THM TECHNISCHE HOCHSCHULE MITTELHESSEN | CAMPUS GIESSEN | MNI Mathematik, Naturwissenschaften und Informatik

Ausarbeitung

WebAssembly

Kurs "Hauptseminar"

von

Alex Ruhl

am 19. Juni 2020 vorgelegt

Modulverantwortlicher: Sebastian Süß

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Gießen, 19. Juni 2020

Abstract

Diese Arbeit befasst sich mit WebAssembly (abgekürzt Wasm). WebAssembly ist eine neue low-level Sprache die nahezu native Laufzeitperformanz bietet und von allen großen Browsern unterstützt wird. Nach einer Einführung, die einen ersten Überblick bietet, wird WebAssembly erläutert. Hierbei wird hauptsächlich auf die Architektur, die Konzepte und unterstützte Programmiersprachen eingegangen.

Im Anschluss geht es um die Verwendung im Browser und zukünftige Features von WebAssembly. Außerdem werden aktuelle Zahlen zur Verbreitung analysiert. Nach dem ein größeres Verständnis zu der Technologie vorliegt, werden die Anwendungsgebiete erläutert.

Der Ausblick zeigt, welche Aussichten WebAssembly in der Zukunft hat und argumentiert die Erfolgsaussichten. In der Diskussion wird ein kritisches Fazit gezogen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einführung

Für sehr lange Zeit war JavaScript die einzige Möglichkeit für interaktive Anwendungen im Browser. Rechenintensive Aufgaben, wie z. B. Spiele, wurden von der schlechten Perfomanz JavaScripts zurückgehalten. Trotz der unaufhörlichen Optimierung von JavaScript-Engines genügt die Performanz nicht immer den hohen Anforderungen für das Web [16]. Ein erster Versuch dieses Problem anzugehen war asm.js. Mit asm.js sollten low-level Code Sprachen im Browser laufen können. In der Theorie konnte mit dieser Technologie mit den Sprachen nahezu native Performance erzielt werden [20], jedoch war asm.js nie in allen Browsern konsistent schnell [17]. Der Hauptgrund hierfür war, dass asm.js kein offizieller Web-Standard war. Bei WebAssembly ist dies anders, es ist seit 2017 offiziell von allen großen Browsern (Chrome, Edge, Firefox und Webkit) unterstützt [19] und ist seit 2019 offizieller Standard für das Web [18].

WebAssembly ist ein Binärinstruktionsformat für eine stackbasierte virtuelle Maschine. Es ist entworfen um als Kompilierziel von low-level Programmiersprachen wie C/C++/Rust zu dienen, hierdurch wird die Auslieferung im Web für Client- und Server-Anwendungen ermöglicht [1]. Die Nutzungsmöglichkeiten beschränken sich nicht nur ausschließlich auf das Web, jedoch liegt das Hauptaugenmerk auf diesem Gebiet. Das Ziel von WebAssembly ist clientseitige Applikationen, auf verschiedenen Plattformen, in der Ausführungszeit so nah wie möglich an native Applikationen zu bringen und dabei effizient zu sein.

Mit WebAssembly wird nicht versucht JavaScript zu ersetzen. JavaScript ist und bleibt für die meisten Anwendungsfälle die dominante Sprache im Web. WebAssembly und JavaScript sollen jedoch auf verschiedene Weise zusammen arbeiten. Zum Beispiel können Internetseiten wie üblich aufgebaut bleiben aber um schnelle WebAssembly-Module ergänzt werden. Diese Module können dann rechenintensive Aufgaben wie Simulationen, Bild-/Ton-/Video-Verarbeitung, Visualisierung, Animationen, Kompression und Verschlüsselung übernehmen [2].

Ziel dieser Arbeit ist es, WebAssembly zu untersuchen und dem Leser einen Einblick in diese noch neue Technologie zu geben.

2 WebAssembly

WebAssembly wurde 2015 öffentlich angekündigt [21] und basierte anfangs auf dem Funktionssatz von asm.js [23]. Der Hauptvorteil von WebAssembly gegenüber asm.js ist die schnellere Ladezeit, da die Parsezeit minimal ist. Erste Tests ergaben mehr als 20-mal schnellere Parsezeiten für WebAssembly [21]. Diese Performanz konnte bereits 2016 durch das Ausführen von Angry Bots, einem Unity Spiel, in Firefox, Google Chrome und Microsoft Edge bewiesen werden [24, 25, 26]. Seit dem hat WebAssembly einige Entwicklungsschritte durchgemacht. Es ist unter anderem offizieller W3C-Standard und von allen großen Browsern unterstützt [18, 19]. Dies sind bereits Errungenschaften, welche darauf hindeuten das WebAssembly eine Technologie ist, die längerfristig eine Rolle in der Web-Entwicklung spielen wird.

Abbildung 2.1: WebAssembly Logo [34].

2.1 Architektur

In diesem Abschnitt wird darauf eingegangen, was die Architektur von WebAssembly so schnell, effizient und plattformunabhängig macht. Wie bereits erwähnt ist WebAssembly eine stackbasierte virtuelle Maschine, diese führt Programme durch push und pop Operation auf dem Stack aus. Da bei so einer Vorgehensweise immer einzelne Instruktionen ausgeführt werden, ähnelt es in dieser Hinsicht Assembly, wodurch der Name zustande kam. Jedoch agiert WebAssembly nur mit der stackbasierten virtuellen Maschine und nicht wie Assembly, mit den CPU-Registern.

2.1.1 Binärinstruktionsformat

Durch die Nutzung des Binärinstruktionsformat kann die Parsezeit von WebAssembly minimal gehalten werden. JavaScript wird im Browser zu einem Abstrakten Syntaxbaum umgewandelt. Von hier aus wird der Abstrakte Syntaxbaum zu dem speziellen Bytecode der JavaScript-Engine transformiert. Bei Webassembly fallen alle diese Schritte weg, da es bereits fertig im Binärinstruktionsformat beim Browser ankommt. Der Browser muss es nur noch decodieren und zur Fehlervermeidung validieren [27]. Außerdem spart das Binärinstruktionsformat Speichergröße. Eine Datei, die kleiner ist, wird schneller vom Server bereitgestellt und vom Client geladen. Obwohl JavaScript-Dateien durch Komprimierungsalgorithmen beachtlich an Größe abnehmen können, ist das komprimierte Binärformat von Webassembly trotzdem kleiner [27].

2.1.2 Virtuelle stack basierte Maschine

Das Binärinstruktionsformat von WebAssembly wird in einer stackbasierten virtuellen Maschine ausgeführt.

Ein Computer, wie wir ihn kennen ist eine Registermaschine. Er verfügt über Register, welche von der CPU geschrieben und gelesen werden. Instruktionen verändern Werte in den Registern und führen dadurch ein Programm aus [6].

Bei einer Stackmaschine hingegen befinden sich die Instruktionen mit den Werten auf dem Stack, die Werte befinden sich nicht in Registern. Ein Stack ist ein Stapel, der nach dem LIFO-Prinzip (Last In, First Out) funktioniert. Die Instruktionen manipulieren hier Werte die sich auf dem Stack befinden. Es gibt zwei Arten von Instruktionen. Die erste Art führt simple Datenoperationen aus und wird simple Instruktionen genannt. Mit diesen simplen Instruktionen werden Argumente vom Stack entfernt (pop) und Ergebnisse oder Argumente auf den Stack gelegt (push). Kontrollinstruktionen sind die zweite Art von Instruktionen, diese verändern den Kontrollfluss des Programms. Bei WebAssembly ist der Kontrollfluss

strukturiert. Das bedeutet, dass der Kontrollfluss durch verschachtelte Konstrukte wie Blöcke, Schleifen und Bedingungen realisiert wird [3].

2.1.3 Typen

WebAssembly verfügt über vier Typen:

Typ	Beschreibung
i32	32 Bit Ganzzahl
i64	64 Bit Ganzzahl
f32	32 Bit Fließkommazahl nach IEEE-754-2019
f64	64 Bit Fließkommazahl nach IEEE-754-2019

Tabelle 2.1: Typen in WebAssembly.

Ganzzahlen werden in WebAssembly vorzeichenlos gesehen, bis es zu einer Instruktion kommt. Falls ein Operator verwendet wird, der eine vorzeichenbehaftete Interpretation der Ganzzahl benötigt, so wird die Darstellung der Zahl mit dem Zweierkomplement umgewandelt. Für diese vier Datentypen gibt es die üblichen unären und binären Operatoren aber auch alle Umwandlungsoperationen [4].

2.2 Konzepte

Der Aufbau von WebAssembly unterliegt einigen Grundkonzepten. Typen und Instruktionen sind zwei dieser Konzepte und wurden bereits erläutert. Dieser Abschnitt widmet sich den restlichen grundlegenden Konzepten von WebAssembly. Die erläuterten Konzepte sind alle im Minimum Viable Product (MVP) von WebAssembly enthalten, diese Version ist momentan auch von den Browsern unterstützt [1].

2.2.1 Module

Der Unterschied zwischen Programm oder Bibliothek ist bei WebAssembly nicht bekannt, beide werden Module genannt. Module sind eines der wichtigsten Konzepte von WebAssembly, da so die Form von den Binärdateien spezifiziert wird. Sie sind ausführbare, ladbare und zu versendende Teil von WebAssembly [5]. Module bestehen aus verschiedenen Abschnitten [7] manche von diesen müssen angegeben sein und andere sind optional.

Benötigte Abschnitte:

1. **Typen**: Dieser Bereich beinhaltet Signaturen von Methoden, die im Modul definiert werden oder importierten Funktionen.

2. **Function**: Hält einen Index zu jeder im Modul definierten Funktion.

3. **Code**: Beinhaltet die Funktionskörper der im Modul enthaltenen Funktionen.

Optionale Abschnitte:

1. **Export**: Macht Funktionen, Speicher, Tabellen und globale Variablen für andere WebAssembly Module und JavaScript verfügbar.

2. **Import**: Spezifiziert Funktionen, Speicher, Tabellen und globale Variablen, welche von anderen WebAssembly Modulen oder JavaScript importiert werden sollen.

3. **Start**: Diese Funktion startet automatisch, wenn das Modul geladen ist (quasi eine Mainfunktion).

4. **Global**: Deklariert globale Variablen für das Modul.

5. **Memory**: Definiert den vom Modul genutzten Speicher.

6. **Table**: Macht es möglich auf, Elemente außerhalb des WebAssembly Moduls zuzugreifen, sowie JavScript Objekte. Die ist besonders für indirekte Funktionsaufrufe nützlich.

7. **Data**: Initialisiert importierten oder lokalen Speicher.

8. **Element**: Initialisiert importierte oder lokale Tables.

2.2.2 Linearer Speicher

Moderne Programmiersprachen bieten einen New-Operator, mit dem eine neue Instanz auf dem Heap erzeugt werden kann. Der Speicher wird also für Objekte angelegt und der Compiler kennt intern die Größe des Objekts oder findet diese heraus. Wenn eine Instanz an eine Funktion übergeben wird, weiß der Compiler außerdem ob es ein Pointer ist oder ein Wert und ob dieser auf dem Stack oder Heap ist.

WebAssembly verfügt über keinen traditionellen Heap und es gibt kein Konzept eines New-Operators. Der Speicher wird auch nicht für Objekte allokiert, da es keine Objekte in traditioneller Hinsicht gibt. Außerdem verfügt WebAssembly noch nicht über eine eigene Garbage Collection (GC).

Für Webassembly ist Speicher nur ein großes Array von Bytes, welches mit direkten Zugriffen arbeitet. Wenn mehr Speicher benötigt wird, kann dieser pageweise, in Schritten von 64 KB erhöht werden. Die Vergrößerung erfolgt jedoch nicht automatisch nach Bedarf, sondern muss händisch durch einen Aufruf der grow Instruktion gestartet werden. Der direkte Zugriff auf Speicher ist nicht nur schnell, sondern bietet auch Sicherheitsvorteile, denn der Host kann zwar jederzeit den

linearen Speicher des Moduls lesen und schreiben aber das Wasm-Module hat niemals Zugriff auf den Speicher des Hosts [12].

2.2.3 Tabellen

Kurzgesagt sind Tabellen vergrößerbare Arrays von Referenzen, die per Indexzugriff aus dem Wasm-Modul aufgerufen werden können. Dieses Konzept wurde entwickelt, um Funktionspointer aus C/C++ in WebAssembly verwendbar zu machen [8]. Um eine Funktionsreferenz aufzurufen, wird in WebAssembly die call_indirect Instruktion in Verbindung mit einem Index aufgerufen.

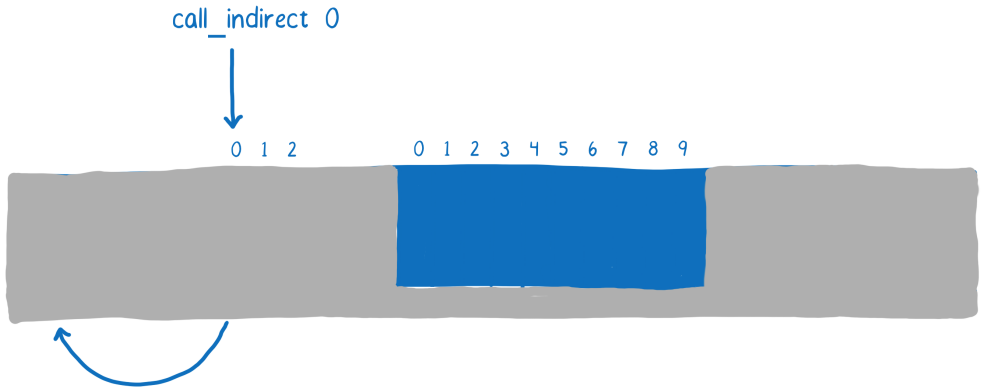

Abbildung 2.2: Illustration eines call indirect Aufrufs [8].

Der blaue Bereich aus Abbildung 2.2 zeigt den linearen Speicher des WebAssembly-Moduls. Die Indizes im grauen Bereich stellen die Tabelle der Funktionsreferenzen dar. Alles Weitere sind Speicheradressen, die nicht zum WebAssembly-Speicher gehören und deswegen niemals bekannt sein sollten.

Durch die Tabelle, welche die Funktionsreferenzen hält, in Kombination mit der call_indirect Operation, kennt ein Modul zu keiner Zeit die geanue Adresse einer Funktionsreferenz im Speicher und kann diese deswegen nicht verändern. Es greift nur per Index auf die Adressen zu. Dadurch das diese Referenzen nicht per Hand veränderbar sind, können diese Adressen auch nicht böswillig überschrieben werden und somit nicht durch einen Angriff auf Schadcode verweisen [8].

2.3 Unterstützte Sprachen

Das anfängliche Ziel war die Unterstützung von C und C++ Dateien. Dies wurde realisiert, indem Emscription um einen extra Wasm Parameter erweitert wurde, welcher ermöglicht, C oder C++ mithilfe des LLVM backend, nach Wasm zu kompilieren [35]. Im Laufe der Zeit sind weitere Sprachen dazu gekommen. Eine Auflistung aller zur Zeit unterstützten Sprachen findet sich hier [30].

.Net
AssemblyScript
Brainfuck
C
C#
C++
Clean
COBOL
Forth
Go
Lobster
Lua
Rust
Typescript
Wah
Zig

Tabelle 2.2: Aktuell vollständig unterstützte Sprachen.

2.4 Zukünftige Features von WebAssembly

Bereits geforderte und gewünschte wichtige Features wurden bewusst aus dem MVP herausgehalten, da diese Version nur die Mindestanforderungen beinhaltet. Anfangs sollte WebAssembly nur ein Kompilierziel sein, schnelle Ausführungszeit haben, ein kompaktes Format sein und über linearen Speicher verfügen. Mit diesen Eigenschaften konnten schon sehr viele Desktop-Anwendungen oder Spiele so im Browser laufen, als würden sie nativ auf dem PC ausgeführt werden. Jedoch gibt es viele Funktionalitäten, die noch nicht bereitgestellt werden. Dieser Abschnitt zeigt nur einen Teil der zukünftigen Features und eine genaue Auflistung lässt sich in folgendem Blogpost finden [28].

2.4.1 Threads

Der Threadsupport ist im MVP außer acht gelassen worden. Heutige Computer oder Mobilgeräte verfügen über eine Multiprozessorarchitektur. Nur mit Threads ist es möglich, eine Anwendung bedarfsgerecht auf verschiedene Prozessorkerne aufzuteilen und hierdurch die Gegebenheiten moderner Architekturen optimal zu nutzen. Ohne Threadunterstützung können WebAssembly-Module nur auf einem Core ausgeführt werden.

Für den Threadsupport besteht bereits ein offener Vorschlag zur Umsetzung im Github-Repository [22]. Dieser soll WebAssembly um einen geteilten linearen Speicher erweitern um zu ermöglichen, dass verschiedene Instanzen außerhalb

des Main-Threads, gleichzeitig auf den Speicher zugreifen können. Es besteht also bereits ein Konzept für die Umsetzung dieses wichtigen Features und es wird in Zukunft in die offizielle Spezifikation WebAssembly einfließen.

2.4.2 Garbage Collection

Die meisten modernen Programmiersprachen verfügen über einen Garbage Collector (GC) der den Speicher verwaltet. Ein Programmierer muss sich keine Gedanken mehr darüber machen, wann ein Objekt nicht mehr benötigt wird und gelöscht werden muss, da der GC diese Aufgabe übernimmt.

WebAssembly verfügt über keine Funktionen zur Speicherverwaltung oder Bereinigung, es stellt wie unter 2.2.2 erläutert, nur einen linearen Speicher. Sprachen die keinen GC verwenden, benötigen deswegen trotzdem einen Mechanismus um Speicher zu allokieren, wie z. B. Rust den WebAssembly optimised allocator braucht [29].

Sprachen die nicht ohne einen auskommen GC können, müssen diesen momentan zu einem WebAssembly-Modul kompilieren und als Teil der Binärdatei bereitstellen oder können nicht verwendet werden. Auch für dieses Feature von WebAssembly besteht bereits ein Vorschlag zur Umsetzung im Github-Repository [22].

3 Anwendung und Verbreitung

Nachdem nun ein größeres Verständnis zur Architektur von WebAssembly vorliegt, wird in diesem Abschnitt auf die Nutzung eingegangen.

3.1 Verwendung im Browser

Anhand eines Beispiels wird gezeigt, wie ein WebAssembly-Modul im Browser verwendet werden kann. Aus Gründen der Übersichtlichkeit und Verständnis wurde sich für eine einfache Subtraktionsfunktion in C, zur Erläuterung der Vorgehensweise entschieden.

```
int sub(int x, int y) {
  return x - y;
}
```

Listing 3.1: sub.c

Diese C-Funktion muss nun zu einem .wasm Modul kompiliert werden. Zum Kompilieren genügt für das Beispiel bereits ein Onlinetool wie WasmFiddle [32]. Da es sich bei dem Output jedoch um ein Binärformat handelt, wird dieses an der Stelle nicht gezeigt. WebAssembly bietet zur Veranschaulichung das WebAssembly Text-Format (WAT) an. Dieses Format erstellt, wie der Name schon sagt, eine textuelle Erscheinung des Moduls, welche den Aufbau der Binärdatei zeigt.

```
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "sub" (func $sub))
 (func $sub (; 0 ;) (param $0 i32) (param $1 i32) (result i32)
  (i32.sub
   (get_local $0)
   (get_local $1)
  )
 )
)
```

Listing 3.2: sub.wat

Der Großteil ist an diesem Punkt schon erledigt. Der C-Code wurde erfolgreich zu einem WebAssembly-Modul kompiliert und muss nur noch innerhalb von JavaScript aufgerufen werden.

Im Script-Tag der example.html wird sub.wasm durch den Aufruf von WebAssembly.instantiateStreaming vom Server geladen. Wenn diese Datei ankommt, wird die Instanz des Moduls und ihre exportierte Methode sub mit den Parametern aufgerufen. Das Ergebnis der Subtraktion wird nun in die Konsole geschrieben.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
  </head>
  <body>
    <script>
    WebAssembly.instantiateStreaming(fetch('sub.wasm'))
               .then(prog => {
                   console.log(prog.instance.exports.sub(5, 2));
               });
    </script>
  </body>
</html>
```

Listing 3.3: example.html

3.2 Verbreitung

Im Jahr 2019 wurden 947.704 der Alexa Top 1 Millionen Webseiten besucht und auf die Verwendung von WebAssembly-Modulen untersucht.

Insgesamt wurden 1639 Internet seiten erfasst, die 1950 verschiedene Wasm-Module geladen haben. Hiervon waren 150 Module einzigartig. Einige Module sind beliebt genug, um auf verschiedenen Seiten aktiv zu sein. Ein Modul ist sogar insgesamt 346-mal erschienen. Andererseits gab es 87 Module, die nur auf einer Seite vorgekommen sind. Dies deutet darauf hin, dass die meisten speziell für eine Website entwickelt wurden. Abbildung 3.1 zeigt, dass weniger beliebte Seiten dazu tendierten eher Wasm-Module zu verwenden [31].

Diese Zahlen sind sehr ernüchternd und weit von einer allgemeinen Verbreitung entfernt. Jedoch ist zu beachten, dass dieser Ausschnitt nur einen kleinen Teil des World Wide Web widerspiegelt. Viel wichtiger sind an dieser Stelle, die Projekte, die WebAssembly bereits voll nutzen. Zu diesen zählen unter anderem AutoCAD, QT und Google Earth [33]. Denn diese guten Beispiele zeigen Entwicklern, dass WebAssembly eine Daseinsberechtigung hat und bei performanceabhängigen und rechenintensiven Aufgaben sinnvoll ist.

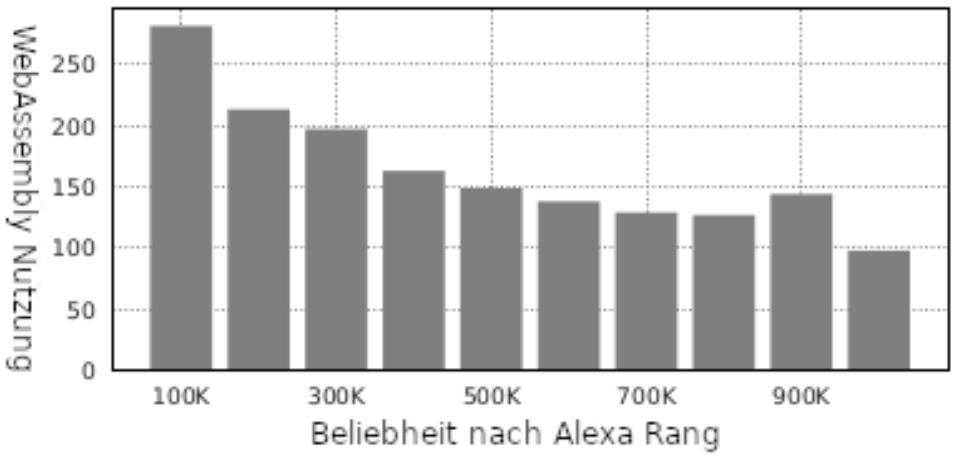

Abbildung 3.1: Verteilung von Alexa Websiten die WebAssembly nutzen in Hunderttausender schritten [31].

3.3 Nutzungszwecke

Die 150 einzigartigen Module aus Abschnitt 3.2 wurden weitergehend auf ihre Verwendung untersucht. Es kam heraus, dass 48 dieser Wasm Module, auf über 50 Prozent der Seiten, für das Schürfen von Kryptowährungen genutzt wurden [31]. Mit WebAssemly können diese Algorithmen natürlich performanter als mit JavaScript agieren. Jedoch sind Module dieser Untersuchung nicht nur für bösartige Zwecke verwendet worden. Unter anderem wurden 25 Bibliotheksmodule auf 38,8 Prozent der untersuchten Seiten verwendet [31].

Die unter Abschnitt 3.2 genannten Projekte, sind alles rechenintensive Anwendungen und verfügen über eine C/C++ Codebasis. Es sind also alles perfekte Beispiele, um aufzuzeigen zu welchem Zweck WebAssembly entwickelt wurde. Sie nutzen WebAssembly um die Performance der nativen Applikation ins Web zu bringen. Gerade bei Google Earth und AutoCAD ist dies von immenser Bedeutung, denn beides sind grafische 3-D-Anwendungen, die sehr rechenintensiv sind. Ein weiterer Vorteil ist natürlich, dass beide Anwendungen einfach die bestehende Codebasis zu WebAssembly kompilieren können.

WebAssembly wird außerdem für die Bereitstellung von Spielen, Spiele Engines oder Konsolen-Emulatoren im Web genutzt [33].

4 Ausblick

WebAssembly ist bereits offizieller Webstandard, wird von allen Browsern unterstützt und verzeichnet bereits einige große Projekte als seine Nutzer. Diese Errungenschaften scheinen auf dem ersten Blick eine perfekte Zukunft für die Technologie zu ebnen.

WebAssembly wurde entwickelt, um mit JavaScript zu harmonieren, es benötigt immer JavaScript-Code zur Unterstützung. Bei der Ausführung von rechenintensiven Aufgaben kann WebAssembly glänzen aber alles Weitere ist dann eben immer noch Aufgabe von JavaScript. Dies stellt einen Web-Entwickler mit bestehender Code-Basis vor die Fage, ob er nicht versucht, seinen JavaScript-Algorithmus zu optimieren. Ehe er die rechenintensiven Stellen erneut aber diesmal in C schreibt und zu WebAssembly kompiliert. Bei zukünftigen Projekten erwarte ich jedoch einen Anstieg der Nutzung von WebAssembly, weil die performancekritischen Algorithmen hier von Anfang an in C/C++/Rust geschrieben werden können.

Durch die zukünftige Erweiterung des GC vermute ich außerdem einen Anstieg der nativen Applikationen, die auf das Web portiert werden. Momentan sind diese, wie in Abschnitt 3.3 erwähnt, hauptsächlich mit C/C++ entwickelt. Durch die Erweiterung des GC werden weitere moderne Sprachen WebAssembly als stabiles Kompilierziel verwenden können. Hierdurch können wiederum mehr Anwendungen als Web App bereitgestellt werden.

Außerdem ist die Bereitstellung von Spielen oder Emulatoren im Web durch WebAssembly zu verfolgen. Vermutlich werden in diesem Bereich noch viele Anwendungen dazu kommen.

Alles in allem erwartet WebAssembly wirklich eine gute Zukunft, auch wenn es auf noch keiner breiten Masse von Webseiten verwendet wird [31].

5 Diskussion

Diese Arbeit zeigt die stärken und schwächen von WebAssembly. Überzeugen konnte WebAssembly durch die sehr gute Spezifikation und die Performanz. Die Konzepte sind gut durchdacht, wenn auch noch nicht vollständig. Durch die Ernennung zum Webstandard hat WebAssembly zwar alle Voraussetzungen um viel genutzt zu werden, jedoch beschränkt sich die Zahl der Anwender noch. Hier ist zu beobachten, wie sich die Nutzungszahlen verändern werden. Eine Einarbeitung in die Thematik fällt leicht, da es viele gut geschriebene Artikel zu WebAssembly gibt. Abschließend ist zu sagen, dass WebAssembly das Potenzial besitzt, um eine tragende Rolle im Web der Zukunft zu spielen.

Literaturverzeichnis

[1] WebAssembly Webiste. https://webassembly.org/, 17. Mai 2020.

[2] WebAssembly FAQ. https://webassembly.org/docs/faq/, 17. Mai 2020.

[3] WebAssembly Sepzifikation. *Instructions*, https://webassembly.github.io/spec/core/syntax/instructions.html, 19. Juni 2020.

[4] WebAssembly Spezifikation. *Types*, https://webassembly.github.io/spec/core/syntax/types.html, 19. Juni 2020.

[5] WebAssembly Spezifikation. *Modules*, https://webassembly.github.io/spec/core/syntax/modules.html, 19. Juni 2020.

[6] Sparsh Mittal. *A Survey of Techniques for designing and managing CPU register file*, https://zenodo.org/record/1229149 2016.

[7] Lin Clark. Mozilla Hacks. *Creating and working with WebAssembly modules*, https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/, 28. Februar 2017.

[8] Lin Clark. Mozilla Hacks. *WebAssembly table imports... what are they?*, https://hacks.mozilla.org/2017/07/webassembly-table-imports-what-are-they/, 19. Juli 2017.

[9] Conrad Watt. *Mechanising and Verifying the WebAssembly Specification*, https://www.cl.cam.ac.uk/~caw77/papers/mechanising-and-verifying-the-webassembly-specification.pdf, 2018.

[10] Micha Reiser und Luc Bläser. *Accelerate JavaScript Applications by Cross-Compiling to WebAssembly*, https://dl.acm.org/doi/pdf/10.1145/3141871.3141873, 2017.

[11] Abhinav Jangda, Bobby Powers, Emery D. Berger und Arjun Guha. emphNot So Fast: Analyzing the Performance of WebAssembly vs. Native Code, https://www.usenix.org/system/files/atc19-jangda.pdf, 2019.

[12] Andreas Haas, Andreas Rossberg, Derek l. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai und JF Bastien. *Bringing the Web up to Speed with WebAssembly*, https://people.mpi-sws.org/~rossberg/papers/Haas,Rossberg,Schuff,Titzer,Gohman,Wagner,Zakai,Bastien,Holman-BringingtheWebuptoSpeedwithWebAssembly.pdf, 2017.

[13] Conrad Watt, Andreas Rossberg und Jean Pichon-Pharabod. *Weakening WebAssembly*, https://www.cl.cam.ac.uk/~jp622/weakening_webassembly.pdf, 2019.

[14] Marius Musch, Christian Wresnegger, Martin Johns und Konrad Rieck. *New Kid on the Web: A Study on the Prevelance of WebAssembly in the Wild*, https://www.sec.cs.tu-bs.de/pubs/2019a-dimva.pdf, 2019.

[15] Christoph Alladoum. *Understanding WebAssembly*, https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/understanding-web-assembly.pdf, 2018.

[16] Mozilla. *ARE WE FAST YET?*, https://arewefastyet.com, 2017

[17] Alon Zakai. Mozilla Hacks. *Why WebAssembly is Faster Than asm.js*, https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/, 2017.

[18] W3 Website. *World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation*, https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en, 2019.

[19] W3 Lists Website. *WebAssembly consensus and end of Browser Preview*, https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html, 2017.

[20] Alon Zakai und Robert Nyman. Mozilla Hacks. *Gap between asm.js and native performance gets even narrower with float32 optimizations*, https://hacks.mozilla.org/2013/12/gap-between-asm-js-and-native-performance-gets-even-narrower-with-float32-opti 2013.

[21] Github. *Going public launch bug*, https://github.com/WebAssembly/design/issues/150, 2015.

[22] Github. *WebAssembly proposals*, https://github.com/WebAssembly/proposals, 19. Juni 2020.

[23] Dr. Axel Rauschmayer. 2ality - JavaScript and more. *WebAssembly: a binary format for the web*, https://2ality.com/2015/06/web-assembly.html, 18. Juni 2015.

[24] Limin Zhu. Microsoft Edge Developer Blog, *Previewing WebAssembly experiments in Microsoft Edge*, https://blogs.windows.com/msedgedev/2016/03/15/previewing-webassembly-experiments/, 15. März 2016.

[25] Luke Wagner. Mozilla Hacks. *A WebAssembly Milestone: Experimental Support in Multiple Browsers*, https://hacks.mozilla.org/2016/03/a-webassembly-milestone/, 14. März 2016.

[26] Seth Thompson. V8 Developer Blog. *Experimental support for WebAssembly in V8*, https://v8.dev/blog/webassembly-experimental, 15. März 2016.

[27] Lin Clark. Mozilla Hacks. *What makes WebAssembly fast?*, https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/, 28. Februar 2017.

[28] Lin Clark. Mozilla Hacks. *WebAssembly's post-MVP future: A cartoon skill tree*, https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/, 22. Oktober 2018.

[29] Github. *wee_alloc*, https://github.com/rustwasm/wee_alloc, 16. Juni 2020.

[30] Github. *Awesome WebAssembly Languages*, https://github.com/appcypher/awesome-wasm-langs#java, 17. Juni 2020.

[31] Marius Musch, Christian Wressnegger, Martin Johns und Konrad Rieck. *New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild*, https://www.sec.cs.tu-bs.de/pubs/2019a-dimva.pdf, 2019.

[32] WasmFiddle. https://wasdk.github.io/WasmFiddle/ 18. Juni 2020.

[33] Made with WebAssembly. *All Projects*, https://madewithwebassembly.com/all-projects/, 18. Juni 2020.

[34] GitHub. *WebAssembly Logo*, https://github.com/carlosbaraza/web-assembly-logo, 2017.

[35] V8 Developer Blog. *Emscripten and the LLVM WebAssembly backend*, https://v8.dev/blog/emscripten-llvm-wasm, 01. Juli 2017.