Refining the I2C proposal for WebAssembly System Interface

Robin Verbeelen
Graduate Student, Faculty of Engineering and Architecture
Ghent University
Ghent, Belgium
robin.verbeelen@ugent.be

prof. dr. Bruno Volckaert
Department of Information Technology
Ghent University – imec
Ghent, Belgium
bruno.volckaert@ugent.be

dr. ing. Merlijn Sebrechts
Department of Information Technology
Ghent University – imec
Ghent, Belgium
merlijn.sebrechts@ugent.be

ing. Michiel Van Kenhove
Department of Information Technology
Ghent University – imec
Ghent, Belgium
michiel.vankenhove@ugent.be

***Abstract*—Resource-constrained embedded systems face critical challenges adopting WebAssembly (Wasm) technologies for hardware communication. The current I²C Proposal for WebAssembly System Interface (WASI) depends on Preview 2's sophisticated component model architectures, creating deployment barriers for embedded environments. This work develops a comprehensive WASI Preview 1 implementation ecosystem that eliminates these constraints. Manual Rust bindings translate WIT-defined interface semantics to Preview 1 function calls, enabling WebAssembly Micro Runtime (WAMR) integration on constrained platforms. Comparative evaluation against Wasmtime's Preview 2 approach demonstrates that Preview 1 implementations achieve embedded compatibility while maintaining functional equivalence with Preview 2 specifications. WAMR delivers 77 × faster startup times (253 µs vs 19,559 µs) and dramatically lower memory requirements (10 kB vs 2.7 MB peak usage) than Wasmtime. Hardware validation through Raspberry Pi and Arduino testbeds confirms practical I²C communication, establishing performance baselines for embedded WebAssembly deployment.**

***Index Terms*—WebAssembly, WASI, I²C, Embedded, IoT, WAMR, Wasmtime**

I. Introduction

Modern embedded systems must simultaneously meet stringent security, reliability, and updatability requirements while operating within strict resource constraints. The European Union's Cyber Resilience Act mandates effective vulnerability management and long-term security updates throughout product lifecycles, while automotive industries increasingly adopt Over-The-Air (OTA) software distribution requiring robust rollback mechanisms [2]. These regulatory and technical demands create a fundamental challenge: enabling secure, reliable, and updatable software for resource-constrained Internet of Things (IoT) devices.

WebAssembly (Wasm) and its System Interface specification (WASI) address these challenges directly. Originally designed for web browsers, Wasm now provides portable, secure, and efficient runtime technology across diverse environments. WASI Preview 2's release introduced the component model architecture, enabling modular and interoperable Wasm application development. However, a critical gap persists in WASI's hardware interface coverage: despite I²C communication's ubiquity in embedded systems, protocol standardization remains in early stages.

Preview 2's dependency on sophisticated runtime capabilities conflicts directly with embedded resource limitations. Current WASI I²C standardization efforts focus primarily on component model architectures that exceed embedded platform constraints. This research demonstrates how the proposed WASI I²C interface can be effectively adapted for resource-constrained environments through a comprehensive Preview 1 implementation approach.

II. Problem Statement and Research Questions

Current WASI I²C standardization efforts, building upon prior work by Vandenberghe et al. [3], focus primarily on WASI Preview 2 environments. These implementations demonstrate functional I²C communication but reveal critical limitations for embedded deployment: runtime complexity requiring sophisticated component model support exceeds embedded platform capabilities, while binary size overhead challenges memory-constrained systems. The standardization process demands Preview 1 compatibility to meet embedded deployment requirements specified in the proposal's portability criteria [4].

Three fundamental research questions drive this investigation:

RQ1: How can the WASI Preview 2 I²C interface be effectively mapped to WASI Preview 1 environments while preserving functional compatibility?

RQ2: How do Preview 1 and Preview 2 approaches compare regarding developer experience, maintainability, and implementation complexity?

RQ3: What performance differences exist between WASI Preview 2 and Preview 1 implementations for embedded I²C applications in terms of startup latency, memory consumption, and execution overhead?

III. Implementation Methodology

This approach develops a complete Preview 1 I²C ecosystem through five integrated components that collectively demonstrate embedded deployment feasibility.

A. Manual Bindings Library (wasip1-i2c-lib)

This Rust library is a result of manually translating the WIT-defined I²C interface semantics to Preview 1 function call mechanisms. Conditional compilation supports both guest and host environments while preserving type safety through optimized error encoding schemes. Error types compress into single-byte values using bit manipulation, providing errno-style interfaces that reduce memory usage while maintaining complete semantic information from WIT specifications. For experimental reasons, there was a deviation from the Canonical ABI compliance, investigating the flexibility of Preview 1.

B. Preview 1 Guest Implementation

This guest implementation targets `wasm32-wasip1` with `#![no_std]` configuration for minimal binary size. RAII patterns ensure automatic resource cleanup while `lol_alloc` provides embedded-appropriate memory management. The optimized Wasm module compiles to 4.8 kB compared to 38 kB with standard library inclusion.

C. WAMR Runtime Integration

The host implementation — embedding WAMR with its Rust SDK — bridges Rust's memory safety model with WAMR's C-based architecture. Per-instance, per-handle access control implements capability-based security while critical memory translation between WebAssembly linear memory and native memory spaces maintains safe pointer operations.

D. Comparative Preview 2 Framework

Parallel Preview 2 implementations using Wasmtime and the cargo-component toolchain enable direct performance comparison while maintaining functional equivalence in I²C communication capabilities. The Wasm component compiles to 16.2 kB, without support for a `#![no_std]` version.

E. Performance Evaluation Infrastructure

The benchmarking framework employs Criterion.rs for statistical analysis with automatic outlier detection. DHAT profiling tracks memory allocation patterns during both runtime setup and execution phases, providing comprehensive resource utilization analysis. Both tools allowed comprehensive profiling of each implementation throughout different stages of execution: Runtime setup and Cold/Hot start Wasm guest execution. The Wasm guests perform a complete I²C communication cycle: resource acquisition, writing, reading, and finally destroying the resource.

IV. Experimental Setup and Evaluation

A. Hardware Configuration

The testbed consists of a Raspberry Pi 5 (ARM64) acting as I²C controller connected to an Arduino Uno R3 serving as I²C target. Two Arduino firmware versions support evaluation phases: correctness verification firmware with serial logging enables transaction validation, while performance-optimized firmware eliminates serial overhead during benchmarking.

B. Evaluation Methodology

The evaluation follows a three-phase approach, ensuring both correctness and comprehensive performance characterization:

1) **Correctness Verification:** Functional validation across all implementations via Arduino serial monitor confirms successful I²C transactions before performance measurement.
2) **Performance Measurement:** Statistical benchmarking over 100 samples using Criterion.rs measures three execution scenarios: runtime setup (initialization overhead), cold execution (first operation after setup), and hot execution (repeated steady-state operations).
3) **Memory Analysis:** DHAT profiling captures heap allocation patterns during setup and execution phases. Binary size analysis compares standard and optimized builds across implementation variants.

The ping-pong test routine, implemented in the Wasm guests, exercises complete system functionality through resource creation, bidirectional I²C communication — write `"hello"`, read response — and automatic cleanup. This sequence validates the entire WebAssembly I²C ecosystem from resource lifecycle management through hardware communication to performance characterization.

V. Results and Analysis

A. Runtime Setup Performance

Initialization overhead demonstrates stark performance disparities between WebAssembly runtimes. WAMR achieves setup in 253 µs (130 × native overhead), while Wasmtime requires 19,559 µs (approximately 10,000 × native overhead, 77 × slower than WAMR). These characteristics directly impact embedded deployment: Wasmtime's millisecond-scale startup can exclude strict ultra-low-power applications, while WAMR's sub-millisecond initialization remains viable. Fig. 1 visualizes these findings.

B. Execution Performance

Both runtimes achieve nearly identical steady-state performance with 2 × native overhead (WAMR: 1,185 µs, Wasmtime: 1,184 µs vs Native: 589 µs). Cold execution shows minimal optimization: WAMR improves 2.1 %, Wasmtime 9.0 %, indicating modest caching effects. Coefficient of Variation analysis confirms excellent measurement consistency (<1 %) across all implementations. Fig. 2 visualizes these findings.

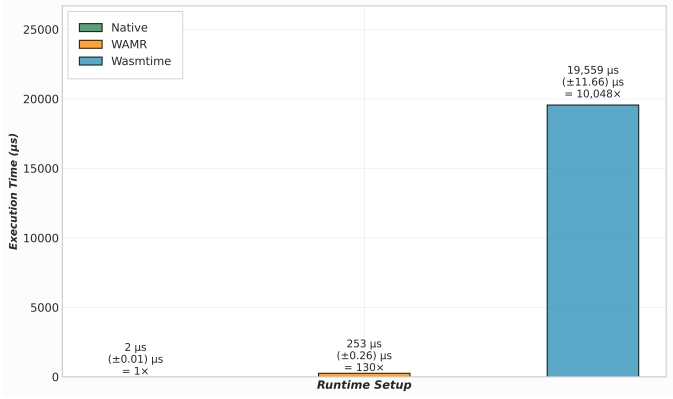

Fig. 1. Relative performance comparison across all implementations during runtime setup, showcasing significant latency for Wasmtime

C. Memory Utilization

Memory consumption patterns expose architectural trade-offs. WAMR requires 10 kB across 16 allocations during setup (99.6 % persistent), while Wasmtime allocates 14.25 MB during initialization, maintaining 2.72 MB peak usage (81 % temporary allocation). Wasmtime's peak usage exceeds RAM limitations of ESP32-C3 (400 kB) and Nucleo F412ZG (256 kB) platforms specified in standardization portability criteria. Fig. 3 visualizes these findings, needing logarithmic scaling for clear representation.

Execution memory usage validates both approaches for embedded deployment: WAMR consumes 327 B across 6 allocations, Wasmtime uses 416 B across 10 allocations. This represents 20 × to 26 × overhead versus native baseline while confirming viability for long-running embedded applications.

D. Binary Size Analysis

Experimental compile-time optimization effectiveness varies significantly across implementations. WAMR achieves 93.7 % reduction from 10.44 MB to 660 kB, while Wasmtime manages 70.4 % reduction from 12.20 MB to 3.61 MB.

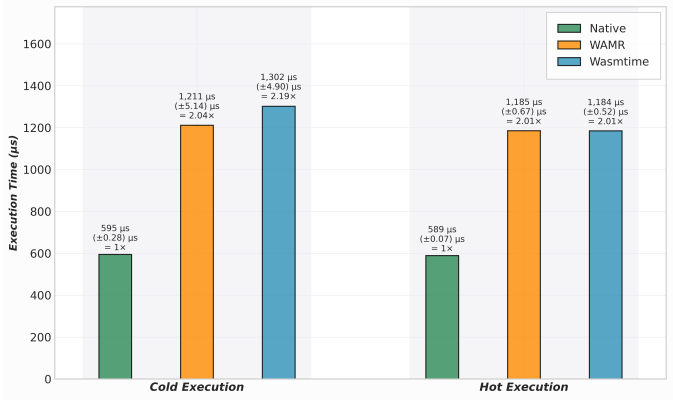

Fig. 2. Relative performance comparison across all implementations during execution of the Wasm guest, showcasing similar behavior for both Wasm runtimes

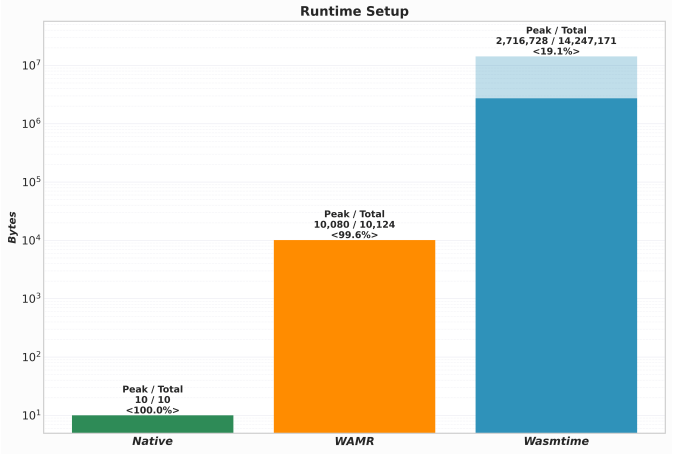

Fig. 3. Memory usage visualization showing orders of magnitude higher utilization by Wasmtime during runtime setup. WAMR also makes a significant jump in memory requirements compared to Native. Notice the logarithmic step size.

WAMR's optimized footprint targets moderately constrained embedded systems, while Wasmtime's minimum requirement significantly exceeds the flash memory constraints of target platforms.

VI. Impact on WASI I²C Standardization

This research advances the WASI I²C proposal toward Phase 3 standardization by addressing critical embedded system compatibility requirements. The Preview 1 implementation eliminates deployment barriers created by component model dependencies, demonstrating that the interface accommodates resource-constrained environments without functionality compromise.

Concrete standardization contributions include:

- **Embedded compatibility validation:** WAMR integration meets portability criteria, even for the more constrained ESP32-C3 and Nucleo F412ZG platforms specified in the proposal.
- **Performance benchmarks:** Quantitative baselines establish deployment feasibility metrics across startup latency and execution overhead.
- **Reference implementation:** Complete Preview 1 codebase enables standardization testing and validation workflows.
- **Adaptation methodology:** Systematic approach for translating WIT-defined interfaces to Preview 1 environments applies to other WASI hardware proposals

Beyond I²C-specific contributions, this work establishes general methodologies for bridging WASI Preview 2 interfaces to embedded environments. The manual bindings approach, capability-based security implementation, and performance evaluation framework provide templates for standardizing SPI, PWM, UART, and other hardware interfaces targeting resource-constrained deployment scenarios.

VII. Limitations and Future Work

This investigation acknowledges specific limitations that constrain result generalizability. The evaluation targets simple ping-pong I^2C operations rather than complex production workloads involving bulk transactions, concurrent device management, or sophisticated error recovery scenarios. Hardware validation uses Raspberry Pi/Arduino configurations exclusively; performance characteristics will vary across different embedded platforms, I^2C implementations, and microcontroller architectures. The simplified interface implementation enables focused core functionality evaluation but excludes transaction sequences and write-read operations present in the complete WASI I^2C specification.

Future research directions address both standardization requirements and deployment optimization:

Standardization advancement: Comprehensive test suite development covering complete WASI I^2C specification complexity, realistic embedded application workloads, and error handling scenarios will support Phase 3 progression requirements.

Runtime optimization: WAMR startup latency and memory footprint reduction strategies target ultra-constrained deployments, while Wasmtime embedded viability investigation through ahead-of-time compilation and custom component model configurations remains unexplored.

Interface evolution: WASI 0.3 async support integration and Canonical ABI compliance analysis will improve interoperability when other WebAssembly components interact with I^2C interfaces, enhancing production deployment viability.

VIII. Conclusion

This research establishes that WASI Preview 2 I^2C interface specifications can be effectively adapted for resource-constrained embedded systems through comprehensive Preview 1 implementation approaches.

RQ1 - Interface Mapping: The wasip1-i2c-lib library demonstrates successful adaptation of Preview 2 interface semantics to Preview 1 environments. Manual Rust bindings effectively translate resource-based I^2C controller management, WIT-defined error handling, and bidirectional communication patterns to Preview 1's function call mechanisms while maintaining complete type safety and functional equivalence with Preview 2 specifications.

RQ2 - Implementation Complexity Comparison: Analysis reveals fundamental trade-offs between developer convenience and deployment flexibility. Preview 2's automatic code generation through wit-bindgen provides superior developer experience but requires sophisticated runtime support, revealing incompatibility with embedded constraints. Preview 1 approaches demand significant manual implementation effort but enable deployment transparency, runtime simplicity, and optimization opportunities critical for resource-constrained environments.

RQ3 - Performance Analysis: Quantitative evaluation establishes clear performance distinctions with direct embedded implications. WAMR delivers 77 × faster startup times (253 µs vs 19,559 µs) and dramatically lower memory requirements (10 kB vs 2.7 MB peak usage) compared to Wasmtime. Both runtimes achieve equivalent steady-state execution performance with 2 × native overhead, validating WebAssembly viability for embedded I^2C communication.

This implementation directly addresses advancing the WASI I^2C standardization toward Phase 3 requirements while establishing systematic methodologies for broader embedded WebAssembly adoption. By demonstrating embedded system compatibility without functionality compromise, this work enables secure, portable, and efficient embedded software solutions addressing contemporary regulatory and technical challenges in I^2C communication.

References

[1] European Parliament, "Cyber Resilience Act," Official Journal of the European Union, Mar. 2024.

[2] UNECE, "World Forum for Harmonization of Vehicle Regulations, Global technical regulation on cybersecurity," ECE/TRANS/WP.29/2020/79, Jun. 2020.

[3] F. Vandenberghe, M. Sebrechts, T. Goethals, F. De Turck, and B. Volckaert, "Advancing the I2C proposal for WebAssembly System Interface," Master's thesis, Ghent University, 2024.

[4] WebAssembly Community Group, "WASI I2C Proposal," https://github.com/WebAssembly/wasi-i2c, 2024.

[5] W3C WebAssembly Community Group, "WebAssembly Specification," https://webassembly.github.io/spec/core/, 2023.

[6] Bytecode Alliance, "WebAssembly Micro Runtime (WAMR)," https://github.com/bytecodealliance/wasm-micro-runtime, 2024.

[7] Bytecode Alliance, "Wasmtime: A WebAssembly Runtime," https://wasmtime.dev/, 2024.

[8] NXP Semiconductors, "UM10204: I2C-bus specification and user manual," Rev. 6, Apr. 2014.