a representative embedded CPU with mere in-order-execution, a development board based on an 84MHz Atmel SAM3X8E (ARM Cortex M3, Thumb-2 ISA) was used. Due to flash alignment affecting performance on that CPU, the alignment leading to fastest execution was chosen for each benchmark. Bounds checking (for the WebAssembly stack and linear memory) was disabled for the ARM Cortex M3 benchmarks in order to allow comparison of raw performance. Bounds checking is, unlike on x86-64, necessary due to the lack of virtual memory (and thus lack of efficient sandboxing via guard pages). Problem size was adjusted for ARM to account for the significantly slower CPU and for less available RAM. Input data for programs was chosen in such a way that execution time is orders of magnitude greater than startup time. Startup time can therefore be neglected. Execution times are listed in Table II.

1) Benchmarked Runtimes and Compilers:

- **V8 Liftoff, V8 Turbofan (v8.0.317):** WebAssembly baseline and optimizing compilers, respectively, in Google’s V8 JavaScript engine.
- **Wasmer Singlepass/Cranelfit/LLVM (v0.10.1):** Standalone desktop WebAssembly runtime with three different compiler backends.
- **Lightbeam (Wasmtime v0.9):** Part of the Wasmtime repository, a streaming compiler that uses a custom intermediate representation called Microwasm.
- **Clang (v10.0.0):** The C-language compiler included in the LLVM project. Highest optimization level enabled.
- **GCC (ARM, v4.8.3):** GNU Tools for ARM Embedded Processors. Highest optimization level enabled.

2) Benchmarks:

- **fib:** Calculates the Fibonacci sequence by recursion.

- **mandelbrot:** Calculates a portion of the Mandelbrot set and stores the resulting bitmap in memory.
- **pollardrho:** Performs the Pollard Rho integer factorization algorithm for a set of numbers.
- **nussinov, floydwarshall, jacobi1d, correlation:** Arbitrarily selected benchmark programs from the Poly-BenchC benchmark suite.

B. Results

As was expected, the performance of V8 Turbofan was consistently better than V8 Liftoff. The same trend can be observed for the three different Wasmer backends with LLVM being the most mature and fastest and the Singlepass backend being the slowest. Valent-block compilation fares on average only 21% slower than the highly optimized version of Clang (with a maximum overhead of 87%) and shows the best average performance among the benchmarked compilers. We attribute that edge over the competing compiler solely to the simplicity of the tests and highly optimized input WebAssembly modules. V8 Turbofan shows the second best (38% overhead) and Wasmer LLVM the third best (53% overhead) average performance. The two other singlepass compilers V8 Liftoff and Wasmer Singlepass exhibit an average overhead of 166% and 261%, respectively, whereas Lightbeam has an average overhead of 129%. Unfortunately, we were unable to execute mandelbrot, fib and nussinov with Lightbeam due to seemingly erroneous compilation and subsequent errors. For fib, this was solved by manually fixing the machine code, which was not possible for the benchmarks without considerable changes that would have significantly impacted execution time. The results show that even on RISC embedded CPUs with in-order-execution, like the used ARM Cortex M3, performance of valent-block compilation lies in the order of native execution with an average slowdown of only 21%.

V. FUTURE WORK

We plan to extend the benchmarks to provide a better view of the strengths and weaknesses of this compiler topology. This includes increasing the complexity of the benchmark

TABLE II: Execution times (mean of 10 runs, rel. standard deviation consistently less than 1%), best result for each row and platform is marked in bold. Average performance and performance range are given relative to Clang and GCC. Problem size adjusted for ARM to account for a slower CPU and less available RAM limiting the largest possible result set size.

Benchmark	x86-64								ARM Thumb-2	
	V8			Wasmer					VB	GCC
	VB	Liftoff	Turbofan	Singlepass	Craneflight	LLVM	Lightbeam	Clang		
fib	1.98s	3.43s	3.13s	3.93s	2.97s	2.05s	2.14s	1.82s	1.93s	2.09s
mandelbrot	2.93s	3.71s	2.78s	5.02s	3.14s	6.00s	–	2.72s	3.80s	3.89s
pollardrho	2.59s	2.78s	2.54s	2.95s	2.75s	2.14s	2.69s	2.51s	2.33s	1.97s
nussinov	5.25s	15.14s	6.88s	14.63s	10.17s	5.72s	–	5.31s	0.24s	0.15s
floydwarshall	3.83s	14.37s	4.38s	21.85s	12.63s	5.12s	12.70	2.05s	1.54s	0.90s
jacobi1d	2.82s	5.98s	2.84s	10.45s	4.80s	3.98s	7.53s	1.94s	3.53s	3.36s
correlation	5.79s	7.97s	5.97s	7.59s	6.49s	5.40s	8.27s	5.96s	2.04s	2.00s
Avg. perf.	1.21	2.66	1.38	3.61	2.22	1.53	2.29	1.00	1.21	1.00
Perf. range	0.97–1.87	1.11–7.01	1.00–2.14	1.18–10.66	1.09–6.16	0.85–2.50	1.07–6.20	1.00–1.00	0.92–1.71	1.00–1.00

programs insofar as to include function calls to the runtime, passing complex data to and from the runtime and assessing the actual overhead imposed by WebAssembly’s sandboxing mechanisms on different platforms. We further want to benchmark actual compilation speed and memory consumption during both the compilation and execution phase. Additionally, we intend to develop a formal proof of correctness for the proposed compilation logic and want to provide an analysis of the persistence of different compiler optimizations across the various down- and upstream compiler stages. We also aim to retarget the compiler to other common CPU architectures (e.g. AArch32, AArch64, Power Architecture).

VI. CONCLUSION

By employing lazy machine code emission, instruction merging and peephole optimization, in a manner that takes advantage of the strict structure of WebAssembly modules, a highly efficient and compact streaming quasi-singlepass compiler can be designed. This compilation logic is well suited not only for desktop-, but most notably for embedded systems.

Our benchmarks confirm that optimizations performed by an upstream compiler carry over to WebAssembly bytecode and that minimal compilation logic, building on pre-optimization and inherent restrictions of structured programming languages, can produce highly efficient executables with predictable performance. In these benchmarks, valent-block compilation fares exceptionally well against popular WebAssembly runtimes and we expect future benchmarks (specifically those assessing memory consumption and performance during compilation) to also produce favorable results and to be equally competitive.

We thus conclude that valent-block compilation is a feasible option to bring on-target compilation and nanoprocess virtualization to embedded systems. This can act as an enabler for future developments in the field of minimal virtualized sandboxed execution environments and allows efficient execution of untrusted code on resource-constrained systems.

REFERENCES

- [1] F. Marchiò, B. Vittorelli, and R. Colombo, “Automotive electronics: Application technology megatrends”, in *Proc. 44th Europ. Solid*

- State Device Research Conf. (ESSDERC)*, 2014, pp. 23–29. [Online]. Available: <https://doi.org/10.1109/ESSDERC.2014.6948749>
- [2] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, “Consolidate IoT edge computing with lightweight virtualization”, *IEEE Network*, vol. 32, no. 1, pp. 102–111, Jan 2018. [Online]. Available: <https://doi.org/10.1109/MNET.2018.1700175>
- [3] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with WebAssembly”, *Proc. 38th ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*, pp. 185–200, 2017. [Online]. Available: <https://doi.org/10.1145/3062341.3062363>
- [4] M. Jacobsson and J. Wåhlén, “Virtual machine execution for wearables based on WebAssembly”, in *Proc. 13th EAI Int. Conf. on Body Area Networks (BodyNets)*, 2018, pp. 381–389. [Online]. Available: https://doi.org/10.1007/978-3-030-29897-5_33
- [5] E. Rohou, B. N. Swamy, and A. Sezec, “Branch prediction and the performance of interpreters: Don’t trust folklore”, in *Proc. of the 13th Annual IEEE/ACM Int. Symp. on Code Generation and Optimization*. IEEE Computer Society, 2015, pp. 103–114. [Online]. Available: <https://dl.acm.org/doi/10.5555/2738600.2738614>
- [6] N. Brouwers, K. Langendoen, and P. Corke, “Darjeeling, a feature-rich VM for the resource poor”, *Proc. of the 7th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pp. 169–182, 2009. [Online]. Available: <https://doi.org/10.1145/1644038.1644056>
- [7] N. Reijers, J. Ellul, and C.-S. Shih, “Making sensor node virtual machines work for real-world applications”, *IEEE Embedded Systems Letters*, vol. 11, no. 1, pp. 13–16, March 2019. [Online]. Available: <https://doi.org/10.1109/LES.2018.2837685>
- [8] N. Reijers and C.-S. Shih, “Improved ahead-of-time compilation of stack-based JVM bytecode on resource-constrained devices”, *ACM Transactions on Sensor Networks*, vol. 15, no. 3, 2019. [Online]. Available: <https://doi.org/10.1145/3341170>
- [9] N. Shaylor, “A just-in-time compiler for memory-constrained low-power devices”, in *Proc. of the 2nd Java Virtual Machine Research and Technology Symp.* USENIX Association, 2002, pp. 119–126. [Online]. Available: <https://dl.acm.org/doi/10.5555/648042.744884>
- [10] C. Hammacher, “Liftoff: a new baseline compiler for WebAssembly in V8”, V8 JavaScript engine, 2018, accessed Nov 20, 2019. [Online]. Available: <https://v8.dev/blog/liftoff>
- [11] J. Ellul, “Poster: Towards WebAssembly for wireless sensor networks”, in *Proc. of the 2017 Int. Conf. on Embedded Wireless Systems and Networks*, 2017, pp. 208–209. [Online]. Available: <https://dl.acm.org/doi/10.5555/3108009.3108043>
- [12] R. Gurdeep Singh and C. Scholliers, “WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers”, *Proc. of the 16th ACM SIGPLAN Int. Conf. on Managed Programming Languages and Runtimes (MPLR)*, pp. 27–36, 2019. [Online]. Available: <https://doi.org/10.1145/3357390.3361029>
- [13] Intel, “WebAssembly Micro Runtime (WAMR)”, Bytecode Alliance, 2019, accessed Nov 20, 2019. [Online]. Available: <https://github.com/bytecodealliance/wasm-micro-runtime>
- [14] “WebAssembly troubles part 4: Microwasm”, 2018, accessed Feb 20, 2020. [Online]. Available: <http://troubles.md/posts/microwasm/>
- [15] A. Donovan, R. Muth, B. Chen, and D. Sehr, “PNaCl: Portable Native Client executables”, MIT CSAIL Computer Systems Security Group, 2010. [Online]. Available: <https://www.css.csail.mit.edu/6.858/2018/readings/pnacl.pdf>