Riku Rundelin

Mitä uutta WebAssembly tuo web-ympäristöön?

Tietotekniikan kandidaatintutkielma

16. heinäkuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Riku Rundelin

Yhteystiedot: riku.h.rundelin@student.jyu.fi

Ohjaaja: Antti-Juhani Kaijanaho

Työn nimi: Mitä uutta WebAssembly tuo web-ympäristöön?

Title in English: What new does WebAssembly bring to the web environment?

Työ: Kandidaatintutkielma

Sivumäärä: 21+0

Tiivistelmä: JavaScript on pitkään ollut ainoa web-selainten tukema ohjelmointikieli. Suorituskykykriittisissä sovelluksissa sen epätasainen suorituskyky jää kuitenkin usein puuttelliseksi. Tässä tutkielmassa tutustutaan WebAssemblyyn, joka on hyväksi matalan tason kohdekieleksi suunniteltu uusi binäärikoodiformaatti. Tutkielman tavoitteena on selvittää, mitä uutta WebAssembly tuo suorituskykyä vaativien web-sovellusten näkökulmasta. Tutkielmassa esitellään myös lyhyesti WebAssemblyä edeltäneitä teknologioita. Lähdekirjallisuuden perusteella WebAssembly näyttäisi tuovan merkittäviä suorituskykyparannuksia JavaScriptiin verrattuna. Sille löytyy lukuisia käyttökohteita web-sovelluksista ja sitä voidaan hyödyntää monin eri tavoin.

Avainsanat: WebAssembly, matalan tason kieli, kohdekieli

Abstract: JavaScript has been the only programming language supported by web browsers for a long time. In performance critical applications its irregular performance is often insufficent. In this thesis we take a look at WebAssembly, which is a binary code format designed to be a good low-level compilation target. The goal of this thesis is to find out what new does WebAssembly bring to the web environment from the point of view of performance demanding web applications. The thesis also briefly introduces technologies preceeding WebAssembly. According to the source literature WebAssembly seems to bring notable performance improvements compared to JavaScript. It has several use cases in web applications and it can be utilized in many different ways.

Keywords: WebAssembly, low-level language, compilation target

Sisältö

1 Johdanto

Web on nykypäivänä yksi tärkeimmistä alustoista ohjelmistokehitykselle. Web-ympäristö on jatkuvassa muutoksessa. Yksi asia on kuitenkin pysynyt muuttumattomana jo 1990-luvun loppupuolelta. JavaScript on ainoa ohjelmointikieli, jota kaikki selaimet tukevat ilman ylimääräisiä liitännäisiä. Muiden teknologioiden, kuten Javan tai Flashin, käyttäminen vaatii liitännäisiä, mikä tuo omat ongelmansa. Liitännäiset eivät välttämättä asetu täydellisesti ympäröivään HTML-koodiin, eivätkä ne toimi kaikilla alustoilla (Zakai 2011). Näiden teknologioiden käyttö onkin laskussa niitä piinaavien turvallisuus- ja suorituskykyongelmien takia (Haas ym. 2017). Haasin ym. (2017) mukaan niiden käyttö onkin ollut laskussa, sillä niitä piinaavat suorituskyky- ja turvallisuusongelmat.

JavaScript suunniteltiin alun perin korkean tason dynaamiseksi skriptikieleksi (Watt 2018). Siitä on kuitenkin tullut niin merkittävä ohjelmointikieli web-sovellusten kehityksessä, että se on päätynyt kohdekieleksi (*compilation target*) muille kielille. Esimerkiksi Emscriptenillä voi kääntää C- ja C++-koodia web-selaimissa toimivaksi JavaScript-koodiksi (Zakai 2011).

Vaikka JavaScript-moottorit kehittyvät ja nopeutuvat koko ajan, ei JavaScriptin suorituskyky ole aina riittävää. Suorituskyvyn puute vaikuttaa etenkin moniin laskennallisesti raskaisiin sovelluksiin, kuten kryptografiaan, simulaatioihin ja peleihin (Reiser ja Bläser 2017). Sen lisäksi, että JavaScriptiksi käännetty koodi on yleensä paljon hitaampaa kuin vastaava natiivikoodi, on JavaScriptin tehokas kääntäminen tunnetusti hankalaa (Jangda ym. 2019).

Tässä tutkielmassa tutustutaan WebAssemblyyn, jonka on tarkoitus ratkaista nämä JavaScriptiä piinaavat ongelmat. Se on uusi kieli- ja alustarippumaton binäärikoodiformaatti, jonka on tarkoitus tuoda natiivikoodia vastaava suorituskyky web-sovelluksiin. WebAssemblyä kehittää W3C-yhteisöryhmä, ja sitä kehitetään avoimena standardina (Rossberg 2019, 1.1. Introduction). Sen kehittämiseen ovat osallistuneet kaikki merkittävät selainvalmistajat (Rossberg 2016) ja kaikki merkittävät selaimet tukevat sitä. WebAssembly on suunniteltu täyttämään vaatimukset, jotka matalan tason kohdekielen tulisi täyttää. Tutkielmassa perehdytään erityisesti siihen, kuinka WebAssemblyä voidaan hyödyntää laskentaintensiivisissä, suorituskykykriittisissä web-sovelluksissa.

Tutkielma rakentuu seuraavasti. Luvussa 2 tutustutaan WebAssemblyyn ja sen suunnitteluperiaatteisiin sekä teknisiin periaatteisiin. Luvussa 3 tutustutaan WebAssemblyn edeltäjiin ja vertaillaan sen suorituskykyä. Luvussa 4 tutustutaan, mihin WebAssemblyä on tähän mennessä käytetty, ja pohditaan, mihin sitä voisi tulevaisuudessa käyttää. Luvussa 5 tiivistetään tutkielman olennaisimmat asiat, sekä esitetään niistä tehdyt johtopäätökset.

2 WebAssembly

WebAssembly, lyhenne Wasm, on avoin standardi, joka määrittelee binäärikoodiformaatin. Sitä kehittää W3C-yhteisöryhmä, johon kuuluu kaikkien suurien selainvalmistajien työntekijöitä. Se julkistettiin 17. heinäkuuta 2015 (Bastien 2015). Pienin toimiva tuote -statuksen se saavutti maaliskuussa 2017 (webassembly.org 2019b, Roadmap) ja tällä hetkellä kaikki merkittävimmät selaimet tukevat sitä suurimmilta osin (MDN web docs 2019, Browser compatibility). Tässä luvussa esitellään WebAssemblyn tärkeimmät suunnitteluperiaatteet. Niiden lisäksi tutustutaan WebAssemblyn keskeisimpiin teknisiin piirteisiin.

2.1 Suunnitteluperiaatteet

WebAssembly on suunniteltu mahdollistamaan suorituskykyä vaativien sovellusten ajaminen web-selaimissa (Rossberg 2019, 1.1. Introduction). Sen on tarkoitus olla hyvä matalan tason kohdekieli muille ohjelmointikielille. Haasin ym. (2017) mukaan ominaisuusvaatimukset hyvälle matalan tason kohdekielelle ovat nopeus, turvallisuus, siirrettävyys sekä tehokas ja turvallinen esitysmuoto.

Hyvä suorituskyky on oleellista raskaiden sovellusten kannalta, ja nopeus onkin tärkeä WebAssemblyn tavoite. Ohjelmointikielen suorituskyvystä puhuttaessa vertailukohtana käytetään yleensä natiivikoodia. WebAssemblyn tavoitteena on saavuttaa lähes natiivikoodia vastaava suorituskyky (Rossberg 2019, 1.1.1. Design Goals). Tällaiseen suorituskykyyn on mahdollisuus, sillä Herreran ym. (2018) mukaan WebAssembly on rakennettu abstraktioksi modernien laitteistoarkkitehtuurien päälle, mikä mahdollistaa kielen optimoitavuuden ja sen myötä suorituskykyhyödyt.

Webissä oleva ohjelmakoodi on lähtökohtaisesti epäluotettavaa, joten turvallisuus on erittäin tärkeä asia verkkosivulla olevaa ohjelmakoodia suoritettaessa. WebAssemblyssä datan korruptoituminen ja turvallisuusrikkomukset pyritään estämään validoimalla koodi ja ajamalla se muistiturvallisessa hiekkalaatikkoympäristössä (Rossberg 2019, 1.1.1. Design Goals).

Siirrettävyys on tärkeä WebAssemblyn suunnitteluperiaate. Haasin ym. (2017) mukaan we-

biin tarkoitetun koodin tulee olla laitteisto- ja alustariippumatonta, jotta sovelluksen käyttäytyminen on samanlaista riippumatta käytetystä selaimesta tai laitteistosta. Näin samaa versiota sovelluksesta voisi käyttää ilman asennuksia millä tahansa laitteella, jossa on ajantasainen web-selain. WebAssemblyn käyttömahdollisuudet eivät rajoitu kuitenkaan pelkästään webiin, sillä sen suunnittelu ei ole riippuvainen webistä tai JavaScript-ympäristöstä (Haas ym. 2017).

Ohjelmakoodin esitysmuodolla voi olla merkittäviä vaikutuksia web-sovelluksessa erityisesti ohjelman latausajan kannalta. Mitä pienempään tilaan ohjelman voi pakata, sitä vähemmän aikaa ja kaistanleveyttä sen välittämiseen asiakkaalle menee. Haasin ym. (2017) mukaan web-sovellusten koodi välitetään yleensä palvelimelta JavaScript-lähdekoodina. WebAssembly-koodia on kuitenkin nopeampi välittää, sillä se on kompaktimpaa kuin tyypilliset tekstiformaatit ja natiivikoodiformaatit (Rossberg 2019, 1.1.1. Design Goals). Näin ollen sovellusta pääsee käyttämään nopeammin.

2.2 Keskeiset tekniset piirteet

WebAssembly on staattisesti tyypitetty kieli. Sen tyyppijärjestelmä on yksinkertainen, sillä siinä on vain neljä tyyppiä. Tyypit ovat kokonaisluvut ja IEEE 754 -liukuluvut 32- ja 64-bittisinä (Haas ym. 2017). Tyypeille rakennetuissa operaattoreissa on pyritty optimoimaan suorituskykyä, sillä Rossbergin (2016) mukaan ne mukailevat modernien suoritinten konekielisiä käskyjä, minkä ansiosta niiden suorituskyky ja kuvautuminen konekielisiksi käskyiksi on ennakoitavaa.

WebAssembly-ohjelmakoodi paketoidaan moduuleihin. Haasin ym. (2017) mukaan moduuli sisältää funktioiden määrittelyt, globaalit muuttujat, taulut ja muistin. Määritelmiä voi sekä tuoda (import), että viedä (export). Moduuli on ohjelman staattinen esitys, josta saadaan dynaaminen luomalla siitä instanssi. Moduulista voidaan luoda instanssi ympäröivässä koodissa, jolloin voidaan käyttää moduulin viemiä funktioita ja muuttujia.

Moduulin sisällä ohjelmakoodi järjestetään funktioihin. Funktiot ottavat tietyn tyyppisiä arvoja parametrinä ja palauttavat funktiotyyppinsä mukaisia arvoja. Funktiot voivat määritellä muutettavia lokaaleja muuttujia, joita myös funktion saamat parametrit ovat. Haasin ym.

(2017) mukaan WebAssemblyssä funktiot eivät ole ”ensimmäisen luokan kansalaisia”, eli niitä ei voi esimerkiksi viedä parametrinä funktiolle tai palauttaa funktiosta. Funktioita ei voi määritellä funktioiden sisällä, mutta ne voivat kuitenkin kutsua toisiaan ja rekursiivisesti itseään.

Taulu (Table) on WebAssembly tapa tarjota C- ja C++-kielissä laajasti käytettyjä funktio-osoittimia vastaavaa toiminnallisuutta. Rossberg (2019, 1.2.1. Concepts) kertoo taulun oleva taulukko, joka sisältää viitteitä moduulin funktioihin. Taulu mahdollistaa funktioiden epäsuoran kutsumisen. Näin kutsuttaessa funktion tyyppi tarkastetaan dynaamisesti ennen hyppyä sen ohjelmakoodiin. Mikäli tarkastus onnistuu, on hyppy funktioon turvallinen. Muuten, jos esimerkiksi tyyppi on väärä tai pyydetty indeksi on taulukon ulkopuolella, tuotetaan ansa.

Ansa (trap) on WebAssemblyssä eräänlainen poikkeus. Haasin ym. (2017) mukaan tietyissä tilanteissa ohjelma tuottaa ansan, jolloin ohjelman suoritus keskeytyy välittömästi. Esimerkiksi jakolasku ja muunnos liukuluvusta kokonaisluvuksi voivat tuottaa epäkelpoja tuloksia, jolloin tuotetaan ansa. WebAssembly-koodi ei voi itse käsitellä ansaan joutumista, vaan virheenkäsittely on tehtävä koodissa, johon WebAssembly-koodi on upotettu.

Haasin ym. (2017) mukaan moduulin määrittelemä muisti on lineaarista muistia. Se on muutettava taulukko peräkkäisiä tavuja. Sille alustetaan jokin koko, mutta se voi tarvittaessa kasvaa dynaamisesti. Muistin koon yksikkönä on 64 KiB sivu (page). Tämä koko on valittu siirrettävyyssyistä, sillä se on pienin yhteinen jaettava modernissa laitteistossa esiintyvistä sivun koista.

Haas ym. (2017) kertovat moduulin muistin olevan erillään ohjelmakoodialueesta, suorituspinosta ja suoritusmoottorin tietorakenteista. Näin virheellinen tai vihamielinen ohjelmakoodi ei pysty korruptoimaan suoritusympäristöä tai aiheuttamaan muuta määrittelemätöntä käytöstä. Pahimmassa tapauksessa haavoittuva ohjelma voi sekoittaa vain oman muistinsa. Mikäli ohjelma yrittää päästä käsiksi lineaarisen muistinsa ulkopuoliseen alueeseen, tuotetaan ansa.

Koska WebAsssembly ei pääse käsiksi kohdealustan tärkeisiin rajapintoihin, kuten webselaimen DOM- tai WebGL-rajapintoihin, on se yleensä upotettuna johonkin isäntäympäristöön, kuten JavaScriptiin. Haasin ym. (2017) mukaan upottaja (embedder) toimii linkkinä

isäntäympäristön ja WebAssembly-moduulin välillä. Se hoitaa moduuli-instanssien luomisen isäntäympäristössä ja määrittelee, kuinka moduulin tuodaan (import) tarvittavat määritelmät ja kuinka sen tarjoamia määritelmiä käytetään.

3 Edeltäjät ja suorituskyky

WebAssembly ei ole ensimmäinen yritys tuoda matalan tason koodia web-selaimeen. Sitä ovat edeltäneet erilaiset selainliitännäiset sekä teknologiat kuten asm.js. Mitkä ovat olleet näiden teknologioiden puutteita? Minkälainen WebAssemblyn suorituskyky on muihin teknologioihin verrattuna? Tässä luvussa tutustutaan WebAssemblyä edeltäneisiin matalan tason koodin web-teknologioihin ja vertaillaan sen suorituskykyä JavaScriptiin ja natiiviin C:hen.

3.1 Aikaisemmat matalan tason koodin web-teknologiat

Ajatus matalan tason koodista web-ympäristössä on syntynyt jo paljon ennen WebAssemblyä. Eräs aikaisimmista tähän ajatukseen perustuvista teknologioista on Microsoftin ActiveX. Se mahdollisti x86-binäärien ajamisen selaimessa ja sen turvallisuus perustui kokonaan koodin allekirjoitusvarmenteisiin (Haas ym. 2017). Tämä ei ole kovin hyvä asia turvallisuuden kannalta, sillä allekirjoitusvarmenne ei takaa, että koodissa ei olisi esimerkiksi turvallisuuden kannalta kriittisiä ohjelmointivirheitä tai että ohjelmakoodi olisi luotettavaa.

Googlen Native Client pyrkii ratkaisemaan turvallisuuteen liittyvän ongelman hieman järkevämmin. Native Client mahdollistaa epäluotettavan natiivikoodin ajamisen web-selaimessa hiekkalaatikkoympäristössä. Haas ym. (2017) mukaan se on ensimmäinen teknologia, joka mahdollistaa konekielisen koodin ajamisen selaimessa hiekkalaatikkoympäristössä lähes natiivikoodia vastaavalla nopeudella. Se tarjoaa myös monia suorituskykyyn liittyviä, web-ohjelmointiympäristöistä usein puuttuvia, ominaisuuksia, kuten säikeistyksen ja assemblykoodin kirjoittamisen käsin (Yee ym. 2009).

Native Clientin ongelmana on sen siirrettävyys, sillä se on pohjimmiltaan tietyn arkkitehtuurin konekielen osajoukko (Haas ym. 2017). Tästä syystä Native Client -sovelluksessa kehittäjällä on iso työ testata ja julkaista kaikilla tuetuilla laitealustoilla (Donovan ym. 2010). Ratkaisuna NaCl:n siirrettävyysongelmaan kehitettiin Portable Native Client. Donovan ym. (2010) mukaan PNaCl on työkalu, jonka avulla voidaan kehittää NaCl-sovelluksia kohdealustaneutraalilla tavalla. Haasin ym. (2017) mukaan siinä olevat alustariippuvaiset yksityis-

kohdat ja sen saatavuuden rajoittuminen Chrome-selaimeen rajoittavat sen siirrettävyyttä merkittävästi.

Myös JavaScriptistä on pyritty luomaan matalan tason kohdekieli suorituskykyä vaativien sovellusten tarpeiden täyttämiseksi. Asm.js on JavaScriptin osajoukko, josta on karsittu pois korkean tason kielen ominaisuudet kuten dynaamisuus. JavaScriptin osajoukkona asm.js toimii selainten olemassa olevissa JavaScript moottoreissa ja pystyy hyödyntämään niiden kehittyneitä JIT-kääntäjiä (Haas ym. 2017).

Asm.js mahdollistaa paljon tavallista JavaScriptiä paremman suorituskyvyn. Khanin ym. (2015) mukaan asm.js-ohjelmakoodi voidaan kääntää suoraan assembly-koodiksi, mikä mahdollistaa lähes natiivisovellusta vastaavan suorituskyvyn selaimessa. JavaScriptin osajoukkona oleminen tuo kuitenkin omat haittapuolensa. Esimerkiksi ominaisuuksien laajentaminen on hankalaa, sillä Haasin ym. (2017) mukaan se vaatii, että ensin ominaisuudet on tuotava JavaScriptiin.

3.2 Suorituskyky

WebAssemblyn tärkeä suunnitteluperiaate on nopeus, mutta miten se näkyy käytännössä? Tässä alaluvussa vertaillaan WebAssemblyn suorituskykyä natiivikoodiin ja JavaScriptiin. Tässä luvussa esitetyt suorituskykytulokset perustuvat Herreran ym. (2018) ja Jangdan ym. (2019) tutkimukseen.

Herrera ym. (2018) vertaavat WebAssemblyn numeerisen laskennan suorituskykyä sekä JavaScriptiin, että natiiviin C:hen erilaisilla alustoilla. Testaamiseen on käytetty Ostrich Benchmark Suitea, joka on työkalu ohjelmointikielten numeerisen laskennan suorituskyvyn testaamiseen (Khan ym. 2015). Jangda ym. (2019) puolestaan vertaavat WebAssemblyn ja natiivikoodin suorituskykyä laaja-alaisesti käyttäen SPEC CPU benchmark suitea. SPEC CPU on testausohjelmisto, jolla testataan suorituskykyä laskennallisesti intensiivisillä työkuormituksilla (*SPEC's Benchmarks*).

Web-selaimissa tärkein vertailukohta WebAssemblylle on JavaScript, koska JavaScript on sen lisäksi ainoa selaimissa tuettu ohjelmointikieli. Herrera ym. (2018) vertailevat WebAs-

semblyn ja JavaScriptin suorituskykyä useilla selaimilla erilaisilla mobiililaitteilla ja tietokoneilla. Suorituskykytesteistä selvisi, että WebAssembly on keskimäärin 1,5-2 kertaa nopeampaa, kuin JavaScript samassa selaimessa. Suurimmat erot syntyivät Windows- ja Mac-työasemilla käyttöjärjestelmien omilla selaimilla, joissa ero nousi jopa 2,5-kertaiseksi. Herrera ym. (2018) epäilevät Edge-selaimella syntyneen eron johtuvan ainakin osittain Edgen JavaScript-moottorin hitaudesta.

Miten WebAssembly sitten pärjää natiiviin C:hen verrattuna? Jangdan ym. (2019) mukaan WebAssemblyn suorituskyky on keskimäärin noin 67 % natiivi C:n suorituskyvystä Firefox-selaimella, Chrome-selaimella suorituskyky jää 52 %:iin. Herrera ym. (2018) tuloksissa WebAssembly pääsi hieman lähemmäksi natiivikoodin suorituskykyä, sillä Firefox-selaimella päästiin keskimäärin noin 85%:iin ja Chrome-selaimella reiluun 70 %:iin siitä. Käyttöjärjestelmäkohtaisista selaimista Edgen suorituskyky oli hiukan Chromea parempi, mutta Firefoxia huonompi. Safari puolestaan jäi hitaimmaksi jääden 58%:iin natiivikoodin suorituskyvystä.

Myös asm.js on tärkeä vertailukohta WebAssemblylle sen edeltäjänä. WebAssemblyn suorituskyky on korkeampaa kuin asm.js:n. Niiden ero ei ole valtava, mutta silti huomattava. Haasin ym. (2017) mukaan WebAssembly on keskimäärin 33,7 % suorituskykyisempää kuin asm.js heidän suorittamissaan PolyBenchC-suorituskykytesteissä. Jangdan ym. (2019) tutkimustulokset näyttävät vahvistavan WebAssemblyn ja asm.js:n välisen suorituskykyeron olevan noin 30 % WebAssemblyn hyväksi. He havaitsivat kuitenkin merkittäviä eroja selainten välillä, sillä Firefoxilla WebAssembly oli keskimäärin jopa 68 % nopeampaa, kun taas Chromella ero jäi vain 10 %:tiin.

WebAssembly koodin kompakti koko ja validoinnin nopeus ovat myös merkityksellisiä etuja JavaScriptiin verrattuna. Haasin ym. (2017) mukaan WebAssembly ohjelman validointi vie vain 3 % vastaavan asm.js ohjelman validoinnin vaatimasta ajasta. Tämä nopeuttanee huomattavasti ohjelman suorittamisen aloittamista. Lisäksi WebAssembly ohjelmakoodi vie keskimäärin 37,5 % vähemmän tilaa kuin vastaava asm.js-ohjelmakoodia ja 14,7 % vähemmän kuin vastaava x86-64 natiivikoodi.

4 WebAssemblyn käyttökohteet

WebAssembly on mielenkiintoinen uusi teknologia. Se mahdollistaa muidenkin ohjelmointikielten kuin JavaScriptin käyttämisen web-sovelluksien kehittämiseen. Se myös nostaa websovellusten suorituskyvyn uudelle tasolle. Onko WebAssemblyn tarkoituksena korvata JavaScript kokonaan? Miten WebAssemblyä voi käyttää web-sovelluksissa? Minkälaisiin sovelluksiin WebAssemblyä on tähän mennessä käytetty? Näihin kysymyksiin on tarkoitus vastata tässä luvussa.

4.1 WebAssembly kohdekielenä

Yksi kiinnostavimmista WebAssemblyn tuomista mahdollisuuksista lienee valinnanvara websovelluksen kehittämiseen käytettävän ohjelmointikielen suhteen. Kehittäjä voi kirjoittaa sovelluksen haluamallaan ohjelmointikielellä ja kääntää sen sitten WebAssemblyksi, jolloin sovellus toimii web-selaimessa lähes natiivitoteutusta vastaavalla suorituskyvyllä. Kuten jo aikaisemmin todettiin, WebAssembly on suunniteltu nimenomaan hyväksi kohdekieleksi webselaimeen muille ohjelmointikielille. Tiettyjä rajoitteita WebAssemblyn käytössä kohdekielenä kuitenkin on vielä tällä hetkellä.

Akinyemin (2019) ylläpitämä lista pitää kirjaa kielistä, jotka tällä hetkellä kääntyvät WebAssemblyksi tai joiden virtuaalikone on käännetty WebAssemblyksi. Listan mukaan tällä hetkellä 11 kieltä toimii tarpeeksi vakaasti WebAssemblyn kanssa tuotantokäyttöön. Näihin kieliin kuuluvat muun muassa C, C++, C# ja Rust. Haasin ym. (2017) mukaan WebAssemblyn onkin aluksi tarkoitus keskittyä tukemaan matalan tason koodia matalamman tason ohjelmointikielistä kuten C- ja C++-kielistä käännettynä.

Tuotantokäyttöön soveltuvien kielten lisäksi Akinyemin (2019) listan mukaan 18 ohjelmointikieltä toimii jollain tavalla WebAssemblyn kanssa ja 14 ohjelmointikielen kohdalla tuki on työn alla. Haasin ym. (2017) mukaan WebAssemblystä onkin tavoite tehdä hyvä kohdekieli myös korkean tason ohjelmointikielille. Tällä saralla tärkeä tavoite on mahdollistaa selaimiin sisäänrakennettujen roskien kerääjien (garbage collector) hyödyntäminen, sillä roskien keruun puute on yksi WebAssemblyn suurimmista puutteista JavaScriptiin verrattuna.

Myös JavaScript-kehittäjät hyötyvät WebAssemblystä. WebAssemblyn JavaScript API:n ansiosta JavaScript-kehittäjä voi hyödyntää esimerkiksi C++-kielellä kirjoitettua WebAssemblyksi käännettyjä kirjastoa lähes yhtä helposti kuin mitä tahansa muuta JavaScript-moduulia (webassembly.org 2019a). WebAssemblyn avulla web-kehittäjä voivat hyödyntää mittavaa määrää esimerkiksi C- ja C++-kielillä kirjoitettuja valmiita kirjastoja.

4.2 WebAssemblyn käyttötarkoitukset

WebAssemblylle löytyy valtava määrä käyttökohteita web-ympäristöstä. Sen tuoma suorituskyky mahdollistaa entistä monipuolisempien ja raskaampien sovellusten tarjoamisen webselaimen välityksellä. Web-sovellusten suurena etuna on, että sovellus toimii turvallisessa hiekkalaatikkoympäristössä millä tahansa alustalla ilman erillisiä asennuksia. Seuraavaksi on tarkoitus pohtia WebAssemblyn mahdollisia käyttökohteita ja käyttötapoja.

webassembly.org (2019c) listaa WebAssemblyn suunnittelussa huomioituja käyttökohteita. Listattuja käyttökohteita ovat muun muassa kuvan- ja videonkäsittely, pelit, musiikkisovellukset, VR- ja AR-sovellukset sekä CAD-sovellukset. Näitä ja suurta osaa muista käyttökohteista yhdistää tarve laskennalliselle suorituskyvylle, mikä onkin yksi WebAssemblyn valteista. Tällaiset sovellukset hyötyvät todennäköisesti myös WebAssemblyn tasaisemmasta suorituskyvystä. Esimerkiksi peleissä tasaunen suorituskyky on erittäin tärkeää, sillä yksin suorituskyvyn notkahdus voi pilata pelaajan immersion.

Matalan tason kielenä WebAssembly soveltuu hyvin myös erityistä turvallisuutta vaativiin sovelluksin, kuten kryptografisten algoritmien toteuttamiseen. Wattin ym. (2019) mukaan JavaScriptin monimutkaisuuden takia sillä on erittäin haastavaa toteuttaa kryptografiaa turvallisesti. WebAssembly sopii tähän tehtävään heidän mukaansa paremmin, sillä matalan tason ohjelmointikielenä WebAssemblyn käskyt ovat lähempänä laitteistoa, mikä antaa paremman varmuuden esimerkiksi ohjelman ajoitusominaisuuksista. Myös WebAssemblyn formaali määrittely ja vahva staattinen tyypitys edesauttavat turvallisten sovellusten toteuttamista.

WebAssemblyä voi hyödyntää web-sovelluksissa monella tapaa. webassembly.org (2019c) mainitsee kolme erilaista tapaa käyttää WebAssemblyä web-sovelluksessa. WebAssemblyllä

voi rakentaa koko sovelluksen, mutta tämä ei ole vielä ainakaan kovin käytännöllistä, sillä WebAssemblyn kautta ei pääse käsiksi selaimen rajapintoihin kuten DOM:iin. Kätevämpää onkin ehkä rakentaa sovelluksen logiikka WebAssemblyn avulla ja toteuttaa käyttöliittymä JavaScriptin ja HTML-koodin avulla. WebAssemblyä voi myös hyödyntää osana JavaScript-sovellusta, esimerkiksi toteuttamalla vain jonkin laskennallista suorituskykyä vaativan sovelluksen osan WebAssemblyllä.

4.3 Esimerkkejä WebAssemblyn käytöstä

WebAssemblyä on ehditty hyödyntää jo monissa tosielämän sovelluksissa. Sovelluksia yhdistää tarve saavuttaa korkea suorituskyky web-selaimessa. Seuraavaksi esitellään muutama näistä sovelluksista.

Chromatic on WebAssemblyllä toteutettu selaimessa toimiva bioinformatiikan työkalu, joka mahdollistaa syöpägenomien tarkastelun (Finney ja Meerzaman 2018). Finneyn ja Meerzamanin (2018) mukaan WebAssemblyllä toteutetun työkalun suuri etu on ohjelman käyttöönottamisen helppous. Ohjelmaa ei tarvitse ensin ladata ja asentaa jostain, vaan tarvitsee vain avata verkkosivu, jolla sovellus on. WebAssemblyn siirrettävyyden ja selainten hyvän tuen ansiosta sovellusta voi käyttää nopeasti lähes millä tahansa alustalla.

Göttl, Gagel ja Grubert (2018) ovat hyödyntäneet WebAssemblyä hieman erilaisessa sovelluksessa. He käyttävät WebAssemblyä vauhdittamaan selainpohjaista kuvanseurantaliukuhihnaa. Heidän mukaansa sovelluksen tarjoaminen web-selaimen välityksellä helpottaa kehittäjien työtä, koska siten ei tarvitse kirjoittaa jokaiselle alustalle omaa koodia. Aiemmin konenäköalgoritmeja on pidetty liian raskaina toteutettaviksi web-ympäristössä, mutta Göttlin ym. (2018) mukaan heidän WebAssemblyä hyödyntävä järjestelmänsä pystyy seuramaan kuvakohteita jopa 25-65 Hz kuvanpäivitysnopeudella laitteesta riippuen. Heidän tulostensa perusteella näyttäisi siltä, että WebAssemblyyn perustuvat lisätyn todellisuuden websovellukset ovat mahdollisia.

Myös peliteollisuudessa on huomattu WebAssemblyn tuomat edut. Suosittu pelimoottori Unity toi WebAssembly-tuen selaimeen tarkoitetuille WebGL-buildeille vuonna 2018 (Trivellato 2018). Unity-pelien tarjoaminen selaimessa on onnistunut aikaisemminkin asm.js:n

avulla. WebAssembly tuo kuitenkin huomattavia etuja etenkin pelien suorituskykyyn ja latausaikoihin. Trivellaton (2018) mukaan 3D-grafiikkaa sisältävä projekti vie WebAssemblyksi käännettynä noin 18 % vähemmän tilaa kuin vastaava asm.js käännös. Tämä säästää tilaa ja mahdollistaa esimerkiksi sovelluksen nopeamman latautumisen. WebAssembly on myös tehokkaampi muistin käytön suhteen ja mahdollistaa esimerkiksi Unityn kekomuistin dynaamisen kasvattamisen. Lisäksi WebAssemblyn suorituskyky on parempaa, etenkin selaimissa, joissa ei ole asm.js-spesifejä optimointeja.

5 Yhteenveto

Tässä tutkielmassa tutustuttiin WebAssemblyyn, joka on uusi matalan tason ohjelmointikieli, jonka on tarkoitus tuoda natiivikoodia vastaava suorituskyky web-selaimiin. Se on suunniteltu hyväksi kohdekieleksi muille ohjelmointikielille, ja mahdollistaa niiden käyttäminen web-sovellusten kehittämiseen. WebAssemblyn tärkeimmät suunnitteluperiaatteet ovat nopeus, turvallisuus, siirrettävyys sekä kompakti ja turvallinen esitysmuoto.

WebAssembly on suorituskyvyllisesti huomattava parannus asm.js:ään verrattuna ja etenkin JavaScriptiin verrattuna. Natiivikoodin suorituskykyyn WebAssemblyllä on kuitenkin vielä matkaa, vaikka parhaimmillaan ero jääkin hyvin pieneksi. Suorituskykymielessä WebAssemblyn etuja ovat myös sen nopea validointi sekä hyvin kompakti esitysmuoto. Suorituskyky tulee todennäköisesti vielä paranemaan selainten WebAssembly-toteutusten kehittyessä.

WebAssemblylle on lukuisia käyttökohteita web-sovelluksissa. Se mahdollistaa ohjelmointikielten, kuten C:n, C++ ja Rustin käyttämisen web-sovellusten kehittämiseen. Lisäksi se mahdollistaa näillä ohjelmointikielillä luotujen jo olemassa olevien kirjastojen käyttämisen. Siitä hyötyvä eniten laskennallista suorituskykyä vaativat sovellukset, kuten kuvan- ja videonkäsittely, pelit, musiikkisovellukset, VR- ja AR-sovellukset sekä CAD-sovellukset.

Tässä tutkielmassa keskityttiin käsittelemään WebAssemblyä web-kontekstissa. WebAssembly ei nimestään huolimatta ole riippuvainen web-selaimesta tai muista web-teknologioista. Mielenkiintoista voisikin olla selvittää, miten WebAssemblyä voidaan hyödyntää muualla kuin web-sovelluksissa.

Lähteet

Akinyemi, Steve. 2019. *A curated list of languages that compile directly to or have their VMs in WebAssembly.* https://github.com/appcypher/awesome-wasm-langs. Accessed: 2019-04-29.

Bastien, JF. 2015. *Going public launch bug.* https://github.com/WebAssembly/design/issues/150. Accessed: 2019-02-18.

Donovan, Alan, Robert Muth, Brad Chen ja David Sehr. 2010. *PNaCl: Portable native client executables.* Tekninen raportti. https://css.csail.mit.edu/6.858/2012/readings/pnacl.pdf.

Finney, Richard, ja Daoud Meerzaman. 2018. "Chromatic: WebAssembly-Based Cancer Genome Viewer". *Cancer informatics* 17. doi:10.1177/1176935118771972.

Göttl, Fabian, Philipp Gagel ja Jens Grubert. 2018. "Efficient pose tracking from natural features in standard web browsers". *arXiv preprint arXiv:1804.08424.* https://arxiv.org/pdf/1804.08424.pdf.

Haas, Andreas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai ja JF Bastien. 2017. "Bringing the Web Up to Speed with WebAssembly". Teoksessa *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation,* 185–200. PLDI 2017. Barcelona, Spain: ACM. ISBN: 978-1-4503-4988-8. doi:10.1145/3062341.3062363.

Herrera, David, Hanfeng Chen, Erick Lavoie ja Laurie Hendren. 2018. "Numerical computing on the web: benchmarking for the future". Teoksessa *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages,* 88–100. ACM. doi:10.1145/3276945.3276968.

Jangda, Abhinav, Bobby Powers, Arjun Guha ja Emery Berger. 2019. "Mind the Gap: Analyzing the Performance of WebAssembly vs. Native Code". *arXiv preprint arXiv:1901.09056.* https://arxiv.org/abs/1901.09056.

Khan, Faiz, Vincent Foley-Bourgon, Sujay Kathrotia, Erick Lavoie ja Laurie Hendren. 2015. “Using JavaScript and WebCL for numerical computations: a comparative study of native and web technologies”. *ACM SIGPLAN Notices* 50 (2): 91–102. doi:10.1145/2661088.2661090.

MDN web docs. 2019. *WebAssembly*. https://developer.mozilla.org/en-US/docs/WebAssembly. Accessed: 2019-02-18.

Reiser, Micha, ja Luc Bläser. 2017. “Accelerate JavaScript applications by cross-compiling to WebAssembly”. Teoksessa *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages,* 10–17. ACM. doi:10.1145/3141871.3141873. http://concurrency.ch/Content/publications/Reiser_Blaeser_Accelerate_JavaScript_WebAssembly_VMIL_2017.pdf.

Rossberg, Andreas. 2016. “WebAssembly: high speed at low cost for everyone”. Teoksessa *ML16: Proceedings of the 2016 ACM SIGPLAN Workshop on ML*. https://c10109cf-a-62cb3a1a-s-sites.googlegroups.com/site/mlworkshoppe/2016-1.pdf.

———. 2019. *WebAssembly Specification*. https://webassembly.github.io/spec/core/bikeshed/index.html. Accessed: 2019-02-18.

SPEC’s Benchmarks. https://www.spec.org/benchmarks.html. Accessed: 2019-03-25.

Trivellato, Marco. 2018. *WebAssembly is here!* https://blogs.unity3d.com/2018/08/15/webassembly-is-here/. Accessed: 2019-04-29.

Watt, Conrad. 2018. “Mechanising and verifying the WebAssembly specification”. Teoksessa *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs,* 53–65. ACM. doi:10.1145/3167082.

Watt, Conrad, John Renner, Natalie Popescu, Sunjay Cauligi ja Deian Stefan. 2019. “CT-wasm: Type-driven Secure Cryptography for the Web Ecosystem”. *Proc. ACM Program. Lang.* (New York, NY, USA) 3, numero POPL (): 77:1–77:29. ISSN: 2475-1421. doi:10.1145/3290390.

webassembly.org. 2019a. *FAQ*. https://webassembly.org/docs/faq. Accessed: 2019-05-06.

———. 2019b. *Roadmap*. https://webassembly.org/roadmap/. Accessed: 2019-02-18.

———. 2019c. *Use Cases*. https://webassembly.org/docs/use-cases/. Accessed: 2019-05-06.

Yee, Bennet, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula ja Nicholas Fullagar. 2009. "Native client: A sandbox for portable, untrusted x86 native code". Teoksessa *2009 30th IEEE Symposium on Security and Privacy,* 79–93. IEEE. doi:10.1109/SP.2009.25.

Zakai, Alon. 2011. "Emscripten: an LLVM-to-JavaScript compiler". Teoksessa *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion,* 301–312. ACM. doi:10.1145/2048147.2048224.