Accelerating WebAssembly Interpreters in Embedded Systems through Hardware-Assisted Dispatching*

Matthias Rupp[0000-0001-6359-5839], Jonathan Schröter[0009-0007-7435-146X], and Stefan Wallentowitz[0000-0003-3182-4929]**

Hochschule München University of Applied Sciences
Department of Computer Science and Mathematics
{matthias.rupp,stefan.wallentowitz}@hm.edu

Abstract. WebAssembly is a promising bytecode virtualization technology for embedded systems. WebAssembly interpreters for embedded demonstrate strong isolation and portability. However, they come with a significant performance penalty compared to direct bare-metal execution or compiled WebAssembly code. This creates demand for interpreter optimization. In this work, we present an approach that increases the execution speed of interpreted WASM code by offloading computed GOTOs, as used in interpreters, to a hardware accelerator. We describe the accelerator's hardware design, as well as the integration in the popular *WebAssembly Micro Runtime (WAMR)*. To prove the functionality and effectiveness of our approach, we integrate the accelerator into an open-source RISC-V processor core and evaluate WASM interpretation using different benchmarks. Our results show that the approach significantly increases execution speed while only creating minimal code size overhead. Finally, we give an outlook on possible future improvements of the accelerator.

Keywords: WebAssembly · RISC-V · Interpreter · Accelerator · Hardware/Software Codesign · Computed GoTo

1 Introduction

Porting complex software between different embedded platforms is increasingly challenging due to the large variety of platforms and architectures. Bytecode virtualization is a promising technology for modern embedded system development. The use of interpreters and runtime environments allows the same bytecode to run on different embedded devices, which increases the portability of embedded software. Furthermore, bytecode virtualization has strong isolation properties. A popular choice for such a runtime has been the *Java VM (JVM)* [16] which

* This project received funding from the Chips Joint Undertaking project *ISOLDE* (No. 101112274) and BMBF project 16MEE0372.
** Corresponding author

has been adopted to many different embedded platforms. A problem with this solution is the fact that the JVM only supports the use of Java and some related languages which might not be well-suited for all problems and also limit the amount of developers that can work on the software. WebAssembly (*WASM*) [13] can solve this problem because, despite it being a relatively new technology, there already exist plenty of compilers for a variety of different languages. While its name suggests WebAssembly is focused on browser use cases, the bytecode is much nearer to the hardware than JVM is in comparison. WebAssembly, as all other bytecode, can be executed natively with *AoT* (Ahead of Time) or *JiT* (Just in Time) compilation for the target platform. However, the highest portability is archived by the use of an interpreter for the bytecode. The downside of bytecode interpretation is a significant performance overhead since multiple native instructions are needed for the interpretation of a single WASM instruction. Therefore, the optimization of WebAssembly interpreters is an important task in scenarios, where embedded software needs to be fast and very portable.

This paper evaluates a generic hardware feature in the context of WebAssembly. The execution of each bytecode instruction is split in the actual operation and the calculation of the next instruction's memory address. As the calculation of the jump target of the next instruction affects each instruction and is identical in all cases, it is an excellent candidate for hardware acceleration. We evaluate a hardware offload using the open source cv32e40x RISC-V core and the popular *WebAssembly Micro Runtime (WAMR)* for embedded system use cases. cv32e40x is an open-source RISC-V processor written in SystemVerilog. It's an in-order 32-bit core with a four-stage pipeline. It supports multiple ISA extensions, for example the M extensions (for math operations) or compressed instructions. A FPU is not available. The *eXtension* interface allows to tightly couple a co-processor with the core to offload new instructions. It is developed by OpenHW Group, ETH Zurich and University of Bologna. It is a fork from the cv32e40p core, which is based on the PULP platform. [6]

The paper is structured as follows. We present the state of the art and related work in the field. Then, the basic concept the so called computed goto function is presented and the potential impact justified by analyzing the interpreter's dynamic execution behavior. Then the hardware design of the proposed accelerator as well as the integration in an existing WebAssembly runtime is presented. The paper concludes with an in-depth evaluation and an outlook on future work.

2 Related Work

Depending on the form of execution, WebAssembly code is significantly slower than native code for the target platform. This can be observed for embedded[19], as well as browser-based applications[15].

There currently exist plenty of WASM runtimes for running WASM outside a web browser. They focus on different environments (high- or low-power embedded devices, for example). There are significant performance differences between runtimes dependent on the tested benchmark. AoT and JiT compilation is a way to optimize performance. [20]

Especially when WebAssembly bytecode is interpreted on the embedded device, hardware optimizations are a viable way of speeding up execution. This strategy is well-known in the JVM world where embedded performance is a heavily researched topic. For example, Java bytecode interpretation can be offloaded to hardware that specializes on parallel execution, for example a GPU [12]. Another approach is a hardware implementation of the JVM itself [17]. In ARM processors, *Jazelle* is used to optimize the execution of JVM bytecode. Some JVM instructions are binary-translated to ARM instructions in hardware and executed directly, while different instructions have to be handled in software. In the latter case, *Jazelle* automatically performs a jump to the corresponding handler. [1] A more general approach is taken with the *ThumbEE* instruction set in ARM, which provides a compressed instruction set specifically tailord towards the optimization of JiT-compiled languages[8]. For RISC-V, the RISC-V-J extension[7] is currently under specification, which aims to optimize RISC-V for various interpreted and JIT-compiled bytecodes.

In comparison to these approaches, our optimization does not focus on the faster execution of interepreted bytecode itself, but instead speeds up the execution speed of the runtime by calculating the address of the next instruction handler in hardware. It is completely agnostic of the interpreted bytecode.

Apart from JVM-specific solutions, there exist more general approaches to speed up the execution of interpreted code: Erven Rohou et al. identify the instruction fetch as one of the main performance overheads in bytecode interpretation. Their solution is built around vectorization and the use of SIMD instructions to reduce the number of fetches for the same amount of data. The use of existing hardware SIMD instructions allow for a further optimization of bytecode interpretation.[18]

While this is a promising way of increasing execution performance, it requires a vectorization step during program compilation. Since we focus on maintaining maximum portability, altering the interpreted bytecode is not necessary with our solution.

Currently, only few works are available that are specifically about WASM optimization such as *Wasmachine* [21], an OS that is tailored towards efficient WASM execution. JiT compiled WASM is executed directly in kernel mode to speed up system calls.

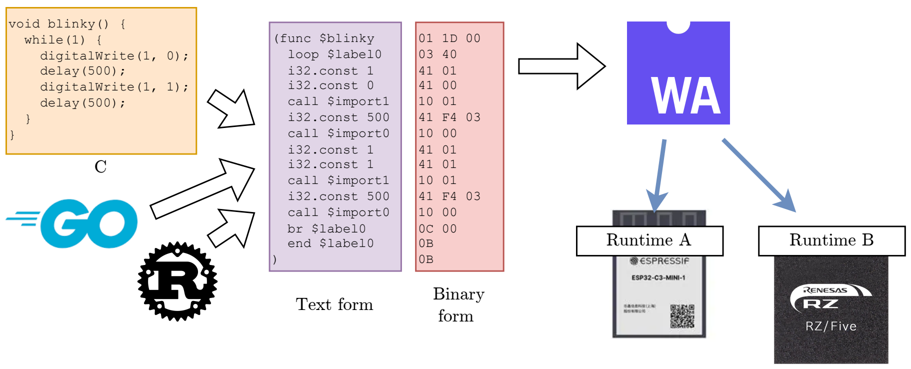
Hardware Acceleration of WebAssembly is a novel topic and we are not aware of previous work in this area.

3 WebAssembly

WebAssembly was introduced in 2014 as a joint standard for complex, faster programs in the browser [13]. Contrary to asm.js [2], which was the predominant approach at the time, WebAssembly is an entirely new portable, standardized format. As a bytecode virtualization format, WebAssembly programs are not only independent from the underlying platform, but also allow to be compiled from a wide range of source programming languages. As a bytecode virtualization it has excellent strong isolation properties.

WebAssembly is maintained and further developed as a standard by the W3C consortium [9]. Based on this, the Bytecode Alliance [4] emerged as a non-profit organization that maintains software that runs outside of the browser and defines standards beyond the web. In addition, an extremely rich open source ecosystem has emerged, which has already reached a respectable state with a large number of runtime systems, compilers and other tools.

WebAssembly is a stack-based virtual machine. Bytecode operates on this stack, operands to operations are always at the top of the stack, as in Java. In contrast to Java, there are (currently) no objects and garbage collection, but a flat memory model like in C or Rust.



```
void blinky() {
  while(1) {
    digitalWrite(1, 0);
    delay(500);
    digitalWrite(1, 1);
    delay(500);
  }
}
```

```
(func $blinky
  loop $label0
  i32.const 1
  i32.const 0
  call $import1
  i32.const 500
  call $import0
  i32.const 1
  i32.const 1
  call $import1
  i32.const 500
  call $import0
  br $label0
  end $label0
)
```

```
01 1D 00
03 40
41 01
41 00
10 01
41 F4 03
10 00
41 01
41 01
10 01
41 F4 03
10 00
0C 00
0B
0B
```

Fig. 1: Overview of WebAssembly as bytecode virtualization layer in embedded systems

The translation and execution of WebAssembly are shown as an example in Figure 1. The starting point in this example is a simple embedded program in C. A wide range of other programming languages can also be used. This is particularly interesting because it allows companies with large embedded system projects to introduce new programming language such as Rust, even if the platform itself is not supported by a Rust compiler.

A bytecode in binary form (*wasm*) is created from the source programming language. In addition to the binary form, there is also a text representation (*wat*), which is compared to the binary format in Figure 1. Without going into the exact commands, the format of a stack-based virtual machine is clearly visible here. This example calls functions that can be implemented either in the same WebAssembly module or, as in the example, as imported functions natively on the platform or in other modules.

A WebAssembly module in binary format can now run in any WebAssembly runtime environment. Common ones are interpreters, interpreters with just-in-time compilers, and ahead-of-time runtime environments. Due to the resource limitations of embedded systems, just-in-time approaches are not addressed, so embedded-focused runtime environments are usually interpreters that occasionally also allow modules to be translated ahead of time. The most common interpreters for embedded systems are currently the WebAssembly Micro Runtime from the Bytecode Alliance [11] and WASM3 [10]. These runtime environments can either be instantiated bare metal or integrated into operating systems.

In this paper we use the most widely used WebAssembly Micro Runtime (WAMR) as a WebAssembly runtime suitable for embedded platforms. It supports WASM interpretation, Ahead-of-Time (AoT) and Just-in-Time (JiT) compilation. Its development is focused towards delivering a small runtime size of around 85kB. It runs on different architectures, including RISC-V. See section 4 for a detailed analysis of the interpreter's working principle.

4 Computed GoTo

The WebAssembly micro runtime ships two different implementations of the WASM interpreter: The "classic interpreter" and "fast interpreter". The classic interpreter implementation is a traditional interpreter that iterates through the program in a loop and interprets WASM *in-place* through a large switch-case statement. For better performance, the fast interpreter optimizes the bytecode during load. In this paper, we exclusively focus on the fast interpreter. In this interpreter implementation, each WASM instruction is computed by a handler which is marked by a label.

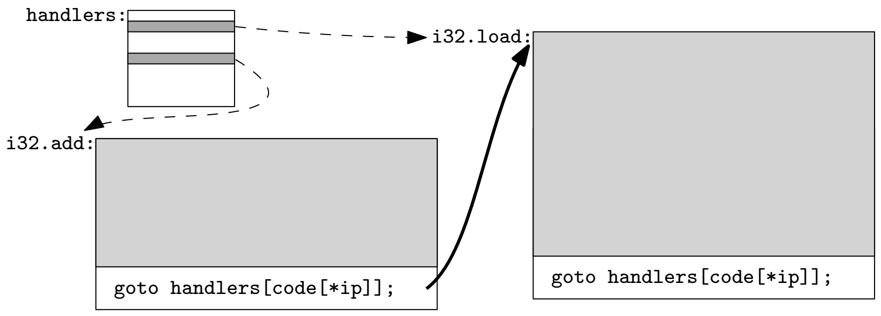


Fig. 2: Optimized control flow between opcodes using computed goto.

Figure 2 illustrates the execution of the WebAssembly interpreter through the interpreted code. A table (`handlers`) with the addresses of the labels inside the so called dispatch loop is generated by the compiler (`&&label` syntax). Each opcode is processed with implementation of the opcode semantics (light gray parts in Figure 2). To avoid multiple jump instructions, the interpreter then *directly* jumps to the next opcode handler. This is done by fetching the opcode of the next instruction and looking up the handler address, a so called *computed goto*. WAMR optimizes this by pre-computing the handler address for each opcode in the program.

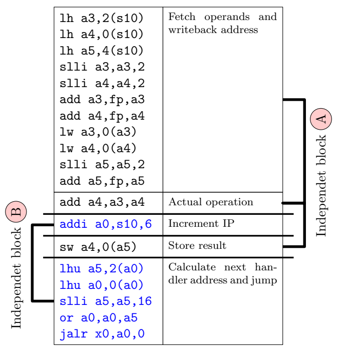
As this computation repeats both *in each opcode* and *identically* for each opcode this computed goto in the interpreter implementation is an interesting candidate. To analyze if it is suitable and promising for hardware acceleration, we analyzed the machine instructions in RISC-V that are executed in such a handler. An `i32.add` instruction, for example, is interpreted using the RISC-V instructions shown in Fig. 3a. This allows to analyze different parts of the program (operand fetch, operation, storing the result and jumping to the next opcode handler) separately. Comparing source- and destination registers of each instructions allows to identify parts of the code that are independent from the rest of the program and are therefore a candidate for offloading to an accelerator. It's observed that the code decomposes into two independent blocks: Block Ⓐ is responsible for operand fetch, operation and write back while block Ⓑ calculates the next handler's address and performs the jump. This is observed in all arithmetic and logic opcode handlers. Only a few instructions, like branches, have dependencies between block Ⓐ and Ⓑ. This makes Ⓑ a candidate for offloading to a co-processor.

To estimate the impact of this approach, the number of machine instructions for each block is measured for different WASM opcodes. The handler runtimes were extracted from a run of a simple benchmark with just a few instructions (*crc32* benchmark, see section 6.2) are shown in Fig. 3b. As expected, the impact varies between integer, memory and control flow instructions, and is a static offset. For integer operations (e.g. `i32.add`) around 30% of instructions are used for handler address calculation and jumping. The relative impact is smaller though once the extra implementation of the semantics, such as `return` and `load` in this example.

This analysis serves as an illustrative example and indicates that offloading the handler jump to a co-processor for parallel calculation is an approach worth researching. The following chapter describes hardware- and software modifications needed for implementation, along with a more in-depth analysis of the impact.

5 Accelerator Design

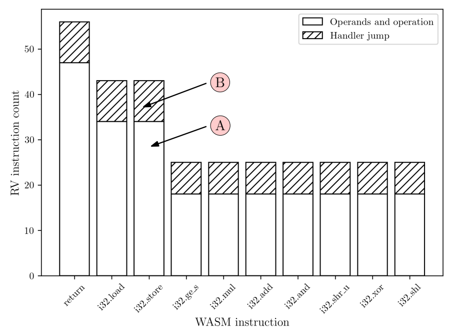
The accelerator consists of a co-processor and minor hardware modifications of the cv32e40x . Furthermore the interpreter implementation needs to be adopted to make use of the accelerator. The interface between the accelerator hardware and the software is implemented with control and status registers (CSR) and a new instruction `wasmacc_jmp`. Writing an address to the CSR triggers the fetching of the next handler's address in the background. Once the jump address is available, `wasmacc_jmp` triggers the actual jump. The co-processor is connected



```
lh a3,2(s10)
lh a4,0(s10)
lh a5,4(s10)
slli a3,a3,2
slli a4,a4,2
add a3,fp,a3
add a4,fp,a4
lw a3,0(a3)
lw a4,0(a4)
slli a5,a5,2
add a5,fp,a5
add a4,a3,a4
addi a0,s10,6
sw a4,0(a5)
lhu a5,2(a0)
lhu a0,0(a0)
slli a5,a5,16
or a0,a0,a5
jalr x0,a0,0
```

Fetch operands and writeback address
Actual operation
Increment IP
Store result
Calculate next handler address and jump
A
Independet block
B
Independet block

(a) Assembly code for the `i32.add` WASM opcode handler. Blue instructions are responsible for jumping to the next opcode handler.



(b) Number of instructions in handlers for different WASM opcodes.

Fig. 3: Analysis of RV instructions in WASM opcode handlers

to cv32e40x using the eXtension interface. The jump address is passed to the core via a new port and the jump is performed internally, since the cv32e40x is unable to offload jump instructions via the eXtension interface.

5.1 Hardware Design

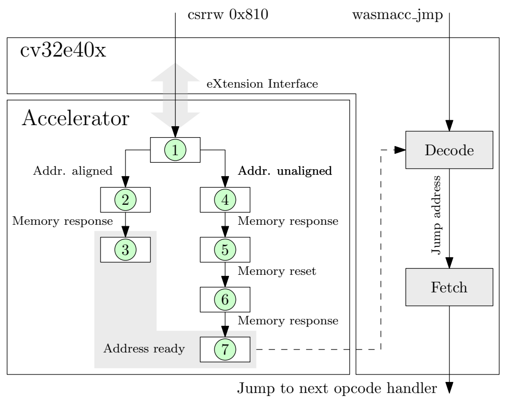


Fig. 4: Accelerator design

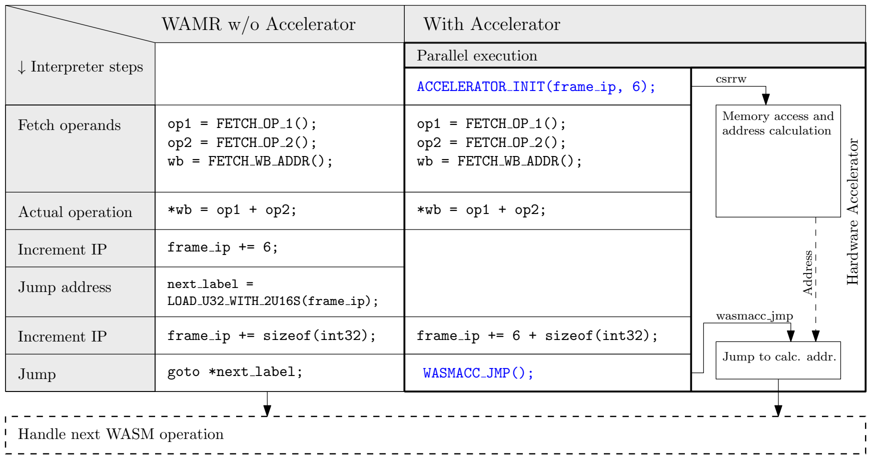
Figure Fig. 4 shows the hardware design of the accelerator. It is integrated with the cv32e40x via the eXtension interface. The accelerator is designed as a state machine. The accelerator is activated by a write to CSR `0x810` containing the current instruction pointer plus the desired increment (ip_{next}) as value. The cv32e40x is designed in a way that CSR writes always reach co-processors connected via xif. The write triggers a transition to state ① where the accelerator first checks the pointer's alignment. If the pointer is aligned ($ip_{next} \mod 4 = 0$), the accelerator transitions to state ② and issues a memory request at address ip_{next}. On successful response, a transition to state ③ is made and the address is available at the accelerator's output.

If an unaligned memory address is detected in stage ① ($ip_{next} \mod 4 \neq 0$), the accelerator needs 2 consecutive memory accesses at ip_{next} and $ip_{next} + 4$ (stage ④...⑥). After a successful second response, state machine reaches ⑦ where the address is calculated from both results and the alignment offset.

The final jump address is then available to the core's *decode* stage. When the `wasmacc_jmp` command is issued, the core executes the jump to this address by passing the address to fetch stage which then reads the instructions required for handling the next WASM opcode.

The cv32e40x extension interface ensures that memory requests are only issued when the memory is not in use by the main program running on the core.

5.2 Software Integration



↓ Interpreter steps	WAMR w/o Accelerator	With Accelerator
		Parallel execution
		ACCELERATOR_INIT(frame_ip, 6);

Fetch operands	op1 = FETCH_OP_1();
	op2 = FETCH_OP_2();
	wb = FETCH_WB_ADDR();
		op1 = FETCH_OP_1();
		op2 = FETCH_OP_2();
		wb = FETCH_WB_ADDR();

Actual operation	*wb = op1 + op2;	*wb = op1 + op2;
Increment IP	frame_ip += 6;	
Jump address	next_label = LOAD_U32_WITH_2U16S(frame_ip);	
Increment IP	frame_ip += sizeof(int32);	frame_ip += 6 + sizeof(int32);
Jump	goto *next_label;	WASMACC_JMP();

csrrw
Memory access and address calculation
Address
Hardware Accelerator
wasmacc_jmp
Jump to calc. addr.
Handle next WASM operation

Fig. 5: One iteration of the WAMR interpreter for an add (e.g. `i32.add`) instruction with and without the hardware accelerator in operation.

Fig. 5 shows the comparison between the interpretation of a WASM opcode in WebAssembly micro runtime with and without the accelerator in place. If available, the accelerator must be provided with a valid instruction pointer and the increment at the beginning of the opcode handler. For this reason, it cannot be used for opcodes which alter the instruction pointer (e.g. branches). In the specific case illustrated in Fig. 5, the instruction pointer is incremented by 6, which is the case for most arithmetic operations. In both cases, the handler needs to fetch operands and a write back address. The actual operation (for example an addition) is then performed in software.

Normally, the runtime would then increment the instruction pointer and calculate the next label's address. This is done by reading two 16-bit values from memory, which is faster for unaligned addresses, however, it comes at the cost of 4 native instructions (observed in our RISC-V-based setup). Then, the `ip` is incremented again and the jump to the next label is triggered.

With the accelerator in place, the jump address is fetched in parallel, so no additional memory access has to be performed in software. Another benefit is that there is no dependency between instruction pointer and jump address calculation any more, since the accelerator keeps its own copy. Therefore, after compiler optimizations, only one increment of the instruction pointer (in this case by 10) is needed, which saves an additional instruction. Finally, the `wasmacc_jmp` instruction is called to instruct the cv32e40x to perform the jump to the next WASM opcode handler.

The interpreter modification is straight-forward: On naive dispatch loops, the fetch of the next instruction occurs at the end of the loop after the current opcode is interpreted. Optimized designs like the interpreter under consideration here include the calculation in each handler and jump between those handlers. Interpreters make heavy use of macros or are generated, which makes the accelerator extension easy to integrate.

6 Evaluation

To quantify the actual impact of the proposed acceleration approach on the performance of WASM interpretation, the simulation and evaluation pipeline as shown in Fig. 6 is used. For benchmarking, the well-established benchmark suite *embench-iot* is used, as well as some custom benchmarks. The *wasi-sdk* compiler is used to compile these benchmarks to WebAssembly binaries which are then compiled into the embedded firmware together with *WebAssembly micro runtime*. After compilation to a RISC-V binary, the firmware is exported as a Verilog memory file. Memory, cv32e40x and the accelerator are compiled by *Verilator*. The simulation's binary is then executed and the results are analyzed.

6.1 Binary size

First, the influence of the accelerator on the firmware binary size is evaluated by comparing firmware images with- and without the accelerator enabled. It is observed that the accelerator increases the `.text` segment by a small amount of 240 bytes, compared to an untouched version of the WebAssembly micro runtime. This is likely due to the fact, that accelerator code can't be used for app opcode

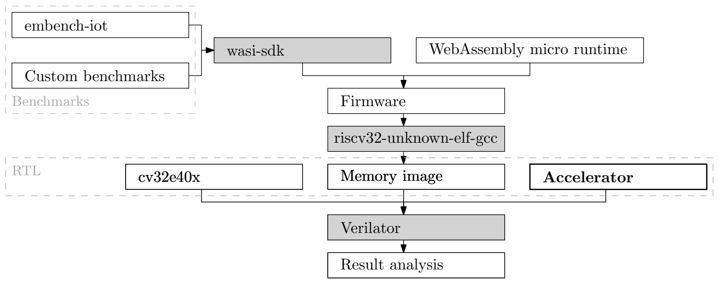


Fig. 6: Simulation and evaluation pipeline with artifacts and tools

handlers. Therefore, code for offloading an operation to the accelerator has to be added to all accelerated opcode handlers individually. Accelerator-specific code is only added to the interpreter itself, other parts of the firmware do not need to be changed and thus don't change in size.

6.2 Effect on Benchmark Runtime

To measure the effectiveness of the hardware accelerator, the execution delay of benchmarks from the *embench-iot* suite is measured. Five benchmarks that utilize a wide variety of different instructions were selected for this work:

nbody Simulates a planetary system using many floating point and vector operations.[5]
st Benchmark from the BEEBS suite[3]. Calculates statistic metrics (mean, standard deviation etc.) on random data.
crc32 Also from BEEB. Computes CRC checksums on an array of data.
ud From the BEEBS suite. Performs LU decomposition on linear equations.
primecount Originally developed by Bruce Hoult[14]: Counts prime numbers.

Only the actual execution is measured, program load, function lookup etc. is excluded since it is not influenced by the accelerator. Fig. 7 shows the results (in clock cycles) for all evaluated benchmarks. The accelerator is able to reduce the execution time of all tested benchmarks. However, the reduction varies from 4% (*st* and *nbody*) to 11% (*ud*, *primecount*).

This shows that the acceleration approach works well in increasing execution speed by over 10%. However, the amount depends on the interpreted program. This is further investigated in section 6.3. A case in which the accelerator had a negative impact on execution performance was never observed.

6.3 Acceleration by WASM Opcode

The results of section 6.2 show that the accelerator's effectiveness highly depends on the interpreted program. To further investigate this behavior, the impact of

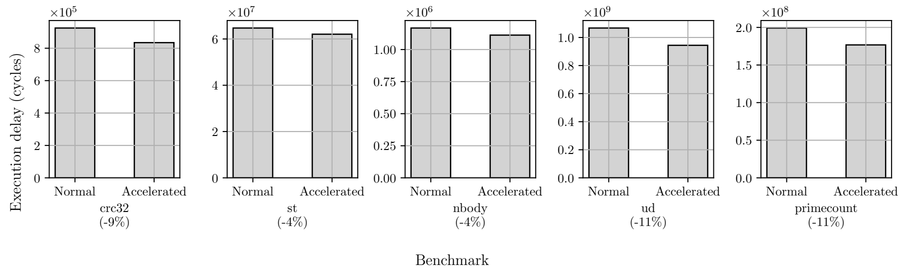
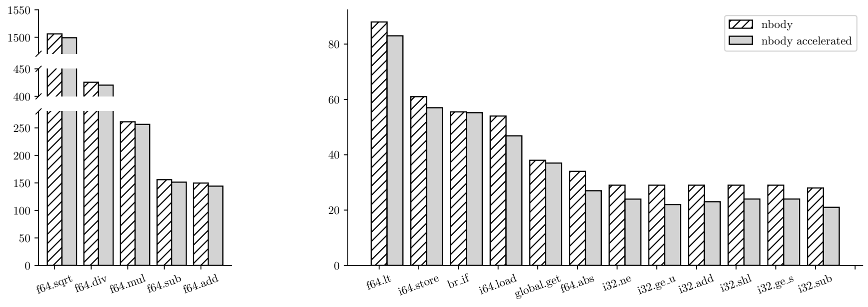


Fig. 7: Benchmark execution delay with and without accelerator

the accelerator is measured for each WASM opcode individually. Fig. 8 shows the clock cycles required for each taken opcode handler in the *nbody* benchmark. While the effect is clearly visible for fast arithmetic operations like `i32.add` (-20%), the relative benefit is much smaller for long running operations like `f64.sqrt` (-0.5%) and other floating point operations. The accelerator shows no effect for `br-if`, since it can't be optimized due to it being a branching operation. These results already show that the optimization approach is capable of reducing the interpretation time of most WASM operations.

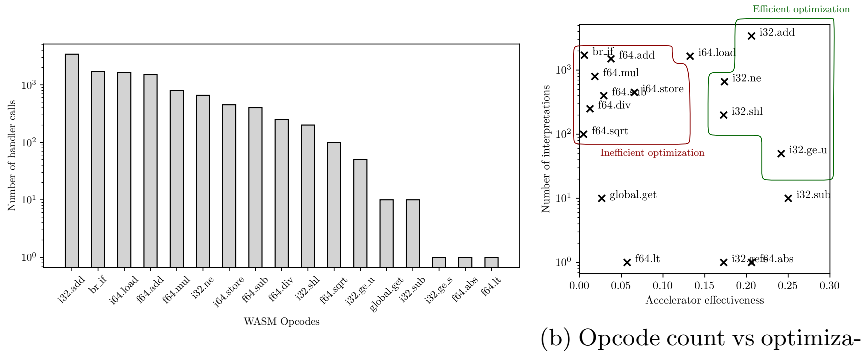


(a) Opcodes > 100 cycles

(b) Opcodes ≤ 100 cycles

Fig. 8: *nbody* benchmark: Number of clock cycles needed for the interpretation of each WASM opcode

To estimate the actual impact on the benchmarks overall efficiency, the number of handler calls is counted for each opcode. Fig. 9a shows the results for the *nbody* benchmark. With 3420 interpretations, `i32.add` is the most common operation followed by `br_if` (1721 interpretations). Fig. 9b combines both metrics by plotting quantity and optimization effectiveness of all WASM opcodes used in the benchmark. This allows to identify opcodes that allow for an efficient optimization of the program (frequently used and high optimization effect) and WASM opcodes hindering the optimization (frequently used and low benefit of the accelerator). In this benchmark, the overall greatest optimization effect is caused by integer (arithmetic, logical, comparison) operations, while branch and floating point operations reduce the accelerator's impact on program performance.



(a) WASM opcode count

(b) Opcode count vs optimization effectiveness

Fig. 9: *nbody* benchmark:Opcode count and optimization effectiveness

For comparison, this measurement is repeated for the *primecount* benchmark, which had greater speed improvements with the accelerator in place. The results are visualized in Fig. 10. It shows that *primecount* benchmark has much less frequent opcodes that are not optimizable than *nbody*. Most frequent opcodes in *primecount* are fast and easy to optimize integer operations.

Our results show that the accelerator can improve the performance of WASM opcode interpretation. However, the effect varies between opcodes. The findings indicate that programs performing mainly integer operations are easy to optimize using our approach while floating point operations and branches hinder optimization effectiveness. It must be noted that the optimization effectiveness of floating point operations might be greater in processors with a FPU.

7 Conclusion and Outlook

The use of interpreted WebAssembly code is a promising way of creating very portable embedded software. However, the performance overhead generates demand for optimization strategies. In this work, we presented a concept and first implementation of a hardware accelerator to optimize the interpretation of embedded WebAssembly code. Evaluation showed that embedding the accelerator into an existing WebAssembly runtime only creates a code minimal size overhead. When in use, the accelerator can increase the execution speed of interpreted WASM code by over 10%. However, the effectiveness highly depends on

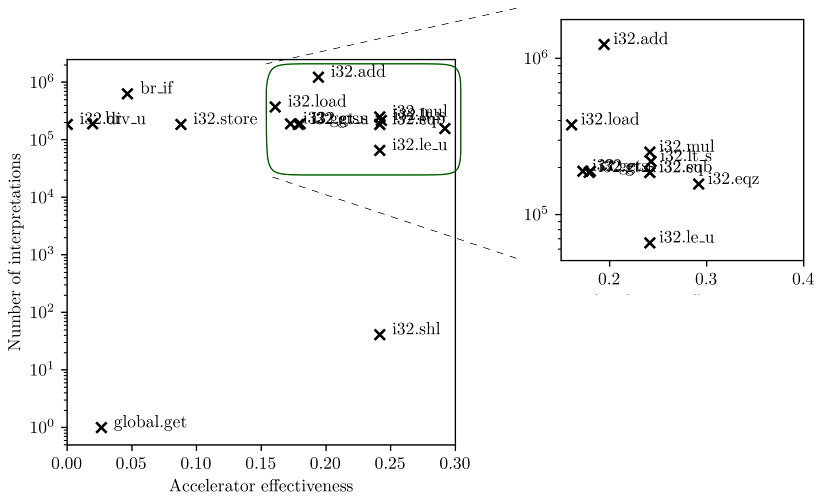


Fig. 10: *primecount* benchmark: Optimization effectiveness

the interpreted program. While integer operations can be optimized by a significant amount, there is only a tiny effect with more complex operations. Currently, there is no way of optimizing branching operations.

Considering the very small size penalty and the benefit in execution speed, using a hardware accelerator to offload handler table jumps to the hardware is a promising optimization strategy. Topic of future research could be the use of the accelerator with a processor containing a FPU to also effectively optimize floating point operations. Since the general concept of offloading parts of the interpreter to hardware proved to be working well, it might also be worthwhile to identify additional blocks for offloading, for example the fetching of source operands or result write back.

In future work we will add support to other interpreters, such as wasm3. Furthermore we will generalize the approach to support other interpreter designs. Finally, the results indicate that there is much more potential for bytecode interpreter acceleration.

References

1. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, `https://developer.arm.com/documentation/ddi0406/b/Application-Level-Architecture/Application-Level-Programmers--Model/Execution-environment-support/Jazelle-direct-bytecode-execution-support?lang=en`
2. asm.js, `https://asmjs.org`
3. Bristol/Embecosm Embedded Benchmark Suite, `https://beebs.mageec.org`
4. Bytecode Alliance, `https://bytecodealliance.org/`
5. n-body description (Benchmarks Game), `https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/nbody.html`

6. OpenHW Group CV32E40X User Manual, https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/index.html
7. RISC-V J Extension, https://github.com/riscv/riscv-j-extension
8. Thumb Execution Environment (ThumbEE), https://developer.arm.com/documentation/den0013/d/ARM-Architecture-and-Processors/Architecture-history-and-extensions/Thumb-Execution-Environment--ThumbEE-
9. W3C WebAssembly Working Group, www.w3.org/wasm/
10. wasm3, https://github.com/wasm3/wasm3
11. WebAssembly Micro Runtime, https://github.com/bytecodealliance/wasm-micro-runtime
12. Fumero, J., Stratikopoulos, A., Kotselidis, C.: Running parallel bytecode interpreters on heterogeneous hardware. In: Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming. pp. 31–35. ACM, Porto Portugal (Mar 2020). https://doi.org/10.1145/3397537.3397563, https://dl.acm.org/doi/10.1145/3397537.3397563
13. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with webassembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 185–200 (2017)
14. Hoult, B.: Program to count primes, http://hoult.org/primes.txt
15. Jangda, A., Powers, B., Berger, E.D., Guha, A.: Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 107–120. USENIX Association, Renton, WA (Jul 2019), https://www.usenix.org/conference/atc19/presentation/jangda
16. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java virtual machine specification. Addison-wesley (2013)
17. O'Connor, J., Tremblay, M.: picoJava-I: the Java virtual machine in hardware. IEEE Micro **17**(2), 45–53 (Apr 1997). https://doi.org/10.1109/40.592314, http://ieeexplore.ieee.org/document/592314/
18. Rohou, E., Williams, K., Yuste, D.: Vectorization technology to improve interpreter performance. ACM Transactions on Architecture and Code Optimization **9**(4), 1–22 (Jan 2013). https://doi.org/10.1145/2400682.2400685, https://dl.acm.org/doi/10.1145/2400682.2400685
19. Wallentowitz, S., Kersting, B., Dumitriu, D.M.: Potential of WebAssembly for Embedded Systems. In: 2022 11th Mediterranean Conference on Embedded Computing (MECO). pp. 1–4 (2022). https://doi.org/10.1109/MECO55406.2022.9797106
20. Wang, W.: How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes. In: 2022 IEEE International Symposium on Workload Characterization (IISWC). pp. 228–241. IEEE, Austin, TX, USA (Nov 2022). https://doi.org/10.1109/IISWC55918.2022.00028, https://ieeexplore.ieee.org/document/9975423/
21. Wen, E., Weber, G.: Wasmachine: Bring IoT up to Speed with A WebAssembly OS. In: 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). pp. 1–4. IEEE, Austin, TX, USA (Mar 2020). https://doi.org/10.1109/PerComWorkshops48775.2020.9156135, https://ieeexplore.ieee.org/document/9156135/