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 protool. 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 calllee 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 platform- 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 geinteracteerd.

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 alllocater 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.

```rust
#![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 alllocater 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:

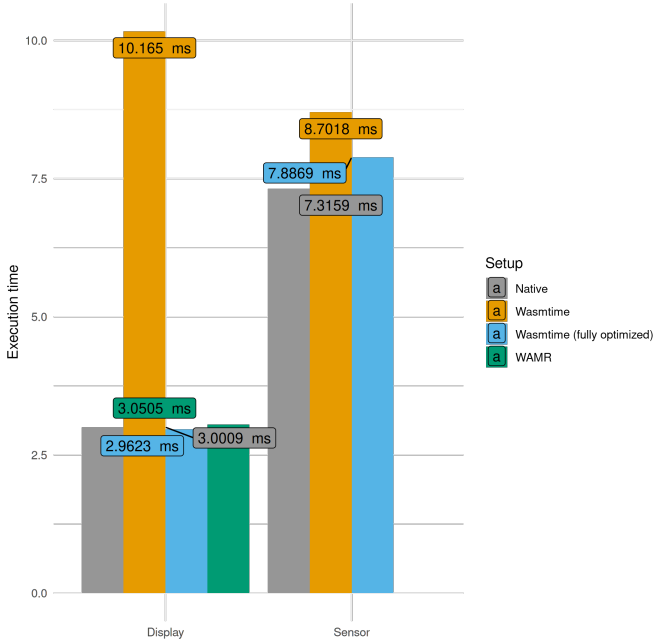
- Op de Raspberry Pi 3 wordt de temperatuur uitgelezen. Deze waarde wordt handmatig geverifieerd met de native implementatie.
- Op de Raspberry Pi 4 wordt de string `"1234"` naar het scherm geschreven. Dit wordt visueel gecontroleerd.

C. Benchmarks

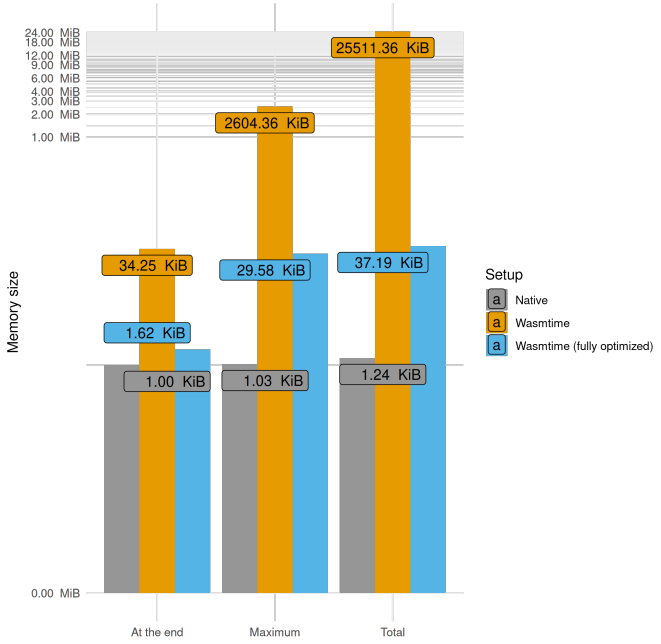
Het draaien in Wasmtime zou slechter moeten presteren door de opgelegde overhead op zowel uitvoeringstijd als geheugengebruik, maar deze overhead zal te verwaarlozen zijn. Voor WAMR zou de overhead nog kleiner moeten zijn.

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 Cranelift.

Cranelift 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



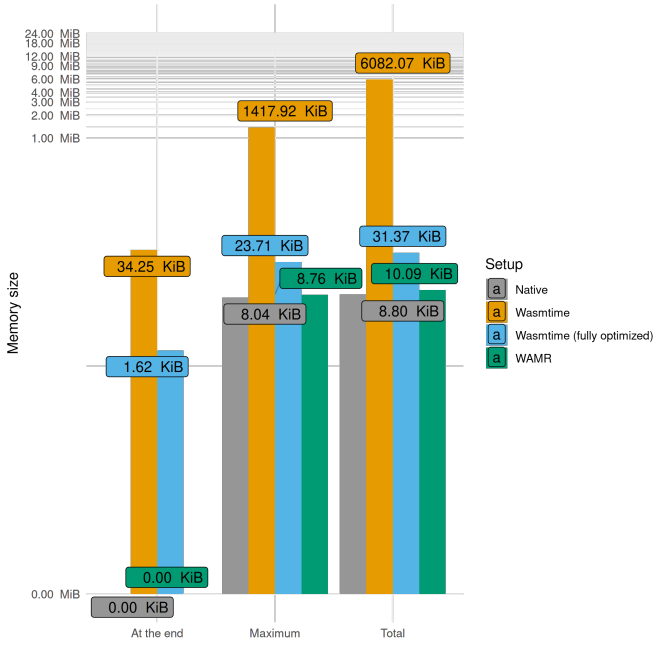
Figuur 2: Gebruikt geheugen bij het lezen van de sensor

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 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 vertragingen. 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 runtim

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)

[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.