Poster: Towards WebAssembly for Wireless Sensor Networks

Joshua Ellul
Department of Computer Science
University of Malta
joshua.ellul@um.edu.mt

Abstract

WebAssembly has been proposed as a binary encoding for the web which aims to be compact, platform and language independent and provide an efficient means of execution. The encoding requirements perfectly match the requirements of wireless sensor network applications. However, the encoding was not designed with typical resource constrained microcontroller applications in mind. In this paper, we propose a variant of WebAssembly, wasm16, targeted for resource constrained systems.

1 Introduction

Over the past two decades JavaScript has emerged as the de facto standard for webpage client-side scripting (whether you love it, or hate it). Although it is popular, ubiquitous and has been around for so long, it is still not the ideal encoding to send over-the-wire for webpage client-side scripting due to its large footprint and execution overheads. WebAssembly [7] is a new platform- and language-independent binary encoding aimed at decreasing the footprint of client-side script sent over-the-wire and also execution overheads inherent in JavaScript.

It would be ideal for such a web based platform- and language-independent, compact and execution efficient encoding to also be used for wireless sensor network (WSN) applications. However, many WSN devices are limited in resources, and WebAssembly's encoding was not optimised for 8 and 16-bit applications. WebAssembly requires more processing, stack space, and program space for run-time compiled applications, than what is required for typical resource constrained WSN applications. In this paper we propose additional instructions that are more suited for typical WSN applications.

International Conference on Embedded Wireless Systems and Networks (EWSN) 2017
20–22 February, Uppsala, Sweden

ISBN: 978-0-9949886-1-4

2 WebAssembly

WebAssembly is an open source development effort by major browser vendors and other developers of previous attempts at a platform independent encoding for the web (including asm.js and PNaCl [5]). WebAssembly [7] supports four basic data types: 32 and 64 bit integers and 32 and 64 bit IEEE floating point numbers. Memory is exposed to the intermediate representation as a linear array of bytes. In addition to the four basic data types, memory can be accessed as 8 and 16 bit integers. Any byte within the linear memory array is accessible (whether or not the byte forms part of a larger data type). The binary format is encoded as a stack machine (at least at the time of writing this paper as WebAssembly is still in a state of flux). Constants are represented within the encoding as variable-length integers. Due to the lowest bit-width operations being of a 32-bit nature, excessive overhead will be incurred for applications that tend to use a majority of 16-bit operations. In terms of footprint, whilst the encoding allows for optimisation of constant and in-memory values, program space requirements for run-time compiled 16-bit targeted applications will incur a heavy penalty due to the 32-bit operations.

3 wasm16 and Compiler Toolchain

Whilst the mapping of operations closely matches modern CPUs, the 32 and 64-bit operations do not closely map operations typically required in resource constrained systems. In this work a 16-bit WebAssembly operation set, wasm16, is proposed. WebAssembly (at the time of writing) uses 171 opcodes from 256 possible (first level) opcodes. Adding support for 16-bit operations requires use of 35 opcodes. This leaves a further 49 unused opcodes. It would be ideal to have the opcodes included as part of the WebAssembly specification so that a single format can be used irrespective of whether it is intended for resource constrained or traditional 32/64 bit architectures. Other alternatives are to define a different version of WebAssembly which supports 16-bit operations (the WebAssembly binary encoding contains a WebAssembly version identifier).

To demonstrate the overheads, consider the addition of two 16-bit local variable integers. Figure 2 below depicts the WebAssembly (wasm) that would be generated and associated unoptimised run-time compiled code that would be generated for the MSP430 architecture (a 16-bit architecture). Note that even though the source program specifies the addi-

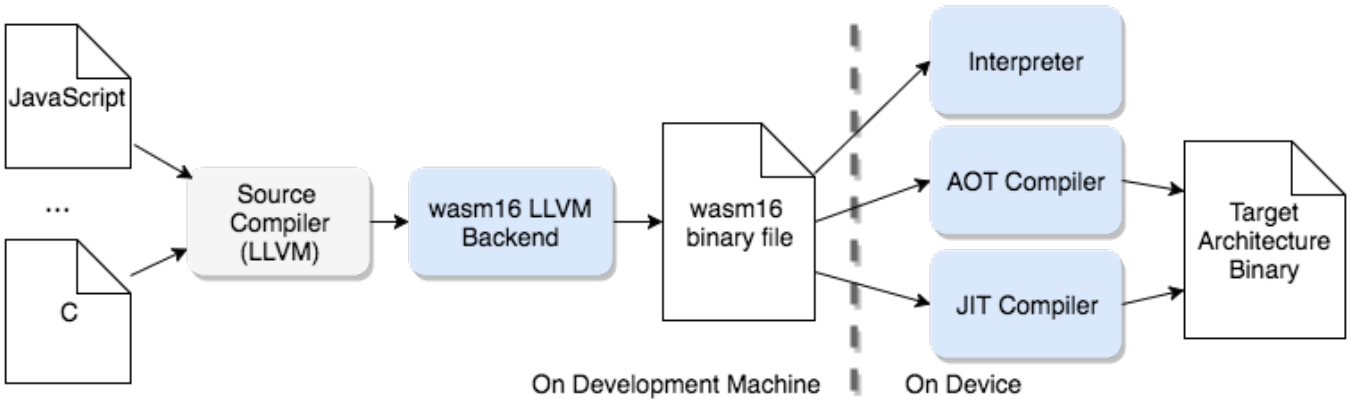

Figure 1. wasm16 compiler and execution toolchain.

```
wasm                wasm32-MSP430-AOT
get_local 0         PUSH.W 0x0000(R4)
                    PUSH.W 0x0002(R4)
get_local 1         PUSH.W 0x0004(R4)
                    PUSH.W 0x0006(R4)
i32.add             POP.W R15
                    POP.W R14
                    POP.W R13
                    POP.W R12
                    ADD.W R14,R12
                    ADDC.W R15,R13
                    PUSH.W R12
                    PUSH.W R13
```

Figure 2. WebAssembly and run-time compiled MSP430 code generated for an addition of two 16-bit local variables.

```
wasm16              wasm16-MSP430-AOT
get_local 0         PUSH.W 0x0000(R4)
get_local 1         PUSH.W 0x0002(R4)
i16.add             POP.W R13
                    POP.W R12
                    ADD.W R13,R12
                    PUSH.W R12
```

Figure 3. Corresponding wasm16 and run-time compiled MSP430 code generated for an addition of two 16-bit local variables.

tion of two 16-bit integers, the variables are promoted to 32-bits since operations in WebAssembly operate on the smallest granularity of 32-bits.

Figure 3 depicts the wasm16 and associated unoptimised run-time compiled code that would be generated for the MSP430 architecture. Note that the 16-bit variables and operations do not require the extra overhead inherent from a minimal specification of 32-bit operations. A 50% decrease in execution overhead and program footprint is gained for code that is inherently 16 or 8-bits. Optimised versions of run-time compiled native code would result in similar savings for a 16-bit implementation over a 32-bit implementation.

The compiler and execution toolchain proposed is depicted in Figure 1. WebAssembly's LLVM back-end compiler was altered to output wasm16. By providing a wasm16 LLVM back-end, any input source language can be compiled to LLVM Intermediate Representation (LLVM IR), such that an LLVM front-end exists. Thereafter the generated LLVM IR can be compiled to wasm16. wasm16 can then be interpreted or compiled and executed at run-time (be it Ahead-of-Time or Just-in-Time).

4 Related Work

Darjeeling [2] and TakaTuka [1] are two virtual machines proposed for resource constrained systems that allowed for the interpretation of a specific high-level language, Java. Darjeeling proposed an altered Java Bytecode encoding that was more tailored for 16-bit architectures that allowed for a smaller footprint. TakaTuka further attempted to minimise code footprint by optimising the interpreter with superinstructions [4]. The two virtual machines were tailored for Java. In contrast, MoteRunner [3] proposed a virtual machine interpreter that supported multiple languages. Due to the overheads inherent in interpretation, run-time compilation techniques were thereafter presented that demonstrated a substantial decrease in execution overhead [6].

5 References

[1] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rührup, and Z. A. Uzmi. Optimized java binary and virtual machine for tiny motes. In *International Conference on Distributed Computing in Sensor Systems*, pages 15–30. Springer, 2010.

[2] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 169–182. ACM, 2009.

[3] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov. Mote runner: A multi-language virtual machine for small embedded devices. In *Sensor Technologies and Applications, 2009. SENSORCOMM'09. Third International Conference on*, pages 117–125. IEEE, 2009.

[4] K. Casey, D. Gregg, M. A. Ertl, and A. Nisbet. Towards superinstructions for java interpreters. In *International Workshop on Software and Compilers for Embedded Systems*, pages 329–343. Springer, 2003.

[5] A. Donovan, R. Muth, B. Chen, and D. Sehr. Pnacl: Portable native client executables. *Google White Paper*, 2010.

[6] J. Ellul. *Run-time compilation techniques for wireless sensor networks*. PhD thesis, University of Southampton, 2012.

[7] A. Rossberg. Webassembly: high speed at low cost for everyone. In *ML16: Proceedings of the 2016 ACM SIGPLAN Workshop on ML*, 2016.