

Advancing the I2C proposal for WebAssembly System Interface

Friedrich Vandenberghe

Student number: 01806624

Supervisors: Prof. dr. Bruno Volckaert, Prof. dr. ir. Filip De Turck

Counsellors: Dr. ing. Merlijn Sebrechts, Dr. ing. Tom Goethals

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science

Academic year 2023-2024

Dankwoord

Allereerst wil ik mijn promotoren, prof. dr. Bruno Volckaert en prof. dr. ir. Filip De Turck, en begeleiders, dr. ing. Merlijn Sebrechts en dr. ing. Tom Goethals, bedanken. Zonder hen zou deze masterproef nooit tot stand gekomen kunnen zijn. In het bijzonder wil ik dr. ing. Merlijn Sebrechts bedanken voor de opvolging en het oneindige geduld om mijn zinnen met een passive voice aan te duiden.

Daarnaast wil ik graag mijn ouders bedanken voor de steun, en het proeflezen. Zonder hen zou dit ook nooit tot stand gekomen kunnen zijn en waren er ongetwijfeld nog talloze grammaticale fouten aanwezig.

Tot slot wil ik ook graag iedereen in de Bytecode Alliance waarmee ik in contact gekomen ben bedanken voor diens onschatbare feedback en hulp, hieruit wil ik Dan Gohman, Chris Woods en Maximilian Seidler uitlichten.

Permission for Usage

The author gives permission to make this master's thesis available for consultation and to copy parts of this master's thesis for personal use. Every other use is subject to copyright terms, in particular with regard to the obligation to explicitly state the source when quoting results from this master's thesis.

Samenvatting

Dit proefschrift richt zich op de dringende behoefte aan robuuste en veilige softwareoplossingen in kritieke infrastructuursystemen, waar plotselinge storingen ernstige gevolgen kunnen hebben. Het onderstreept het belang van tijdige software-updates, vooral in industrieën als de auto-industrie, gezondheidszorg en transport. Het benadrukt de noodzaak van naadloze rollback-mechanismen in geval van storingen.

Het onderzoek introduceert WebAssembly en WebAssembly System Interface als veelbelovende technologieën voor dergelijke toepassingen, maar identificeert een kritiek gat in hun ondersteuning voor het veelgebruikte I2C-protocol in IoT-apparaten.

Het primaire doel van dit onderzoek is om deze kloof te overbruggen door I2C-connectiviteit mogelijk te maken voor Wasm-applicaties, met inachtneming van strenge beveiligings-, betrouwbaarheids- en overheadbeperkingen. Daarnaast heeft het proefschrift als doel om deze aanpak te standaardiseren, waardoor de wijdverspreide toepassing in de industrie wordt bevorderd en de duurzaamheid op lange termijn van kritieke infrastructuursystemen wordt gegarandeerd.

Om dit te vergemakkelijken worden de `wasi-i2c` interfaces voorgesteld. Deze interfaces volgen het capability-based security design principe, dat centraal staat binnen WASI. Om de deugdelijkheid van `wasi-i2c` vast te stellen worden implementaties geleverd. Deze implementaties variëren van verschillende runtimes, platformen en architecturen. Daarna wordt een evaluatie van de gemiddelde uitvoeringstijd en het geheugengebruik uitgevoerd.

Summary

This dissertation addresses the pressing need for robust and secure software solutions in critical infrastructure systems, where sudden failures could have dire consequences. It underscores the importance of timely software updates, especially in industries like automotive, healthcare, and transportation, while emphasizing the necessity for seamless rollback mechanisms in case of failures.

The study introduces WebAssembly and WebAssembly System Interface as promising technologies for such applications, but identifies a critical gap in their support for the widely used I2C protocol in IoT devices.

The primary objective of this research is to bridge this gap by enabling I2C connectivity for Wasm applications while adhering to stringent security, reliability, and overhead constraints. Additionally, the dissertation aims to standardize this approach, thereby promoting its widespread adoption in the industry and ensuring long-term sustainability of critical infrastructure systems.

To facilitate this, the `wasi-i2c` interfaces are proposed. These interfaces follow the capability-based security design principle, which is key inside WASI. To ascertain the soundness of `wasi-i2c` implementations are provided. These implementations range over a variety of runtimes, platforms, and architectures. Afterwards, an evaluation on the mean execution time and memory usage is performed.

Verderzetten van het I2C-voorstel voor WebAssembly System Interface

Friedrich Vandenberghe

dr. ing. Merlijn Sebrechts

dr. ing. Tom Goethals

prof. dr. ir. Filip De Turck

prof. dr. Bruno Volckaert

Abstract— W3C WebAssembly (Wasm), en WebAssembly System Interface (WASI), is de technologie die tegemoet wil komen aan de behoefte aan technologie die veilig, betrouwbaar en gemakkelijk te updaten is, terwijl ze slechts een minimale overhead vereist. Maar op dit moment zijn er nog geen interfaces voor het I2C protocol. Dit artikel biedt het wasi-i2c voorstel, dat zich in de tweede fase van het WASI standaardisatieproces bevindt. De geleverde implementaties bevestigen de haalbaarheid van wasi-i2c en laten ook een verwaarloosbare overhead zien, behalve het geheugengebruik van de Wasmtime runtime.

Index terms—WebAssembly, I2C, WASI, Embedded devices

I. INTRODUCTIE

Er draaien talloze softwareoplossingen op kritieke infrastructuur. Daarnaast zijn consumentenproducten, variërend van babymonitoren tot smartwatches, alomtegenwoordig in ieders leven. Daarom heeft het Europees Parlement op twaalf maart 2024 de Cyber Resilience Act goedgekeurd [1]. Deze wet verplicht fabrikanten er onder andere voor te zorgen dat kwetsbaarheden tijdens de ondersteuningsperiode effectief worden aangepakt. Bovendien moeten beveiligingsupdates beschikbaar worden gesteld aan gebruikers gedurende de periode dat het product naar verwachting in gebruik zal zijn [2]. Daarnaast is er in de auto-industrie een snel groeiende praktijk van het draadloos distribueren van software-updates naar voertuigen, over-the-air [3] genaamd. Wanneer een dergelijke update niet werkt, moet het mogelijk zijn om deze wijziging terug te draaien zonder gevolgen voor de eindgebruiker. Beveiliging en betrouwbaarheid zijn dus van het grootste belang in deze use-cases. Bovendien zijn deze oplossingen vaak nauw geïntegreerd met IoT-apparaten met beperkte hardwaremogelijkheden. Dit verhoogt de eisen met de vraag naar een minimale overhead.

W3C WebAssembly (Wasm) en WebAssembly System Interface (WASI) is de technologie die aan al deze eisen voldoet. Gemaakt om binaire code in de browser uit te voeren, wordt het nu actief buiten deze omgeving gebruikt via WASI. Deze systeeminterface is een verzameling API's die interactie met

het bestandssysteem, HTTP-aanroepen, een Command-Line Interface enz. mogelijk maakt. Met de komst van de tweede preview release kwam ook het component model uit, een architectuur voor het bouwen van interoperabele Wasm bibliotheken, applicaties en omgevingen. Maar op dit moment zijn er nog geen interfaces voor het I2C protocol. Dit protocol definieert een seriële communicatiebus die veel gebruikt wordt in het Internet of Things (IoT) ecosysteem.

Het doel van dit artikel is om de I2C-verbinding tussen Wasm-applicaties en de onderliggende hardware en sensoren te vergemakkelijken, terwijl nog steeds aan de eisen wordt voldaan. Bovendien is deze methode ook gestandaardiseerd. Dit leidt tot de volgende onderzoeksvragen:

- RQ1.** Hoe kan een WebAssembly-component een apparaat besturen met behulp van I2C, terwijl de migratiekosten van bestaande toepassingen naar Wasm zo laag mogelijk worden gehouden?
- RQ2.** Hoe kan deze methode als onderdeel van WASI gestandaardiseerd worden?
- RQ3.** Hoe kan capabiliteit-gebaseerde Beveiliging toegepast worden op deze API?
- RQ4.** Wat is de overhead van deze methode?
- RQ5.** Hoe geschikt is het Component Model voor apparaten met beperkte mogelijkheden?

Hoofdstuk II voorziet een overzicht van het standaardisatieproces. Hoofdstuk III toont enkele implementaties die gebruik maken van I2C als deel van Wasm. Hierop volgend, Hoofdstuk IV geeft een zicht op de overhead.

II. ARCHITECTUUR EN STANDAARD

In het begin was het I2C-voorstel [4] slechts een idee, zonder voorstel of implementatie en dus nog steeds in fase nul. Maar nu bevindt het zich in de tweede fase, met voortdurende inspanningen om te voldoen aan de criteria om de stemming naar de derde fase door te laten gaan. Aan de volgende eisen moet nog worden voldaan:

- In het voorstel ontbreekt een testsuite die de functie dekt.

- Er zijn nog enkele pull requests en issues open die itereren op het ontwerp van de functie.
- Updates voor de referentie-interpreter zijn nog niet vereist maar wel aanbevolen.

Deze inspanning wordt geleid door bepaalde leiders, voor dit voorstel zijn dat Friedrich Vandenberghe, Merlijn Sebrechts en Maximilian Seidler. Friedrich en Merlijn komen allebei van de UGent, Maximilian is van Siemens. Deze mix van academici en mensen uit de industrie zorgt voor voortdurende standaardisatie-inspanningen en daadwerkelijk gebruik van de functie.

Het voorstel bevat drie Wasm Interface Type (WIT) bestanden die het componentenmodel volgen: `delay`, `i2c` en `world`. De eerste twee volgen nauwgezet de corresponderende interfaces uit `embedded-hal` versie 1, maar deze crate is niet noodzakelijk voor een implementatie. Ze definiëren ook twee handles, die vrij brede toegang bieden. Een fijnmaziger besturingsmodel is mogelijk, maar daar is momenteel geen vraag naar.

A. Asynchrone I2C

Momenteel is het voorstel puur synchroon. Om een asynchrone (async) API te bieden, zijn er twee mogelijke manieren om dit aan te pakken:

- Er zou een aparte WIT API kunnen komen die afhankelijk is van `wasi-io` polling. Dit kan worden geïmplementeerd met de huidige tooling.
- Wacht tot het Component Model async van nature integreert. Het grote voordeel van deze benadering is dat een enkele WIT beschrijving zowel een sync als async API kan beschrijven. Caller en callee kunnen elk onafhankelijk kiezen of ze sync of async willen zijn, en ze kunnen gekoppeld worden.

Omdat hier niet direct behoefte aan is, hebben de leiders besloten dat het het beste is om te wachten tot de geïntegreerde optie mainstream is.

B. Portabiliteits criteria

Platform	Architectuur	Hardware
Linux	ARM64	Raspberry Pi 3 Model B
RTOS	RISC-V	ESP32-C2
RTOS	ARM32	Nucleo F412ZG

Tabel 1: `wasi-i2c` capabiliteit criteria

De overdraagbaarheidscriteria laten zien dat het voorstel niet overgespecificeerd is voor een bepaald platform. De leiders hebben, samen met mensen van Siemens, Intel, Aptiv, Xiaomi, Amazon, Midokura, Sony, Atym en Bosch, overeenstemming bereikt over een reeks criteria voor het voorstel. Deze criteria zijn te vinden in Tabel 1. Voor elk criterium wordt referentiehardware geleverd, zonder deze zou er nog steeds veel variatie mogelijk zijn in RAM-grootte, flash-grootte en CPU-

snelheid. Hierin ontbreekt Intel x86, maar ondersteuning hiervoor wordt geïmpliceerd omdat dit de gebruikte architectuur is voor de ontwikkeling van Wasmtime. De Nucleo is gekozen omdat dit het referentiebord van Siemens is. De andere twee zijn gekozen om ondersteuning voor zowel RTOS en Linux, als ARM en RISC-V te laten zien. Als extra vereiste naast de targets, is het belangrijk dat de interface zo ontworpen is dat het geheugengebruik beperkt wordt tot het hoogst noodzakelijke. Dit om ervoor te zorgen dat er genoeg RAM overblijft voor de applicatie zelf.

III. IMPLEMENTATIES

De implementaties controleren de deugdelijkheid van de WIT-interfaces van `wasi-i2c` [5]. Binnen I2C zijn er drie basistypen methoden, waarvan er twee zijn geïmplementeerd, nl. schrijven en lezen. Voor het schrijven gebruiken de implementaties een 4-cijferig 7-segment display en voor het lezen een HTS221 sensor, waarvan de huidige temperatuur en vochtigheid kan worden afgelezen.

Verbonden apparaat	Hardware
HTS221 sensor	Raspberry Pi 3 Model B
4-digit 7-segment display	Raspberry Pi 4 Model B
4-digit 7-segment display	Raspberry Pi Pico

Tabel 2: Concise overview of the setups

De sensor maakt deel uit van een hardware Attached on Top (HAT) die bovenop een Raspberry Pi 3 Model B is gemonteerd via de 40-pins GPIO header. Hoewel het op deze manier is aangesloten, gebeurt het uitlezen van de sensor nog steeds via I2C. Het display daarentegen is een zelfstandig apparaat en kan dus eenvoudig worden aangesloten op een andere controller. Als controller gebruiken we een Raspberry Pi 4 Model B en een Raspberry Pi Pico. Zowel de Pi 3 als de Pi 4 draaien op een ARM64-architectuur en een Linux-platform. De Pico daarentegen is een microcontroller die gericht is op de RP2040. Een overzicht is te vinden in Tabel 2. De gebruikte platformen en architectuurcombinaties verschillen van de overdraagbaarheidscriteria, omdat deze zijn vastgesteld voordat de criteria werden vastgesteld.

A. Native implementaties

De Pi 4 kan de codebase bouwen maar de andere twee vereisen cross-compilatie. In Rust wordt dit uitgevoerd met de target vlag. Voor ARM64 is dit `aarch64-unknown-linux-gnu` en voor RP2040 is dit `thumbv6m-none-eabi`. De Pico heeft nog enkele andere vereisten. Ten eerste ondersteunt het alleen `no_std` builds. Ten tweede moet de geheugenlayout gespecificeerd worden in een bestand genaamd `memory.x`.

Om eenvoudig te intraheren met de HTS221 sensor, kan de HTS221 Rust crate gebruikt worden. Voor `embedded-hal` ver-

sie 1 integratie moest een fork [6] worden aangemaakt. Verder gebruikt de implementatie `linux-embedded-hal` voor een I2C-verbinding. Voor de implementatie op de Pi 4 schrijven we de huidige lokale tijd. In dit geval gebruikt de I2C-verbinding `rppal`. Deze bibliotheek volgt ook de `embedded-hal` API, maar is speciaal gemaakt voor Raspberry Pi-apparaten. Op de Pico wordt in plaats daarvan een eenvoudige oplopende teller weergegeven. Hiervoor wordt de `rp-pico crate` gebruikt.

B. Implementaties binnen WebAssembly

Om de overstap van native naar WebAssembly te maken, wordt alles met betrekking tot de I2C-verbinding zelf aan de host-zijde gehouden. De apparaatspecifieke logica wordt verplaatst naar de gast. Idealiter komt elke implementatie, conceptueel gezien, neer op het schema van een gastcomponent en een host runtime die aan elkaar zijn gelijmd via een WIT-wereld. Maar door de talloze verschillen tussen Wasmtime en WAMR kan er geen WebAssembly-specifieke code tussen beide worden hergebruikt. Deze implementatie houdt daarom alles waarmee elke runtime interageert in twee afzonderlijke mappen.

1) Implementaties binnen Wasmtime:

De Wasmtime-implementatie bestaat uit drie mappen: `gast`, `host` en `wit`. De `wit` directory specificeert twee werelden, die overeenkomen met de twee apparaten waarmee wordt geïnteractueerd.

WIT world:

```
world sensor {
  import wasi:i2c/i2c@0.2.0-draft;
  export hts: interface {
    use wasi:i2c/i2c@0.2.0-draft.{i2c, error-code};
    get-temperature: func(connection: i2c) ->
      result<string, error-code>;
    get-humidity: func(connection: i2c) ->
      result<string, error-code>;
  }
}
```

Listing 1: One van de WIT worlds waarmee guest en host connecteren.

Zowel de gastcomponenten als de Wasmtime hostimplementatie binden aan werelden die verwant zijn aan die van Listing 1. De HTS221 koppelt aan deze wereld en `wasi:i2c` is het voorstel.

Guest code:

Om de grootte van de gegenereerde Wasm-binaire bestanden te beperken, maken de gastimplementaties geen gebruik van Rust's `std`. Voor het genereren van de gastcomponenten is een adapter nodig. Typisch maakt deze adapter het gebruik van `std` mogelijk, maar in dit geval wordt een lege adapter meegegeven. Merk op dat dit `#![feature(...)]` gebruikt, wat het gebruik van de nightly versie van Rust vereist. Listing 2

beschrijft hoe, conceptueel, elke componentimplementatie eruit ziet.

Host code:

De WIT-interface importeert (delen van) de `wasi:i2c`-interfaces. Daarom is het niet mogelijk om componenten te draaien via de `wasmtime` commandoregel en is een aangepaste host nodig. De meest eenvoudige versie van deze aangepaste host zou simpelweg een standalone implementatie van elke wereld zijn. Elke wereldimplementatie heeft dan zijn eigen invulling van de imports, wat resulteert in veel codevernietiging. Daarom is een meer betrokken hostimplementatie praktischer. Hier kunnen de wereldimplementaties hun gebruik van de I2C-bindingen voor het vervullen van de imports delen via het `with` sleutelwoord. Het is belangrijk dat deze paden volledig gekwalificeerd zijn. Anders zal de Rust compiler fouten geven.

2) Implementaties binnen WAMR:

WebAssembly Micro-Runtime (WAMR) heeft geen ondersteuning voor WASI preview twee. Deze applicatie kan dus geen gebruik maken van WIT, noch een goede I2C-verbinding doorgeven aan de gast omdat er geen notie is van een WIT-bron. Daarom wordt de verbinding nu globaal gehouden. In Rust is er geen standaard manier om een variabele globaal te houden, maar hier gebruiken de implementaties de `once_cell` krat.

```
#![no_std]
#![no_main]
mod bindings;

// To make the generated bindings compatible with the
// `embedded-hal` API
// Imported from the `wasi-embedded-hal` crate
add_i2c_hal!(i2c);

struct Component {}

impl Guest for Component {
  // Omitting implementations of the exports inside
  // of the binded world

  // Omitting the definition of a global allocator and
  // a panic handler,
  // needed by Rust because of `no_std`.

  bindings::export!(Component with_types_in bindings);
}
```

Listing 2: Gestripte versie van een gastcomponent voor Wasmtime.

```

#![no_std]
#![no_main]

#[link(wasm_import_module = "host")]
extern "C" {
    // The term _slave_ is still used because `rppal`
    // still uses this.
    fn host_i2c_write(slave_address: u16, data: u8);
}

#[export_name = "setup"]
pub fn setup() { /* Omitting definition */ }
#[export_name = "write"]
pub fn write(d0: i32, d1: i32, d2: i32, d3: i32) { /
/* Omitting definition */ }

// Omitting the definition of a global allocator and
// a panic handler,
// needed by Rust because of `no_std`.

```

Listing 3: Gestripte versie van een gastcomponent voor WAMR.

Alle gast-naar-heer en heer-naar-gast communicatie wordt als onveilig beschouwd op basis van geheugenveiligheid. Dit komt omdat WAMR is geschreven in C. Bovendien schrijft de WAMR Rust SDK voor dat imports en exports worden aangeroepen en ontwikkeld als externe C-functies. Dit leidt tot een module-implementatie geconceptualiseerd door Listing 3. De host kan niet direct binden met de resulterende binary, in plaats daarvan moet het de import registreren en de exports vinden. Voor het doorgeven van parameters aan de exports wordt een vector van `WasmValues` doorgegeven. Een `WasmValue` kan een void, 32-bit signed integer, 64-bit signed integer, 32-bit float, 64-bit float of een 128-bit signed integer zijn. Elke waarde wordt onderliggend opgeslagen als een array van 32-bits unsigned integers. De geëxporteerde setup functie kan dan aangeroepen worden met een array die een `WasmValue::Void` bevat, voor write wordt een vector van vier `WasmValue::I32`'s doorgegeven.

Op de Raspberry Pi Pico is er geen bestandssysteem, dus geen mogelijkheid om de Wasm binary van schijf te lezen. In plaats daarvan kan er een hexdump van de binary genomen worden, waarna deze bytes hard gecodeerd kunnen worden als een vector van unsigned 8-bit integers in de host.

Onder de motorkap gebruikt Rust `rust-lld` voor de cross-compilatie naar de Pico [7], maar WAMR gebruikt de `arm-none-eabi-gcc` linker. Dit resulteert in een CMAKE-fout bij het bouwen van de SDK [8], waardoor cross-compilatie wordt uitgeschakeld.

IV. EVALUATIE

A. Evaluatie setup

De evaluatieopstelling is gebaseerd op de opstellingen die zijn gebruikt voor de implementaties, zoals weergegeven in Hoofdstuk III. Deze bestaat uit het volgende:

- Een Raspberry Pi 4 Model B met 8 GB RAM aangesloten op een 4-cijferige 7-segment LED.
- Een Raspberry Pi 3 Model B met een aangesloten HTS221 sensor.

De opstelling evalueert het display op zichzelf en met Wasmtime en WAMR als runtime. Hetzelfde geldt voor de sensor, maar zonder de WAMR runtime.

B. Functionele evaluatie

Er worden twee scenario's getest:

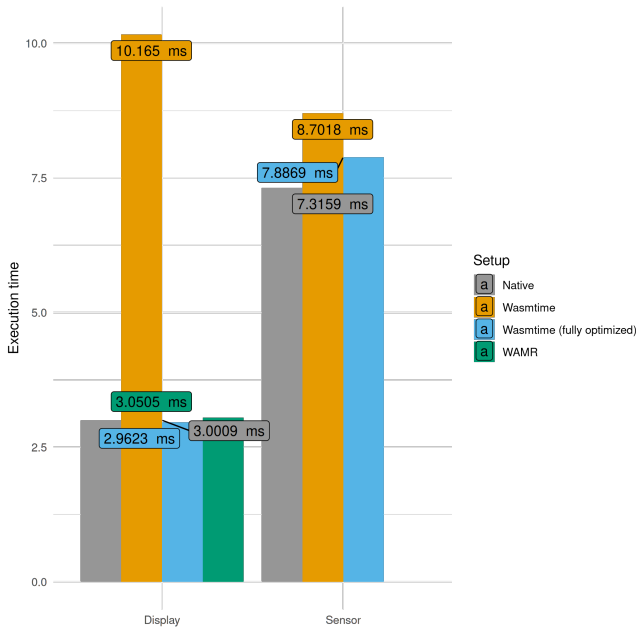
- Op de Raspberry Pi 3 wordt de temperatuur uitgelezen. Deze waarde wordt handmatig geverifieerd met de native implementatie.
- Op de Raspberry Pi 4 wordt de string "1234" naar het scherm geschreven. Dit wordt visueel gecontroleerd.

C. Benchmarks

Het draaien in Wasmtime zou slechter moeten presteren door de opgelegde overhead op zowel uitvoeringstijd als geheugengebruik, maar deze overhead zal te verwaarlozen zijn. Voor WAMR zou de overhead nog kleiner moeten zijn.

Eerst wordt de uitvoeringstijd geïnspecteerd. Een overzicht is beschikbaar in Figuur 1. Dit overzicht laat een verbazingwekkend langere uitvoeringstijd zien wanneer het programma in Wasmtime wordt uitgevoerd. Het grootste deel van deze tijd wordt doorgebracht in functies die gerelateerd zijn aan Canelift.

Canelift van Bytecode Alliance is een compiler backend die onder andere gebruikt wordt door Wasmtime voor just-in-time en ahead-of-time compilatie. Via de ahead-of-time functionaliteit is het mogelijk om de gemiddelde executietijd sterk te reduceren. Dit kan worden gedaan via `wasmtime compile` of de `Component::serialize` functie in Wasmtime. Om een voorgecompileerd bestand te laden, moet de functie `Component::deserialize_file` worden gebruikt. Dit bestand bevat native, niet-overdraagbare binaire code die niet zal werken op een andere architectuur, en mogelijk zelfs niet werkt op verschillende processormodellen binnen dezelfde architectuur. Het is dus niet haalbaar om een voorgecompileerde Wasm binary te distribueren in plaats van de Wasm binary zelf.



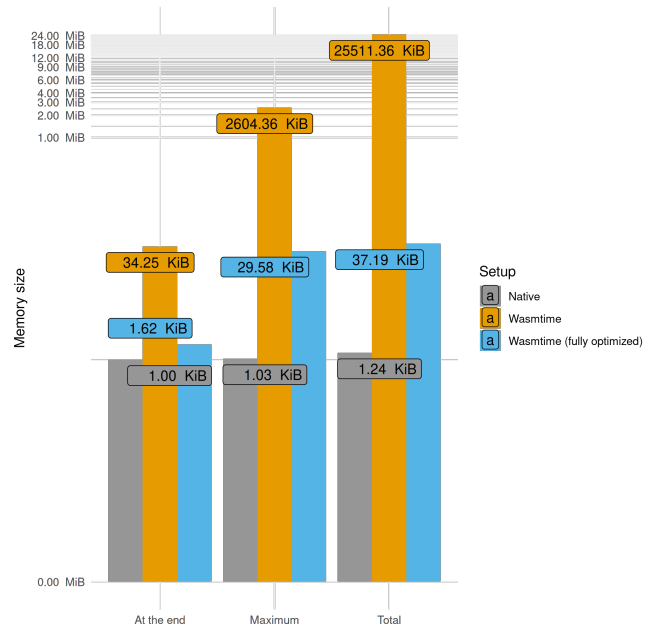
Figuur 1: Overzicht van alle gemiddelde uitvoeringstijden

Een verdere optimalisatie kan worden gedaan door de opdrachtwereld niet langer te koppelen. Deze wereld is niet nodig vanwege de aangepaste hostopstelling. Nu is de volledig geoptimaliseerde Wasmtime build iets sneller bij het schrijven naar het scherm dan de native versie. Deze versnelling kan te wijten zijn aan ruis of kernel intrinsics.

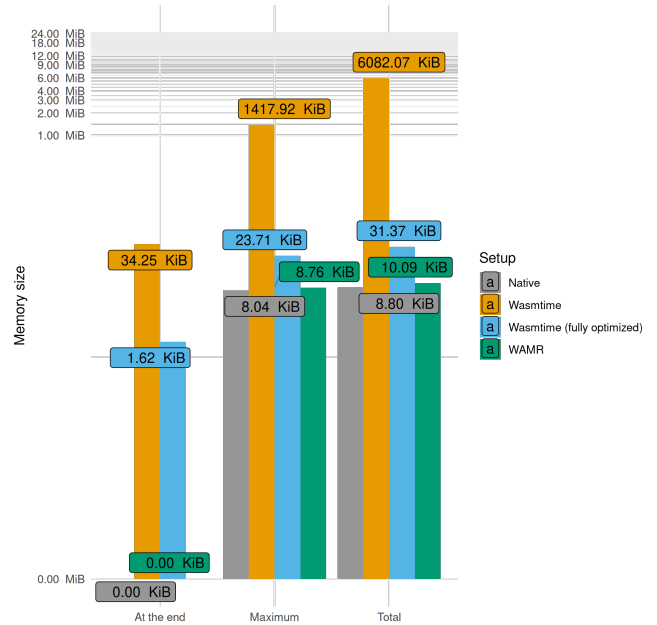
Als we WAMR vergelijken met de snelste Wasmtime-versie voor het schrijven naar de display, zien we dat Wasmtime beter presteert, dit met 88,2 microseconden. Dit verschil is niet groot genoeg om de ene sneller of langzamer te laten zijn dan de andere.

Als we kijken naar het geheugengebruik, Figuur 2 en Figuur 3, vallen drie verschijnselen op. Ten eerste hebben de uitgevoerde optimalisaties om de Wasmtime-implementatie sneller te laten werken ook een groot voordeel voor het geheugengebruik. Ten tweede heeft WAMR bijna native geheugengebruik, terwijl Wasmtime bijna vier keer meer geheugen gebruikt in vergelijking met de native versie. Als derde en laatste is het geheugengebruik van de volledig geoptimaliseerde versie van Wasmtime redelijk vergelijkbaar tussen het schrijven naar de display en het lezen van de sensor. Dit laat een basishoeveelheid geheugen zien die nodig is om het componentmodel uit te voeren.

Concluderend hebben beide oplossingen voor Wasmtime veel baat gehad bij het voorcompileren van de binary van Wasm. Bovendien zorgt de voorcompilatie ervoor dat Wasmtime qua tijd op gelijke hoogte komt met WAMR en native, maar Wasmtime heeft nog steeds een aanzienlijk grotere hoeveelheid geheugen nodig om succesvol te draaien.



Figuur 2: Gebruikt geheugen bij het lezen van de sensor



Figuur 3: Gebruikt geheugen bij het schrijven naar het scherm

V. CONCLUSIE

Dankzij de interfaces in wasi-i2c is het mogelijk voor een Wasm component om een apparaat te besturen via I2C. Dit beantwoordt dus de eerste onderzoeksvraag. Om eventuele migratiekosten van bestaande toepassingen naar WebAssembly te minimaliseren, wordt de wasi-embedded-hal crate voorzien. Met deze crate is het mogelijk om eenvoudig samen te werken met apparaatcrates binnen het Rust embedded-hal

ecosysteem. Hoewel het voorstel sterk geïnspireerd is door dit ecosysteem, is het zo ontworpen dat geen enkele implementatie deze API hoeft te gebruiken om succesvol te zijn.

Verder, om de tweede onderzoeksvraag op te lossen, zijn de bovengenoemde interfaces gebracht van een louter idee tot volwaardige interfaces klaar voor feedback van de WebAssembly Community Group. Dus, van fase nul naar fase twee. Er zijn twee mogelijkheden, één voor I2C en één voor vertragen. Meer verfijnde controle over deze mogelijkheden is iets dat momenteel ontbreekt, maar zeker haalbaar is.

Om de vierde vraag te beantwoorden is de overhead gedefinieerd als een dichotomie van de gemiddelde uitvoeringstijd en het piekgeheugengebruik. Met slechts een toename van 1,65% in de gemiddelde uitvoeringstijd en een 8,93% hoger piekgeheugengebruik over de volledige levensduur van de applicatie, kan de overhead van de WAMR runtime als verwaarloosbaar worden beschouwd. Voor Wasmtime gelden enkele overwegingen. Ten eerste heeft elke WASI-interface die aan de linker wordt toegevoegd extra overhead. Ten tweede en als laatste, alvorens de component zelf uit te voeren, moet de runtime

Dit geeft aan dat Wasmtime, en bij uitbreiding het componentenmodel, een flinke geheugenoverhead heeft. Er wordt aan gewerkt om deze overhead meer in overeenstemming te brengen met die van WAMR, maar dit is nog in een vroeg stadium. Dus, om de vijfde onderzoeksvraag te beantwoorden, in zijn huidige staat is het WASI componentenmodel niet geschikt om te draaien op apparaten met beperkte middelen.

REFERENCES

- [1] B. Chatain, 'Cyber Resilience Act: MEPs adopt plans to boost security of digital products', *European Parliament*, mrt. 2024, [Online]. Beschikbaar op: <https://www.europarl.europa.eu/news/en/press-room/20240308IPR18991/cyber-resilience-act-meps-adopt-plans-to-boost-security-of-digital-products>
- [2] E. Commission, 'EU Cyber Resilience Act'. december 2023.
- [3] Aptiv, 'What Is Over-the-Air (OTA)?', *Mobility Insider*, jun. 2020, [Online]. Beschikbaar op: [https://www.aptiv.com/en/insights/article/what-is-over-the-air-\(ota\)](https://www.aptiv.com/en/insights/article/what-is-over-the-air-(ota))
- [4] F. Vandenberghe, L. Clark, en J. Schilling, 'WASI I2C'.
- [5] F. Vandenberghe, 'i2c-wasm-components'.
- [6] F. Vandenberghe, 'hts221'.
- [7] E. R. W. Group, 'PSA: Cortex-M Breakage (LLD as the default linker)'. augustus 2018.
- [8] F. Vandenberghe, 'Compiling wamr-sys for thumbv6m-none-eabi fails'. april 2024.

Advancing the I2C proposal for WebAssembly System Interface

Friedrich Vandenberghe

dr. ing. Merlijn Sebrechts

dr. ing. Tom Goethals

prof. dr. ir. Filip De Turck

prof. dr. Bruno Volckaert

Abstract— W3C WebAssembly (Wasm), and WebAssembly System Interface (WASI), is the technology that aims to fulfil the need for technology that is secure, reliable and easily updatable, while requiring only a minimal overhead. But currently, there aren't any interfaces yet for the I2C protocol. This paper provides the `wasi-i2c` proposal, which is in the second phase of the WASI standardization process. The provided implementations ascertain the feasibility of `wasi-i2c`, and also showcase a negligible overhead. Except for the memory usage of the Wasmtime runtime.

Index terms—WebAssembly, I2C, WASI, Embedded devices

I. INTRODUCTION

There are countless software solutions running on critical infrastructure. Additionally, consumer products ranging from baby-monitors to smart-watches are omnipresent in everyone's life. That's why the European Parliament has approved the Cyber Resilience Act on the twelfth of March 2024 [1]. This Act will obligate manufacturer's to ensure, among others, that for the duration of the support period, vulnerabilities are handled effectively. Furthermore, security updates need to be made available to users for the time the product is expected to be in use [2]. Besides, in the automotive industry, there's a rapidly growing practice of wirelessly distributing software updates to vehicles, called over-the-air [3]. When such an update fails to perform, it should be possible to roll back this change without impact to the end-user. Thus, security and reliability are of utmost importance in these use-cases. Additionally, these solutions are frequently tightly integrated with IoT devices with limited hardware capabilities. This augments the requirements with the demand for a minimal overhead.

W3C WebAssembly (Wasm), and WebAssembly System Interface (WASI), is the technology that aims to fulfill all these requirements. Created to execute binary code inside the browser, it is now actively used outside this environment via WASI. This system interface is a set of APIs that facilitates interaction with the filesystem, HTTP calls, a Command-Line Interface etc. With the advent of its second preview release also came the release of the component model, an architecture for building interoperable Wasm libraries, applications, and environments. But currently, there aren't any interfaces yet

for the I2C protocol. This protocol defines a serial communication bus widely used in the Internet of Things (IoT) ecosystem.

The goal of this paper is to facilitate the I2C connection between Wasm applications and its underlying hardware and sensors, while still adhering to the requirements. Furthermore, this method is also standardized. This leads to the following research questions:

- RQ1.** How can a WebAssembly component control a device using I2C, while keeping the migration cost of existing applications to Wasm as low as possible?
- RQ2.** How can this method be standardized as part of WASI?
- RQ3.** How can capability-based security be applied to this API?
- RQ4.** What is the overhead of this method?
- RQ5.** How suitable is the WASI Component Model for micro-controllers with constrained resources.

Section II provides an overview of the standardization effort. Section III demonstrates implementations that make use of I2C as part of Wasm. Following up, Section IV gives an in-depth review of the overhead.

II. ARCHITECTURE AND STANDARD

At the start, the I2C proposal [4] was merely an idea, lacking any proposal or implementation, and thus still residing in phase zero. But now it is in the second phase, with ongoing effort to fulfill the criteria to pass the vote to the third phase. The following requirements are yet to be met:

- The proposal lacks a test suite that covers the feature.
- Some pull requests and issues are still open that iterate on the design of the feature.
- Updates to the reference interpreter are not yet required, but recommended.

This effort is led under the guidance of certain champions, for this proposal these are Friedrich Vandenberghe, Merlijn Sebrechts and Maximilian Seidler. Both Friedrich and Merlijn are from UGent, Maximilian is from Siemens. This mix of academicians and people from the industry ensures ongoing standardization effort and actual usage of the feature.

The proposal contains three Wasm Interface Type (WIT) files that follow the component model: `delay`, `i2c` and `world`. The first two closely follow the corresponding interfaces from

embedded-hal version 1, but this crate is not necessary for an implementation. They also define two handles, which provide pretty broad access. A more fine-grained control model is possible, but there's currently no demand for this.

A. Asynchronous I2C

Currently, the proposal is purely synchronous. To provide an asynchronous (async) API, there are two possible ways to tackle this:

- A separate WIT API could be provided that depends on `wasi-io` polling. This could be implemented with the current tooling.
- Wait for the Component Model to natively integrate async. The major upside to this approach is that a single WIT description can describe both a sync and async API. Caller and callee can each independently choose if they want to be sync or async, and they can be linked.

As there's no immediate need for this, the champions have decided that it is best to wait until the integrated option is mainstream.

B. Portability criteria

Platform	Architecture	Reference hardware
Linux	ARM64	Raspberry Pi 3 Model B
RTOS	RISC-V	ESP32-C2
RTOS	ARM32	Nucleo F412ZG

Table 1: `wasi-i2c` capability criteria

The portability criteria show that the proposal isn't overspecified for a particular platform. The champions, together with people from Siemens, Intel, Aptiv, Xiaomi, Amazon, Midokura, Sony, Atym and Bosch, have agreed upon a set of criteria for the proposal. These criteria can be found in Table 1. For each criterium, reference hardware is provided, without these there would still be a lot of possible variety in RAM size, flash size and CPU speed. Herein Intel x86 is missing, but support for this is implied as this is the used architecture for developing Wasmtime. The Nucleo is chosen because this is the reference board for Siemens. The other two are chosen to showcase support for both RTOS and Linux, and ARM and RISC-V. As an extra requirement besides the targets, it is important that the interface is designed in such a way that memory usage is limited to the bare needed minimum. To ensure enough RAM left for the application itself.

III. IMPLEMENTATIONS

The implementations ascertain the soundness of the WIT interfaces from `wasi:i2c` [5]. Inside I2C there are three basic types of methods, from which two are implemented, write and read. For writing, the implementations use a 4-digit 7-segment

display, and for reading a HTS221 sensor, from which the current temperature and humidity can be read.

Connected device	Hardware
HTS221 sensor	Raspberry Pi 3 Model B
4-digit 7-segment display	Raspberry Pi 4 Model B
4-digit 7-segment display	Raspberry Pi Pico

Table 2: Concise overview of the setups

The sensor is part of a hardware Attached on Top (HAT) mounted atop of a Raspberry Pi 3 Model B via its 40-pin GPIO header. Although it is connected this way, reading from the sensor is still done via I2C. The display, on the other hand, is a standalone device and can thus easily be hooked up with a different controller. As a controller, we use a Raspberry Pi 4 Model B and a Raspberry Pi Pico. Both the Pi 3 and the Pi 4 run on an ARM64 architecture and a Linux platform, contrarily, the Pico is a microcontroller targeting RP2040. An overview can be found in Table 2. The used platform and architecture combinations are different from the portability criteria because these were decided before the criteria were decided upon.

A. Native implementations

The Pi 4 can build the codebase, but the other two require cross-compilation. In Rust, this is performed with the `target` flag. For ARM64 this is `aarch64-unknown-linux-gnu`, and for RP2040 it is `thumbv6m-none-eabi`. The Pico has some further requirements. First, it only supports `no_std` builds. Second, the memory layout needs to be specified inside a file called `memory.x`.

To easily interface with the hts221 sensor, the hts221 Rust crate can be used. For embedded-hal version 1 integration, a fork had [6] to be created. Furthermore, to have an I2C connection, the implementation uses `linux-embedded-hal`. For the implementation running on the Pi 4 we write the current local time. In this case, the I2C connection uses `rppal`. This library also follows the embedded-hal API, but is specifically tailored for Raspberry Pi devices. On the Pico, a simple incrementing counter is instead displayed. For this, it uses the `rp-pico` crate.

B. Implementations inside WebAssembly

To make the move from native to WebAssembly, everything related to the I2C connection itself is kept host-side. The device-specific logic, is moved to the guest. Ideally, each implementation, conceptually, comes down to the scheme of a guest component and a host runtime glued together via a WIT world. But due to the myriad of differences between Wasmtime and WAMR, no WebAssembly-specific code can be reused between them. This implementation, therefore, keeps everything each runtime interacts with inside two separate directories.

1) Implementations inside Wasmtime:

The Wasmtime implementation consists of three directories: guest, host and wit. The wit directory specifies two worlds, corresponding with the two devices being interacted with.

WIT world:

```
world sensor {
    import wasi:i2c/i2c@0.2.0-draft;
    export hts: interface {
        use wasi:i2c/i2c@0.2.0-draft.{i2c, error-code};
        get-temperature: func(connection: i2c) ->
            result<string, error-code>;
        get-humidity: func(connection: i2c) ->
            result<string, error-code>;
    }
}
```

Listing 1: One of the WIT worlds to which guest and host bind.

Both the guest components, as the Wasmtime host implementation, bind to worlds akin to the one from Listing 1. The HTS221 maps to this world, and wasi:i2c is the proposal.

Guest code:

To reduce the size of the generated Wasm binaries, the guest implementations don't make use of Rust's std. For the generation of the guest components, an adapter is required. Typically this adapter enables the usage of the std, but in this case, an empty adapter is given. Note that this uses `#![feature(...)]`, which requires the usage of the nightly version of Rust. Listing 2 describes how, conceptually, each component implementation looks like.

Host code:

The WIT interface imports (parts of) the wasi:i2c interfaces. Therefore, it is not possible to run components via the wasmtime command line, and a custom host is necessary. The most basic version of this custom host would simply be a standalone implementation of each world. Every world implementation then has their own fulfilment of the imports, resulting in lots of code redundancy. Therefore, a more involved host implementation, is more practical. Here, the world implementations can share their usage of the I2C bindings for the fulfilment of the imports via the `with` keyword. It is important that these paths be fully qualified. Otherwise, errors will be thrown by the Rust compiler.

2) Implementations inside WAMR:

WebAssembly Micro-Runtime (WAMR) has no support for WASI preview two. This application, thus, cannot make use of WIT, nor pass a proper I2C connection to the guest because there's no notion of a WIT resource. Therefore, the connection is now kept global. In Rust, there's no default way to keep a variable global, but here the implementations use the `once_cell` crate.

```
#![no_std]
#![no_main]
mod bindings;

// To make the generated bindings compatible with the
// `embedded-hal` API
// Imported from the `wasi-embedded-hal` crate
add_i2c_hal!(i2c);

struct Component {}

impl Guest for Component {
    // Omitting implementations of the exports inside
    // of the binded world
}

// Omitting the definition of a global allocator and
// a panic handler,
// needed by Rust because of `no_std`.
```

`bindings::export!(Component with_types_in bindings);`

Listing 2: Stripped down version of a guest component for Wasmtime.

```
#![no_std]
#![no_main]

#[link(wasm_import_module = "host")]
extern "C" {
    // The term _slave_ is still used because `rppal`
    // still uses this.
    fn host_i2c_write(slave_address: u16, data: u8);
}

#[export_name = "setup"]
pub fn setup() { /* Omitting definition */ }
#[export_name = "write"]
pub fn write(d0: i32, d1: i32, d2: i32, d3: i32) { /
    * Omitting definition */ }

// Omitting the definition of a global allocator and
// a panic handler,
// needed by Rust because of `no_std`.
```

Listing 3: Stripped down version of a guest module for WAMR.

All guest-to-host and host-to-guest communication is considered unsafe on a memory-safety basis, this is because WAMR is written in C. Furthermore, the WAMR Rust SDK mandates that imports and exports are called and developed as extern C functions. This leads to a module implementation conceptualized by Listing 3. The host cannot directly bind with the resulting binary, instead it has to register the import and find the exports. For passing parameters to the exports, a vector of `WasmValues` is passed. A `WasmValue` can be a void, 32-bit signed integer, 64-bit signed integer, 32-bit float, 64-bit float or a 128-bit signed integer. Each value is underlying stored as an array of 32-bit unsigned integers. The exported setup function can

then be called with an array containing a `WasmValue::Void`, for write a vector of four `WasmValue::I32`'s are passed.

On the Raspberry Pi Pico, there's no filesystem, thus no capability of reading the Wasm binary from disk. Instead, a hexdump can be taken from the binary, and then these bytes can be hardcoded as a vector of unsigned 8-bit integers inside the host.

Under the hood, Rust uses `rust-ld` for the cross-compilation to the Pico [7], but WAMR uses the `arm-none-eabi-gcc` linker. This results in a CMAKE error when building the SDK [8], and thus disabling cross-compilation.

IV. EVALUATION

A. Evaluation setup

The evaluation setup is based upon the setups used for the implementations, as shown in Section III. It consists of the following:

- A Raspberry Pi 4 Model B with 8 GB of RAM hooked up with a 4-digit 7-segment LED
- A Raspberry Pi 3 Model B with an attached HTS221 sensor

The setup evaluates the display natively and with Wasmtime and WAMR as a runtime. The same applies for the sensor, but without the WAMR runtime.

B. Functional evaluation

Two scenarios are tested:

- On the Raspberry Pi 3 the temperature gets read out. This value is manually verified with the native implementation.
- On the Raspberry Pi 4 the string "1234" gets written to the display. This is visually verified.

C. Benchmarks

Running in Wasmtime should perform worse due to the imposed overhead on both execution time and memory usage, but this overhead will be negligible. For WAMR, there should be an even slimmer overhead.

First, an inspection is performed on the execution time. An overview is available in Figure 1. This overview showcases an astonishing longer execution time when running inside Wasmtime. Most of this time is spent inside functions related to Cranelift.

Bytecode Alliance's Cranelift is a compiler backend that is among others in use by Wasmtime for just-in-time and ahead-of-time compilation. Via its ahead-of-time functionality, it is possible to greatly reduce the average execution time. This can be done via `wasmtime compile` or the `Component::serialize` function inside Wasmtime. To load a precompiled file, the `Component::deserialize_file` function needs to be used.

This file contains native, non-portable binary code which will not work on a different architecture, and might not even work across different processor models within the same architecture. Thus, it is not feasible to distribute a precompiled Wasm binary, instead of the Wasm binary itself.

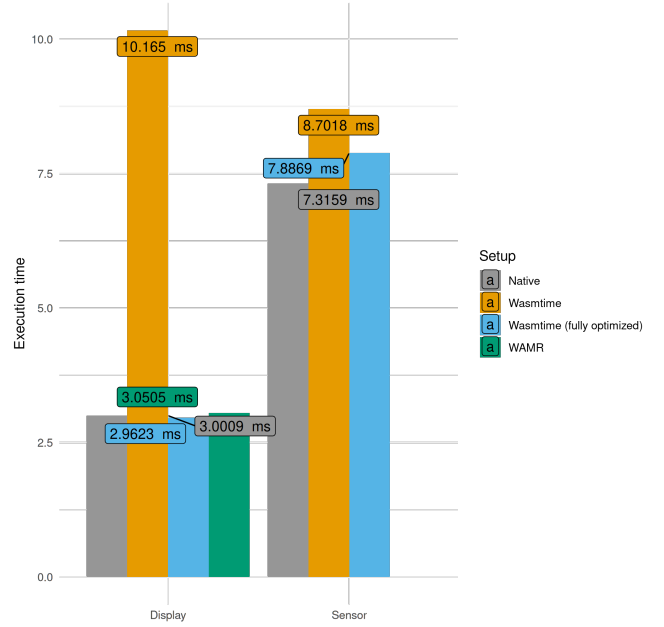


Figure 1: Overview of all the mean execution times

A further optimization can be done through no longer linking the command world. This world isn't necessary because of the custom host setup. Now, the fully optimized Wasmtime build is slightly faster when writing to the display than the native version. This speedup could be due to noise or kernel intricacies.

Comparing WAMR with the fastest Wasmtime version for writing to the display, shows that Wasmtime outperforms, with 88.2 microseconds. This is not significant enough of a difference for one to be faster or slower than the other.

When considering the memory usage, Figure 2 and Figure 3, three phenomena are striking. First, the performed optimizations to make the Wasmtime implementation run faster also greatly benefit the memory usage. Second, WAMR has near-native memory usage, while Wasmtime uses nearly four times more memory when compared to the native version. Third and last, the memory usage of the fully optimized version of Wasmtime is fairly similar between writing to the display and reading from the sensor. This showcases a baseline amount of memory required to run the component model.

To conclude, both solutions for Wasmtime greatly benefited from precompiling the Wasm binary. Furthermore, pre-compilation makes Wasmtime on par with WAMR and native time-wise, but Wasmtime still requires a substantial larger amount of memory to successfully run.

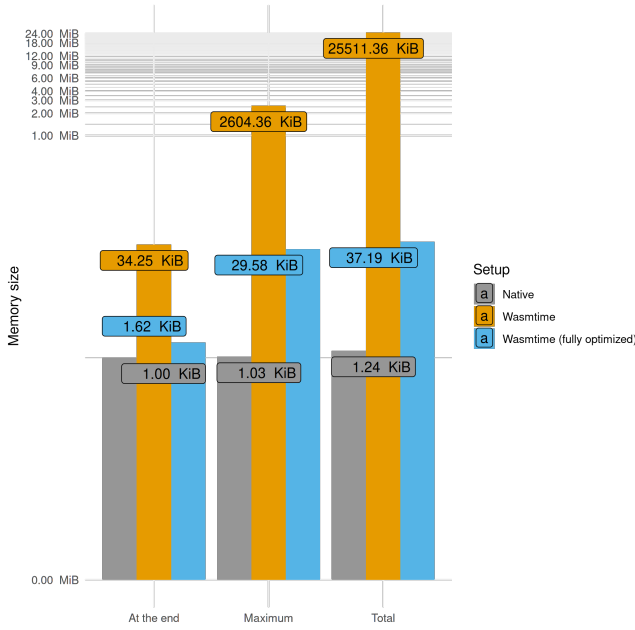


Figure 2: Used memory when reading from the sensor

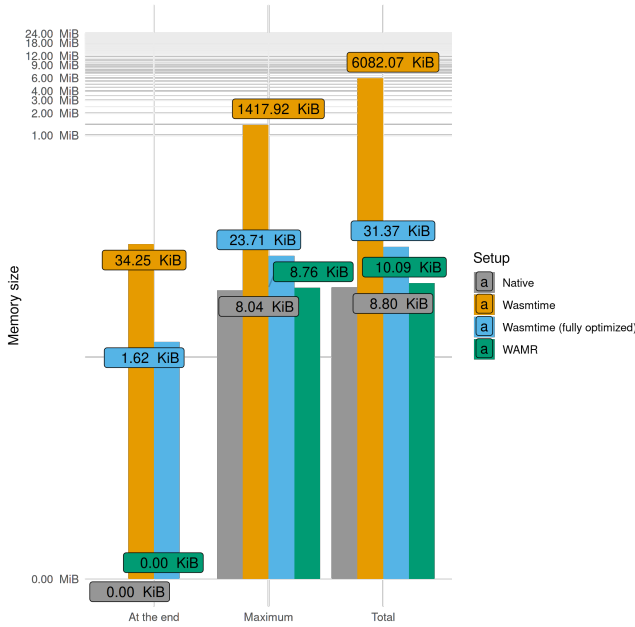


Figure 3: Used memory when writing to the display

V. CONCLUSION

Thanks to the provided interfaces inside `wasi-i2c`, it is possible for a Wasm component to control a device using I2C. Thus this answers the first research question. To minimize any migration costs from existing applications to WebAssembly, the `wasi-embedded-hal` crate is provided. With this crate, it is possible to easily interoperate with device crates inside the Rust embedded-hal ecosystem. Although the proposal is heav-

ily inspired by this ecosystem, it is designed in such a way that no implementation should need to use this API to be successful.

Furthermore, to resolve the second research question, the aforementioned interfaces have been brought from a mere idea to fully-fledged interfaces ready for feedback from the WebAssembly Community Group. Thus, from phase zero to phase two. Two capabilities are provided, one for I2C and one for delays. More fine-grained control over these capabilities is something that's currently lacking, but certainly feasible to achieve.

To address the fourth question, the overhead is defined as a dichotomy of the mean execution time and the peak memory usage. With only an increase of 1.65% in mean execution time, and having an 8.93% higher peak memory usage over the complete lifespan of the application, the overhead of the WAMR runtime can be seen as negligible. For Wasmtime, some considerations apply. First, each WASI interface added into the linker incurs extra overhead. Second and last, before executing the component itself, the runtime first instantiates it. This instantiation can be done a priori but is not guaranteed to work on other machines. For this, the runtime engine should be configured in a peculiar way. This is seen as an open challenge. With these considerations, the mean execution time is -3.86% and peak memory usage is 15.87 times more when compared to native

This indicates that Wasmtime, and by extension the component model, incurs a hefty memory overhead. Efforts are underway to align this overhead more with WAMR's, but this is in an early stage. Thus, to answer the fifth research question, in its current state the WASI component model is not suitable to run on devices with constrained resources.

REFERENCES

- [1] B. Chatain, "Cyber Resilience Act: MEPs adopt plans to boost security of digital products," *European Parliament*, Mar. 2024, [Online]. Available: <https://www.europarl.europa.eu/news/en/press-room/20240308IPR18991/cyber-resilience-act-meps-adopt-plans-to-boost-security-of-digital-products>
- [2] E. Commission, "EU Cyber Resilience Act." Dec. 2023.
- [3] Aptiv, "What Is Over-the-Air (OTA)?" *Mobility Insider*, Jun. 2020, [Online]. Available: [https://www.aptiv.com/en/insights/article/what-is-over-the-air-\(ota\)](https://www.aptiv.com/en/insights/article/what-is-over-the-air-(ota))
- [4] F. Vandenberghe, L. Clark, and J. Schilling, "WASI I2C."
- [5] F. Vandenberghe, "i2c-wasm-components."
- [6] F. Vandenberghe, "hts221."
- [7] E. R. W. Group, "PSA: Cortex-M Breakage (LLD as the default linker)." Aug. 2018.
- [8] F. Vandenberghe, "Compiling wamr-sys for thumbv6m-none-eabi fails." Apr. 2024.

Contents

Samenvatting	iv
Summary	v
List of Figures	xix
List of Tables	xx
List of Acronyms	xxi
List of Code Fragments	xxiii
1 Introduction	1
2 Background	3
2.1 WebAssembly	3
2.1.1 JavaScript integration	3
2.2 WebAssembly System Interface	4
2.2.1 Design principles	4
2.2.2 The standardization process	5
2.2.3 WIT	5
2.2.4 Versions	6
2.2.5 Runtimes	7
2.2.6 Alternatives	8
2.3 The Component Model	9
2.3.1 Toolchain	9
2.3.2 Running components	10
2.4 I2C fundamentals	11
2.4.1 Operations	11
2.4.2 SMBus	13
2.5 Interfacing hardware	14

2.5.1	Different versions of <code>embedded-hal</code>	14
2.5.2	Peripheral Access Crates	14
2.5.3	Running the solution	15
3	Architecture and standard	16
3.1	Asynchronous I2C	17
3.2	Portability criteria	18
4	Implementations	19
4.1	Native implementations	20
4.2	Implementations inside WebAssembly	21
4.2.1	Implementations inside Wasmtime	21
4.2.2	Implementations inside WAMR	25
5	Evaluation	27
5.1	Evaluation setup	27
5.2	Functional evaluation	28
5.3	Benchmarks	28
	Conclusion	35
	References	36

List of Figures

2.1	Example sequence of a read operation.	12
4.1	Directory structure of the implementations repository.	20
4.2	Setups of....	21
4.3	Schematic overview of an implementation in WebAssembly.	22
4.4	Structure of the <code>wit</code> subdirectory inside Wasmtime	22
4.5	Structure of the Wasmtime host directoy structure vis-à-vis with its logical structure.	24
5.1	Visual verification of the display	28
5.2	Overview of all the mean execution times	29
5.3	Probability density function of execution time when writing to the display on....	30
5.4	Probability density function of execution time when reading from the sensor on....	31
5.5	Execution time when writing to the display inside WAMR	31
5.6	Used memory size when....	32
5.7	Flamegraphs of writing to the display	33
5.8	Flamegraph of reading the sensor inside Wasmtime with a precompiled component.	34

List of Tables

3.1	<code>wasi-i2c</code> portability criteria	18
4.1	Concise overview of the setups	19
4.2	Concise overview of the setups	22

List of Acronyms

ABI	application binary interface. , 9
ACK	Acknowledgement. , 11, 12
API	Application Programming Interface. , 3, 4, 8, 35
GPIO	general-purpose input/output. , 14
HAL	Hardware Abstraction Layer. , 14
HAT	Hardware Attached on Top. , 19
I2C	Inter-Integrated Circuit. , 1, 8, 11–14, 16, 17, 19, 35
IDL	Interface Description Language. , 5, 6
IoT	Internet of Things. , 1, 7
MCU	Microcontroller Unit. , 11, 15, 19
NACK	Negative-acknowledgement. , 11
OS	Operating System. , 3, 9, 14, 15
SCL	Serial Clock Line. , 11–13
SDA	Serial Data Line. , 11, 12
SIG	Special Interest Group. , 4
SMBus	System Management Bus. , 12, 13
SPI	Serial Peripheral Interface. , 14, 35
SVD	System View Description. , 14
SWD	Serial Wire Debug. , 15
UART	universal asynchronous receiver-transmitter. , 14
VM	Virtual Machine. , 3, 5, 7

WAMR WebAssembly Micro Runtime. , 7, 27, 28, 35

WASI WebAssembly System Interface. , 1–7, 9, 25, 35

Wasm WebAssembly. , 1, 3, 4, 6, 7, 9, 10, 23–25, 29, 35

WIT Wasm Interface Type. , xxiii, 4–6, 9, 16, 19, 22, 23, 25

List of Code Fragments

3.1	The <code>world.wit</code> file inside the <code>wasi-i2c</code> proposal.	16
4.1	The Wasm Interface Type (WIT) worlds to which guest and host bind.	23
4.2	Stripped down version of a guest component for Wasmtime.	24
4.3	Usage of the <code>with</code> keyword inside the <code>bindgen</code> 's of <code>device.rs</code> and <code>display.rs</code>	25
4.4	Stripped down version of a guest module for WAMR.	26

1

Introduction

There are countless software solutions running on critical infrastructure, where sudden failure could result in the loss of people's lives, e.g. programs guiding surgeons during operations, infotainment systems inside cars or train control systems. Additionally, consumer products ranging from baby-monitors to smart-watches are omnipresent in everyone's life. Due to a variety of reasons, the longevity of this hardware often spans multiple decades [1]. That's why the European Parliament has approved the Cyber Resilience Act on the twelfth of March 2024 [2]. This Act aims to see inadequate security features of such products and software to become a thing of the past. To this end, it will obligate manufacturers to ensure, among others, that for the duration of the support period, vulnerabilities are handled effectively. Furthermore, security updates need to be made available to users for the time the product is expected to be in use [3]. Besides, in the automotive industry, there's a clear trend towards more advanced infotainment systems and more software-defined vehicles. Ideally, these applications should be kept up to date regularly with security enhancements or feature improvements. To this end, there's a rapidly growing practice of wirelessly distributing software updates to vehicles, called over-the-air [4]. When such an update fails to perform, it should be possible to roll back this change without impact to the end-user. Thus, security and reliability are of utmost importance in these use-cases. Additionally, these solutions are frequently tightly integrated with Internet of Things (IoT) devices with limited hardware capabilities. This augments the requirements with the demand for a minimal overhead.

WebAssembly (Wasm), and WebAssembly System Interface (WASI), is the technology that aims to fulfill all these requirements. Created to execute binary code alongside JavaScript inside the browser, it is now actively used outside this environment via WASI. This system interface is a set of APIs that facilitates interaction with the filesystem, HTTP calls, a Command-Line Interface etc. With the advent of its second preview release also came the release of the component model, an architecture for building interoperable Wasm libraries, applications, and environments. But currently, these technologies merely aim to, and thus not satisfy, these requirements. Specifically, there aren't any interfaces yet for the Inter-Integrated Circuit (I2C) protocol. This protocol defines a serial communication bus widely used in the IoT ecosystem, like smart cars, health machines etc.

1 Introduction

The goal of this dissertation is to facilitate the I2C connection between Wasm applications and its underlying hardware and sensors, while still adhering to the requirements. Furthermore, this dissertation also tries to standardize this method. Effectively enabling its use in the industry and ensuring sustainability. This leads to the following research questions:

RQ1 How can a WebAssembly component control a device using I2C, while keeping the migration cost of existing applications to Wasm as low as possible?

RQ2 How can this method be standardized as part of WASI?

RQ3 How can capability-based security be applied to this API?

RQ4 What is the overhead of this method?

RQ5 How suitable is the WASI Component Model for microcontrollers with constrained resources.

Chapter 2 provides a writeup of all the necessary background knowledge. Chapter 3 gives an overview of the standardization effort. Chapter 4 demonstrates implementations that make use of I2C as part of Wasm. Following up, Chapter 5 gives an in-depth review of the overhead.

2

Background

2.1 WebAssembly

Wasm is a binary instruction format for a stack-based Virtual Machine (VM). It is designed as a portable compilation target for programming languages. Binaries have a `.wasm` file extension, there's also a textual representation which has a `.wat` extension. This enables deployment on the web for client and server applications. Examples of web applications using this technology are Adobe Photoshop [5] and Google Earth [6].

Although the name implies it, Wasm is not merely limited to the web. There are runtimes that enable execution on a myriad of platforms, ranging from Linux devices to smartphones or even microcontrollers, via a system interface that enables direct Operating System (OS) communication, called WASI.

2.1.1 JavaScript integration

Initially, Wasm was designed for near-native code execution speed in the web browser¹. Therefore, it was designed to run alongside JavaScript, allowing both to work together. In this early stage, compiling to Wasm was supplemented with the generation of the required JavaScript glue code. For this, the emscripten [7] compiler is used.

Outside the web browser, there are also platforms that provide a JavaScript runtime environment, i.e. Node.js [8] and Deno [9]. They both, too, have Wasm support, but no manner of accessing OS functionality directly on its own. For this the WASI Application Programming Interface (API) needs to be utilized.

For Node.js, integration with WASI is experimental and does not provide the comprehensive security properties provided by dedicated WASI runtimes. It is uncertain whether full support will ever be implemented. In Deno official support has been deprecated due to a lack of interest [10].

¹To be precise, it's the JavaScript engine inside the browser that added support for WebAssembly.

2.2 WebAssembly System Interface

WASI [11] is a modular collection of API proposals defined with the WIT IDL. It provides a secure and portable way to access several operating-system-like features such as filesystems, networking, clocks and random numbers. This collection is developed under the governance of the WASI Subgroup, a subgroup of the W3C's, the World Wide Web Consortium, WebAssembly Community Group.

This subgroup doesn't provide any implementations, for this the Bytecode Alliance [12] exists, a nonprofit organization of companies containing Fastly, Fermion, Cosmonic, Intel, Microsoft, Siemens and more. Inside this organization, there's also a subcommunity of people interested in the combination of Wasm and embedded devices. To ratify this subcommunity, a request for a Special Interest Group (SIG) Embedded has been opened with the Bytecode Alliance.

2.2.1 Design principles

The following design principles are core:

- **Capability-based security:** All access to external resources is provided by capabilities.
- **Interposition:** A Webassembly instance can implement a given WASI interface, and the consumer WebAssembly instance can then use this implementation transparently.
- **Compatibility:** If possible, keep the API free of Compatibility concerns, and provide compatibility through libraries.
- **Portability:** The exact meaning of this is specific to each API, but in globo it means that no engine should need to implement every API in WASI.
- **Modularity:** The component model's worlds mechanism is used, in order to allow specific sets of APIs to be described which meet the needs of different environments. See chapter 2.3.

From these, capability-based security can be seen as the key defining feature. Capability-based security dictates that access to external resources must be denied, unless it is authorized via capabilities [13]. In the context of WASI, there are two kinds of capabilities:

- **Handles:** Dynamically identify and provide access to resources via a 32-bit integer index. This is analogue to file descriptors.
- **Link-time capabilities:** Functions that require no handle arguments. These are used in situations where it's not necessary to identify more than one instance of a resource at runtime. Used sparingly.

2 Background

For this security model to work correctly, its default to block all external accesses must be airtight, i.e. the runtime needs to be secure.

2.2.2 The standardization process

The WASI subgroup is further split up into the Community Group and the Working Group. The purpose of the Community Group is to attempt to address all concerns, but no 100% consensus is needed. The Working Group, on the other hand, is there to finalize and ratify mostly complete specifications plus test suites from the Community Group.

The process is split up into five stages of standardization:

Phase 0. Pre-Proposal: The Community Group decides whether the pre-proposal is in scope for WASI.

Phase 1. Proposement of the feature: An overview document must be produced that specifies the feature with reasonably precise and complete language.

Phase 2. Specification text is available: A test suite should be added, and it should pass on the prototype or some other implementation.

Phase 3. The specification gets implemented by engines.

Phase 4. The feature is being standardized: Ownership gets transferred from the Community Group to the Working Group, and two or more Web VM's have implemented the feature.

Phase 5. The feature is standardized: Editors perform final editorial tweaks and merge the feature into the main branch of the primary specification repository.

To go from the one stage to the following, a vote in the subgroup needs to be passed. Except to enter phase 0, here the proposal is still merely an idea.

2.2.3 WIT

WIT is an Interface Description Language (IDL). This means that it is a format that defines how the interface of a component should look like. To this end, it uses the following set of concepts: types, functions, interfaces, worlds and packages. From these, worlds are the most key.

A world describes the capabilities and needs of a component - it says which interfaces are available for outside code to call, the `exports`, and which interfaces it depends on, the `imports`. Thus, only the surface of a component is defined, not the internal behaviour. The internal behaviour is determined when the world is targeted by a component an application or library developer creates. For a component to run, its imports must be fulfilled, by a host or by other components.

2 Background

On the other hand, a world defines an environment in which a component can be instantiated, and its functionality can be invoked.

Regarding the other concepts, functions can only be declared as part of an interface, or as an import or export in a world. Finally, a package is not a world, but can be seen as more like a namespace. It's a way of grouping related interfaces and worlds together for ease of discovery and reference.

WIT also provides built-in types, including primitives like signed and unsigned integer types, floats, strings, and more complex types like results, options and lists. In these, there are two small intricacies worth pointing out. First, both the `char` and `string` types are Unicode. Second, there's the user-defined `resource` type. This type can be seen as an object that implements an interface, and therefore behaviour is only exposed through methods.

To make managing dependencies inside your WIT definition easier, the `wit-deps` [14] project can be used. It makes it possible to lock your dependencies to a certain version and to check if they're the most recent one.

`witx`

In older tooling, it is possible to come across `witx` instead of WIT. This was the IDL used during `Preview 1`. It was derived from `wat`, see chapter 2.1, and had a low-level C-like type system that emphasized raw pointers, and callees were expected to have access to the entire linear memory of the caller.

2.2.4 Versions

At the time of writing, WASI is in `Preview 2`. The flagship feature of this preview is the release of the component model, see section 2.3. Furthermore, two WIT worlds are now included:

- `wasi-cli`: A command-line interface, roughly corresponding to POSIX.
- `wasi-http`: An HTTP proxy.

The major banner of the upcoming 0.3 is asynchronous support. The exact details of what this asynchrony entails is yet to be determined. Following up will be a stable 1.0 release.

2 Background

2.2.5 Runtimes

A runtime system is a binary that is accountable for the running of Wasm binaries. Analogue to the role of a hypervisor for a virtual machine. Depending on the capabilities of the runtime, it contains the following noteworthy high-level items:

Engine: An engine stores and configures global compilation settings like optimization level, enabled wasm features, etc.

Module: This represents a compiled form of the input preview 1 wasm module.

Component: A compiled preview 2 component ready to be instantiated.

Linker: A component-style location for defining host functions. This is not the same as `wasmtime::Linker` for modules.

Instance: An instance of a component, or module, is the actual object on which the functions are called.

Store: The store owns the instances, functions, globals, etc.

The Bytecode Alliance maintains two runtimes, Wasmtime [15] and WAMR [16].

Wasmtime can be seen as a general-purpose runtime, focussing on server-side and non-web embeddings with components. It has full component model support, first-class support for eight languages, and community support for a further two. This makes it the de-facto runtime.

On the other hand, WebAssembly Micro Runtime (WAMR) is specifically designed to be as lightweight as possible, targeting embedded devices and the edge. This translates itself into the provided features and the supported guest languages. Support for the component model is planned for the end of 2024, and it only has robust support for C/C++. A toolkit for Rust has been published in March 2024 [17], but this is still novel.

In Rust, the standard library, abbreviated as `std`, is a set of minimal shared abstractions for the broader Rust ecosystem. This library is enabled by default, and can be opted out via the `no_std` attribute. Rejection is useful when targeting a platform that does not support the library or purposely doesn't use the capabilities of `std`. Historically, the community surrounding Wasmtime has been strongly opposed to the inclusion of a `no_std` build. This is no longer the case. When such a build will eventually be available, the gap between Wasmtime and WAMR will be reduced.

Besides these two, there are numerous ones provided by other parties, in varying degrees of completeness, targeting other use-cases. There's for example Jco [18], specialized for JavaScript, componentize.py [19], for Python, or Chicory [20], which runs on the Java VM. These are out-of-scope for this dissertation.

2 Background

2.2.6 Alternatives

Inside the Wasm ecosystem, there's `mechanoid` [21], a framework for applications on embedded systems and IoT devices that had its first release in March 2024. The framework itself is written in Go, and it has builtin support for the `wazero` [22] and `wasman` [23] runtimes. Both runtimes specifically designed for Go developers. It doesn't use WASI, nor does it have any plans for this. This makes it unsuitable for us.

In the Node.js ecosystem, the most prominent package available for an I2C connection is `i2c-bus` [24]. This is an addon written in C++ that operates directly on a given file descriptor.

For the web, `chirimen-drivers` [25] could be used, but the documentation is completely written in Japanese. Alternatively, combined use of the WebUSB and Web Serial API could be done, but unfortunately the Web Serial API is only available in Chrome, Edge and Opera.

On Windows, I2C communication can be performed via the `HIDI2C.sys` [26] driver. Linux allows access to a device from userspace, through the `/dev` [27] interface. For this, the `i2c-dev` kernel module needs to be loaded. Each registered adapter gets a number, starting at 0. To start the communication, first the device file needs to be opened and then the target address must be set with `ioctl(file, I2C_SLAVE, addr)`. Finally, on macOS the connection is controlled with the `IOI2CInterface.h` [28] interface inside the IOKit framework.

2.3 The Component Model

The WebAssembly Component Model is an architecture for building interoperable Wasm libraries, applications and environments. These components can be seen as a wrapper around a core module, or other components, which express their interfaces and dependencies via WIT and the canonical application binary interface (ABI). Unlike core modules, components may not export Wasm memory, reinforcing Wasm sandboxing and facilitating interoperability between languages with different memory assumptions.

An ABI can be seen as an agreement on how to pass around data in a binary format, specifically concerned with the data layout at the bits-and-bytes level. The Canonical ABI defined by the component model, specifies how the WIT type definitions are translated to bits and bytes. Internally, a C and a Rust component might represent strings in a quite different way, but the canonical ABI provides a format for them to pass strings across the boundary between them.

In regard to WASI, the component model is the staple feature of its second preview, but it is possible to make use of the WASI interfaces without the component model, and thus this model is entirely optional. By way of comparison to a traditional OS, the Component Model fills the role of an OS's process model, defining how processes start up and communicate with each other, while WASI fills the role of an OS's many I/O interfaces.

To compose multiple components together `wasm-tools` [29] can be used, or visually using the builder [30] app. Specifically for Rust, `cargo-component` [31] is also an option.

2.3.1 Toolchain

In Rust, `cargo-component` can be used to compile code to a preview 2 component. In essence, compiling to Preview 2 means compiling to `wasm32-wasi` and then converting it to a component via an adapter and the `wasm-tools component new` subcommand. This component then adheres to the WIT interface specified in the configuration file. The adaption is needed because there's no first-class support for Preview 2 yet. Mainstream support for this is planned for early 2025 [32].

Under the hood, `cargo-component` relies upon `wit-bindgen` [33] for binding with the interface. Besides Rust, `wit-bindgen` also supports the following languages: C, Java, Go and C#. For JavaScript, `ComponentizeJS` [34] can be used.

2 Background

Adapter modules

The Wasmtime runtime publishes adapter modules with each release, they provide the bridge between the Preview 1 ABI and the Preview 2 ABI. The following three modules are provided:

- `Command`: For command-line applications
- `Reactor`: Applications that don't have a `main` function
- `Proxy`: For applications fed into `wasmtime serve`

The `wasmtime serve` subcommand runs a component inside the `wasi:http/proxy` world, supporting the sending and receiving of HTTP requests.

2.3.2 Running components

Running a component is done by calling one of its exports. This can require a custom host, otherwise the `wasmtime` command line can be used.

The job of a custom host is to load a component and execute it through the usage of a Wasm runtime. See section 2.2.5 for a shortlist of the available ones. To guarantee a correct execution, it is important to make sure that any missing interface imports are filled in here, see the earlier section 2.2.3. When using `wit-bindgen`, this is done via the `with` option inside the `bindgen` [35] macro.

When the component exports the `wasi:cli/run` interface, and imports only interfaces listed in the `wasi:cli/command` world, it is considered a command component. Command components can be executed by the `wasmtime run` subcommand. This will compile the module to native code, instantiate it and optionally execute an export.

2.4 I2C fundamentals

As we will be leveraging the I2C protocol, it is worth looking into the inner workings.

I2C [36] is a serial communication bus invented in the eighties by Philips Semiconductors. It uses only two bidirectional lines, a Serial Data Line (SDA) and a Serial Clock Line (SCL). Typically, 7-bit addressing is used, but there exists a 10-bit extension. This extension is fully backwards compatible, allowing a software-emulated 10-bit addressing implementation if the hardware only supports 7-bit addressing.

This communication bus has no minimum frequency, but can go as fast as 5 Mbit/s. Not every Microcontroller Unit (MCU) supports every frequency though, for example the PCF8523 Real-Time Clock only supports up to 1 MHz.

Besides a 0 or 1 data bit, there are two special START and STOP signals which act as message delimiters.

2.4.1 Operations

A node on the bus can have one of two roles²:

- Controller: Generates the clock, via required minimum periods for the low and high phases of the SCL, and initiates communication with targets.
- Target: Receives the clock and responds when addressed by the controller.

Any number of any type can be present, and these may be changed between messages. They can also both receive and send data, when in the corresponding mode.

Initial communication is established by a controller that sends a START followed by the address of the target it wishes to communicate with, which is finally completed by a single bit indicating if it wishes to write (0) or read (1) from the target. If the target exists on the bus, it will respond with an acknowledgement. This Acknowledgement (ACK) corresponds with transmitting a single 0 bit, there is also a Negative-acknowledgement (NACK) which is a single 1 bit.

Further communication is performed by one party sending data, most significant bit first, and the other sending an ACK bit.

²In earlier literature, the terminology master and slave were used for respectively controller and target.

2 Background

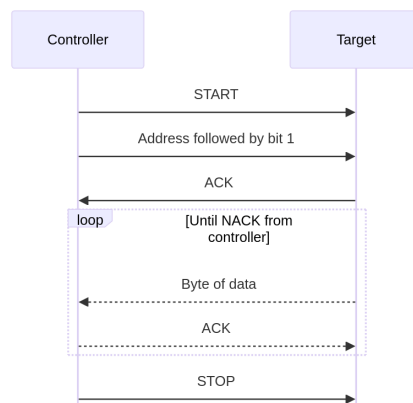


Figure 2.1: Example sequence of a read operation.

Bus sharing

In the case of multiple targets linked with one controller, the controller needs to indicate which target it wants to interact with. To achieve this, each target compares the address sent by the controller with its own. If the address matches, it sends a low voltage ACK bit back to the controller. If the address doesn't match, the target does nothing and the SDA line remains high.

When there are multiple controllers, issues can arise, precisely when they try to send or receive data at the same time over the SDA line. To solve this problem, each controller needs to detect if the SDA line is low or high before transmitting a message. If the SDA line is low, another controller is in control of the bus, and it should wait until a STOP has been received to send the message. If the SDA line is high, then it's safe to transmit the message.

Methods

The protocol defines three basic types:

- Write: Controller writes bytes to the target.
- Read: Controller reads bytes from the target until a given buffer is full.
- Write-read: Controller first writes bytes and then reads enough bytes to fill the buffer in a single transaction.

It is also possible to combine a list of write and read operations inside a transaction contract. Figure 2.1 showcases more in detail how a read operation works.

2 Background

2.4.2 SMBus

The System Management Bus (SMBus) [37] is a subset derived from I2C by Intel. Its main application is to monitor critical parameters on PC motherboards and in embedded systems. On the surface they are quite similar, but there are some subtle differences worth mentioning. For one, SMBus will time out when SCL is held low for more than 35 milliseconds. I2C doesn't have an established timeout value, implicating that a target or controller can hold SCL as long as necessary to process data.

On account of this timeout, SMBus has a minimum clock speed of 10 kHz. Leading to a maximum of 100 kHz. As stated earlier, I2C can go as fast as 5 Mbit/s and no minimum frequency is specified.

Another difference is in terms of voltage levels. For I2C the typical levels are +5 V, +3.3 V or even +1.8 V and below. In contrast, in an SMBus system the supply ranges are restricted between +1.8 V and +5 V. In general, even with the different specifications for the input logic voltage thresholds, I2C and SMBus devices will be interoperable over the supply voltages permitted by the SMBus specification.

Sometimes libraries that provide methods for I2C communication, also provide ones for SMBus. But, thus, in the context of driving an I2C target device, these can be safely ignored.

2.5 Interfacing hardware

Embedded devices have a high degree of diversity of possible constraints, e.g. 64-bit support, memory size and the availability of hardware units like a memory protection unit. Making it difficult for drivers to support any number of target platforms, unless these platforms are abstracted away behind a shared API. This is the purpose of a Hardware Abstraction Layer (HAL). It is important that this layer hides device-specific details and that it is generic across devices.

For Rust, this HAL is, aptly, named `embedded-hal` [38] and provides traits for using peripherals commonly available in microcontrollers such as general-purpose input/output (GPIO), universal asynchronous receiver-transmitter (UART), Serial Peripheral Interface (SPI) or I2C. There exists many crates that implement these interfaces for a certain microcontroller family or a system running some OS. Furthermore, there are also loads of driver crates that use the `embedded-hal` interface to support all these families and systems. A curated list can be found in the Awesome Embedded Rust [39] repository.

Sadly, the notion of a community-wide shared interface is not universally present in all embedded communities. The C/C++ community is such an example, where there isn't one HAL to rule them all.

2.5.1 Different versions of `embedded-hal`

Unfortunately, there are two major versions of `embedded-hal`, i.e. 0.2.7 and 1.0, which are incompatible with one another. As version 1.0 was only released on the ninth of January 2024, it is still fairly novel. Thus, crates have a wildly varying degree of compliance with this version.

Broadly speaking, there are four major changes [40]. Firstly, traits have been simplified and others have been merged to remove interoperability gotchas. Secondly, async versions of the blocking traits are now available in the `embedded-hal-async` crate. Thirdly, there is now support for SPI bus sharing. Lastly and fourthly, there is improved error handling.

There is a crate [41] that tries to provide a compatibility layer between these two versions, but the latest supported version is merely a release candidate of 1.0. Thus, the crate is not really practically useful.

2.5.2 Peripheral Access Crates

System View Description (SVD) files are XML files typically provided by silicon vendors which describe the memory map of a device. Via the `svd2rust` crate [42] it is possible to generate a mostly-safe Rust wrapper. Further discussion is out of scope as this is a very thin wrapper, and usually depended upon by HAL authors.

2 Background

2.5.3 Running the solution

When the target platform is an OS, it is typically fairly easy to build and execute a software solution, plainly by doing this on the target device itself, or by cross-compilation from a more potent device. This is not the case for an MCU, here, only cross-compilation is possible. Due to the constrained nature of memory on an MCU, the memory-layout also needs to be specified.

In the case of a Raspberry Pi Pico, compilation results in an UF2 and an ELF file. The former is a file format developed by Microsoft for flashing microcontrollers over mass storage connections. The latter is used by the debugger.

To program the flash on the Pico, the BOOTSEL button needs to be held, forcing it into USB Mass Storage Mode. Then, you can move a UF2 file onto it. Whereupon the RP2040 processor of the Pico will reboot, unmount itself, and run the flashed code. Other boards could require pulling down the flash CS pin, which is how the BOOTSEL button works on the pico, using an exposed Serial Wire Debug (SWD) interface, also an option for the Pico, or have a reset button that needs to be double-pressed.

SWD is a standard interface on Cortex-M based microcontrollers, which the host machine can use to reset the board, load code into flash, and set the code running. Without the need to manually reset the board or hold the BOOTSEL button. The easiest way to connect with this interface on a Pico is to make use of a debug probe via `probe-rs` [43]. This also unlocks the ability to print to `STDOUT` or even utilize the Debug Adapter Protocol [44].

3

Architecture and standard

Currently, the I2C proposal [45] is in the second phase, with ongoing effort to fulfill the criteria to pass the vote to the third phase. Specifically, the following requirements are yet to be met:

- The proposal lacks a test suite that covers the feature.
- Some pull requests and issues are still open that iterate on the design of the feature.
- Updates to the reference interpreter are not yet required, but recommended.

This effort is led under the guidance of certain champions, for this proposal these are Friedrich Vandenberghe, Merlijn Sebrechts and Maximilian Seidler. Both Friedrich and Merlijn are from UGent, Maximilian is from Siemens. This mix of academicians and people from the industry ensures ongoing standardization effort and actual usage of the feature.

At the start of this thesis `wasi-i2c` was merely an idea, lacking any proposal or implementation, and thus technically, still residing in phase zero. But now it contains three WIT files that follow the component model: `delay`, `i2c` and `world`. The first two closely follow the corresponding interfaces from `embedded-hal` version 1, `embedded-hal` is explained in section 2.5, but this crate is not necessary for an implementation. The philosophy behind this is the same as for why one would want to have an `embedded-hal`.

```
package wasi:i2c@0.2.0-draft;
```

```
world imports {  
    import i2c;  
    import delay;  
}
```

Codefragment 3.1: The `world.wit` file inside the `wasi-i2c` proposal.

The proposal defines two handles, for an explainer on handles see 2.2.1, one for I2C and one for delays. These provide pretty broad access, e.g. it is not possible to limit the I2C connection to certain addresses. A more fine-grained control model is possible, but there's currently no demand for this.

3 Architecture and standard

Inside the `i2c` interface, no explicit constructor is defined because `i2c` resources can be constructed in many different ways, so worlds that include this interface should also include a way to obtain I2C handles. Typically, this is done by either having handles passed into exported functions as parameters, or by having handles returned from imported functions. The definition for `world.wit` can be found in codefragment 3.1. The purpose of this world is to provide a way to import all the defined interfaces in the proposal at once.

Although delays are not inherently linked with I2C, some guests implementations require this, and thus it is part of the proposal. In the future, it could be that delays get split up into their own proposal, or that this proposal gets merged with the other embedded-related proposals. The biggest drawback to a merger would be that this, possibly, could heavily increase the burden on the champions. Furthermore, this would require that all the champions are interested in maintaining all of these other communication protocols. On the other hand, this would centralize the proposals and could make it easier for potential implementors.

3.1 Asynchronous I2C

Currently, the proposal is purely synchronous. To provide an asynchronous (async) API, there are two possible ways to tackle this:

Explicit A separate WIT API could be provided that depends on `wasi-io` [46] polling. This could be implemented with the current tooling.

Integrated Wait for the Component Model to natively integrate async. The major upside to this approach is that a single WIT description can describe both a sync and async API. Caller and callee can each independently choose if they want to be sync or async, and they can be linked.

As there's no immediate need for this, the champions have decided that it is best to wait until the integrated option is mainstream.

3 Architecture and standard

3.2 Portability criteria

An important part of a proposal is its portability criteria, with these the champions show that their specification isn't overspecified for a specific platform. Commonly, these are two complete independent implementations, but due to the diversity in platforms that can interact with I2C this isn't suitable for the `wasi-i2c` proposal.

Platform	Architecture	Reference hardware
Linux	ARM64	Raspberry Pi 3 Model B
RTOS	RISC-V	ESP32-C2
RTOS	ARM32	Nucleo F412ZG

Table 3.1: `wasi-i2c` portability criteria

The champions, together with people from Siemens, Intel, Aptiv, Xiaomi, Amazon, Midokura, Sony, Atym and Bosch, have agreed upon a set of criteria for the proposal. These criteria can be found in Table 3.1. For each criterium, reference hardware is provided, without these there would still be a lot of possible variety in RAM size, flash size and CPU speed. Herein Intel x86 is missing, but support for this is implied as this is the used architecture for developing Wasmtime. The Nucleo is chosen because this is the reference board for Siemens. The other two are chosen to showcase support for both RTOS and Linux, and ARM and RISC-V. As an extra requirement besides the targets, it is important that the interface is designed in such a way that memory usage is limited to the bare needed minimum, to ensure enough RAM left for the application itself.

4

Implementations

Implementations are provided to ascertain the soundness of the WIT interfaces from `wasi:i2c`, an overview can be found in Table 4.1. These solutions reside inside the `i2c-wasm-components` repository [47]. Figure 4.1 contains the parts that are noteworthy.

Connected device	Hardware
HTS221 sensor	Raspberry Pi 3 Model B
4-digit 7-segment display	Raspberry Pi 4 Model B
4-digit 7-segment display	Raspberry Pi Pico

Table 4.1: Concise overview of the setups

As stated in section 2.4.1, there are three basic types of methods, from which two are implemented, `write` and `read`. For writing, a 4-digit 7-segment display¹ is used, and for reading a HTS221 sensor, from which the current temperature and humidity can be read. These are respectively `segment_led` and `hat` in the implementation.

The sensor is part of a Hardware Attached on Top (HAT) mounted atop of a Raspberry Pi 3 Model B via its 40-pin GPIO header, see Figure 4.2a. Although it is connected this way, reading from the sensor is still done via I2C. The display, on the other hand, is a standalone device and can thus easily be hooked up with a different controller. As a controller, we use a Raspberry Pi 4 Model B, Figure 4.2b, and a Raspberry Pi Pico, Figure 4.2c. In this figure, the Pico is connected with a debug probe, see section 2.5.3. Both the Pi 3 and the Pi 4 run on an ARM64 architecture and a Linux platform, contrarily, the Pico is an MCU targeting RP2040. Table 4.2 shows a concise overview. The used platform and architecture combinations are different from the portability criteria because these were decided before the criteria were decided upon.

¹Because each digit is represented by seven segments, parts of the latin alphabet can also be shown.

4 Implementations

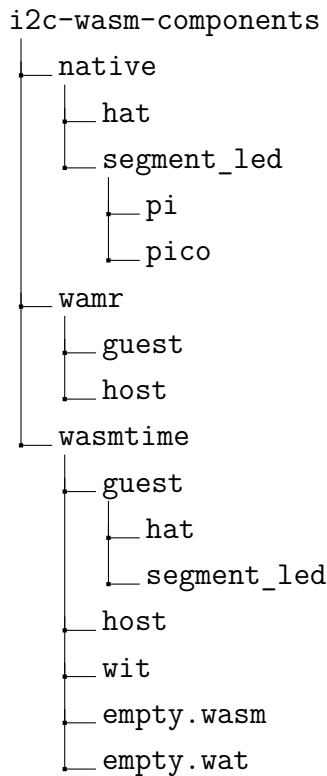


Figure 4.1: Directory structure of the implementations repository.

4.1 Native implementations

The Pi 4 can build the codebase, but the other two lack the capabilities for this, thus cross-compilation needs to be performed. Fortunately, Rust provides great support for this via the `target` flag. ARM64 can be targeted with `aarch64-unknown-linux-gnu`, and RP2040 with `thumbv6m-none-eabi`.

As the Pico is an MCU, it has some further requirements. First, it only supports `no_std` build, see section 2.2.5. Second, the memory layout also needs to be specified inside a file called `memory.x`.

To easily interface with the HTS221 sensor, the homonymous `hts221` Rust crate [48] can be used. Unfortunately, for `embedded-hal` version 1 integration, a fork [49] had to be created. The `hts221` crate requires an I2C connection that follows the HAL specification, for this the `linux-embedded-hal` crate [50] is used.

For the implementation running on the Pi 4 we write the current local time. To establish the I2C connection the `rppal` crate [51] is used. This library also follows the `embedded-hal` API, but is specifically tailored for Raspberry Pi devices. Due to the limited capabilities of the Pico, a simple incrementing counter is instead

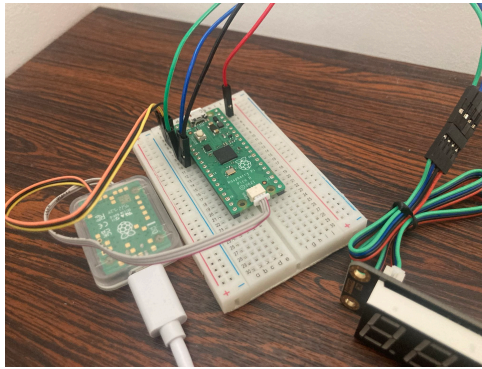
4 Implementations



(a) Pi 3



(b) Pi 4



(c) Pi Pico

Figure 4.2: Setups of....

displayed. For the connection, the `rp-pico` crate [52] is now utilized.

4.2 Implementations inside WebAssembly

To make the move from native to WebAssembly, everything related to the I2C connection itself is kept host-side. The device-specific logic, like reading the current temperature, is moved to the guest. Ideally, each implementation, conceptually, comes down to the schematic defined in Figure 4.3. But due to the myriad of differences between Wasmtime and WAMR, outlined in section 2.2.5, no WebAssembly-specific code can be reused between them. This implementation, therefore, keeps everything each runtime interacts with inside two separate directories.

4.2.1 Implementations inside Wasmtime

The Wasmtime implementation consists of three directories: `guest`, `host` and `wit`. The `wit` directory specifies two worlds, corresponding with the two devices being interacted with. This implementation thus contains two

4 Implementations

Connected device	Hardware
HTS221 sensor	Raspberry Pi 3 Model B
4-digit 7-segment display	Raspberry Pi 4 Model B
4-digit 7-segment display	Raspberry Pi Pico

Table 4.2: Concise overview of the setups

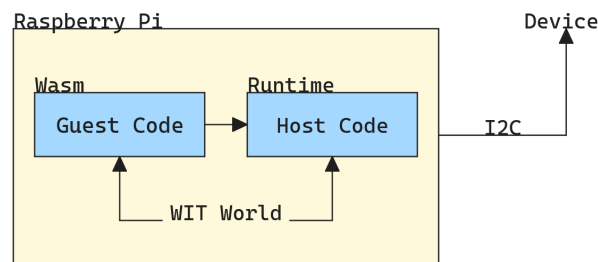


Figure 4.3: Schematic overview of an implementation in WebAssembly.

implementations that follow the schematic described by Figure 4.3.

WIT world

```
wit
├── device.wit
│   ├── deps/i2c
│   ├── deps.toml
│   └── deps.lock
```

Figure 4.4: Structure of the `wit` subdirectory inside Wasmtime

Both the guest components, as the Wasmtime host implementation, make use of the WIT from codefragment 4.1, as stipulated in section 2.3.1 and corresponding with the `device.wit` from Figure 4.4. Herein, `wasi:i2c` is the proposal, see chapter 3. The display is represented by the `screen` world, which needs both `i2c` and `delay` from the proposal. The HTS221 is mapped with the `sensor` world, that only needs `i2c`. Therefore, the former just includes the imports from `wasi:i2c`, while the latter only imports `i2c`. The idea behind this WIT description is to keep it as generic as possible, this is why the `write` function accepts a string while the display only has four digits.

The `deps/i2c` contains the WIT interfaces from the proposal, both the `deps.lock` and `deps.toml` are files required for the correct workings of `wit-deps`.

4 Implementations

```
world sensor {
  import wasi:i2c/i2c@0.2.0-draft;
  export hts: interface {
    use wasi:i2c/i2c@0.2.0-draft.{i2c, error-code};
    get-temperature: func(connection: i2c) -> result<string, error-code>;
    get-humidity: func(connection: i2c) -> result<string, error-code>;
  }
}

world screen {
  include wasi:i2c/imports@0.2.0-draft;
  export display: interface {
    use wasi:i2c/i2c@0.2.0-draft.{i2c};
    use wasi:i2c/delay@0.2.0-draft.{delay};
    write: func(connection: i2c, delay: delay, message: string);
  }
}
```

Codefragment 4.1: The WIT worlds to which guest and host bind.

Guest code

To reduce the size of the generated Wasm binaries, the guest implementations don't make use of Rust's std. For the generation of the guest components `cargo-component`, see section 2.3.1, requires an adapter. Typically this adapter enables the usage of the std, but as the guest implementations doesn't make use of this an adapter is no longer needed. To solve this conundrum, an empty adapter is given to `cargo-component` via `empty.wasm` and `empty.wat`. Note that this uses `#![feature(...)]`, which requires the usage of the nightly version of Rust.

It would be ideal if the Wasm implementation of the sensor read would still use the `hts221` crate. For this, code needs to be supplemented to the generated bindings to make them compatible with the `embedded-hal` API. This supplementation is done via the `add_i2c_hal` macro inside the `wasi-embedded-hal` crate [53].

Codefragment 4.2 describes how, conceptually, each component implementation looks like.

Host code

The WIT interface imports (parts of) the `wasi:i2c` interfaces. Therefore, it is not possible to run components via the `wasmtime` command line, explained in section 2.3.2, and a custom host is necessary. The most basic version of this custom host would simply be a standalone implementation of each world. Every world implementation then has their own fulfillment of the imports, resulting in lots of code redundancy. Therefore, a more involved host implementation, like the structure defined in Figure 4.5, is more practical. Here, the world implementations

4 Implementations

```
#![no_std]
#![no_main]
mod bindings;

// To make the generated bindings compatible with the `embedded-hal` API
add_i2c_hal!(i2c);

struct Component {}

impl Guest for Component {
    // Omitting implementations of the exports inside of the binded world
}

// Omitting the definition of a global allocator and a panic handler,
// needed by Rust because of `no_std`.

bindings::export!(Component with_types_in bindings);
```

Codefragment 4.2: Stripped down version of a guest component for Wasmtime.



Figure 4.5: Structure of the Wasmtime host directory structure vis-à-vis with its logical structure.

can share their usage of `device.rs` for the fulfillment of the imports via the `with` keyword, as demonstrated inside codefragment 4.3. It is important that these paths be fully qualified. Otherwise errors will be thrown by the Rust compiler.

Each world implementation in itself is a module that splits the standard procedure, see 2.2.5, for executing a method from a Wasm binary in two functions:

1. `new`: Responsible for adding the implemented `i2c` and `delay` structs from `device` to the linker, creation of the host state and populating it, creation of the store and instantiation of the bindings.
2. `run`: Executes a certain function from the component. The parameters are hardcoded.

The purpose of this deviation from the standard procedure is twofold. First, it allows calling the exported function without the need to instantiate the component each time. Second, it provides a shared function signature for running a component.

4 Implementations

```
// device.rs
bindgen!({
    path: "../wit/deps/i2c",
    world: "wasi:i2c/imports",
    with: {
        "wasi:i2c/i2c/i2c": device::I2c,
        "wasi:i2c/delay/delay": device::Delay,
    }
});

// display.rs
bindgen!({
    path: "../wit",
    world: "screen",
    with: {
        "wasi:i2c/i2c/i2c": device::I2c,
        "wasi:i2c/delay/delay": device::Delay,
    }
});
```

Codefragment 4.3: Usage of the `with` keyword inside the `bindgen`'s of `device.rs` and `display.rs`.

4.2.2 Implementations inside WAMR

This implementation represents the schematic from Figure 4.3 for WAMR, however there's no support for WASI preview two. This application, thus, cannot make use of WIT, nor pass a proper I2C connection to the guest because there's no notion of a WIT resource. Therefore, the connection is now kept global. In Rust, there's no default way to keep a variable global, but for this implementation the `once_cell` [54] crate is used.

All guest-to-host and host-to-guest communication is considered unsafe on a memory-safety basis, this is because WAMR is written in C, a language infamous for how unsafe it is. Furthermore, the WAMR Rust SDK mandates that imports and exports are called and developed as extern C functions. This leads to a module implementation conceptualized by codefragment 4.4. The host cannot directly bind with the resulting binary, instead it has to register the import and find the exports. For passing parameters to the exports, a vector of `WasmValues` is passed. A `WasmValue` can be a void, 32-bit signed integer, 64-bit signed integer, 32-bit float, 64-bit float or a 128-bit signed integer. Each value is underlying stored as an array of 32-bit unsigned integers. The exported `setupfunction` can then be called with an array containing a `WasmValue::Void`, for `write` a vector of four `WasmValue::I32`'s are passed.

On the Raspberry Pi Pico, there's no filesystem, thus no capability of reading the Wasm binary from disk. Instead, a `hexdump` can be taken from the binary, and then these bytes can be hardcoded as a vector of unsigned 8-bit integers inside the host.

4 Implementations

```
#![no_std]
#![no_main]

#[link(wasm_import_module = "host")]
extern "C" {
    // The term _slave_ is still used because `rppal` still uses this.
    fn host_i2c_write(slave_address: u16, data: u8);
}

#[export_name = "setup"]
pub fn setup() { /* Omitting definition */ }

#[export_name = "write"]
pub fn write(d0: i32, d1: i32, d2: i32, d3: i32) { /* Omitting definition */ }

// Omitting the definition of a global allocator and a panic handler,
// needed by Rust because of `no_std`.
```

Codefragment 4.4: Stripped down version of a guest module for WAMR.

Under the hood, Rust uses `rust-lld` for the cross-compilation to the Pico [55], but WAMR uses the `arm-none-eabi-gcc` linker. This results in a CMAKE error when building the SDK [56], and thus disabling cross-compilation.

5

Evaluation

5.1 Evaluation setup

The evaluation setup is based upon the setups used for the implementations, as shown in Chapter 4. It consists of the following:

- A Raspberry Pi 4 Model B with 8 GB of RAM hooked up with a 4-digit 7-segment LED
- A Raspberry Pi 3 Model B with an attached HTS221 sensor

The display is evaluated natively and with Wasmtime and WAMR as a runtime. The sensor will be evaluated in the same manner, but without the WAMR runtime.

The execution time is measured with the `criterion.rs` crate [57]. This crate performs a hundred measurements, each containing many iterations of a routine, and then accumulates this into a probability density function. It indicates the estimated probability of an iteration taking a certain amount of time. Furthermore, the plot also contains a vertical line, this indicates the mean execution time. Thanks to the immense range of factors that can influence the execution time, outliers will always occur, and the resulting density functions aren't set in stone. Due to these outliers, sometimes a smaller extra peak can appear. Before measuring, `criterion` first performs a warmup phase. Here, the routine is executed repeatedly to give the system time to adapt to the new workload. This helps prevent things like cold caches.

For memory profiling, `dhat-rs` [58] is used. The profiler makes use of a global allocator that tracks allocations and deallocations on the heap. After execution, the total number of bytes, the maximum amount and the size at the end is printed. Together with the number of allocations.

The `flamegraph` crate [59] is used to provide an in-depth overview of the functions an implementation uses under the hood. This overview is provided in fashion of flamegraphs. Many times per second, the threads in a program are interrupted and the current location in the code is recorded, along with the chain of functions that were called to get there. These samples are then processed and stacks that share common functions are added

5 Evaluation

together. Then a figure is generated showing the call stacks that were measured. The x-axis doesn't show the passing of time. The left to right ordering has no meaning. The width of each block shows the total time that that function is on the CPU. A wider block means that it consumes more CPU per execution than other functions, or that it's called more. Each block's color is chosen at random.

5.2 Functional evaluation

This dissertation evaluates the applications on their execution time and used memory compared between running natively versus inside the Wasmtime and WAMR runtimes.

For this, two scenarios are tested:

- On the Raspberry Pi 3 the temperature gets read out. This value is manually verified with the native implementation.
- On the Raspberry Pi 4 the string "1234" gets written to the display. This is visually verified, see Figure 5.1.

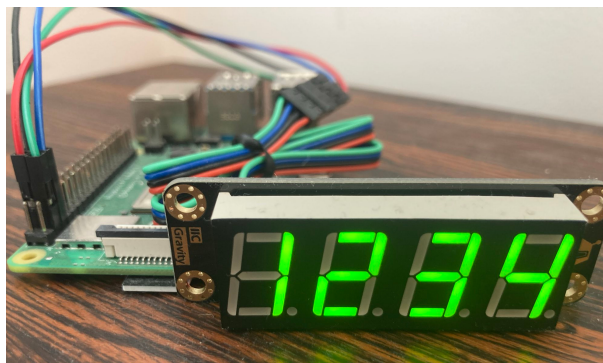


Figure 5.1: Visual verification of the display

5.3 Benchmarks

It is expected that running in Wasmtime will perform worse due to the imposed overhead on both execution time and memory usage, but that this overhead will be negligible. For WAMR, an even slimmer overhead is expected.

First, the execution time is inspected. An overview is available in Figure 5.2. Both Figure 5.3 and Figure 5.4 showcase an astonishing longer execution time when running inside Wasmtime. The flamegraph in Figure 5.7b gives the probable explanation, namely writing to the display is done inside the `_start` block, almost all the other blocks are related to Cranelift.

5 Evaluation

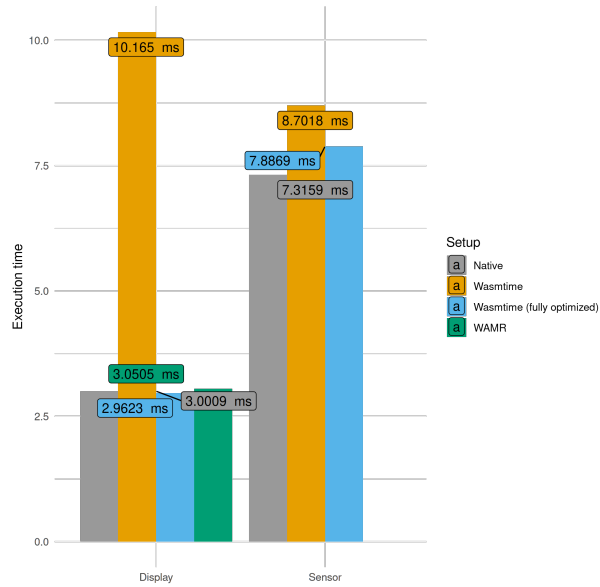


Figure 5.2: Overview of all the mean execution times

Bytecode Alliance’s Cranelift is a compiler backend that is among others in use by Wasmtime for just-in-time and ahead-of-time compilation. Via its ahead-of-time functionality, it is possible to greatly reduce the average execution time. There are two options to make use of this, through `wasmtime compile` or the `Component::serialize` function inside Wasmtime. To load such a precompiled file, the `Component::deserialize_file` function needs to be used, instead of `Component::from_file`. This file contains native, non-portable binary code which will not work on a different architecture, and might not even work across different processor models within the same architecture. Thus, it is not feasible to distribute a precompiled Wasm binary, instead of the Wasm binary itself¹.

Figure 5.3c shows the execution time with a precompiled binary. It now is far more comparable to the average time of the native execution, with only an estimated slower mean run time of 0.3062 milliseconds. The same holds true for reading the sensor, Figure 5.4c, where the slowdown is now merely 1.3859 milliseconds. But, when compared to the experienced slowdown of the display implementation, this is still too much. The flamegraph inside Figure 5.8 shows that nearly 14% of the time is spent with adding the command world to the linker, using the `command::sync::add_to_linker` function, while this world technically isn’t needed because of the custom host setup. Without this world, reading the sensor in Wasmtime is now only 0.571 milliseconds slower, Figure 5.4d, and writing to the display is 38.6 microseconds faster than native, Figure 5.3d. The executed functions, see Figure 5.7a and Figure 5.7c, do not show any prominent differences. Thus, the speedup could be due to noise or kernel intrinsics.

¹Technically, it is possible, but this requires careful configuration and is too complex for this dissertation.

5 Evaluation

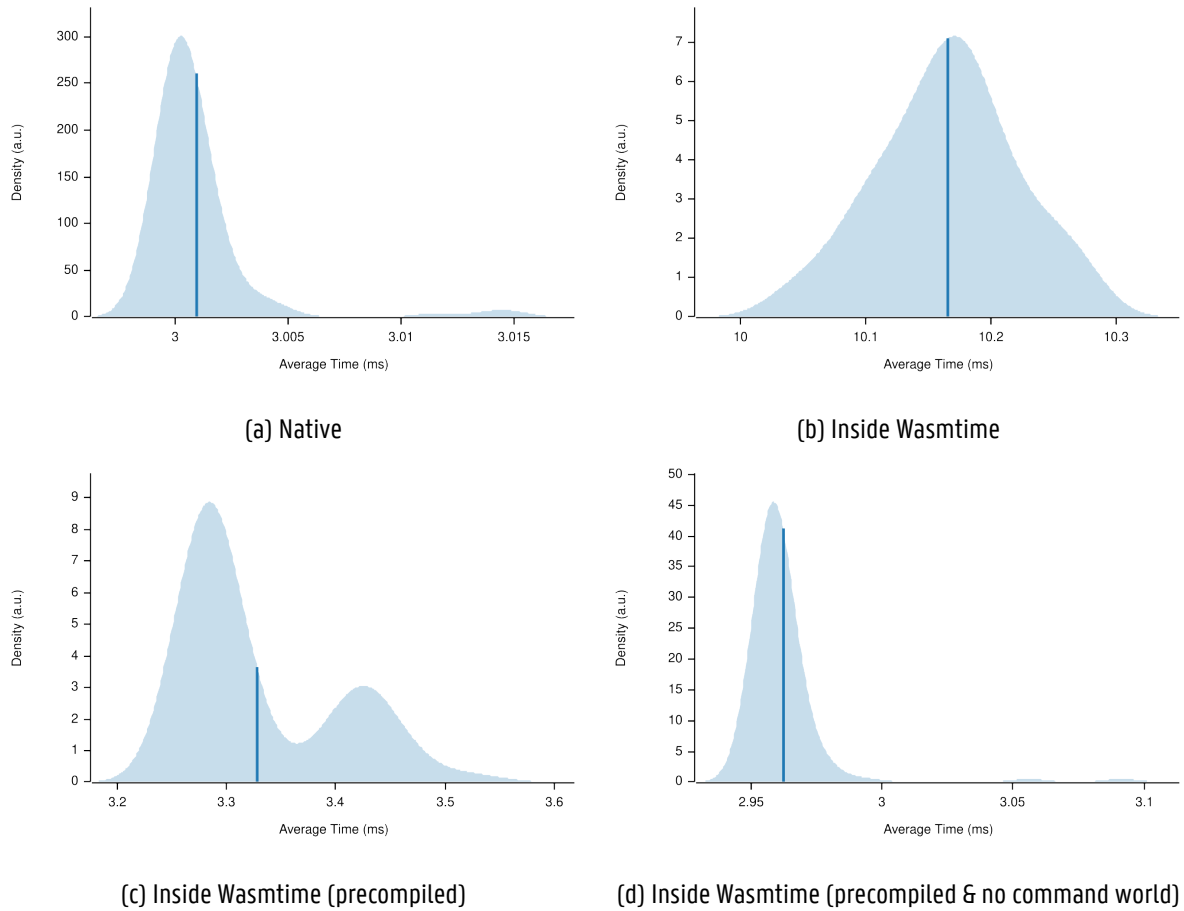


Figure 5.3: Probability density function of execution time when writing to the display on....

Comparing WAMR with the fastest Wasmtime version for writing to the display, thus Figure 5.5 with 5.3d, shows that Wasmtime outperforms with 88.2 microseconds. This is not significant enough of a difference for one to be faster or slower than the other. Looking at the flamegraph for WAMR, Figure 5.7d, it is apparant that a considerable amount of time is spent on a page translation fault.

When considering the memory usage, Figure 5.6, three phenomena are striking. First, the performed optimizations to make the Wasmtime implementation run faster also greatly benefit the memory usage. Second, WAMR has near-native memory usage, while Wasmtime uses nearly four times more memory when compared to the native version. Third and last, the memory usage of the fully optimized version of Wasmtime is fairly similar between writing to the display and reading from the sensor. This showcases a baseline amount of memory required to run the component model.

To conclude, both solutions for Wasmtime greatly benefitted from precompiling the Wasm binary. Furthermore, precompilation makes Wasmtime on par with WAMR and native time-wise, but Wasmtime still requires a substantial larger amount of memory to succesfully run.

5 Evaluation

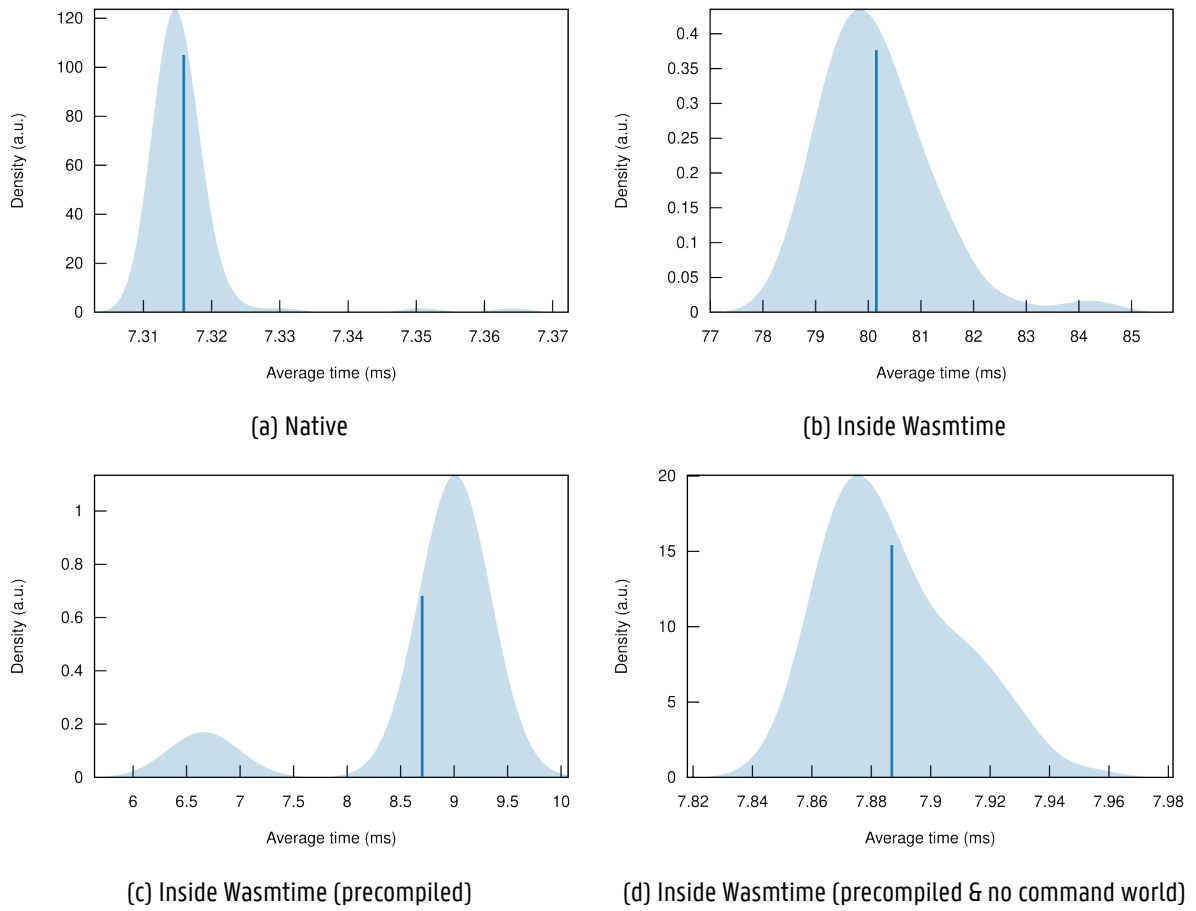


Figure 5.4: Probability density function of execution time when reading from the sensor on....

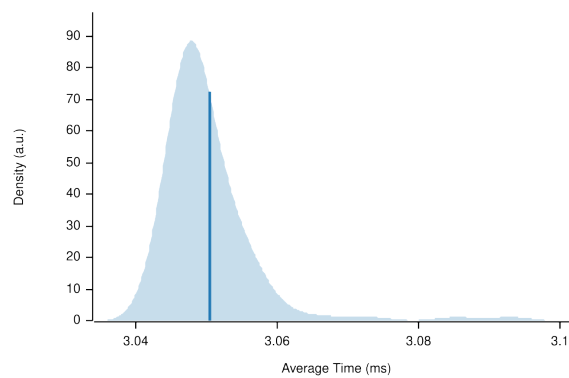


Figure 5.5: Execution time when writing to the display inside WAMR

5 Evaluation

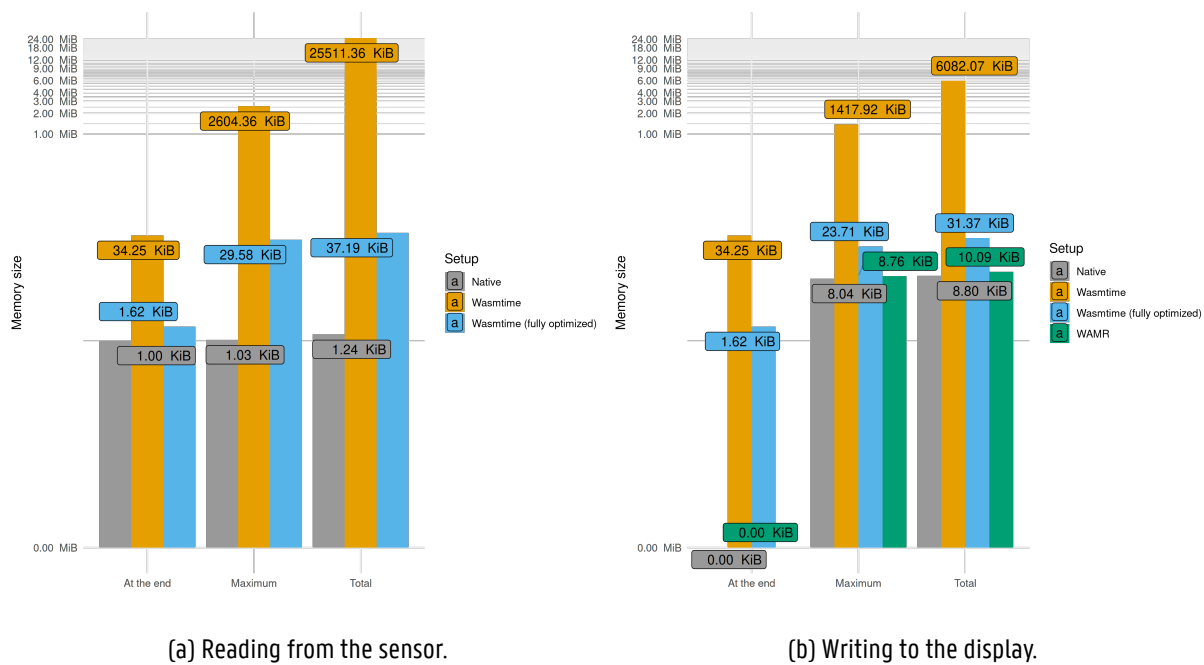


Figure 5.6: Used memory size when...

5 Evaluation

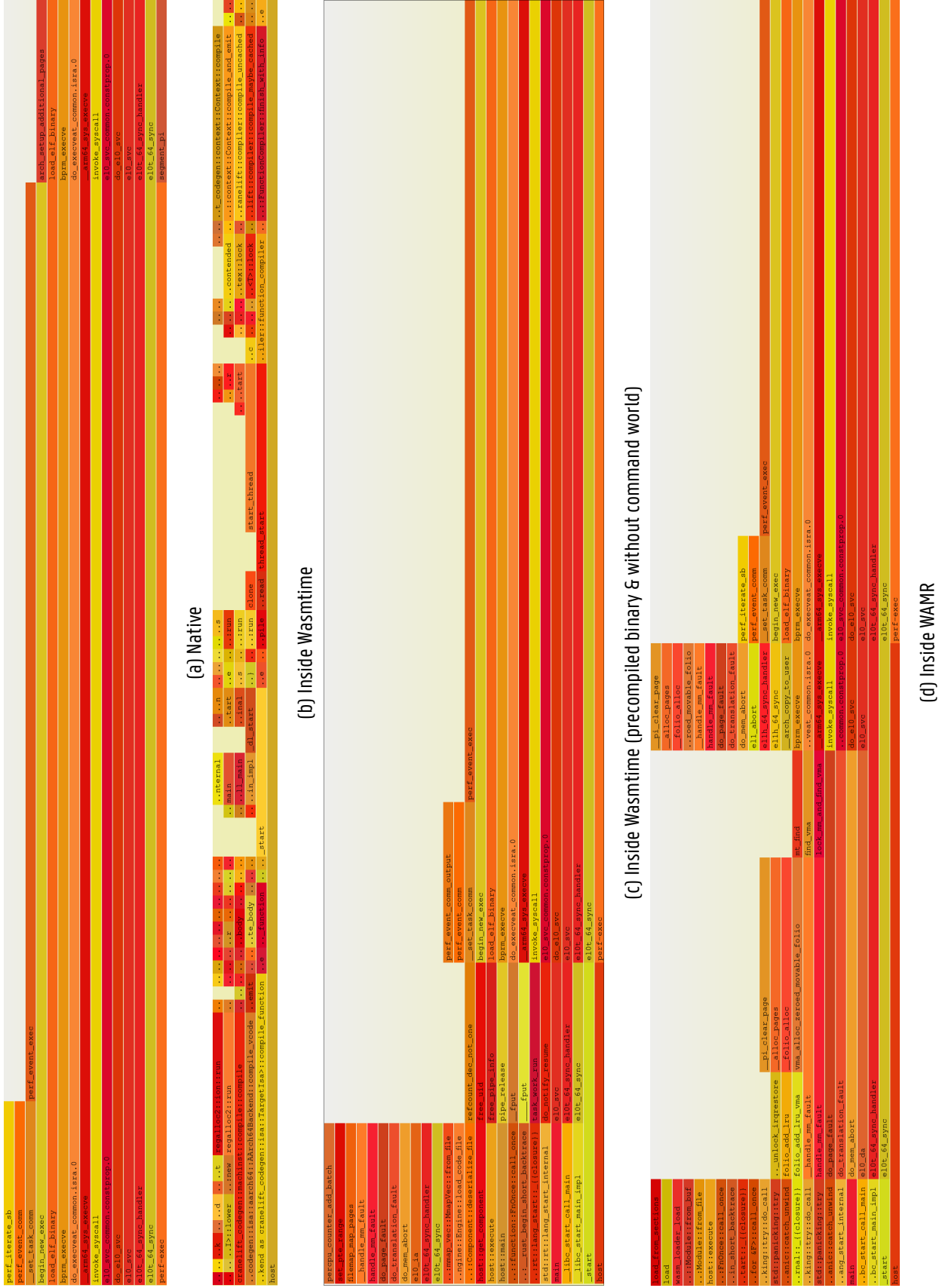


Figure 57: Flamegraphs of writing to the display

5 Evaluation

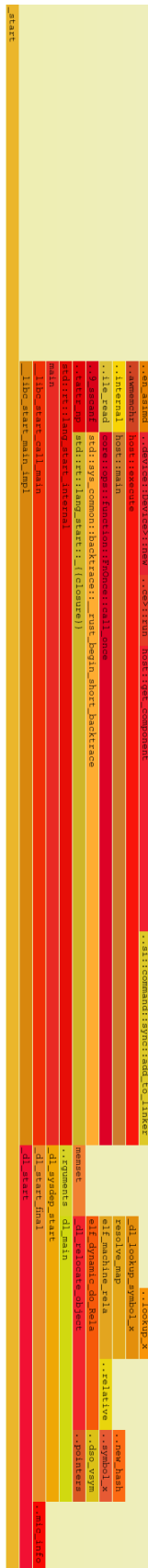


Figure 5.8: Flamegraph of reading the sensor inside Wasmtime with a precompiled component.

Conclusion

As an answer to the first research question, this dissertation demonstrates that, thanks to the provided interfaces inside `wasi-i2c`, it's possible for a Wasm component to control a device using I2C. To minimize any migration costs from existing applications to WebAssembly, the `wasi-embedded-hal` crate is provided. With this crate, it is possible to easily interoperate with device crates inside the Rust `embedded-hal` ecosystem. Although the proposal is heavily inspired by, and integration is offered for, this ecosystem, it is designed in such a way that no implementation should need to use `embedded-hal` in order to be successful.

Furthermore, to resolve the second research question, the aforementioned interfaces have been brought from a mere idea to fully-fledged interface standard proposals ready for feedback from the WebAssembly Community Group. Thus, from phase zero to phase two. In this API, as part of the third research question, two capabilities are provided, one for I2C and one for delays. More fine-grained control over these capabilities is something that's currently lacking, but certainly feasible to achieve.

To address the fourth question, the overhead is defined as a dichotomy of the mean execution time and the peak memory usage. With only an increase of 1.65% in mean execution time, and having an 8.93% higher peak memory usage over the complete lifespan of the application, the overhead of the WAMR runtime is seen as negligible. For Wasmtime, some considerations apply. First, each WASI interface added into the linker incurs extra overhead. Second and last, before executing the component itself, the runtime first instantiates it. It is possible to do this instantiation a priori, but this isn't guaranteed to work on other machines. For this, the runtime engine should be configured in a peculiar way. This is seen as an open challenge. With these considerations, the mean execution time is -3.86% and peak memory usage is 15.87 times more when compared to native.

This stark increase in memory consumption indicates that Wasmtime, and by extension the component model, incurs a hefty memory overhead. Efforts are underway to align this overhead more with WAMR's, but this is in an early stage. Thus, to answer the fifth research question, in its current state the WASI component model is not suitable to run on devices with constrained resources.

Open challenges

The following items are seen as open challenges:

- The design principles of the `wasi-i2c` proposal could be applied to similar protocols, e.g. SPI.
- Not every platform of the capability criteria has a reference implementation yet.
- The efforts to reduce Wasmtime's memory allocations are still in an early stage.
- A finer configuration of the I2C connection is currently not possible in `wasi-i2c`.
- A manageable manner to pre-instantiate components across platforms is missing.

References

- [1] S. Harding, "5.25-inch floppy disks expected to help run san francisco trains until 2030," *Ars Technica*, 2024. [Online]. Available: <https://arstechnica.com/gadgets/2024/04/5-25-inch-floppy-disks-expected-to-help-run-san-francisco-trains-until-2030/>
- [2] B. Chatain, "Cyber resilience act: Meps adopt plans to boost security of digital products," *European Parliament*, 2024. [Online]. Available: <https://www.europarl.europa.eu/news/en/press-room/20240308IPR18991/cyber-resilience-act-meps-adopt-plans-to-boost-security-of-digital-products>
- [3] E. Commission, "Eu cyber resilience act," <https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act>, Dec. 2023.
- [4] Aptiv, "What is over-the-air (ota)?" *Mobility Insider*, 2020. [Online]. Available: [https://www.aptiv.com/en/insights/article/what-is-over-the-air-\(ota\)](https://www.aptiv.com/en/insights/article/what-is-over-the-air-(ota))
- [5] P. Clark, "Adobe photoshop streamlines power and precision for the web; now includes widely popular adobe firefly powered features," <https://blog.adobe.com/en/publish/2023/09/27/photoshop-streamlines-power-precision-web>, 2023.
- [6] G. Shah, "Welcome home to the new google earth," <https://blog.google/products/earth/welcome-home-new-google-earth/>, 2017.
- [7] emscripten core, "emscripten," <https://github.com/emscripten-core/emscripten>.
- [8] O. Foundation, "Node.js," <https://nodejs.org/en>.
- [9] R. Dahl, B. Belder, and B. Iwańczuk, "Deno 1.0," <https://deno.com/blog/v1>, May 2020.
- [10] A. Jiang, B. Iwańczuk, D. Sherret, K. Whinnery, L. Kettmeir, M. Mastracci, N. Rahman, Y. Hinosawa, and M. Hagemeister, "Deno 1.38: Html doc generator and hmr," <https://deno.com/blog/v1.38#deprecation-of-stdwasi>, 2023.
- [11] P. Hickey, J. Konka, D. Gohman, S. Clegg, A. Brown, A. Crichton, L. Clark, C. Ihrig, P. Huene, Y. Yuji, D. Vasilik, J. Triplett, S. Rubanov, S. Akbary, M. Frysinger, A. Turner, A. Zakai, A. Mackenzie, B. Brittain, C. Beyer, D. McKay, L. Wang, M. Mielniczuk, M. Berger, PTrottier, P. Sikora, T. Schneidereit, katelyn martin, and nasso, "Webassembly/wasi: snapshot-01," *Zenodo*, Dec. 2020. [Online]. Available: <https://doi.org/10.5281/zenodo.4323447>
- [12] L. Clark, "Announcing the bytecode alliance: Building a secure by default, composable future for webassembly," <https://bytecodealliance.org/articles/announcing-the-bytecode-alliance>, 2019.

5 References

- [13] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, no. 3, p. 143–155, mar 1966. [Online]. Available: <https://doi.org/10.1145/365230.365252>
- [14] R. Volosatovs, "wit-deps," <https://github.com/bytecodealliance/wit-deps>.
- [15] L. Clark, "Wasmtime reaches 1.0: Fast, safe and production ready!" <https://bytecodealliance.org/articles/wasmtime-1-0-fast-safe-and-production-ready>, 2022.
- [16] B. Alliance, "wasm-micro-runtime," <https://github.com/bytecodealliance/wasm-micro-runtime>.
- [17] —, "wamr-rust-sdk," <https://github.com/bytecodealliance/wamr-rust-sdk>.
- [18] Y. Wuyts, "Announcing jco 1.0," <https://bytecodealliance.org/articles/jco-1.0>, 2024.
- [19] B. Alliance, "componentize-py," <https://github.com/bytecodealliance/componentize-py>.
- [20] B. Eckel and A. Peruffo, "Chicory: Creating a language-native wasm runtime," <https://www.youtube.com/watch?v=jrnMhWiV23w>, 2024.
- [21] hybridgroup, "Mechanoid," <https://mechanoid.io/>.
- [22] A. Cole, "Introducing wazero from tetrade," <https://tetrade.io/blog/introducing-wazero-from-tetrade/>, 2023.
- [23] hybridgroup, "wasman," <https://github.com/hybridgroup/wasman>.
- [24] fivdi, "i2c-bus," <https://github.com/fivdi/i2c-bus/tree/master>.
- [25] C. O. Hardware, "Chirimen drivers," <https://chirimen.org/chirimen-drivers/>.
- [26] M. Hopkins, "Introduction to hid over i2c," <https://learn.microsoft.com/en-us/windows-hardware/drivers/hid/hid-over-i2c-guide>, 2024.
- [27] T. kernel development community, "I2c device interface," <https://www.kernel.org/doc/html/v5.4/i2c/dev-interface.html>.
- [28] "I2cinterface.h user-space," https://developer.apple.com/documentation/iokit/IOI2CInterface_h_user-space.
- [29] B. Alliance, "wasm-tools," <https://github.com/bytecodealliance/wasm-tools>.
- [30] P. Huene, D. Anderson, and J. Mart, <https://wasmbuilder.app/>.
- [31] B. Alliance, "cargo-component," <https://github.com/bytecodealliance/cargo-component>.
- [32] Y. Wuyts, "Changes to rust's wasi targets," <https://blog.rust-lang.org/2024/04/09/updates-to-rusts-wasi-targets.html>, 2024.

5 References

- [33] B. Alliance, “wit-bindgen,” <https://github.com/bytecodealliance/wit-bindgen>.
- [34] —, “Componentizejs,” <https://github.com/bytecodealliance/ComponentizeJS>.
- [35] wasmtime, “Macro wasmtime::component::bindgen,” <https://docs.rs/wasmtime/latest/wasmtime/component/macro.bindgen.html>.
- [36] *I2C-bus specification and user manual*, NXP Semiconductors, 10 2021, rev. 7.
- [37] *System Management Bus (SMBus) Specification*, System Management Interface Forum, 1 2022, rev. 3.2.
- [38] R. Embedded, “embedded-hal,” <https://github.com/rust-embedded/embedded-hal>.
- [39] R. E. R. Team, “Embedded rust,” <https://github.com/rust-embedded/awesome-embedded-rust>.
- [40] E. R. W. Group, “embedded-hal v1.0 now released!” <https://blog.rust-embedded.org/embedded-hal-v1/>, 2024.
- [41] R. Kurte, “Embedded hal compatibility layer,” <https://github.com/ryankurte/embedded-hal-compat>.
- [42] J. Aparicio, “svd2rust,” <https://crates.io/crates/svd2rust>.
- [43] probe rs, “probe-rs,” <https://probe.rs/>.
- [44] Microsoft, “Debug adapter protocol,” <https://microsoft.github.io/debug-adapter-protocol/>.
- [45] F. Vandenberghe, L. Clark, and J. Schilling, “Wasi i2c,” <https://github.com/WebAssembly/wasi-i2c>.
- [46] D. Gohman, “Wasi i/o,” <https://github.com/WebAssembly/wasi-io>.
- [47] F. Vandenberghe, “i2c-wasm-components,” <https://github.com/Zelzahn/i2c-wasm-components>.
- [48] D. Gallagher, “hts221,” <https://github.com/danielgallagher0/hts221>.
- [49] F. Vandenberghe, “hts221,” <https://github.com/Zelzahn/hts221>.
- [50] R. E. L. team, “linux-embedded-hal,” <https://github.com/rust-embedded/linux-embedded-hal>.
- [51] R. van der Meer, “Rppal - raspberry pi peripheral access library,” <https://github.com/golemparts/rppal>.
- [52] rp rs, “rp-pico,” <https://github.com/rp-rs/rp-hal-boards>.
- [53] F. Vandenberghe, “wasi-embedded-hal,” <https://github.com/Zelzahn/wasi-embedded-hal>.
- [54] A. Kladov, “once_cell,” https://github.com/matklad/once_cell.
- [55] E. R. W. Group, “Psa: Cortex-m breakage (lld as the default linker),” <https://blog.rust-embedded.org/2018-08-2x-psa-cortex-m-breakage/>, Aug. 2018.

5 References

- [56] F. Vandenberghe, "Compiling wamr-sys for thumbv6m-none-eabi fails," <https://github.com/bytecodealliance/wamr-rust-sdk/issues/30>, 2024.
- [57] B. Heisler, "criterion.rs," <https://github.com/bheisler/criterion.rs/>.
- [58] N. Nethercote, "dhat-rs," <https://github.com/nnethercote/dhat-rs>.
- [59] flamegraph rs, "flamegraph," <https://github.com/flamegraph-rs/flamegraph>.