Tampereen yliopisto

Aapo Hakamaa

WEBASSEMBLYN JA JAVASCRIPTIN SUORITUSKYKYVERTAILU

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintutkielma
Kesäkuu 2021

TIIVISTELMÄ

Aapo Hakamaa: WebAssemblyn ja JavaScriptin suorituskykyvertailu
Kandidaatintutkielma
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma
Kesäkuu 2021

Verkkopalveluiden kasvavan käytön myötä verkkosivujen toiminnalle asetetaan yhä enemmän vaatimuksia. Selaimella suoritettava verkkosivujen logiikka monimutkaistuu jatkuvasti responsiivisten käyttöliittymien yleistyessä. Vaikka tietokoneiden laskentateho on tasaisessa kasvussa, on tullut tarve kehittää myös täysin uusia teknologioita sujuvoittamaan verkkoselailua.

Jo usean vuosikymmenen ajan verkkosivujen logiikka on perustunut JavaScript-ohjelmointikieleen. Selaimet ovat optimoineet hyvin pitkälle tämän kielen suorittamista. JavaScriptissä on kuitenkin tehty perustavanlaatuisia ratkaisuja, kuten dynaaminen tyypitys ja ihmisen luettavan koodin tulkkaus, jotka asettavat rajoitteita suoritusnopeudelle.

WebAssembly on uusi, kehitteillä oleva teknologia JavaScriptin rinnalle. Sen tavoitteena on kiertää JavaScriptin rajoitteita ja näin tuoda lisätehoa koodin suorittamiseen verkkosivuilla. WebAssemblyn vahvuuksia ovat nopeasti tulkattava tavukoodi ja staattinen tyypitys.

Tämän tutkielman tarkoitus on vertailla WebAssemblyn ja JavaScriptin suorituskykyjä. Vertailu toteutetaan kahdella erilaisella käyttötapauksella. Kummassakin käyttötapauksessa käytetään lisäksi useita eri arvoja parametreille, jotta saadaan tarkempi kuva nopeuteen vaikuttavista tekijöistä. Molemmilla kielillä saman ohjelman suoritukseen kuluva aika mitataan millisekunnin tarkkuudella. Jokaisella parametrijoukolla aika mitataan useaan kertaan ja lopputulokseen otetaan näiden aikojen mediaani.

Tutkielman tuloksista havaitaan, että WebAssembly on suurimmassa osassa tapauksista huomattavasti nopeampi JavaScriptiin verrattuna. Huonoimmillaan WebAssemblyn suorituskyky on samaa luokkaa ja parhaimmillaan jopa yli sata kertaa nopeampi kuin JavaScript. Tämän työn menetelmät kuitenkin antavat vain yksipuolisen kuvan WebAssemblyn ja JavaScriptin eroavaisuuksista. WebAssemblyn käyttöönottoa harkittaessa tulee ottaa huomioon muitakin seikkoja, kuten selainten tuki, kehityskustannukset ja osaaminen.

Avainsanat: WebAssembly, JavaScript, suorituskyky, vertailu, ohjelmointi, front end, verkkosivu, selain

SISÄLLYSLUETTELO

LYHENTEET

API	Application Programming Interface, ohjelmointirajapinta
HTML	Hypertext Markup Language, verkkosivujen merkintäkieli

1 JOHDANTO

Internetin kasvun myötä yhä useammat palvelut ovat siirtymässä verkkoon. Verkkoselaimella pystyy kotoa käsin hoitamaan niin pankkiasiat kuin ostoksetkin, mutta myös viihdesisältö ja pelit ovat saatavilla ilman, että tietokoneelle tarvitsee asentaa raskaita sovelluksia. Pilvipalvelut ovat ulkoistaneet myös tiedostonhallinnan verkkokäyttöliittymien taakse. Mullistuksen myötä selaimet ja verkkoteknologiat ovat kasvaneet keskeiseksi osaksi palveluiden saatavuuden ja toimivuuden takaamista.

JavaScript on nykyään yleisin ohjelmointikieli verkkosivujen logiikan toteuttamiseen. Koska on suuri tarve saada verkkosivut toimimaan mahdollisimman tehokkaasti, selainten JavaScript-koodin suoritus on hyvin pitkälle optimoitua. [1] JavaScriptin suunnitteluperiaatteissa on kuitenkin tehty kompromisseja suorituskyvyn ja koodin kirjoitettavuuden välillä. JavaScriptin dynaaminen tyypitys koituu usein pullonkaulaksi selaimessa suoritettaville ohjelmille [2].

Vaihtoehdoksi JavaScriptille on kehitteillä uusi standardi – WebAssembly, jonka tavoite on parantaa verkkopalveluiden suorituskykyä. WebAssembly muistuttaa sekä tyylillisesti että tehokkuudeltaan tietokoneella natiivisti suoritettavia ohjelmia. Sitä ei kuitenkaan ole tarkoitettu JavaScriptin korvaajaksi vaan käytettäväksi rinnakkain eri käyttötarkoituksiin. [3]

Työn tarkoituksena on selvittää JavaScriptin ja WebAssemblyn eroavaisuuksia lähinnä suorituskyvyn näkökulmasta. Vertailuja suoritetaan kahdesta eri käyttötapauksesta verkkoympäristössä. Tavoitteena on selvittää, missä tapauksissa WebAssemblyä on kannattavaa käyttää JavaScriptin sijaan ja miten paljon sillä on vaikutusta suoritustehokkuuksiin.

WebAssembly on vielä kehitystyön alla, joten tämän työn kaltaiset vertailut ovat vielä harvassa. Toisaalta selainten WebAssembly-koodin optimointiin ja suoritukseen voi tulla parannuksia, joten tässä tutkielmassa saadut tulokset saattavat vanhentua tulevaisuudessa.

Työssä käydään läpi aluksi yleisesti verkko-ohjelmointia selaimen toimintaan rajoitettuna, eli palvelinohjelmointia ei käsitellä. JavaScript ongelmineen esitellään ja WebAssemblyn toimintaperiaatteista kerrotaan tarkemmin. Itse vertailun käyttötapaukset, vertailumenetelmät ja käytetyt ohjelmakoodit käydään läpi. Käyttötapauksina on enemmän käyttäjän interaktiota hyödyntävä ohjelma sekä paljon laskentaa vaativa, raskas ohjelma. Lopuksi tutkitaan vertailujen tuloksia ja pohditaan niiden perusteella sopivuuksia eri käyttötarkoituksiin.

2 FRONT END -VERKKO-OHJELMOINTI

Verkkosivut jaetaan yleensä toteutuksen näkökulmasta kahteen osaan. Front end tarkoittaa verkkosivun sitä osaa, jonka käyttäjä näkee selaimellaan ja jonka kanssa käyttäjä on vuorovaikutuksessa. Se esittää tietoja käyttäjälle graafisesti ja ottaa vastaan käyttäjän komentoja. Back end puolestaan hoitaa käyttäjälle näkymättömiä toimintoja, kuten tiedon tallentamisen tietokantaan ja ajastettujen sähköpostien lähetykset. Front endin ohjelmakoodia suoritetaan yleensä selaimella käyttäjän laitteella ja back endin erillisellä palvelimella. Käyttäjän vieraillessa verkkosivulla front end lähettää pyyntöjä verkkosivun back endille ja on sitä kautta yhteydessä esimerkiksi tietokannan dataan.

Verkkosivujen front end -ohjelmoinnissa on otettava huomioon eri verkkoselaimet ja päätelaitteet. Sama verkkosivu voi eri alustoilla näyttää erilaiselta ja toimia eri tavalla. Ohjelmoinnin helpottamiseksi on kehitetty standardeja, joilla eri alustojen toiminnallisuuksia saadaan yhtenäistettyä. Selainkehittäjät pyrkivät standardien mukaiseen toteutukseen, mutta lisäävät usein myös omia ominaisuuksiaan.

Standardeja verkkoympäristöön kehittää ja ylläpitää pääasiassa World Wide Web Consortium (W3C). Konsortio on muun muassa kehittänyt HTML-standardin sekä useita muita laajassa käytössä olevia standardeja. [4]

2.1 TYYPPIJÄRJESTELMÄT

Tyyppijärjestelmät tarkoittavat ohjelmointikielissä tietorakenteiden ja muuttujien käsittelyyn kohdistuvia sääntöjä. Tietotyypit auttavat ohjelmoijaa luokittelemaan suurempia määriä eri käyttötarkoituksiin tarkoitettua dataa. Tyyppejä voivat olla esimerkiksi kokonaisluku, totuusarvo, merkkijono, funktio tai ohjelmoijan määrittelemä luokka, jolla on annettuja ominaisuuksia.

Tyyppijärjestelmät jaetaan staattisesti ja dynaamisesti tyypitettyihin järjestelmiin. Staattisesti tyypitetyssä järjestelmässä muuttujan tyyppi määritetään luonnin yhteydessä, eikä sitä voi enää muuttaa jälkikäteen. Tämä helpottaa tiedon jäsentelyä ja ohjelman dokumentointia sekä testausta.

Dynaamisesti tyypitetyssä järjestelmässä muuttujan tyypin voi muuttaa kesken ohjelman suorituksen. Dynaamista tyypitystä käytetään muun muassa JavaScript-kielessä. Tyyppejä ja luokkia ei tarvitse esitellä eikä määritellä muuttujalle luonnin yhteydessä. Sen ansiosta tiedostokoot pysyvät pienempinä, mikä on verkko-ohjelmoinnissa merkittävä etu tiedonsiirron optimointiin.

Suurin haittapuoli dynaamisessa tyypityksessä on sen vaikutus ohjelman suoritustehoon. Koska muuttujan tyyppi voi olla mikä tahansa, ennen sille tehtäviä operaatioita täytyy suorittaa enemmän tarkistuksia kuin staattisella tyypityksellä, jolla tarkistukset voidaan tehdä jo ohjelmaa käännettäessä. Kun tarkistukset tehdään vasta suorituksen aikana, myös mahdollisuus virheisiin on suurempi. [2]

2.2 TAVUKOODI

Tavukoodi on konekieltä muistuttavaa koodia, jossa operaatiokoodit ovat yhden tavun pituisia. Tavukoodi on optimaalista tietokoneen tulkattavaksi, eikä sitä ole tarkoitettu ihmisen luettavaksi tai kirjoitettavaksi. Useat ohjelmointikielet käännetään tavukoodiksi ennen tulkkausta tai konekielelle kääntämistä.

Tavukoodi on vähemmän laitteistoriippuvaista kuin konekieli, koska käskyjen ei tarvitse suoraan kuulua minkään prosessorin käskykantaan. Korkean tason ohjelmointikieliin verrattuna tavukoodi on tiiviimpää, joten myös tiedostokoko pysyy pienempänä. Tavukoodin käskyt muistuttavat myös konekielen käskyjä, joten tulkkaus on lähes yhtä nopeaa kuin käännetyn ohjelman suorittaminen.

2.3 JAVASCRIPT

JavaScript on front end -verkko-ohjelmoinnissa yleisimmin käytetty ohjelmointikieli. Sen kehitti Netscapella työskennellyt Brendan Eich vuonna 1995, mutta sittemmin se standardoitiin ECMAScriptinä, jota kehittää Ecma International. Sitä käytetään verkkosivuilla yhdessä HTML-dokumenttien kanssa. HTML:n avulla luodaan sivun staattinen ulkoasu ja JavaScriptillä pystytään lisäämään interaktiivisuutta sekä ohjelmoida monimutkaisempaa logiikkaa. Nykyään JavaScriptin käyttö on yleistynyt lisäksi muun muassa palvelin- ja mobiiliohjelmoinnissa. [5]

JavaScript on dynaamisesti tyypitetty ja oliopohjainen kieli. Sen oliot koostuvat ominaisuuksista, joilla on nimi ja arvo. Ominaisuuksia pystyy lisätä, muokata ja poistaa milloin tahansa, mikä lisää tietorakenteiden dynaamisuutta. Verkkosivuilla JavaScriptiä voidaan käyttää standardoitujen rajapintojen avulla esimerkiksi graafisen näkymän muokkaamiseen tai mikrofonin äänisignaalin tallentamiseen.

JavaScript-koodia ei käännetä, vaan selaimet tulkkaavat sitä JavaScript-moottoreilla. Koodi on siis ulkoasultaan samaa sekä ohjelmoijalle että tietokoneelle. Eri selaimilla on erilaisia toteutuksia JavaScript-tulkeista: Googlen Chrome-selain käyttää V8:aa, Mozillan Firefox SpiderMonkeyta ja

Microsoftin Internet Explorer Chakraa. [2] Tulkkaukseen käytetään paljon eri tekniikoita, joilla JavaScript-koodin suoritusta pystytään optimoimaan. Dynaamista tyypitystä on kuitenkin vaikea optimoida, ja se onkin JavaScriptin suurimpia pullonkauloja. [1][8]

2.4 WEBASSEMBLY

WebAssembly on kehitteillä oleva ohjelmointikieli, jonka tavoitteena on tarjota lähes työpöytäsovellusten tasoista suoritustehoa verkkoympäristössä [6]. Kehitystyö on ollut WebAssembly Community Groupin voimin aktiivisesti käynnissä vuodesta 2015. Tämänhetkinen versio on 1.0 ja tarkoituksena on jatkaa suunnittelua laajamittaisen käyttöönoton ja testauksen siivittämänä. [7]

WebAssembly käyttää staattista tyypitystä, minkä ansiosta sen suorittaminen on potentiaalisesti nopeampaa JavaScriptiin verrattuna. WebAssemblyn tukemat muuttujien tyypit ovat sekä 32- että 64-bittiset kokonaisluvut ja liukuluvut. Koska kyseessä on matalan tason ohjelmointikieli, monimutkaisemmat tyypit ja muuttujien operaatiot ovat ohjelmoijan toteutuksen varassa.

Binäärimuoto	Tekstimuoto
20 00	get_local 0
42 00	i64.const 0
51	i64.eq
04 7e	if i64
42 01	i64.const 1
05	else
20 00	get_local 0
20 00	get_local 0
42 01	i64.const 1
7d	i64.sub
10 00	call 0
7e	i64.mul
0b	end

Koodiesimerkki 1. *WebAssemblyn binääri- ja tekstimuoto [9].*

Toisin kuin JavaScriptiä WebAssemblyä ei ole pääasiassa tarkoitettu suoraan ihmisen kirjoitettavaksi, vaan korkeamman tason ohjelmointikieliä, kuten C ja C++, käännetään WebAssembly-moduuleiksi. WebAssemblylle on kuitenkin määritelty kaksi muotoa: binäärimuoto ja tekstimuoto. Koodiesimerkissä 1 on esitettynä sama funktio molemmilla tavoilla. Binäärimuoto on verrattavissa konekieleen ja tekstimuoto puolestaan assembly-ohjelmointikieleen eli symboliseen konekieleen. [3]

2.4.1 BINÄÄRIMUOTO

Muoto, jota selaimet WebAssemblystä ymmärtävät, on binäärimuoto. Se on tavukoodia ja muistuttaa natiivia konekieltä. Kun operaatiokoodit ovat yhden tavun mittaisia, ne ovat prosessorin helposti käsiteltävissä ja suoritus on nopeaa.

WebAssemblyn binäärimuotoa ei ole tarkoitettu ihmisen kirjoitettavaksi tai luettavaksi, koska siinä on otettu huomioon vain suoritustehoon vaikuttavat tekijät. Ohjelmointi tapahtuu ylemmän tason ohjelmointikielillä tai WebAssemblyn tekstimuodossa, joista se sitten käännetään binäärimuotoon suorittamista varten.

2.4.2 TEKSTIMUOTO

Ihmisystävällisempi formaatti WebAssemblystä on tekstimuoto. Sitä ei pysty suorittamaan selaimessa, vaan ohjelma täytyy kääntää ensin binäärimuotoon. Tekstimuoto muistuttaa paljon assembly-ohjelmointikieltä niin käyttötarkoituksen kuin ulkonäkönsäkin kannalta.

Tekstimuoto on hyödyllinen, kun halutaan täysi kontrolli ohjelman toiminnasta. Kun suorituskyky on kriittinen tekijä, korkeammalta ohjelmointikieleltä kääntämisen jälkeen ohjelmaa pystytään optimoimaan käsin tekstimuodossa ennen binäärimuotoon kääntämistä. Tekstimuoto on kätevä myös debuggauksessa sekä WebAssemblyn opiskelussa.

2.4.3 JAVASCRIPT API

Kehitysvaiheessa WebAssembly-koodia ei vielä pysty yksinään suorittamaan selaimessa. Koodin suorittamiseksi täytyy käyttää hyväksi JavaScript-rajapintaa, joka sisältää tarvittavat luokat sekä metodeja muun muassa virheenkäsittelyyn ja muistinhallintaan. WebAssembly-moduulit ladataan JavaScript-koodissa, minkä jälkeen moduulien funktioita voidaan käyttää.

Importoiminen toimii molempiin suuntiin; sen lisäksi että JavaScriptissä voidaan käyttää WebAssembly-funktioita, WebAssemblyssä pystytään myös kutsumaan JavaScriptillä toteutettua funktiota. Lisäksi WebAssembly-koodista aiheutuvat virheet tuottavat tavallisia JavaScript-virheolioita, joten niitä pystyy tarkastelemaan JavaScriptin puolella. JavaScriptistä pystyy myös käsittelemään WebAssemblyn muistia. Tiivis integraatio kielien välillä on käytännöllinen, koska silloin on mahdollista hyödyntää molempien erilaisia vahvuuksia. [10]

3 VERTAILUMENETELMÄT

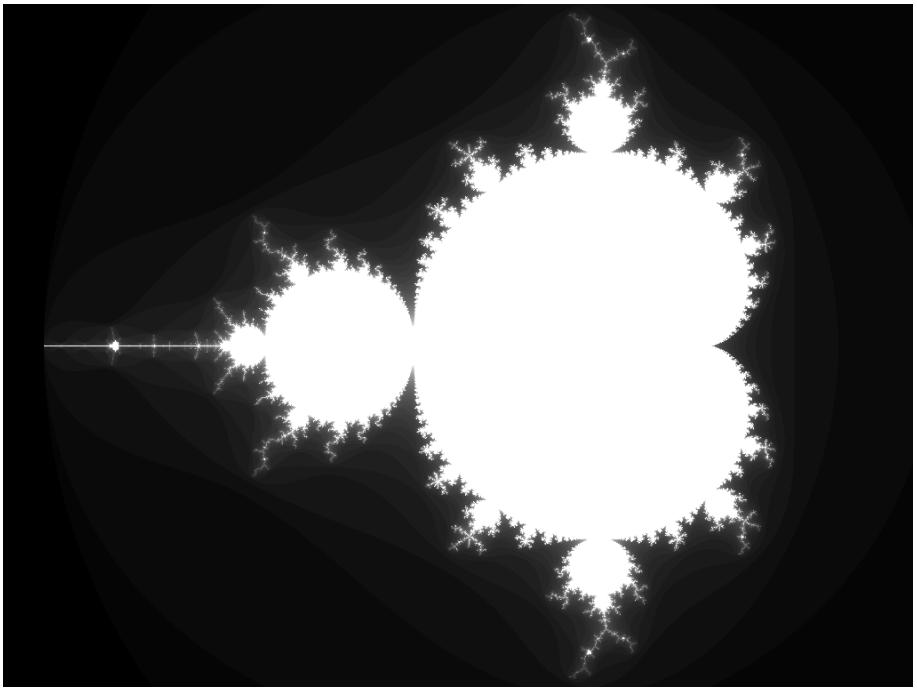
Tässä luvussa käydään läpi menetelmät, joilla JavaScriptin ja WebAssemblyn suorituskykyvertailua on lähestytty. Menetelmistä kerrotaan niiden yhteyksistä verkkosivujen yleisiin skenaarioihin ja esitetään koodit, joilla vertailut suoritettiin. Myös C++-koodi, joka on käännetty WebAssemblyksi, esitellään, koska se muistuttaa enemmän JavaScriptiä ja siten helpottaa vertailua. C++-koodin kääntämiseen WebAssemblyksi on käytetty verkosta löytyvää työkalua WebAssembly Explorer [11].

Kielien vertailu toteutetaan tapaustutkimuksena. Jokaisesta tapauksesta mitataan tehokkuuksia monilla eri muuttujien arvoilla ja usealla selaimella. Mittausperusteena käytetään koodin suorittamiseen kulunutta aikaa.

3.1 TAPAUS 1: MANDELBROTIN JOUKKO

Työpöytäsovellusten kaltaiset palvelut, kuten kuvankäsittelytyökalut, ovat yleistymässä verkossa. WebAssemblyn yksi potentiaalisista käyttökohteista on tämänkaltaiset raskasta laskentaa vaativat tehtävät. [6]

Mandelbrotin joukko on fraktaali, jota usein visualisoidaan tietokoneella. Se perustuu funktioon, jota moneen kertaan iteroitaessa tietyillä kompleksiluvuilla joko jää lähelle nollaa tai kasvaa äärettömäksi. Visualisoinnissa jokainen kuvan pikseli vastaa yhtä kompleksilukua, ja pikselin väri riippuu siitä, kuinka nopeasti arvo lähenee ääretöntä. Kuvassa 1 on esimerkki Mandelbrotin joukon visualisoinnista. Mitä tummempi kuvapikseli on, sitä nopeammin arvo kasvoi iteroidessa. Valkoiset alueet eivät laskentaiteraatioiden aikana kasvaneet yli kynnysarvon. Mandelbrotin joukon laskenta on hyvin raskasta, koska jokaiselle pikselille täytyy tehdä useita iteraatioita rekursiivisella funktiokutsulla.



Kuva 1. *Mandelbrotin joukko.*

Vertailua alustettiin luomalla HTML-tiedosto, joka sisälsi suoritettavat JavaScript-koodit sekä *canvas*-elementin, johon Mandelbrotin joukon kuva piirrettiin. JavaScriptillä pystyi valitsemaan, lasketaanko visualisointi käyttämällä WebAssemblyä vai ilman.

Ohjelma 1 näyttää funktiot Mandelbrotin joukon laskentaan JavaScriptillä. Näistä funktioista kutsutaan *mandelbrot*-funktiota, joka ottaa parametreikseen kuvan leveyden pikseleinä, kuvan korkeuden pikseleinä, kuvan keskipisteen reaaliarvon eli vaakakoordinaatin, kuvan keskipisteen imaginaariarvon eli pystykoordinaatin, yhden pikselin koon reaali- ja imaginaariyksiköissä sekä maksimi-iteraatioiden määrän yhden pikselin kohdalla. Funktio kirjoittaa tulokset *buffer*-taulukkoon, josta ne voidaan sitten piirtää *canvasiin* käyttäjän nähtäville.

```
buffer = new Uint8ClampedArray(4 * width * height);

// Laskee kompleksiluvun neliön
complexSquare = (number) => {
    return [number[0]*number[0] - number[1]*number[1], 2*number[0]*number[1]];
};

// Laskee kompleksiluvun itseisarvon
complexAbs = (x) => {
    return Math.sqrt(x[0]*x[0] + x[1]*x[1]);
};

// Laskee pikselin väriarvon
iterate = (x, y, iterations) => {
    let v = 1;
    let z = [0, 0];

    for (var i = 0; i <= iterations; i++) {
        z = complexSquare(z);
        z[0] += x;
        z[1] += y;
        if (complexAbs(z) > 2) {
            v = i/iterations;
            break;
        }
    return v;
};

// Visualisoi Mandelbrotin joukon
mandelbrot = (width, height, x, y, scale, iterations) => {
    let cursor = 0;
    // Käy läpi kaikki pikselit
    for (var dy = 0; dy < height; dy++) {
        for (var dx = 0; dx < width; dx++) {
            // Laske pikselin väriarvo
            const value = iterate(
                (dx - width/2) * scale + x,
                (dy - height/2) * scale + y,
                iterations) * 255;

            // Tallenna pikselin väri muistiin
            // Kolme värikanavaa ja läpinäkyvyyskanava
            buffer[cursor++] = value;
            buffer[cursor++] = value;
            buffer[cursor++] = value;
            buffer[cursor++] = 255;
        }
    }
};
```

Ohjelma 1. *Mandelbrotin visualisoiminen JavaScriptillä.*

C++:lla toteutetut funktiot näkyvät ohjelmassa 2. Ne muistuttavat hyvin paljon vastaavia JavaScript-funktioita, mutta suurimpina eroavaisuuksina ovat C++:an muuttujien ja funktioiden paluuarvojen staattiset tyypitykset. Lisäksi C++:ssa tuloksia aloitetaan kirjaamaan rivillä 34 määritellyn *cursorin* mukaan suoraan muistiosoitteeseen nolla. WebAssemblyksi käännettynä kirjoitus alkaa varatun muistialueen alusta. Ohjelma 3 on *complexSquare*-funktio esimerkkinä WebAssemblyn tekstimuodosta. Muut WebAssemblyksi käännetyt funktiot on suuren rivimäärän takia jätetty pois.

```
#include <stdio.h>
#include <complex>
using std::complex;

// Laskee kompleksiluvun neliön
complex<double> complexSquare(complex<double> x) {
    return complex<double>(x.real()*x.real() - x.imag()*x.imag(), 2*x.real()*x.imag());
}

// Laskee kompleksiluvun itseisarvon
double complexAbs(complex<double> x) {
    return sqrt(x.real()*x.real() + x.imag()*x.imag());
}

// Laskee pikselin väriarvon
double iterate(double x, double y, int iterations) {
    complex<double> xy(x, y);
    double v = 1;
    complex<double> z(0, 0);

    for (double i = 0; i <= iterations; i++) {
        z = complexSquare(z) + xy;
        if (complexAbs(z) > 2) {
            v = i/iterations;
            break;
        }
    }
    return v;
}

// Visualisoi Mandelbrotin joukon
void mandelbrot(int width, int height, double x, double y, double scale, int iterations) {
    char* cursor = 0;
    // Käy läpi kaikki pikselit
    for (int dy = 0; dy < height; dy++) {
        for (int dx = 0; dx < width; dx++) {
            // Laske pikselin väriarvo
            unsigned char value = iterate(
                (dx - width/2)*scale + x,
                (dy - height/2)*scale + y,
                iterations) * 255;

            // Tallenna pikselin väri muistiin
            // Kolme värikanavaa ja läpinäkyvyyskanava
            *(cursor++) = value;
            *(cursor++) = value;
            *(cursor++) = value;
            *(cursor++) = 255;
        }
    }
}

```

Ohjelma 2. *Mandelbrotin visualisoiminen C++:lla*

```
(func $func0 (param $var0 i32) (param $var1 i32)
    (local $var2 f64) (local $var3 f64)
    get_local $var0
    get_local $var1
    f64.load
    tee_local $var3
    get_local $var3
    f64.mul
    get_local $var1
    f64.load offset=8
    tee_local $var2
    get_local $var2
    f64.mul
    f64.sub
    f64.store
    get_local $var0
    get_local $var2
    get_local $var3
    get_local $var3
    f64.add
    f64.mul
    f64.store offset=8
)
```

Ohjelma 3. *ComplexSquare-funktio WebAssemblyn tekstimuotoon käännettynä.*

Ohjelmassa 4 on esitetty, miten WebAssembly-funktion suoritus tapahtuu JavaScriptillä. Rivillä 7 ladataan WebAssembly-tavukoodi *mandelbrot.wasm*-tiedostosta. Muistina käytetään 64 KiB kokoisia muistisivuja. Rivillä 19 varmistetaan, että muisti riittää koko kuvalle lisäämällä tarvittaessa muistisivuja. Rivin 14 *getVariables*-funktio hakee annetut parametrien arvot *mandelbrot*-funktiota varten. WebAssembly-funktiota, jolla lasketaan Mandelbrotin joukkoa, kutsutaan rivillä 25. Tulokset piirretään näkyville riveillä 37–45.

```
// HTML-elementti <canvas>, johon kuva piirretään
canvas = document.getElementById(”canvas”);
// Canvasin renderöintikonteksti
ctx = canvas.getContext(”2d”);

// Hae WebAssembly-tavukoodia sisältävä tiedosto
fetch(”mandelbrot.wasm”)
    // Luo tavukoodista binääridataa sisältävä puskuri
    .then(response => response.arrayBuffer())
    // Luo binääridatasta instanssi, jonka kautta voidaan kutsua WebAssembly-funktioita
    .then(bytes => WebAssembly.instantiate(bytes))
    .then(result => {
        // Hae muuttujien width, height, x, y, scale sekä iterations arvot
        const variables = getVariables();

        // Laske kuvan säilömiseen tarvittava tavumäärä
        const bytes = 4 * variables.width * variables.height;
        // Alusta riittävästi muistisivuja
        result.instance.exports.memory.grow(
            Math.max(1, Math.floor((bytes - 1) / 65536)));

        // Aloitetaan ajanotto
        const startTime = Date.now();
        // Laske Mandelbrotin joukko WebAssembly-funktiolla
        result.instance.exports.mandelbrot(
            variables.width,
            variables.height,
            variables.x,
            variables.y,
            variables.scale,
            variables.iterations
        );
        // Mandelbrotin joukon laskuun kulunut aika sekunteina
        const elapsed = (Date.now() – startTime)/1000;

        // Alusta puskuri WebAssembly-funktiolla lasketulla datalla
        let buffer = new Uint8ClampedArray(
            result.instance.exports.memory.buffer, 0, bytes);
        // Luo kuva puskurista
        let imageData = new ImageData(buffer, variables.width, variables.height);
        // Muuta canvasin kokoa
        canvas.setAttribute(“width”, variables.width);
        canvas.setAttribute(“height”, variables.height);
        // Renderöi kuva
        ctx.putImageData(imageData, 0, 0);
    });
```

Ohjelma 4. *WebAssembly-funktion käyttäminen JavaScriptissä.*

Taulukko 1. *Mandelbrot-funktion parametrit*

Parametri	Arvot A	Arvot B	Arvot C
width	2048	8000	256
height	2048	8000	128
x	-0.776	-0.5	-0.0305
y	0.12	0	-0.75
scale	0.000008	0.001	0.00001
Iterations	255	10	15000

Sekä WebAssemblyllä että JavaScriptillä mitattiin pelkästään *mandelbrot*-funktion suoritukseen kuluvaa aikaa. Ajat mitattiin kolmella eri parametrijoukolla, jotka ovat lueteltuna taulukossa 1. Käyttämällä useita eri parametreja voi selvitä mahdollisia pullonkauloja, kuten esimerkiksi suuri kuvakoko tai iteraatioiden määrä.

3.2 TAPAUS 2: LISTAN SUODATUS

Responsiiviset käyttöliittymät ovat yleistyneet verkkosivuilla. Enää sivustoja selattaessa ei siirrytä HTML-resurssista toiseen linkkien välityksellä, vaan yhtä sivua muutetaan dynaamisesti näyttämään haluttu data. [12] Tämän lisäksi verkkosivujen tulee reagoida nopeasti käyttäjän toimiin, koska pitkät lataustauot rikkovat immersion ja vaikuttavat negatiivisesti käyttäjäkokemukseen [13]. Nämä tekijät yhdessä luovat vaatimuksia hyvälle suorituskyvylle.

Yleinen skenaario verkkosivuilla on, että käyttäjä kirjoittaa tekstiä hakukenttään. Usein tällaisissa hakukentissä käytetään *autocomplete*-ominaisuutta, eli käyttäjän kirjoittaessa näytetään ehdotuksia siitä, miten käyttäjä aikoo mahdollisesti jatkaa kirjoittamista. Joka kerta käyttäjän syöttäessä uuden merkin järjestelmä hakee merkkijonojen listasta kaikki vaihtoehdot, joiden alku täsmää käyttäjän syötteeseen. Käyttäjä pystyy sitten kesken kirjoituksen valitsemaan jonkun ehdotetuista vaihtoehdoista, jos se täsmää käyttäjän tarkoitukseen.

Tässä tapauksessa tutkimme listan suodattamista käyttäjän syötteen mukaan. Tapauksen tulokset auttavat vetämään johtopäätöksiä siitä, kuinka hyvin WebAssembly soveltuu käyttötapauksiin, joissa on mukana käyttäjän interaktiota. Alkuperäisessä skenaariossa suodatettiin merkkijonolistaa. WebAssemblyssä ei kuitenkaan ole suoraa tukea JavaScriptin merkkijonotyypille, ja listat jouduttaisiin muuntamaan käsin WebAssemblyn ymmärtämään muotoon. Tästä syystä tapausta yksinkertaistetaan hieman käyttämällä kokonaislukuja merkkijonojen sijaan.

Vertailussa alustetaan pitkä lista satunnaisia kokonaislukuja nollan ja sadan väliltä. Käyttäjälle näytetään tekstikenttä, johon pystyy kirjoittamaan kokonaisluvun. Kun käyttäjä on syöttänyt kokonaisluvun, sitä käytetään suodattamaan listasta pois syötettä pienemmät tai yhtä suuret luvut. Aikaa mitataan tekstikentän muutoksesta alkaen suodatetun listan valmistumiseen asti molemmilla kielillä.

Satunnaisessa listassa jokaisella luvulla on yhtä suuri todennäköisyys. Tämän avulla lista saadaan suodatettua suunnilleen haluttuun pituuteen. Esimerkiksi käyttäjän syötteellä 50 jokainen listan luku on 50 prosentin todennäköisyydellä pienempi tai yhtä suuri kuin syöte. Tällöin suodatetun listan pituus on noin puolet alkuperäisestä. Syötteellä 90 lista lyhenee noin 90 prosenttia.

Ohjelmassa 5 on kuvattu JavaScriptillä käytetyt funktiot. Satunnaisen listan *data* alustus haluttuun pituuteen tapahtuu *initializeData*-funktiolla. Listan alustus tehdään molempien kielien suodatusta varten JavaScriptillä, eikä sitä lasketa mukaan ajan mittaukseen. Itse suodatus tapahtuu *filter*-funktiolla, jonka suoritusta mitataan molemmilla kielillä. JavaScript-versio on yksinkertainen: siinä hyödynnetään kieleen sisäänrakennettua *Array.filter*-metodia, jolla saadaan suodatettua käyttäjän syötettä suuremmat arvot uuteen *result*-listaan.

```
data = [];

// Alustaa datan length-parametrin määrittämään pituuteen
initializeData = (length) => {
    for (let i = 0; i < length; i++) {
        // Lisää listaan satunnainen kokonaisluku väliltä 0 - 100
        data.push(Math.floor(Math.random() * 101));
    }
};

filter = (input) => {
    // Suodata datasta input-parametria suuremmat arvot
    // ja tallenna tulos result-muttujaan
    result = data.filter(value => value > input);
};
```

Ohjelma 5. *Listan suodatus JavaScriptillä.*

C++:lla toteutettu *filter*-funktio on näkyvillä ohjelmassa 6. Parametreina annetaan muistiosoitteet alkuperäisen listan ja tuloslistan alkuun sekä käyttäjän syöttämä kokonaisluku. Suodatus tapahtuu käymällä läpi kaikki alkiot listassa ja lisäämällä syötettä suuremmat arvot tuloslistaan. Funktion paluuarvona on tuloslistan pituus.

```
// Suodata datasta input-parametria suuremmat arvot.
// data ja result ovat osoittimia listojen ensimmäisiin alkioihin.
int filter(int* data, int* result, int input)
{
    // Koko data-lista sijaitsee muistissa ennen result-listaa.
    // Laske data-listan pituus ensimmäisten alkioiden osoittimien erotuksesta.
    int dataSize = result - data;
    // resultSize kertoo samalla suodatetun listan pituuden
    // ja seuraavan tyhjän paikan osoitteen result-listassa.
    int resultSize;

    // Iteroi datan läpi
    for (int i = 0; i < dataSize; i++)
    {
        int value = data[i];

        // Jos data-alkion arvo on suurempi kuin input-arvo
        if (value > input)
        {
            // Lisää arvo result-listan loppuun
            result[resultSize] = value;
            // Siirry seuraavaan tyhjään paikkaan result-listassa
            resultSize++;
        }
    }

    // Palauta suodatetun listan koko
    return resultSize;
}
```

Ohjelma 6. *Listan suodatus C++:lla.*

Tässä tapauksessa on käytetty vaihtoehtoista tapaa ladata WebAssembly-koodia. Ohjelman 7 ensimmäisellä rivillä käytetty *instantiateStreaming*-metodi kääntää *filter.wasm*-tiedoston tavukoodin ja luo käytettävän instanssin yhdellä kutsulla. JavaScriptiin verrattuna WebAssemblyn käyttö vaatii ylimääräisiä toimenpiteitä: ennen kuin WebAssemblyllä toteutettua funktiota voidaan kutsua, täytyy jaettuun muistiin kopioida satunnaislukujen lista. Listan suodattamisen jälkeen tuloslista täytyy kopioida WebAssemblyn muistista. Näihin muistinkäsittelytoimenpiteisiin kuluva aika lasketaan mukaan, kun mitataan listan suodatuksen kokonaisaikaa WebAssemblyllä.

```
WebAssembly.instantiateStreaming(fetch("filter.wasm"))
    .then(obj => {
        filter = (input) => {
            // WebAssemblyn muistin koko määritellään sivuina
            // Sivun koko on 65536 tavua
            // Uint32 koko on 4 tavua
            // Maksimi tarvittu muisti on kaksi kertaa datan pituus Uint32-kokonaislukuja
            // Laske montako muistisivua tarvitaan
            const pageCount = Math.ceil(2 * data.length * 4 /
                obj.instance.exports.memory.buffer.byteLength);
            // Muistin koko on aluksi 1 sivu, kasvata haluttuun kokoon
            obj.instance.exports.memory.grow(pageCount - 1);

            // Kirjoita satunnaislukujen lista WebAssemblyn muistiin
            let buffer = new Uint32Array(obj.instance.exports.memory.buffer);
            buffer.set(data);

            // Suodata lista WebAssembly-funktiolla.
            // Parametriksi annetaan alkuperäisen listan ja suodatetun listan
            // muistiosoitteet, tässä tapauksessa indeksit, sekä käyttäjän syöte.
            // Palautusarvona saadaan tuloslistan pituus.
            const resultSize = obj.instance.exports.filter(0, 4*data.length, input);

            // Hae suodatettu lista WebAssemblyn muistista käyttämällä listan alun
            // muistiosoitetta ja listan pituutta.
            result = buffer.slice(data.length, data.length + resultSize);
        };
    }
);
```

Ohjelma 7. *WebAssembly-funktion käyttäminen JavaScriptissä.*

Taulukossa 2 on esitelty mittauksessa käytetyt parametrien arvot. Parametreja ovat satunnaisten kokonaislukujen listan pituus ja käyttäjän tekstikenttään syöttämä luku, jolla listaa suodatetaan. Arvoina käytetään eri kombinaatioita kahdesta listan pituudesta ja kahdesta käyttäjän syötteestä.

Taulukko 2. *Tapauksen parametrit*

Parametri	**Arvot A**	**Arvot B**	**Arvot C**	**Arvot D**
Listan pituus	10 milj.	10 milj.	100 milj.	100 milj.
Käyttäjän syöte	50	99	50	99

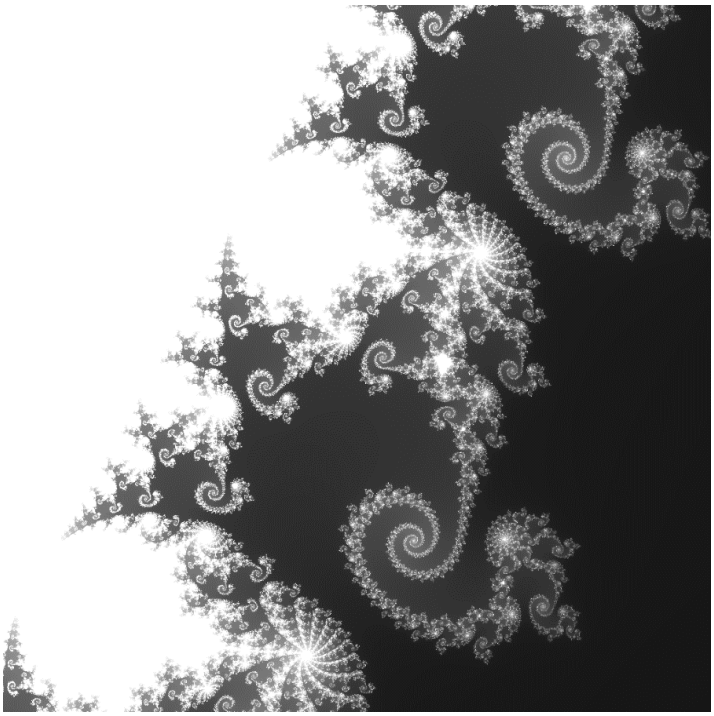
Listan eri pituuksilla pyritään selvittämään, onko esimerkiksi muistin varaamisen nopeudessa eroja. Varsinkin listan kopioimisella WebAssemblyn muistiin ja sieltä pois voi olla vaikutusta suoritusnopeuteen. Käyttäjän syötteiden avulla voidaan vertailla enemmän funktion kutsunopeutta. Syötteellä 99 tuloslistan pituus on noin prosentti alkuperäisen listan pituudesta. Tällöin tuloksen kopioimiseen WebAssemblyn muistista kuluu huomattavasti vähemmän aikaa, jolloin funktion kutsuun kuluva ajan vaikutus näkyy paremmin mittauksissa.

4 TULOKSET

Suoritusaikoja mitattiin neljällä eri selaimella: Google Chrome (versio 81.0.4044.138), Opera (versio 68.0.3618.63), Mozilla Firefox (versio 75.0) ja Microsoft Edge (versio 44.18362.449.0). Mittauksissa käytettiin tietokonetta, jossa on Intel Core i5-3570K prosessori 3,40 GHz kellotaajuudella ja 12 GB DDR3-muistia. Tietokoneen käyttöjärjestelmänä toimi Windows 10 Pro (koontiversio 18362.836). Kaikilla selaimilla aikaa pystyi mittaamaan millisekunneissa, joten tulokset on esitetty taulukoissa sekunteina kolmen desimaalin tarkkuudella.

4.1 TAPAUS 1: MANDELBROTIN JOUKKO

Mandelbrotin joukkoa visualisoitiin eri parametreilla, jotka lueteltiin luvussa 3.1. Kuvassa 2 on tulos, joka saadaan parametreilla A.



Kuva 2. *Parametreilla A visualisoitu Mandelbrotin joukko.*

Visualisoinnin laskemiseen kuluneen ajan vertailuun kerättiin kaikilla parametrien ja selainten kombinaatioilla kahdeksan mittauksen otokset. Otoksista laskettiin mediaaniajat, jotka ovat kirjattuna taulukkoon 3. Joillain selaimilla ensimmäinen mittaus oli melkein tuplasti nopeampi kuin

loput, mikä saadaan tasoitettua laskemalla mediaani keskiarvon sijaan. Myös muut tekijät, kuten satunnaisesti tietokoneella pyörivät taustaprosessit, saadaan melko hyvin kitkettyä vääristämästä aikoja.

Taulukko 3. *Mandelbrotin joukon suoritusaikojen mediaanit sekunteina.*

Selain	Parametrit A (s)	Parametrit B (s)	Parametrit C (s)
Chrome (JavaScript)	21,281	10,321	9,142
Opera (JavaScript)	16,000	10,263	8,853
Edge (JavaScript)	323,015	114,332	181,447
Firefox (JavaScript)	8,660	2,828	4,773
Chrome (WebAssembly)	2,432	0,811	1,347
Opera (WebAssembly)	2,433	0,813	1,348
Edge (WebAssembly)	2,459	0,974	1,358
Firefox (WebAssembly)	2,514	0,750	1,397

Taulukosta 3 nähdään, että WebAssemblyn suoritusajat ovat kaikilla selaimilla ja parametreilla parempia kuin JavaScriptillä. Chromella ja Operalla JavaScript on noin 7–13 kertaa hitaampi ja Edgellä jopa yli sata kertaa hitaampi kuin WebAssembly. Firefoxilla JavaScriptin suoritusajat ovat puolestaan vain noin kolminkertaisia.

4.2 TAPAUS 2: LISTAN SUODATUS

Mandelbrotin joukon tapaan tässäkin tapauksessa jokaisella selaimen, kielen ja parametrien kombinaatiolla mitattiin kahdeksan otosta, joista laskettiin mediaanit. Aikaa mitattiin käyttäjän syötteestä suodatuksen valmistumiseen. Tulokset ovat näkyvillä taulukossa 5.

Taulukko 5. *Listan suodatuksen suoritusaikojen mediaanit sekunteina.*

Selain	**Parametrit A (s)**	**Parametrit B (s)**	**Parametrit C (s)**	**Parametrit D (s)**
Chrome (JavaScript)	0,251	0,146	2,381	1,470
Opera (JavaScript)	0,251	0,148	2,393	1,485
Edge (JavaScript)	0,306	0,095	3,374	0,954
Firefox (JavaScript)	0,113	0,020	1,635	0,180
Chrome (WebAssembly)	0,053	0,017	0,537	0,163
Opera (WebAssembly)	0,053	0,017	0,520	0,163
Edge (WebAssembly)	0,271	0,173	2,656	1,759
Firefox (WebAssembly)	0,075	0,041	0,751	0,405

Kun parametreilla A ja C suodatetaan syötteellä 50, JavaScriptin suoritusajat olivat enintään 5 kertaa hitaampia WebAssemblyyn verrattuna. Millään selaimella JavaScript ei kuitenkaan suoriutunut nopeammin.

Parametreilla B ja D, eli käyttäjän syötteellä 99, tulokset poikkesivat aiemmista. Edge ja Firefox -selaimilla JavaScript oli ensimmäistä kertaa WebAssemblyä nopeampi. WebAssemblyn hitaus saattaa liittyä muistin alustamisen *overheadiin* eli muistinhallinnan funktiokutsujen väistämättömiin resurssikustannuksiin. Chromella ja Operalla JavaScript oli edelleen lähes kymmenen kertaa hitaampi.

Tapausta olisi mahdollista optimoida niin, että muistia ei varata uudestaan joka kerta, kun listaa suodatetaan, vaan ainoastaan kerran sivua ladattaessa. Vain tuloslistan haku WebAssemblyn muistista on pakollinen optimoinninkin jälkeen. Tässä tapauksessa haluttiin kuitenkin pitää avoinna se mahdollisuus, että suodatettava lista muuttuisi, ja näin ottaa huomioon huonoin mahdollinen tapaus.

5 YHTEENVETO

Muutamaa poikkeusta lukuun ottamatta WebAssembly oli kaikissa mittauksissa JavaScriptiä nopeampi. Eri käyttötapauksien välillä oli kuitenkin huomattavia eroja saavutetussa lisätehossa. Parhaimmillaan WebAssembly oli moninkertaisesti nopeampi ja huonoimmillaan samaa tehokkuusluokkaa JavaScriptin kanssa.

Tapauksessa 1 suoritettava funktio sisälsi enemmän koodirivejä ja siten enemmän komentoja tapaukseen 2 verrattuna. WebAssemblyn tavukoodi on nopeammin parsittavissa kuin ihmisen luettava JavaScript, joten pidemmissä funktiossa tämän pitäisi kääntyä WebAssemblyn eduksi. Tapauksen 1 tulokset tukivat tätä oletusta, koska teknologioiden välillä oli huomattava nopeusero. Funktiossa iteroitiin myös kahta sisäkkäistä silmukkaa, mikä moninkertaisti silmukan sisältämän koodin vaikutukset suorituskykyyn.

Tulokset olivat tasaisempia teknologioiden välillä tapauksessa 2. Tämä on mahdollisesti seurausta JavaScriptiin sisäänrakennetun filter-funktion käytöstä. Sisäänrakennettujen funktioiden suoritus voi olla JavaScript-moottoreissa hyvin optimoitua ja lähellä natiivia toteutusta. Tästä syystä WebAssemblyn muistin allokointi ja tavukoodin tulkkaaminen oli joissain tilanteissa hitaampaa.

Suodatettavan listan pidentäminen vaikutti suoraan verrannollisesti suoritusaikoihin. Sekä JavaScriptillä että WebAssemblyllä listan kymmenkertaisella pituudella oli kymmenkertaistava vaikutus mitattuihin aikoihin. Käyttäjän eri syötteillä tuloksissa oli enemmän variaatiota selainten kesken. Edge- ja Firefox-selaimilla syötteen muutos 50:stä 99:ään, eli tuloslistan lyhentäminen 50 prosentista 1 prosenttiin alkuperäisen listan pituudesta, oli suurempi nopeuttava vaikutus JavaScriptiin kuin WebAssemblyyn.

Tuloksista voidaan päätellä, että WebAssemblyn edut saa hyödynnettyä parhaiten, kun tehtävänä on laskea vaativia, pitkiä funktioita tai useaan kertaan toistuvaa ohjelmakoodia. Selaimesta riippuen valmiit JavaScript-funktiot, kuten filter, ovat kuitenkin vielä valideja vaihtoehtoja, koska niiden suoritus JavaScript-moottoreissa on lähellä natiivia. Jos tarve vaatii kustomoidumpaa ohjelmakoodia ja tehokkuus on tärkeässä osassa, WebAssembly on hyvä valinta.

Tässä työssä kieliä vertaillaan puhtaasti suorituskyvyn näkökulmasta. Kun harkitaan kumpi olisi parempi vaihtoehto tiettyyn käyttökohteeseen, täytyy ottaa huomioon myös muut eroavaisuudet. Näitä voi olla esimerkiksi vanhojen selainversioiden tuki ja ohjelmointikustannukset. Jos on tarkoituksena tukea vanhoja selainversioita, jotka eivät vielä tue WebAssemblyä, samasta

järjestelmästä täytyy olla varalla myös JavaScript-versio, jolloin WebAssembly-toteutus tuo pelkästään lisäkustannuksia. JavaScript toimii myös sellaisenaan selaimilla, kun taas WebAssembly täytyy mahdollisesti kääntää C++:sta tai muusta ohjelmointikielestä ja sen jälkeen vielä suorittaa JavaScript API:n avulla. Tästä syystä WebAssemblyn toteutus voi viedä enemmän aikaa ja vaatia laajempaa osaamista ohjelmoijilta. WebAssemblyn hyötyjä kannattaa myös kartoittaa erilaisilla prototyypeillä, ennen kuin se valitaan ohjelmistoprojektin teknologiaksi.

Vaikka tässä työssä tutkittiin kahta varsin erilaista tapausta, ne havainnollistavat vain pientä osaa WebAssemblyn mahdollisista käyttökohteista. Myös käytetty vertailumenetelmä kuvaa vain yhtä puolta teknologioiden eroista. Jatkotutkimuksissa voisi tarkastella useampia erilaisia käyttötapauksia. Suorituskyvyn lisäksi olisi myös mahdollista tutkia muita näkökulmia, kuten WebAssemblyn käyttöönoton haasteita projekteissa.

LÄHTEET

[1] W. Ahn et al. Improving JavaScript performance by deconstructing the type system, ACM, 2014, pp. 496–507. Saatavissa (viitattu 14.11.2017): https://dl-acm-org.libproxy.tuni.fi/citation.cfm?id=2594332

[2] S. Li, B. Cheng ja X. Li, TypeCastor: Demystify dynamic typing of JavaScript applications, AMC, 2011, pp. 55–65. Saatavissa (viitattu 14.11.2017): https://dl-acm-org.libproxy.tuni.fi/citation.cfm?id=1944873

[3] WebAssembly Community Group, WebAssembly Specification, 2018. Saatavissa (viitattu 22.1.2018): https://webassembly.github.io/spec/core/index.html

[4] World Wide Web Consortium, All Standards and Drafts, 2017. Saatavissa (viitattu 14.11.2017): https://www.w3.org/TR/

[5] Ecma International, ECMAScript 2015 Language Specification, 2015. Saatavissa (viitattu 14.11.2017): https://www.ecma-international.org/ecma-262/6.0/

[6] D. Crookes, Our guide to WebAssembly, Web User, (420), 2017, pp. 36. Saatavissa (viitattu 18.9.2017): https://search-proquest-com.libproxy.tuni.fi/docview/1884390479/fulltext

[7] WebAssembly Community Group, Roadmap, 2018. Saatavissa (viitattu 22.1.2018): http://webassembly.org/roadmap/

[8] F. Khan, V. Foley-Bourgon, S. Kathrotia, E. Lavoie, L. Hendren, Using JavaScript and WebCL for numerical computations: A comparative study of native and web technologies, 2014. Saatavissa (viitattu 18.9.2017): http://dl.acm.org.libproxy.tut.fi/citation.cfm?id=2661090

[9] WebAssembly Community Group, Text Format, 2018. Saatavissa (viitattu 22.1.2018): http://webassembly.org/docs/text-format/

[10] WebAssembly Community Group, JavaScript API, 2018. Saatavissa (viitattu 22.1.2018): http://webassembly.org/docs/js/

[11] M. Bebenita, WebAssembly Explorer, 2017. Saatavissa (viitattu 13.2.2018): https://mbebenita.github.io/WasmExplorer/

[12] A. Mesbah ja A. van Deursen, A component- and push-based architectural style for ajax applications, The Journal of Systems & Software, vol. 81, (12), 2008, pp. 2194–2209. Saatavissa (viitattu 28.8.2020): https://doi.org/10.1016/j.jss.2008.04.005

[13] R. Doherty ja P. Sorenson, Keeping Users in the Flow: Mapping System Responsiveness with User Experience, Procedia Manufacturing, vol. 3, 2015, pp. 4384–4391 Saatavissa (viitattu 28.8.2020): https://doi.org/10.1016/j.promfg.2015.07.436