Standaardiseren van USB in de WebAssembly System Interface

Wouter Hennen
Promotoren: prof. dr. Bruno Volckaert, prof. dr. ing. Filip De Turck
Begeleiders: dr. ing. Merlijn Sebrechts, ing. Michiel Van Kenhove

Abstract—Deze thesis onderzoekt de toevoeging van USB-ondersteuning aan de WebAssembly System Interface (WASI). Het doel is om ontwikkelaars in staat te stellen om op een platformonafhankelijke en programmeertaalonafhankelijke manier te communiceren met USB-apparaten, vergelijkbaar met WebUSB en WebAssembly in browsers. Door gebruik te maken van de ingebouwde access control van WASI, kan dit op een veilige manier worden bereikt, waardoor gebruikers efficiënt het beheer van de toegang tot USB-apparaten kunnen regelen. Dit document beschrijft de architectuur, het voorstel, de implementatie en de evaluatie van deze integratie. Aan de hand van benchmarks worden de prestaties van de implementatie geëvalueerd.

Index Terms—WebAssembly, WebAssembly System Interface, Component Model, Wasm, WASI, USB, wasi-usb, access control

I. Inleiding

In Web browsers wordt Wasm gebruikt om gecompileerde code uit te voeren in een lichte virtuele machine. Code wordt gecompileerd naar Wasm, maar kan worden uitgevoerd op elk platform dat Wasm ondersteunt, waardoor het platformonafhankelijk is. Dit model kan ook buiten de browsercontext worden toegepast en kan enkele problemen van native code oplossen. Omdat Wasm primair gericht is op webbrowsers, bevat het alleen API's om te communiceren met browsers. Als Wasm-code buiten de browser moet worden uitgevoerd, zijn er nieuwe API's nodig. Om dit op te lossen, is er een nieuwe specificatie gemaakt: de WebAssembly System Interface (WASI). WASI is een verzameling API-specificaties die door Wasm-toepassingen buiten de browser kunnen worden gebruikt. Dankzij WIT en de canonical ABI kunnen deze API's een meer gebruiksvriendelijke interface aanbieden in vergelijking met Wasm. Deze API's kunnen nog steeds worden gebruikt door elke programmeertaal die WASI ondersteunt. Door gebruik te maken van gegenereerde bindings, kan de in WIT gedefinieerde API worden omgezet naar variant die past bij de programmeertaal en compatibel is met de features van de taal. Zo kunnen bijvoorbeeld talen een verschillende stringrepresentatie hebben maar toch dezelfde API gebruiken. Omdat de API's fungeren als een laag tussen de native API's en de applicatie, kan access control worden toegepast om te bepalen waartoe een applicatie toegang heeft.

Deze thesis breidt de WebAssembly System Interface (WASI) uit met een gestandaardiseerde USB-API, die veilige en efficiënte communicatie met USB-apparaten op verschillende platforms mogelijk maakt. De USB-API maakt gebruik van de access control van WASI om de toegang tot USB-apparaten te beveiligen. Er zal ook aandacht worden besteed aan de draagbaarheid en prestaties van de API.

II. Voorstel

De wasi-usb repository [1] bevat de inhoud van het voorstel om de USB-interface te standaardiseren. Het voorstel bevat een eerste versie van de WIT-interface van de USB-API. In deze sectie worden de belangrijkste onderdelen van de API belicht. Het wasi-usb voorstel bevindt zich ten tijde van het schrijven in de proposal fase 1.

Het voorstel bevat de volgende functies:

- Lijst van USB-apparaten krijgen.
- Descriptors van apparaten lezen.
- Device handles openen.
- Gegevens van apparaten lezen.
- Gegevens naar apparaten schrijven.
- Melding ontvangen wanneer een apparaat aangesloten wordt.
- Melding ontvangen wanneer een apparaat verwijderd wordt.

Figuur 1 toont een algemeen overzicht van de verschillende onderdelen van een programma dat de USB-API gebruikt. De implementatie bevindt zich in het 'WASI USB'-blok; hier gaat het voorstel over.

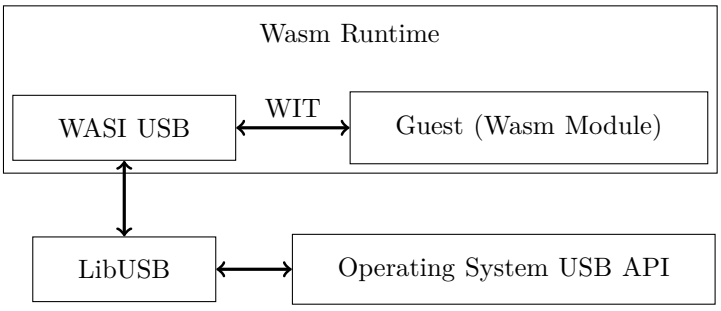

Fig. 1. Interne onderdelen van de WASI USB API. De pijlen vertegenwoordigen communicatie tussen de verschillende gebieden.

A. WIT interface

Het voorstel bevat vier WIT-interfaces: usb, events, descriptors en types. De belangrijkste, usb, wordt hier besproken. Deze wordt getoond in Figuur 2, enkele details zijn weggelaten voor het compact te houden.

De interface bevat twee resources: usb-device en device-handle. De usb-device resource bevat een statische functie enumerate, waarmee alle beschikbare USB-apparaten kunnen worden opgehaald. Deze worden geretourneerd als instanties van usb-device. De andere drie methoden kunnen op een van deze instanties worden gebruikt. De functies device-descriptor en configurations zullen beide extra informatie over een apparaat retourneren en zijn altijd beschikbaar. Dat wil zeggen, ze hoeven geen exclusieve toegang tot deze apparaten te hebben om die informatie te verkrijgen. De functie open is in dit opzicht anders. Deze functie zal een context opzetten waarmee u met het apparaat kunt communiceren. Een belangrijk aspect is dat het openen van een apparaat kan mislukken om verschillende redenen. Sommige besturingssystemen kunnen bijvoorbeeld weigeren een apparaat te openen als de uitvoerbare code niet voldoende rechten heeft. Deze rechten worden afgehandeld op Wasm-runtime niveau en zijn niet specifiek voor Wasm-modules. Om een robuuste en intuïtieve API te creëren, zijn functies die een 'open' apparaat vereisen, gesitueerd in de device-handle resource, die alleen kan worden verkregen als een apparaat succesvol kan worden geopend.

De device-handle resource zal alle functies bevatten die communicatie met een apparaat vereisen, bijvoorbeeld voor de de actieve configuratie van een apparaat op te halen. Er bestaan functies om te lezen en te schrijven met de vier transfer types: read-interrupt, write-interrupt, enz. De meeste functies in device-handle zijn nauw verwant aan vergelijkbare LibUSB-functies [2].

```
package component:usb@0.2.0;

interface usb {
    use types.{usb-error};
    use descriptors.{configuration-descriptor,
        device-descriptor};

    type duration = u64;

    resource usb-device {
        configurations: func() -> ...;

        device-descriptor: func() -> device-
            descriptor;

        open: func() -> result<device-handle, usb-
            error>;

        enumerate: static func() -> list<usb-device
            >;
    }

    resource device-handle {
        reset: func() -> result<_, usb-error>;
        active-configuration: func() -> result<u8,
            usb-error>;

        select-configuration: func(configuration: u8
            ) -> result<_, usb-error>;

        claim-interface: func(%interface: u8) ->
            ...;

        release-interface: func(%interface: u8);

        select-alternate-interface: func(%interface:
             u8, setting: u8) -> ...;

        read-interrupt: func(...) -> ...;
        write-interrupt: func(...) -> ...;

        read-bulk: func(...) -> ...;
        write-bulk: func(...) -> ...;

        read-isochronous: func(...) -> ...;
        write-isochronous: func(...) -> ...;

        read-control: func(...) -> ...;
        write-control: func(...) -> ...;
    }
}
```

Fig. 2. USB interface.

B. Access Control

Het voorstel specificeert access control op een apparaatniveau. Het doet dit door niet-toegestane apparaten te filteren voor usb.usb-device/enumerate, wat een lijst van alle beschikbare apparaten geeft, en events.update, wat updates geeft van apparaatverbindingen. Een gastmodule kan alleen een usb-device resource ontvangen van deze twee functies, dus als ze die resource niet retourneren, kan de gast deze niet gebruiken.

Toegangscontrole kan ook worden gespecificeerd op configuratie-, interface- en eindpuntniveau, maar deze zijn voorlopig niet toegevoegd, omdat nog niet bekend is of ze voordelen zouden bieden.

C. Gebruik in meerdere componenten

Het is mogelijk dat meerdere componenten tegelijkertijd communiceren met de USB-API en proberen toegang te krijgen tot een USB-apparaat. Dit moet worden vermeden, aangezien een component over het algemeen exclusieve toegang tot een apparaat verwacht. Om dit probleem op te lossen, moet de runtime bijhouden welke apparaten door welke componenten worden gebruikt en het openen van een apparaat weigeren wanneer een andere component het al gebruikt. Deze benadering volgt hetzelfde model als de meeste besturingssystemen, waarbij slechts één programma toegang kan hebben tot een USB-apparaat tegelijk.

III. Implementatie

Deze sectie gaat dieper in op de interne aspecten van de implementatie. Eerst wordt de runtime-implementatie

besproken. Vervolgens wordt uitgelegd hoe de API in een gastcomponent kan worden gebruikt.

De wasi-usb [1] repository bevat de implementatie die in deze sectie wordt uitgelegd, evenals een aantal gastcomponenten die hiervan gebruikmaken.

A. Runtime-implementatie

De implementatie is geschreven in Rust [3]. Voor de native implementatie wordt libusb [4] gebruikt om OS-specifieke API's aan te roepen. De Rusb [5] Rust package wordt gebruikt en voegt een thin wrapper rond de libusb-API toe, waardoor het eenvoudiger te gebruiken is in Rust. De implementatie breidt de Wasmtime [6] Wasm-runtime uit en zal Wasmtime in zijn gecompileerde code inbedden. Bij het uitvoeren van het programma kunnen drie parameters worden doorgegeven:

- Gastcomponentpad (vereist): Het pad naar de gastcomponent die zal worden uitgevoerd. De component moet een command component zijn die een run functie bevat.
- '–usb-devices': Een lijst van apparaten waartoe de gastcomponent toegang heeft. De apparaten worden gespecificeerd in het vendor_id:product_id formaat.
- '–usb-use-denylist': Het toevoegen van deze vlag zal de apparatenlijst veranderen van een toestemmingslijst naar een ontkenningslijst.

Standaard heeft de gast geen toegang tot apparaten. Tabel I toont de mogelijke combinaties voor apparaattoegang.

	Devices Specified	No devices specified
Allowlist	specified devices allowed	no device allowed
Denylist	all but specified allowed	all allowed

TABLE I
Opties voor toegangscontrole van apparaten.

B. Gebruik door gastcomponent

Om gebruik te kunnen maken van de USB-API, moet een gast eerst de interfaces in zijn WIT-interface importeren, zoals weergegeven in Figuur 3. Deze component is een Command-component en hoeft daarom geen exports te hebben. Op basis van deze definitie kan een generator bindings aanmaken die natuurlijk aanvoelen voor de taal waarin de gast is geschreven. De gegenereerde bindings zullen de gebruiksvriendelijke API vervolgens omzetten naar een representatie die voldoet aan de canonical ABI [7].

IV. Evaluatie

De API is op drie manieren geëvalueerd. Eerst werd een functionele evaluatie uitgevoerd om te controleren of de API zich gedraagt zoals verwacht. Vervolgens werd een vertragingsevaluatie uitgevoerd om te meten of er merkbare vertragingen zijn bij het gebruik van de API vergeleken met native code. Ten slotte werd

```
package component:usb-component-wasi-stadia;

world root {
    import component:usb/types@0.2.0;
    import component:usb/usb@0.2.0;
    import component:usb/events@0.2.0;
    import component:usb/descriptors@0.2.0;
}
```

Fig. 3. De WIT world voor de gastcomponent.

```
dpad: ,
buttons: assistant_button|l2_button|r2_button,
left stick: x: 128 y: 128,
right stick: x: 128 y: 128,
l2: 255,
r2: 172
```

Fig. 4. Output na het lezen van de staat van de controller.

een geheugenevaluatie uitgevoerd om de impact op het geheugengebruik van het uitvoeren van de code in Wasm te meten in vergelijking met native code.

De code van alle benchmarks is te vinden in de USB_WASI [8] repository.

A. Functionele evaluatie

De functionele evaluatie raakt alle onderdelen van de API: het verkrijgen van verbindings- en ontkoppelingsgebeurtenissen van apparaten, het lezen van descriptors van een apparaat, het openen van een device handle, het lezen van gegevens van het apparaat en het schrijven van gegevens naar het apparaat.

Het algemene idee van het programma is om een gamecontroller te bedienen. Het programma wordt gestart en observeert de verbonden apparaten. Zodra een controller is verbonden die wordt herkend door het programma, zal het programma verbinding maken met de controller. De status van alle knoppen van het apparaat wordt gelezen en geprint. Om het verzenden van gegevens via de USB-interface te testen, stuurt het programma commando's naar de controller om de trilmotoren te activeren. Als het programma een ontkoppelingsgebeurtenis ontvangt, stopt het met het lezen van gegevens en wordt het weer inactief, in afwachting van nieuwe verbindingen.

Dit programma is getest met een Google Stadia-controller. Wanneer een of meerdere knoppen worden ingedrukt, wordt de juiste status afgedrukt. Wanneer een of beide schouderknoppen worden ingedrukt, begint de controller te trillen. Figuur 4 toont een voorbeeldoutput met de status van de controller.

B. Vertragingsevaluatie

Vertraging wordt getest met twee benchmarks: een synthetische benchmark, waarbij een Arduino constant

gegevens zal verzenden, en een benchmark in de echte wereld, waarbij bestanden van een USB mass storage apparaat worden gelezen.

1) Lezen van gegevens van Arduino: De benchmark zal 1 miljoen pakketten van 64 bytes over de bulkinterface van de Arduino lezen. In totaal wordt er 64 MB aan gegevens gelezen. De gegevens worden sequentieel gelezen, zonder vertragingen. Om de ontvangst van de gegevens te starten, stuurt het programma eerst het Device Terminal Ready-signaal naar de Arduino. Zodra de Arduino dit signaal ontvangt, begint hij de gegevens te verzenden. De Arduino voert zo min mogelijk verwerking uit en verzendt een hardcoded array van bytes. De host ontvangt de gegevens en verwerkt deze niet, maar gooit ze weg. Dit is om zoveel mogelijk extra vertragingsfactoren te vermijden.

Figuur 5 toont een boxplot van de vertragingen van beide programma's. Na een miljoen metingen voor elk programma is de mediaan exact hetzelfde, 377 µs. Het gemiddelde van het native programma is marginaal lager, 0,2% sneller dan het WASI-programma. De whiskers en IQR van het WASI-programma zijn iets kleiner, wat een consistenter resultaat betekent. De verschillen in resultaten zijn echter te klein om enige merkbare vertraging te concluderen.

Figuur 5 bevat ook uitschieters, aangeduid met de zwarte cirkels. Sommige van de uitschieters zijn erg groot. Daarom worden de resultaten weergegeven op een logaritmische schaal. Deze uitschieters worden waarschijnlijk veroorzaakt door transmissiefouten bij het ontvangen van gegevens van de Arduino. De Bulk interface wordt gebruikt, die gegevensintegriteit biedt maar geen timinggaranties. Hierdoor wordt dataverlies voorkomen, maar worden hoge latenties geïntroduceerd zoals hier waargenomen. Tabel II toont meer informatie over de uitschieters. Aangezien het percentage uitschieters zeer klein is en in gelijke hoeveelheden voorkomt in beide gevallen, zijn ze uitgesloten van verdere figuren om de duidelijkheid van de grafieken te verbeteren.

Figuur 6 toont een Kernel Density Estimation voor de metingen van de native en WASI-implementatie. Beide grafieken tonen pieken rond de 150 µs en 910 µs. Dit is een interessant resultaat, aangezien men één uniforme verdeling zou verwachten in plaats van twee. De eerste piek is eenvoudig te verklaren: de Arduino is een USB 2.0-apparaat, ook wel bekend als een High-speed USB-apparaat. Een High-speed USB-apparaat stuurt frames met een vaste interval van 125 µs. Daarom ontvangen we ongeveer elke 125 µs nieuwe gegevens. Een extra 25 µs wordt toegevoegd als gevolg van verwerking.

De tweede piek is subtieler. Tabel III toont een fragment van de gemeten vertragingen. Voor zowel de native als de WASI-code zullen de vertragingen tussen beide pieken

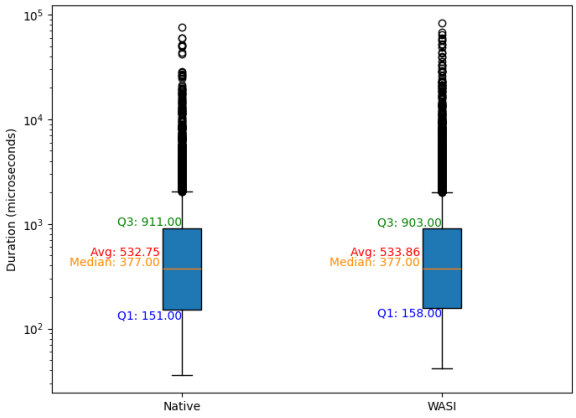

Fig. 5. Boxplot van vertragingen bij het lezen van data van Arduino met read-bulk.

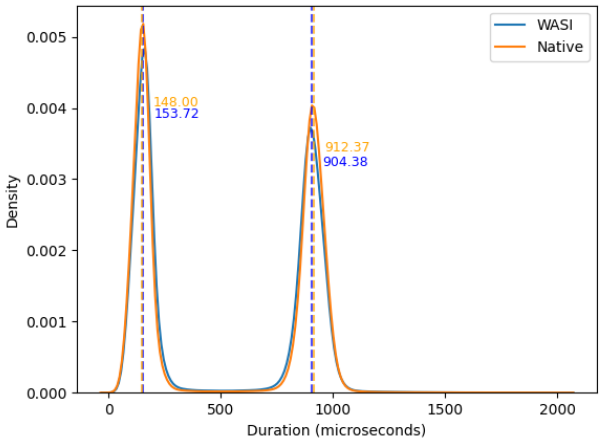

Fig. 6. KDE van vertragingen bij het lezen van data van Arduino met read-bulk.

oscilleren. Op basis van deze resultaten wordt vermoed dat deze piek wordt veroorzaakt door de kleine buffergrootte in de Arduino. De buffer is 64 bytes groot, wat overeenkomt met de grootte van één pakket. Wanneer de buffer vol is, stopt het programma totdat er weer ruimte beschikbaar is in de buffer. Op de host wordt dit waargenomen als een extra vertraging bij het ontvangen van de gegevens.

	Native	WASI
Largest Outlier (us)	75618	83052
Amount of Outliers (>2000 us)	747	1058
Percentage of Total	0.07%	0.10%

TABLE II
Vergelijking van uitschieters bij vertragingen tussen native en WASI.

Op basis van de resultaten kunnen we concluderen dat er geen meetbare verschillen zijn tussen het native en WASI-

Native (us)	WASI (us)
801	918
150	163
885	931
132	241
903	799
110	197
938	820
104	77
962	1003
108	97

TABLE III
Stuk van vertragingsmetingen. De vertragingen oscilleren tussen een lange en korte vertraging.

programma. Dit wordt echter voornamelijk veroorzaakt door de vertraging van het USB-protocol. Deze vertraging weegt ruimschoots op tegen de vertragingen veroorzaakt door de overhead van de Wasm-runtime, waardoor de overhead van Wasm verwaarloosbaar is.

Het is echter mogelijk dat de vertraging van het USB-protocol kleiner is op krachtigere apparaten dan een Arduino of met een modernere USB-versie. Met deze configuraties kan het mogelijk zijn dat een kleine overhead voor Wasm zichtbaar wordt.

2) Bestanden lezen vanaf een Mass Storage-apparaat: Om te bevestigen dat WASI geen merkbare vertraging toevoegt, wordt een andere benchmark uitgevoerd waarbij de inhoud van een USB-massastorageapparaat wordt gelezen. Het voordeel van deze benchmark ten opzichte van de Arduino-benchmark is dat deze beter de real-world-gebruik vertegenwoordigt, in plaats van een synthetische benchmark te zijn.

Er is een programma geschreven dat de bestandsstructuur van een USB-apparaat opsomt en de inhoud van elk bestand in de bestandsstructuur leest. Een Samsung T5 External SSD wordt gebruikt om deze tests uit te voeren. Het apparaat is verbonden met een kabel die een overdrachtssnelheid tot 5 Gbps ondersteunt en is geformatteerd met de MBR-partitiemap en het exFAT-bestandssysteem. Er staan 10 bestanden op het apparaat, waarvan de meeste een paar KB groot zijn. Één bestand heeft een grootte van 679 MB. In totaal is er 680 MB op het apparaat opgeslagen. De benchmark wordt 1000 keer uitgevoerd.

Figuur 7 toont een boxplot van de gemeten totale tijden om de bestandsstructuur te lezen. De gemiddelde uitvoeringstijd van het WASI-programma is 4,2% langzamer dan die van het native programma, en de mediaan 4,3% langzamer. Het WASI-programma heeft ook meer uitschieters, wat aangeeft dat de prestaties van het WASI-programma mogelijk iets minder consistent zijn.

Op basis van deze resultaten kunnen we concluderen dat het uitvoeren van de code met behulp van de WASI USB-API een kleine hoeveelheid overhead met zich meebrengt.

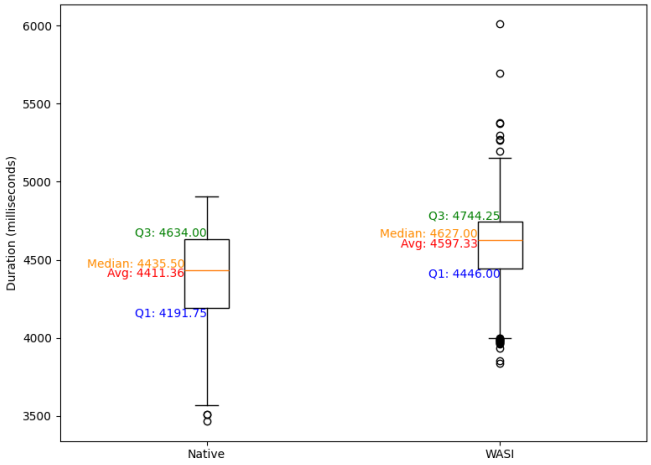

Fig. 7. De vertraging van het lezen van de bestandsstructuur en inhoud na het wijzigen van de benchmarkmethode. Gemiddeld genomen is de WASI USB-implementatie ongeveer 4,2% langzamer.

C. Geheugenevaluatie

In tegenstelling tot standaard Wasm-modules hebben WASI-componenten afzonderlijke geheugenruimtes. Als gevolg hiervan kan één component niet het geheugen van een andere component lezen of schrijven [9]. Het is daarom interessant om te zien welke invloed WASI heeft op het geheugengebruik van een typisch programma.

1) Bestanden lezen vanaf mass storage apparaat: Dezelfde testmethodologie als in Sectie IV-B2 wordt gebruikt om het geheugengebruik te benchmarken. In plaats van de benchmark 1000 keer uit te voeren, wordt deze nu één keer uitgevoerd. Het programma zal ook drie seconden wachten voordat en nadat het apparaat is geopend en gesloten, zodat de evolutie van het geheugengebruik beter zichtbaar is.

Figuur 8 toont de verschillen in geheugengebruik voor beide programma's. Aangezien beide programma's veel geheugen gebruiken, wordt het initiële Wasmtime-geheugengebruik verwaarloosd, omdat dit geen grote invloed heeft op de resultaten. De invloed van het initiële Wasmtime-geheugen wordt besproken in Sectie IV-C2.

Na de eerste drie seconden wachttijd beginnen beide programma's de bestandsstructuur te doorlopen en de inhoud te lezen. Het is duidelijk dat het WASI-programma meer geheugen gebruikt en meer pieken vertoont. Omdat het geheugen niet wordt gedeeld, worden waarden gekopieerd van host naar gast, wat het geheugengebruik verhoogt. Op zijn hoogtepunt zal het 1239 MB gebruiken, terwijl het native programma 718 MB gebruikt. Dit is 72,5% meer. Dit piekgebruik treedt op bij het lezen van het grootste bestand van 679 MB. Omdat de schijf grotendeels leeg is, worden gegevens aaneengesloten op de schijf opgeslagen. Dit verbetert de leesprestaties en zorgt voor een gelijkmatige stroom van gegevens aan het ontvangende einde. Dit is duidelijk zichtbaar in de grafiek voor

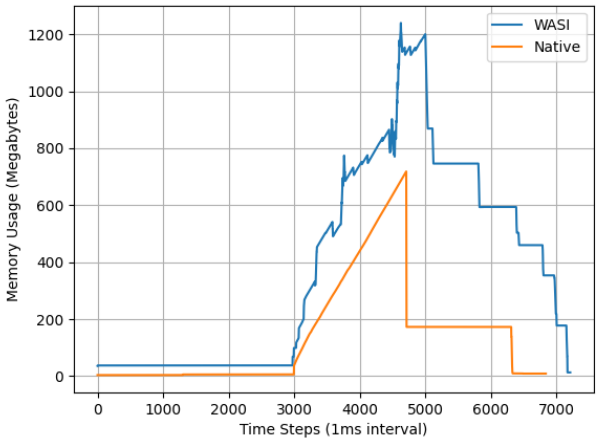

Fig. 8. Het geheugengebruik bij het doorlopen van de bestandsstructuur en het lezen van de bestandsinhoud. Aan het einde van het lezen van het grootste bestand (678 MB) zal het WASI-programma 72,5% meer geheugen gebruiken dan het equivalente native programma.

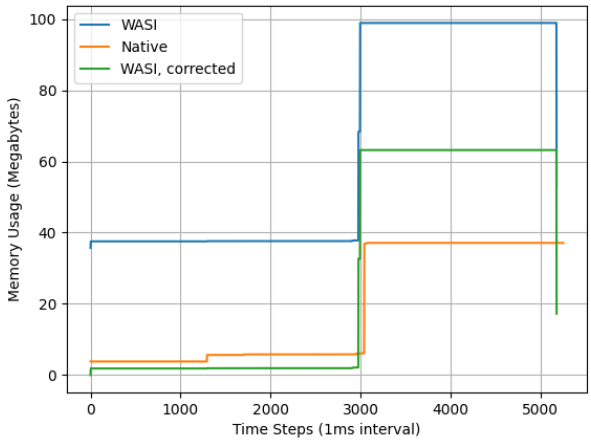

Fig. 9. Geheugengebruik bij het doorlopen van de bestandsstructuur. Het WASI-programma verbruikt 69,8% meer geheugen dan het equivalente native programma.

het native programma, omdat het geheugengebruik een lineaire functie volgt. De WASI-implementatie bevat meer pieken. Deze pieken worden waarschijnlijk veroorzaakt door het kopiëren van waarden, waardoor snel geheugen wordt toegewezen en vrijgegeven.

Nadat de inhoud is gelezen, zal het geheugengebruik voor beide programma's snel dalen. Het geheugengebruik voor het WASI-programma zal echter langzamer dalen dan voor het native programma. Dit gebeurt voornamelijk vanwege het memorypool-achtige gedrag van de Store [10], die het geheugen niet vrijgeeft. Geheugen dat buiten de Store wordt gebruikt, wordt nog steeds vrijgegeven, vandaar de afname van het geheugengebruik. Zodra het programma eindigt, zal het geheugen verder worden vrijgegeven.

2) Overlopen van file tree van mass storage apparaat: Een minder geheugenintensieve benchmark is ook uitgevoerd om te controleren hoe het WASI-geheugengebruik zou zijn bij gebruik van minder geheugen. Deze benchmark gebruikt dezelfde testmethodologie als Sectie IV-C1, behalve dat het de bestandsinhoud niet uitleest. De bestandsstructuur is ook gewijzigd en bevat nu 11947 bestanden, met een totale grootte van 3,25 GB.

Figuur 9 toont de resultaten van de benchmark. De grafiek bevat de resultaten van het totale geheugengebruik van beide programma's, en bevat ook de WASI-resultaten zonder het initiële geheugen dat wordt ingenomen door Wasmtime. Dit is een nuttige maatstaf, omdat een programma dat de USB-API gebruikt deel kan uitmaken van een groter stuk code, waarbij de initiële geheugenoverhead van Wasmtime wordt 'verdeeld' over alle componenten en kan worden verwaarloosd.

De eerste 3000 metingen tonen het geheugengebruik vanaf het moment dat het programma drie seconden slaapt. Het geheugen van zowel de native als de WASI-programma's is vergelijkbaar, als we het initiële Wasmtime-geheugengebruik niet meerekenen. Bovendien is het geheugengebruik van WASI aan het begin iets beter, maar dit wordt waarschijnlijk veroorzaakt door de correctie. Zodra de nuttige programmacode begint, zijn er geheugenpieken voor beide programma's. Het WASI-programma verbruikt echter ongeveer 63 MB geheugen, 69,8% meer dan het geheugengebruik van de native van 37,1 MB.

V. Conclusie

Het voorstel dat gedurende het jaar is ontwikkeld, samen met de initiële werkende implementatie in Wasmtime, heeft de meeste vragen en problemen doorgelicht die men zou verwachten bij het ontwikkelen van een dergelijke API. Hoewel het voorstel nog een lange weg te gaan heeft, heeft het de fundamenten van de API uiteengezet die verder kunnen worden uitgebreid.

Een van de meest voorkomende vragen was hoe access control, een van de belangrijkste kenmerken van WASI, geïntegreerd kon worden in de API. Na het uitproberen van meerdere mogelijkheden werd geconcludeerd dat het beperken van de access control op een apparaatniveau voldoende is geacht. Het idee is ook geïmplementeerd in de implementatie en werkt. Het voorstel bevat ook richtlijnen voor het toevoegen van access control aan andere delen van de interface, mocht dit nodig zijn.

Er zijn meerdere API-ontwerpen getest, waarbij de ene meer leunt op wat libusb biedt, terwijl de andere meer leunt op WebUSB. Na het uitproberen van beide API's en het krijgen van feedback van de gemeenschap, is gekozen voor een API die nauw aansluit bij libusb. Deze API is meer in lijn met hoe USB-apparaten intern werken. Dit maakt het ook gemakkelijker voor bestaande programma's

die libusb gebruiken om hun code over te zetten naar WASI USB.

De prestaties van de API zijn grondig getest en de resultaten zijn over het algemeen positief. Wasm is snel genoeg om geen merkbare prestatieproblemen te hebben bij het gebruik van de USB-API, vooral in vergelijking met de vertraging van de communicatie met een extern apparaat. Echter, vanwege het geïsoleerde geheugen van WASI-componenten, moet data die van en naar een apparaat wordt verzonden worden gekopieerd, wat extra geheugenoverhead met zich meebrengt. Dit kan een probleem worden voor toepassingen die veel geheugen gebruiken.

VI. Toekomstig Werk

Het onderzoek voor deze scriptie heeft de basis gelegd voor het bouwen van een WASI USB API. Echter, het voorstel bevindt zich nog steeds in fase 1 en heeft nog een lange weg te gaan voordat het stabiel kan worden. Verder werk zal moeten plaatsvinden op het gebied van testen, het leveren van meerdere runtime-implementaties, het maken van volledig functionele programma's die de API gebruiken en het verkrijgen van verdere feedback van de gemeenschap.

Bovendien heeft de API momenteel enkele tekortkomingen die in de toekomst moeten worden bekeken:

- Het voorstel negeert momenteel taalondersteuning, maar USB-apparaten kunnen bepaalde gegevens, zoals apparaatnamen, in meerdere talen verstrekken.
- Windows Hotplug-ondersteuning ontbreekt momenteel, waardoor de events-interface van het voorstel momenteel niet beschikbaar is voor Windows. Recentelijke activiteit wijst echter uit dat dit binnenkort mogelijk zal komen [11].

References

[1] WebAssembly, "Webassembly/wasi-usb." [Online]. Available: https://github.com/WebAssembly/wasi-usb

[2] "libusb docs." [Online]. Available: https://libusb.sourceforge.io/api-1.0/group___libusb___dev.html

[3] "Rust." [Online]. Available: https://www.rust-lang.org/

[4] "libusb." [Online]. Available: https://libusb.info/

[5] "Rusb." [Online]. Available: https://docs.rs/rusb/latest/rusb/

[6] "Wasmtime." [Online]. Available: https://wasmtime.dev/

[7] "Canonical abi." [Online]. Available: https://component-model.bytecodealliance.org/advanced/canonical-abi.html

[8] Wouter01, "Wouter01/usb_wasi." [Online]. Available: https://github.com/Wouter01/USB_WASI

[9] "Why the component model?" [Online]. Available: https://component-model.bytecodealliance.org/design/why-component-model.html

[10] "wasmtime::store." [Online]. Available: https://docs.rs/wasmtime/20.0.2/wasmtime/struct.Store.html

[11] "windows: hotplug implementation by sonatique." [Online]. Available: https://github.com/libusb/libusb/pull/1406