

JavaScript ja virtuaalikoneet

Ville Lahdenvuo

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 13. joulukuuta 2015

Tiedekunta — Fakultet — Faculty		Laitos — Institution — Department	
Matemaattis-luonnontieteellinen		Tietojenkäsittelytieteen laitos	
Tekijä — Författare — Author			
Ville Lahdenvuo			
Työn nimi — Arbetets titel — Title			
JavaScript ja virtuaalikoneet			
Oppiaine — Läroämne — Subject			
Tietojenkäsittelytiede			
Työn laji — Arbetets art — Level		Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma		13. joulukuuta 2015	23
Tiivistelmä — Referat — Abstract			
<p>JavaScript-virtuaalikoneet ovat perinteisesti toimineet tulkkamalla lähdekoodista muodostettua abstraktia syntaksipuuta tai tavukoodia. Kaikissa tutkituissa virtuaalikoneissa on otettu käyttöön JIT-kääntäjiä suorituskyvyn parantamiseksi, mutta tulkkia käytetään yhä. Varsinkin suorituksen alussa tulkki on JIT-kääntämistä parempi vaihtoehto vähäisen muistinkäytön ja nopean käynnistymisen ansiosta.</p> <p>JavaScriptin dynaamisuudesta johtuen virtuaalikoneiden kehittäjät ovat joutuneet toteuttamaan monimutkaisia menetelmiä tehokkaan konekoodin tuottamiseksi JavaScript-ohjelmista. Monet optimointimenetelmät käyttävät hyväksi oletusta, että sovellukset eivät hyödynnä JavaScriptin dynaamisuutta liikaa, vaan käyttäytyvät melko staattisesti. Tämän oletuksen nojalla on pystytty hyödyntämään optimointimenetelmiä, joita tyypillisesti käytetään staattisesti tyypitettyjen kielten kanssa.</p> <p>Oletus staattisesta käytöksestä on havaittu ongelmalliseksi. Vaikka yleisesti käytössä olevat suorituskyselytestit käyttäytyvät varsin staattisesti, todelliset sovellukset hyödyntävät kielen dynaamisuutta enemmän, mikä heikentää optimointimenetelmien toimivuutta. Avoi muus ja kilpailu on auttanut virtuaalikoneiden suorituskyvyn kehitystä, mutta parannettavaa on vielä.</p> <p>ACM Computing Classification System (CCS):</p> <p>Software and its engineering → Virtual machines</p> <p>Software and its engineering → Very high level languages</p>			
Avainsanat — Nyckelord — Keywords			
JavaScript, virtuaalikone, suorituskysely			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — Övriga uppgifter — Additional information			

Sisältö

1	Johdanto	1
2	Virtuaalikoneiden toiminta	2
2.1	Kahdenlaisia virtuaalikoneita	3
2.2	Perinteisen JavaScript-virtuaalikoneen anatomia	3
2.3	Suoritusaikainen kääntäminen	4
2.4	Optimoinnin ongelmat	5
3	Optimointimenetelmät	7
3.1	Piiloluokat	7
3.2	Sisällytetty välimuisti	10
3.3	Aktivaatietietueiden manipulointi	11
3.4	Roskienkeruu	13
4	Toteutusten vertailua	14
4.1	Suoritusarkkitehtuuri	14
4.2	Tyyppien päättely	16
5	Tulevaisuus	17
6	Yhteenveto	19
	Lähteet	20

1 Johdanto

JavaScript on ohjelmointikieli, joka suunniteltiin ensisijaisesti verkkosivujen tekijöille. Kielen tavoite oli täydentää Java-ohjelmointikieltä ja HTML-merkkäuskieltä. JavaScript mahdollistaa monipuolisten selaimissa suoritettavien sovellusten kirjoittamisen ilman selainlaajennuksia [17].

Selainten suosio sovellusalustana on kasvanut ja sen ansiosta JavaScriptin käyttö on lisääntynyt. Kasvua siivittää se, että käytännössä kaikissa kuluttajätietokoneissa on jokin selain ja sovellusten käyttämiseen riittää verkkosivuilla vieraileminen. Käyttäjän ei tarvitse asentaa ohjelmaa ennen sovelluksen käyttöä.

Selaimet ovat kehittyneet ja niiden käyttämiä teknologioita on standardoitu. Näistä niin sanotuista Web-teknologioista, joihin JavaScript lasketaan, on tullut varteenotettava vaihtoehto moderniin sovelluskehitykseen. Web-teknologioiden käyttö ei kuitenkaan rajoitu vain selaimiin. Niillä on toteutettu palvelinsovelluksia sekä kokonaan ilman selainta toimivia sovelluksia, kuten esimerkiksi Atom-tekstieditori [8].

JavaScript on dynaaminen oliopohjainen kieli, mutta se tukee myös imperatiivista ja funktionaalista ohjelmointityyliä. JavaScript tarjoaa siis monia tapoja toteuttaa samoja toiminnallisuuksia [2, luku 4.2.1.]. Muuttujat JavaScriptissä ovat dynaamisesti tyyppitettyjä, mikä joidenkin mielestä helpottaa ohjelmakoodin kirjoittamista. Dynaamisuudesta seuraa kuitenkin myös ongelmia, sillä virtuaalikoneiden on vaikea ennustaa dynaamisia muutoksia ja tehdä järkeviä optimointeja, joilla suorituskyykyä voitaisiin parantaa [1, s. 497].

JavaScript-virtuaalikoneet ovat kehittyneet paljon viime vuosina. Niihin on toteutettu monimutkaisia ja kekseliäitä menetelmiä suorituskyyvyn parantamiseksi. Suuri osa virtuaalikoneiden tekemistä optimoinneista perustuu oletukseen, että dynaamisuudesta huolimatta ohjelmat käyttäytyvät suorituksen aikana yleensä melko staattisesti. Keräämällä tietoa ohjelmassa

esiintyvistä tyypeistä suorituksen aikana, virtuaalikoneet pystyvät kääntämään ohjelman optimoiduksi konekoodiksi. On kuitenkin helppo kirjoittaa sellaista koodia, jota virtuaalikoneet eivät ainakaan vielä osaa optimoida.

JavaScriptiä kuitenkin kehitetään jatkuvasti ja siihen on tuotu muun muassa luokka- ja moduulijärjestelmät [2, luvut 14.5. ja 15.2.]. Nämä auttavat yhtenäistämään erilaisia toteutustapoja ja mahdollistavat tällä tavoin aikaisempaa paremmin ennustettavan käytöksen. Esimerkiksi käyttämällä luokkasyntaksia, saadaan yhtenäinen tapa muodostaa olioita. Ennustettavuudesta seuraa parempi optimoitavuus ja suoritussyky [1, s. 497].

JavaScriptin standardoiminen, sen käytön lisääntyminen ja selainvalmistajien keskinäinen kilpailu suoritussyvystä on parantanut kielen mainetta. JavaScriptin rooli on muuttunut skriptikielestä yleiskäyttöiseksi ohjelmointikieleksi [2, luku 4.]. Kieleen on lisätty paljon ohjelmointia helpottavia ominaisuuksia ja keinoja välttää yleisiä ”sudenkuoppia”, joihin varsinkin aloittelevat ohjelmoijat usein törmäävät. Hyvä esimerkki on ”strict”-tila, jonka aktivoinnin jälkeen ei pysty esimerkiksi vahingossa luomaan globaalia muuttujaa unohtamalla `var`-avainsanan muuttujan edestä.

2 Virtuaalikoneiden toiminta

Virtuaalikone on ohjelma, joka tarjoaa todellisen tai hypoteettisen laitteen toiminnallisuuksia muille ohjelmille hyödyntäen sitä suorittavan *isäntäjärjestelmän* abstraktioita ja palveluita. Virtuaalikone voi virtualisoida esimerkiksi CD-asemaa, käyttämällä isäntänsä tiedostojärjestelmää hyväksi, jolloin virtuaalikoneessa suoritettava ohjelma luulee lukevansa CD-levyä, kun todellisuudessa tieto tulee kiintolevyltä.

2.1 Kahdenlaisia virtuaalikoneita

Virtuaalikoneita on kahdenlaisia, *järjestelmä-* ja *prosessivirtuaalikoneita* [22, s. 33]. Järjestelmävirtuaalikone tarjoaa kokonaisen käyttöjärjestelmän palvelut toisin kuin prosessivirtuaalikone, joka tarjoaa yhden prosessin suorittamista varten tarvittavat palvelut. Tässä tutkielmassa virtuaalikoneella tarkoitetaan JavaScript-ohjelmia suorittavaa prosessivirtuaalikonetta.

Yksi suurimmista virtuaalikoneen hyödyistä on, että sille tehty ohjelma tarvitsee kääntää usean alustan sijaan kyseiselle virtuaalikoneelle. Tästä seuraa, että ohjelma toimii kaikilla niillä alustoilla, joille kyseinen virtuaalikone on toteutettu. Virtuaalikoneessa suoritettava ohjelma pääsee käsiksi vain virtuaalikoneen tarjoamiin palveluihin, jolloin ohjelmat on helppo eristää isäntäkäyttöjärjestelmästä ja laitteistosta [22, s. 36]. Pahantahtoisen ohjelman on siis löydettävä haavoittuvuus sekä virtuaalikoneesta että isännästä voidakseen aiheuttaa ongelmia.

2.2 Perinteisen JavaScript-virtuaalikoneen anatomia

Ensimmäisen JavaScript-virtuaalikoneen nimi on SpiderMonkey [4]. Se toteutettiin Netscape-selainta varten vuonna 1995. Nykyään SpiderMonkey-virtuaalikonetta ylläpitää Mozilla ja sitä käytetään muun muassa Mozillan Firefox-selaimessa. Nykyinen SpiderMonkey on kehittynyt paljon, mutta se koostuu yhä kolmesta perustavanlaatuisesta komponentista: *kääntäjä*, *tulkki* ja *roskienkerääjä* [16]. Nykyisessä toteutuksessa on lisäksi muitakin komponentteja, kuten JIT-kääntäjiä, joista kerrotaan seuraavassa luvussa.

SpiderMonkeyn kääntäjä huolehtii koodin *jäsentämisestä* (parsing) ja kääntämisestä *tavukoodimuotoiseksi*. Virtuaalikoneen tavukoodi on todellisen koneen konekoodin kaltaista, ja yksinkertaisista operaatioista muodostuvaa tavukoodia on helpompi suorittaa kuin alkuperäistä tekstimuotoista ohjelmakoodia. Jotkin virtuaalikoneet eivät käytä tavukoodiesitystä, vaan muodostavat koodista ainoastaan *abstraktin syntaksipuun*. Abstrakti syn-

taksipuu on nimensä mukaisesti ohjelmakoodista muodostettu puumainen abstrakti esitystapa.

Kääntäminen suoritetaan ”laiskasti” eli koko ohjelmaa ei käännetä kerralla, vaan koodilohkoja käännetään osissa sitä mukaa kun suoritus etenee. Esimerkiksi suuret kirjastot eivät siis kasvata käännösaikaa niiltä osin, joita ei käytetä ohjelmassa.

Tulkin tehtävä on suorittaa ohjelmaa. Tulkki siis lukee tavukoodia käsky kerrallaan tai käy läpi abstraktia syntaksipuuta ja kutsuu tarvittavia palveluja isäntäjärjestelmästä. Tulvista on siis oltava oma versionsa jokaista eri alustaa varten. Yksi tulkin eduista on, että se on joustavampi kuin todellinen prosessori. Tulkkiin voi toteuttaa monipuolisempia toimintoja helpommin kuin todelliseen prosessoriin.

Roskienkerääjän tehtävä on poistaa muistista muuttujat ja oliot, joihin ohjelmassa ei enää viitata. Roskienkeruun ansiosta ohjelmoijan ei tarvitse vapauttaa muistia itse, vaan järjestelmä hoitaa muistinhallinnan automaattisesti. Automaattinen muistinhallinta vähentää virheiden määrää, kuten muistivuotoja, mutta ei poista kaikkia ongelmia. Esimerkiksi muistivuodot ovat silti mahdollisia, jos ohjelmoija huomaamatta jättää viittauksia olioihin, joita ei enää tarvitse.

2.3 Suoritusaikainen kääntäminen

Riippumatta toteutustavasta tulkki joutuu aina tekemään useita konekäskyjä suorittaakseen yhden tavukoodikäskyn [22, s. 35]. Tämä johtuu siitä, että tavukoodi suoritetaan ohjelmistotasolla, kun konekoodi suoritetaan suoraan laitteistolla. Tämä tekee tulkeista hitaampia kuin haluttaisiin.

Vuonna 2008 Google julkaisi uuden selaimen, Google Chromen, jonka oli tarkoitus parantaa verkkosovellusten käyttökokemusta [12]. Googlen kiinnostus käyttökokemuksen ja ennen kaikkea suorituskyvyn parantamisesta on ymmärrettävää, sillä yhtiöllä on paljon verkkopalveluita, jotka hyötyvät

hyvästä suorituskyvystä. Tästä syystä Google päätyi toteuttamaan Chrome-selaintaan varten oman virtuaalikoneen nimeltään V8.

Mielenkiintoisen V8:sta tekee se, että siinä ei ole lainkaan tulkkia. V8-virtuaalikone kääntää koodin suoraan isäntäjärjestelmän konekoodiksi. Kääntäminen tehdään SpiderMonkeyn tapaan laiskasti niin sanotulla *lähtötilannekääntäjällä* (baseline compiler) [10]. Nopean kääntämisen saavuttamiseksi lähtötilannekääntäjä ei tee monimutkaisia optimointeja, koska ne hidastaisivat ohjelman käynnistymistä. Selainten tapauksessa sivujen nopea lataaminen on kriittistä käyttökokemuksen kannalta.

V8:n innoittamana muut kehittäjät ovat muuttaneet virtuaalikoneidensa toimintaa siten, että tulkkia käytetään vain suorituksen alkuvaiheessa ja ohjelma pyritään kääntämään konekoodiksi mahdollisimman nopeasti *suoritusajalla kääntäjällä* eli *JIT-kääntäjällä* (Just-In-Time compiler).

Varsinkin usein kutsutut funktiot, eli niin sanotut ”kuumat funktiot”, halutaan kääntää mahdollisimman optimoiduksi konekoodiksi. Tätä varten käytetään monissa toteutuksissa erillistä optimoivaa JIT-kääntäjää, joka on hitaampi kuin lähtötilannekääntäjä, mutta tuottaa suorituskypisempää konekoodia.

2.4 Optimoinnin ongelmat

Dynaamisesti tyyppitetillä ohjelmointikielellä toteutetusta ohjelmasta generoitu konekoodi vaatii paljon tyyppitarkastuksia ja poikkeustapauksia muuttujien tyypeille. Virtuaalikoneet tekevät joitain oletuksia ohjelmien käyttäytymisestä. Esimerkiksi ne olettavat, että usein kutsuttuja funktiota kutsutaan usein samantyyppisillä parametreilla [21, s. 2].

Virtuaalikoneiden ei kannata suoraan generoida optimoitua konekoodia JavaScript-ohjelmista, sillä niillä ei ole tietoa muuttujien tyypeistä. Prosessorin kannalta on hyvin tärkeää tietää tehdäänkö jokin operaatio kokonaisluvuille, liukuluvuille tai kenties merkkijonoille. Lisäksi ei ole järkevää käyttää

paljon aikaa koodin optimointiin, jos kyseinen koodi suoritetaan vain kerran tai muutamia kertoja.

Virtuaalikoneet keräävät tietoa ohjelman käyttäytymisestä suorituksen aikana. Tiedon kerääminen hoidetaan yleensä tulkissa, mutta V8:n tapauksessa lähtötilannekääntäjä lisää generoituun konekoodiin käskyjä keräämään tietoa ohjelmassa esiintyvistä tyypeistä [30]. Profiloinnin ansiosta on mahdollista tehdä parempia optimointipäätöksiä.

Avoimen lähdekoodin WebKit-projekti sisältää JavaScriptCore-nimisen virtuaalikoneen, jota käytetään esimerkiksi Applen Safari selaimessa. JavaScriptCore koostuu tulkista, yksinkertaisesta JIT-kääntäjästä optimoivasta JIT-kääntäjästä, jota sen kehittäjät kutsuvat nimellä *DFG-JIT*. Lyhenne DFG tarkoittaa *tietovuokaaviota* (Data Flow Graph) ja se on ohjelman suoritusaikaisen tyyppitiedon tallentava tietorakenne. Siis muiden virtuaalikoneiden tapaan JavaScriptCore kerää ensin tyyppitietoa ja generoi sen avulla optimoitua konekoodia [28].

Automaattinen muistinhallinta eli roskienkeruu on hyödyllinen toiminnallisuus ohjelmoijan kannalta, mutta sen toteuttaminen hyvin ei ole helppoa. Ohjelmalle varatun muistin loppuessa virtuaalikoneen täytyy pysäyttää ohjelman suoritus ja käydä läpi muistin sisältö vapauttaen muistialueita, joihin ei ole enää viittauksia. Selaimen tapauksessa tämä voi aiheuttaa sovelluksen hidastumista. Hidastuminen huonontaa käyttökokemusta etenkin interaktiivisissa sovelluksissa tai animaatioiden aikana.

3 Optimointimenetelmät

Tässä luvussa esitellään muutamia menetelmiä, joilla kehittäjät ovat parantaneet virtuaalikoneidensa tuottaman konekoodin suorituskykyä. Optimointimenetelmiä on kehitetty moniin ongelma-alueisiin kuten kääntämiseen, muistinhallintaan ja kielen erityyspiirteiden toteuttamiseen. Suuri osa menetelmistä hyödyntää virtuaalikoneen suoritusaikana keräämää profiointitietoa. Frankfurtin Goethe-yliopiston YouTube-kanavalta löytyy videoluentoja, joissa selitetään perusteellisesti englanniksi tässä luvussa esiteltyjä aiheita [9].

3.1 Piiloluokat

JavaScriptin oliot käyttävät prototyyppiperintää. Tämä tarkoittaa, että oliolla on käytännössä aina jokin olio prototyyppinään, jolla on toinen olio prototyyppinään ja niin edelleen. Kun oliolta pyydetään jonkin *ominaisuuden* (property) arvoa, etsitään sitä ensin itse oliosta. Jos ominaisuutta ei löydy, käydään olion prototyyppiketjua läpi, kunnes se löytyy tai kaikki prototyypit on käyty läpi, jolloin ominaisuuden arvoksi palautetaan `undefined`.

Ominaisuuden lisääminen tai muuttaminen päivittää oliota, ei koskaan sen prototyyppiä. Jos ominaisuutta ei ole oliossa, se lisätään siihen, vaikka kyseinen ominaisuus löytyisikin prototyyppiketjusta. Tällöin saman prototyypin omaavat oliot säilyttävän vanhan arvon ja vain muutettu olio käyttää uutta arvoa. Olion prototyyppiä pääsee muokkaamaan epästandardin `__proto__`-ominaisuuden kautta tai standardiin lisätyn `Object.getPrototypeOf(obj)`-funktion avulla.

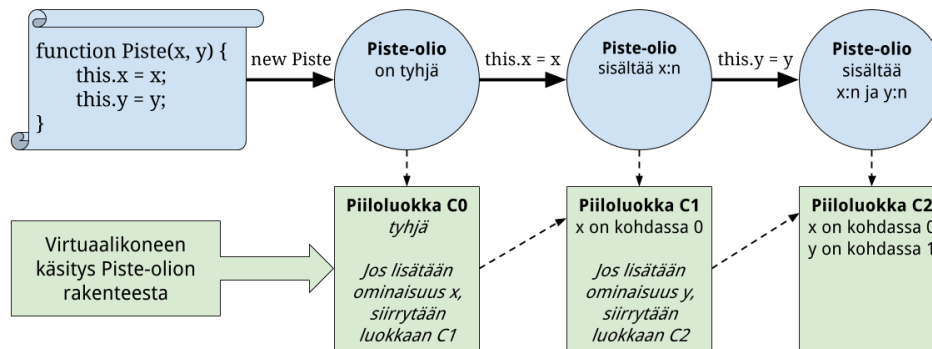
Prototyyppiperintämallista johtuen ominaisuuksien arvot saattavat olla muistissa kaukana toisistaan ja niiden löytämiseen tarvitaan hidasta assosiatiivista hakurakennetta. Staattisissa luokkaperintään pohjautuvissa kielissä olioiden rakenne on helppo muuttaa prosessorille sopivampaan muotoon.

Ongelman ratkaisuksi V8:n kehittäjät esittelivät ominaisuuden nimeltään *piiloluokka* (hidden class) [10] ja sittemmin myös muut virtuaalikoneiden kehittäjät ovat ottaneet samanlaisen mallin käyttöön. Piiloluokat ovat *muuttumattomia* (immutable) luokkia, joita virtuaalikone käyttää JavaScript-olioiden rakenteen kuvaamiseen. Piiloluokka on tavallaan siis olion tyyppi, vaikka JavaScriptissä olioilla ei ole varsinaisesti tyyppiä, paitsi kieleen sisäänrakennetut perustyytit, kuten **String** ja **Number**. Käyttämällä piiloluokkaa olion tyyppinä virtuaalikone voi hyödyntää tehokkaita kääntämistekniikoita, joita käytetään yleisesti staattisesti tyytitetyissä kielissä.

Piiloluokat toimivat siten, että aina kun tyhjä olio luodaan, joko literaalina tai konstruktorissa, virtuaalikone antaa sille tyhjän piiloluokan. Aina kun oloon lisätään ominaisuus, literaalin määrittelyssä, olion konstruktorissa tai myöhemmin koodissa, luodaan sitä vastaamaan uusi piiloluokka tai käytetään olemassa olevaa piiloluokkaa. Piiloluokkia ei voi tehdä etukäteen, koska dynaamisuuden takia on vaikea tietää ennalta millainen olio tulee rakenteeltaan olemaan.

Kuvassa 3.1 näkyy, miten V8 luo piiloluokkia luodessaan Piste-olion ensimmäisen kerran. Kaavion paksut nuolet vastaavat kyseisen koodirivin suorittamista. Katkoviivanuolet kertovat minkälaisen esityksen virtuaalikone muodostaa oliolle ja niiden väliset suhteet. Kun konstruktorifunktiota kutsutaan, luodaan tyhjä olio, johon viitataan avainsanalla **this**. Virtuaalikone kuvaa sitä tyhjällä piiloluokalla **C0**. Luokkaa **C0** käytetään kuvaamaan kaikkia tyhjiä olioita, joilla on sama prototyyppi, joka on oletuksena kieleen sisäänrakennetun **Object**-olion ilmentymä. Konstruktorissa oloon liitetään ominaisuus **x**, jolloin virtuaalikone luo uuden piiloluokan **C1** ja lopulta **y:n** lisäyksen jälkeen **C2:n**. Piiloluokkiin tallennetaan myös tieto mihin piiloluokkaan pitää siirtyä tietyn ominaisuuden lisäämisen jälkeen sekä viittaus olion prototyyppiin.

Piiloluokkia ei tarvitse luoda uudestaan joka kerta, kun halutaan luoda



Kuva 3.1: Esimerkki piiloluokkien toimintaperiaatteesta [10].

uusi Piste-olio. Riittää, että seurataan niihin sisällytetyjä tietoja siirtymistä. Kun uusi Piste-olio luodaan, päädytään lopulta samaan piiloluokkaan C2. Kaikki Piste-oliot ovat siis virtuaalikoneen näkökulmasta samantyyppisiä.

Kun Piste-oliolta pyydetään esimerkiksi ominaisuuden x arvoa, virtuaalikone tietää, että olio on luokan C2 ”tyyppinen” ja arvo löytyy yhdellä konekäskyllä olion muistipaikasta siirtymällä 0 ilman, että täytyy tehdä hidasta hakua assosiatiivisesta hakurakenteesta. Virtuaalikone voi siis generoida tehokkaampaa konekoodia ja muistiviittaukset nopeutuvat huomattavasti.

Syy, miksi luodaan useita piiloluokkia yhden olion luonnin yhteydessä on, että niihin lisätyt siirtymätiedot mahdollistavat erilaisten olioiden luomisen käyttäen hyväksi jo luotuja piiloluokkia. Esimerkiksi Ympyrä-olio voi sisältää ominaisuudet x , y ja r , jolloin sen luomisessa käytetään piiloluokkia C0, C1, C2 ja luodaan uusi C3. Vaihtoehtoisesti voimme luoda kolmiulotteisen Piste, jolla on tasokoordinaattien lisäksi z -ominaisuus. Tällöin piiloluokka C2 sisältää tiedon siirtymästä kummassakin, ympyrän ja kolmiulotteisen pisteen, tapauksessa.

Wonsun Ahn kumppaneineen on tutkinut piiloluokkien hyötyjä ja haittoja todellisissa sovelluksissa [1]. Tutkimuksessa on löytynyt muutamia heikkouksia, jotka vähentävät piiloluokkien hyötyjä todellisissa sovelluksissa. Siinä myös moititaan vahvaa oletusta ohjelmien staattisesta käytöksestä.

Tutkimuksessa havaittiin, miten dynaamisuutta paljon hyödyntävä ohjelma voi helposti luoda paljon piiloluokkia turhaan esimerkiksi alustamalla ominaisuuksia eri järjestyksessä eri kerroilla tai ehdollisesti. Jos piiloluokkia on paljon se haittaa esimerkiksi sisällytettyjen välimuistien toimintaa.

3.2 Sisällytetty välimuisti

Hakupaikaksi (access site) kutsutaan kohtaa koodissa, jossa viitataan olion ominaisuuteen. Koska muuttujilla ei ole tyyppiä, virtuaalikone ei voi tietää ennalta edes löytyykö kyseistä ominaisuutta. Yleinen oletamus on, että samalla hakupaikalla oliot ovat usein samantyyppisiä [21, s. 2]. Tämän takia esimerkiksi V8:ssa on alettu käyttää *sisällytettyä välimuistia* (inline cache) [1, s. 498]. Nimi tulee siitä, että välimuisti on osa generoitua konekoodia. Sen tehtävä on tarkistaa vastaan tulevien arvojen tyyppi ja tallentaa profilointitietoa optimoivaa kääntäjää varten. Sillä ei siis ole suoraan tekemistä prosessorin välimuistin kanssa.

Tarkastellaan seuraavaa pseudokoodia:

```
1 function getX(obj) { return obj.x; }
2 var p = new Piste(1, 2)
3 for (var i = 0; i < 100000; i++) getX(p);
```

Keräämällä tietoa suoritusaikana virtuaalikone voi päätellä, että hakupaikassa `obj.x` on `obj`-muuttujan arvona usein `Piste`-olio. Virtuaalikone voi muokata konekoodia ja lisätä tarkistuksen: ”Onko `obj`:n piiloluokka `C2`?” Jos oliolla on tämä piiloluokka, arvon lukemisen voi suorittaa yhdellä konekäskyllä, sillä ominaisuuden siirtymä muistissa on tunnettu piiloluokan rakenteesta. Sitä voisi kuvata esimerkiksi pseudokoodilla:

```
1 if (getHiddenClass(obj) == 'C2') return obj[C2_X_OFFSET];
2 else return slowPropertySearch(obj, 'x');
```

V8:ssa on kolmen tyyppisiä sisällytettyjä välimuisteja: *lataukselle* (load), *talletukselle* (store) ja *kutsumiselle* (call). Lataaminen tarkoittaa olion

ominaisuuden hakemista ja talletus sen päivittämistä. Kutsuminen tarkoittaa olioön liitetyn funktion kutsumista ja se on samankaltainen lataamisen kanssa, sillä ensin pitää ladata funktio, joka on JavaScriptissä viittaus funktio-olioon.

Jos olion tyyppiä ei löydy sisällytetystä välimuistista, V8 voi lisätä tarkistuksen sisällytettyyn välimuistiin. Esimerkiksi, jos ohjelma alkaa kutsua `getX`-funktia Ympyrä-olioilla, V8 voi lisätä sisällytettyyn välimuistiin tarkistuksen: ”Onko `obj`:n piiloluokka `C3`?”. Tällöin arvon voi ladata taas nopeasti tunnetusta siirtymästä pseudokoodilla:

```
1 if (getHiddenClass(obj) == 'C2') return obj[C2_X_OFFSET];  
2 else if (getHiddenClass(obj) == 'C3') return obj[C3_X_OFFSET];  
3 else return slowPropertySearch(obj, 'x');
```

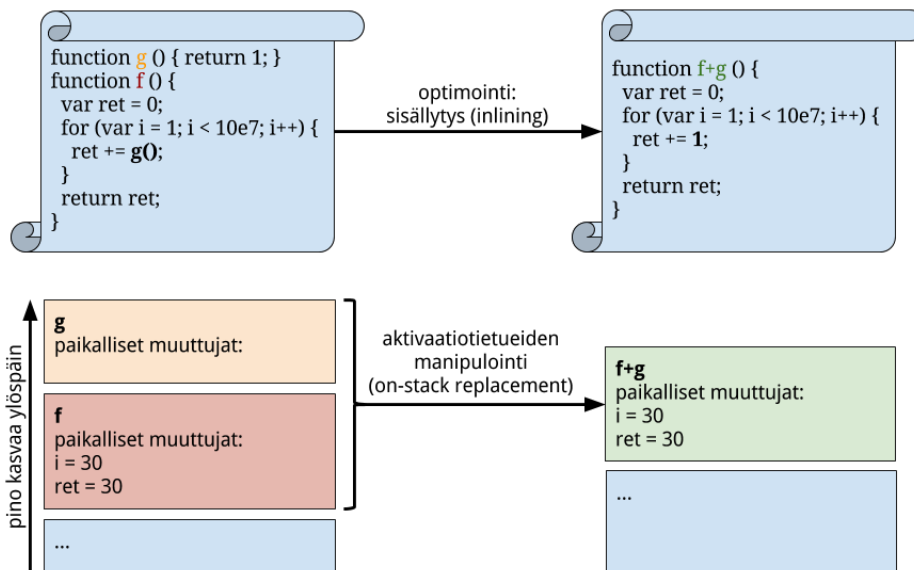
Ylimääräiset tarkistukset tekevät suorituksesta tietenkin hitaampaa. Tutkimuksen mukaan haku välimuistin avulla voi olla parhaimmillaan monta kertaluokkaa nopeampaa kuin ilman välimuistia [1, s. 498], koska silloin ei tarvitse tehdä hidasta hakua ominaisuuden nimellä.

3.3 Aktivaatietietueiden manipulointi

Normaalisti JIT-kääntäjät ottavat optimoidun funktion käyttöön vasta seuraavalla funktion kutsukerralla. Tämä lähestymistapa toimii hyvin monissa tapauksissa, mutta ei silloin kun funktio sisältää esimerkiksi monesti suoritettavan toistorakenteen. Jos funktiota kutsutaan vain kerran, optimoitua koodia ei koskaan saada käyttöön.

Ongelmaa varten on kehitetty menetelmä, jonka englanninkielinen nimi on ”on-stack replacement” [7]. Suomeksi menetelmää voi kutsua aktivaatietietueiden manipuloinniksi. Kuvassa 3.2 näkyy kuinka menetelmää voi käyttää funktion `f` optimointiin.

Kun virtuaalikone alkaa ensimmäistä kertaa suorittaa funktiota `f`, sitä ei ole vielä optimoitu. Virtuaalikone kerää JIT-kääntämistä varten tietoa suorituksen kulusta aika ajoittain ja huomaa, että funktion suoritus vie paljon



Kuva 3.2: Esimerkki aktivaatietietueiden manipuloinnista [29].

aikaa ja funktio pitäisi optimoida. Virtuaalikoneen optimointiheuristiikka päättää sisällyttää funktion **g** funktion **f** sisään eliminoiden turhan funktio-kutsun. Kuvan esimerkissä tämä tapahtuu 30. toiston kohdalla, kun **i**:n arvo on 30.

Kun optimoitu funktio **f+g** on käännetty, suoritus pitää siirtää funktiosta **f** uuteen funktioon **f+g**. Koska funktion **f** suoritus on kesken, täytyy manipuloida funktion **f** aktivaatietietuetta pinossa, siten että se sisältää myös funktion **g** mahdolliset paikalliset muuttujat. Tässä esimerkissä sellaisia ei ole, mutta todellisessa tapauksessa virtuaalikoneen täytyy luoda kuvaus molemmista aktivaatietietueista optimoidun funktion **f+g** aktivaatietietueeksi.

Aktivaatietietueiden manipuloinnin täytyy toimia myös toiseen suuntaan, eli jos koodissa tulee vastaan tapaus, jota optimointi ei ottanut huomioon, täytyy optimointi pystyä peruuttamaan, jolloin aktivaatietietue pitää pilkkoa osiin. Menetelmään liittyy paljon teknisiä yksityiskohtia, kuten miten pysäyttää funktion suoritus ja miten manipuloida aktivaatietietueita oikein.

Tästä huolimatta suuri osa JavaScript-virtuaalikoneista on ottanut menetelmän käyttöön. Toteutuksessa auttaa, jos virtuaalikone hallitsee kaiken konekoodin generoimista, koska silloin se voi esimerkiksi rajoittaa funktiokutsujen konventioita konekoodissa ja vähentää mahdollisten ongelmatilanteiden lukumäärää.

3.4 Roskienkeruu

JavaScript-virtuaalikoneiden *roskienkeruu* (garbage collection) on perinteisesti toiminut melko yksinkertaisella algoritmilla nimeltään *merkitse ja pyyhkäise* (mark-and-sweep). Algoritmi perustuu nimensä mukaisesti kahteen vaiheeseen. Ensimmäinen on merkitsemisvaihe, jossa käydään läpi kaikki ohjelman viittaamat oliot ja merkitään ne. Sitten seuraa pyyhkäisyvaihe, jossa koko muisti käydään läpi ja vapautetaan kaikki oliot, jotka eivät ole merkittyjä ja samalla poistetaan merkit seuraavaa suoritusta varten.

Ohjelman suoritus ei voi jatkua roskienkeruun aikana, koska ohjelma voi käyttää muistia milloin vain. Koko muistialueen käsitteleminen kerralla on hidasta, mikä on suuri ongelma varsinkin animaatioiden ja interaktiivisten sovellusten tapauksessa. Pitkä roskienkeruutauko haittaa käytettävyyttä ja saa sovelluksen tuntumaan hitaalta, vaikka muu suoritus olisikin todella nopeaa.

Ongelman ratkaisemiseksi on kehitetty monenlaisia heuristiikkoja ja optimointeja. Automaattisesta muistinhallinnasta riittää materiaalia useampaankin tutkielmaan [20]. Tässä on lista joitain esimerkkejä menetelmistä, joilla roskienkeruuta on parannettu:

- *sukupolvittainen roskienkeruu* (generational garbage collection) [10]
- *inkrementaalinen roskienkeruu* (incremental garbage collection) [3]
- roskienkeruu ”luppoaikana” eli roskienkeruun aikataulutus [18]
- muistin jakaminen eri osiin, joiden keruuta voi rinnakkaistaa [13].

4 Toteutusten vertailua

Virtuaalikonetoteutukset ovat ottaneet käyttöön toistensa optimointimenetelmiä, mutta niiden kehittäjien erilaiset ajattelutavat ja perinteet ovat ohjanneet virtuaalikoneiden arkkitehtuureja ja käytäntöjä. Toteutuksista löytyy siis yhtäläisyyksiä sekä eroavaisuuksia, jotka vaikuttavat niiden suorituskykyyn, vaikka JavaScriptin toiminta onkin tarkasti määritelty. Vertailuun on valittu Googlen V8-, Applen JavaScriptCore-, Mozillan SpiderMonkey- ja Microsoftin Chakra-virtuaalikoneet, koska niistä löytyy parhaiten tietoa paitsi Chakrasta. Se on toistaiseksi suljettua lähdekoodia, mutta silti mielenkiintoinen ja merkittävä käyttäjämäärällisesti. Microsoft on avaamassa sen lähdekoodin vuoden 2016 alussa [14].

4.1 Suoritusarkkitehtuuri

Ensimmäinen kiinnostava tieto on, mitä virtuaalikone tekee korkealla tasolla koodille suorittaakseen sitä. Virtuaalikoneet eivät ole hylänneet tulkkejaan, vaikka niihin on lisätty JIT-kääntäjiä. Esimerkiksi JavaScriptCoren tulkki on kirjoitettu kokonaan uudelleen paljon edeltäjänsä paremmaksi ja sille on annettu uusi nimi LLInt [27]. Ironista kyllä vaikka Google ei alunperin toteuttanut lainkaan tulkkia V8-virtuaalikoneeseensa, se on nyt kehittämässä tulkkia nimeltä Ignition [11]. Uusi tulkki perustuu JavaScriptCoren LLInt-tulkkiin.

Perustelut muutokselle ovat ymmärrettäviä, sillä uutta tulkkia aiotaan käyttää ainakin alkuun ensisijaisesti mobiililaitteissa, joissa on rajoitettu määrä muistia ja heikompi suorituskyky. Tällaisilla laitteilla kääntäminen konekoodiksi ennen suorittamista on yksinkertaisesti liian hidasta, minkä takia halutaan nopeasti käynnistyvä tulkki [11]. Lisäksi konekoodiksi käännetty koodi vie enemmän muistia kuin tavukoodimuotoinen koodi.

Aikaisemmin V8 on pitänyt JavaScript-lähdekoodia parhaana esitysmuotona ohjelmalle ja siitä on jäsennetty abstrakti syntaksipuu joka kerta ennen

kääntämistä. Uusi tavukoodimuoto voisi kuitenkin toimia myös parempana välikielenä kääntäjille, jolloin alkuperäistä lähdekoodia ei tarvitsisi jäsentää uudelleen joka kerta.

Myös Chakra-virtuaalikone käyttää tulkkia suorituksen alussa. Uusimassa Chakran versiossa on optimoivan JIT-kääntäjän lisäksi yksinkertainen JIT-kääntäjä, joka ei tee monimutkaisia optimointeja mahdollistaen nopeamman käännöksen konekoodiksi [23]. Saatavilla olevien tietojen mukaan Chakra ei tue aktivaatietietueiden manipulointia, joten optimoitu koodi otetaan käyttöön vasta kun funktiota kutsutaan uudestaan. Chakran kehittäjät ovat pyrkineet hyödyntämään laitteistoa mahdollisimman paljon rinnakkais- tamalla käännös- ja roskienkeruuoperaatioita. Myös SpiderMonkey ja V8 suorittaa käännöstyötä rinnakkaisesti ohjelman suorituksen kanssa.

SpiderMonkeyn kehittäjät ovat olleet myös ahkeria ja virtuaalikone on nähnyt jo viisi erilaista JIT-kääntäjää: TraceMonkey, JägerMonkey, IonMonkey, OdinMonkey ja Baseline Compiler (alkutilannekääntäjä) [25]. Kaikki niistä ei ole enää käytössä, vaan osa on korvannut aikaisempia ja osa on siirtynyt eri vaiheeseen suorituksessa. Esimerkiksi alkutilannekääntäjä on korvannut JägerMonkeyn, joka korvasi sitä edeltävän TraceMonkeyn.

SpiderMonkey käyttää yhä tulkkia suorituksen alussa, mutta pyrkii kääntämään koodin mahdollisimman nopeasti alkutilannekääntäjällään [26], IonMonkey keskittyy erityisesti paljon kutsuttujen funktioiden kovakouraiseen optimointiin ja OdinMonkey-kääntäjää käytetään vain asm.js-muotoisen koodin kääntämiseen etukäteen. Asm.js:stä kerrotaan lisää Tulevaisuus-luvussa.

SpiderMonkey ei ole ainoa virtuaalikone, joka on nähnyt useita virtuaalikoneita. V8:n kehittäjät rakentavat uutta JIT-kääntäjää, jota he kutsuvat nimellä TurboFan [24]. Sen on tarkoitus korvata nykyinen optimoiva kääntäjä, Crankshaft, ja olla sitä helpommin jatkokehitettävä. Myös JavaScriptCoren kehittäjät valmistelevat FTL JIT -nimistä (Fourth Tier LLVM JIT) kääntäjää neljänneksi kääntäjäksi [19]. Nimensäkin perusteella se hyödyntää LLVM

kääntäjäinfrastruktuuria, jonka tarkoitus on helpottaa kääntäjien tekemistä tarjoamalla valmis rajapinta. Kääntäjän riittää tuottaa LLVM-yhteensopivaa välikieltä ja LLVM hoitaa sen optimoinnin ja kääntämisen konekoodiksi.

4.2 Tyyppien päättely

Konekoodista ei yksinkertaisesti saa nopeaa, jos jokaisen operaation kohdalla pitää tarkistaa muuttujien tyypit, suorittaa erilaiset tyyppimuunnokset ja varmistaa kaikenlaiset poikkeustapaukset. Tämän takia kaikki vertailun virtuaalikoneet käyttävät jonkinlaista tyyppijärjestelmää, kuten piiloluokkia, taustalla ja pääättelevät muuttujien tyyppejä suoritusaikana kerätyn tiedon perusteella.

V8-virtuaalikoneen tapauksessa kerätty tyyppitieto tallennetaan sisällytettyihin välimuisteihin, jotka ovat osa generoitua konekoodia (eli suoritettavaa muistia) [27]. JavaScriptCoren LLInt-tulkki kerää myös tyyppitietoa sisällytetyillä välimuisteilla, mutta se ei tallenna tyyppitietoa suoritettavan muistin sekaan.

JIT-kääntäjät pystyvät sitten hyödyntämään tätä tyyppitietoa luodessaan optimoituja versioita funktioista. Mutta kuten sisällytetyn välimuistin luvussa mainittiin, täytyy silti varautua poikkeuksiin ja mahdollisesti vaihtaa optimoitu koodi takaisin epäoptimoituun versioon.

Chakran kehittäjät ovat vieneet optimoinnin askelta pidemmälle. Chakra samastaa tyyppejä niin sanotulla ”equivalent object type specilization”-menetelmällä [23]. Menetelmän avulla esimerkiksi aikaisemmin mainitut Piste- ja Ympyräoliot pystyvät käyttämään samaa sisällytettyä välimuistia funktion `getX` hakupaikassa, vaikka niillä on eri piiloluokat. Se onnistuu, koska niiden rakenne on tarpeeksi lähellä toisiaan. Esimerkin tapauksessa ne ovat ekvivalentit tyypeiltään, koska molemmilla on ominaisuus `x` samassa siirtymässä.

5 Tulevaisuus

Brendan Eichin mukaan JavaScriptistä on tullut jo kliseenomaisesti ”Webin assemblykieli” [6]. JavaScript-ohjelmia suoritetaan käytännössä jokaisella alustalla ja kehittäjät ovat alkaneet tehdä kääntäjiä, jotka kääntävät muita ohjelmointikieliä, vanhoja tai uusia, JavaScriptiksi. Tämä lisää painetta parantaa virtuaalikoneiden suorituskykyä ja lisätä matalamman tason rajapintoja kääntäjäohjelmoiden hyödynnettäväksi.

Asm.js [15] on epävirallinen standardi JavaScriptin osajoukosta, jota on mahdollista kääntää tehokkaaksi konekoodiksi. Sen idea on olla toimivaa JavaScript-koodia, mutta mahdollistaa tehokas kääntäminen suoraan konekoodiksi. Se onnistuu kertomalla virtuaalikoneelle, että koodi on asm.js-muotoista ja tarjoamalla tyyppivihjeitä. Tässä esimerkki kokonaislukujen summafunktioista asm.js-koodina:

```
1  "use asm"; // Kerrotaan, että koodi on asm.js-muotoista.
2  function integerSum(x, y) {
3      x = x|0; y = y|0;
4      return (x + y)|0;
5  }
```

Esimerkissä kokonaislukuparametreista annetaan vihjeet tekemällä funktion alussa bittitason tai-operaatio: `x = x|0`. Jos `x`:n arvo ei ole kokonaisluku, esimerkiksi `undefined` tai jokin olio, operaatio muuttaa sen kokonaisluvuksi. Jos virtuaalikone tukee asm.js-kääntämistä, tyyppivihje kertoo virtuaalikoneelle, että parametri tulee aina olemaan kokonaisluku. Virheelliset tyypit voidaan huomata staattisilla työkaluilla suorittamatta koodia. JavaScriptin aritmetiikka toimii aina liukuluvuilla, joten sen takia myös summan tulos pitää muuttaa kokonaisluvuksi.

Mozillan OdinMonkey-kääntäjän lisäksi myös Googlen V8-virtuaalikoneen kehittäjät suunnittelevat tukea asm.js-optimoinneille TurboFan-kääntäjän myötä [24]. Asm.js:n tueksi JavaScriptiin on tuotu lisää suorituskykyä

parantavia toimintoja, kuten *SIMD-käskyt* [6]. SIMD tulee sanoista *Single Instruction Multiple Data*, joka tarkoittaa suomeksi: ”Yksi käsky, useita data-alkioita”. SIMD-käskyjen avulla pystytään hyödyntämään prosessorien mahdollisuutta käsitellä useita data-alkioita, kuten vektoreita, yhdellä konekäskyllä. SIMD-käskyjen hyödyt tulevat esille, kun jokin keskeinen algoritmi täytyy saada mahdollisimman nopeaksi.

Yleensä JavaScript-ohjelmat käyttävät vain yhtä säiettä, joten ne eivät hyödynnä moniytimisiä prosessoreja kovin hyvin. Vaikka kielessä on jo Web Worker -rajapinta, joka mahdollistaa ohjelman jakamisen rinnakkaisiin tehtäviin. Niiden välinen kommunikaatio tapahtuu viestinvälityksellä, joka on melko hidasta. Tukea rinnakkaisohjelmoinnille halutaan parantaa tuomalla muun muassa *SharedArrayBuffer*-rajapinta, eli jaettu taulukkopuskuri, ja atomiset operaatiot. Nämä yhdessä mahdollistavat matalan tason rinnakkaisohjelmoinnin, joka hyödyttää varsinkin raskasta laskentaa vaativia sovelluksia kuten pelejä.

Asm.js alkoi kokeellisena toteutuksena, mutta nyt selainvalmistajat ja standardoijat kehittävät yhdessä virallista Webin konekieltä, jota he kutsuvat nimellä WebAssembly [6]. Sen on tarkoitus tarjota matalan tason binääriformaatti, jota kääntäjät voivat tuottaa. Koska WebAssembly on tiiviissä binäärimuodossa, sitä ei tarvitse pakata ja purkaa, kuten tavallista JavaScript-koodia verkon käytön optimoimiseksi. Sitä ei myöskään tarvitse jäsentää uudelleen selaimessa. Jos selain ei tue WebAssemblyä, se on mahdollista muuntaa kirjaston avulla selaimessa asm.js-tyyliseksi, joka on toimivaa JavaScript-koodia. WebAssemblyn tavoite ei ole korvata JavaScript-koodia ja nykyistä kehitystapaa, vaan tarjota parempi tuki myös käännetyille ohjelmille, jotka aikaisemmin ovat toimineet helposti haavoittuvina selainlaajennuksina.

6 Yhteenveto

JavaScript on löytänyt tiensä monille eri alustoille ja se on kasvattanut suosiotaan kehittäjien keskuudessa. Kukaan tuskin osasi ennustaa JavaScriptin tulevaisuutta, kun se luotiin. Se on käytännössä korvannut Javan ja muut laajennuksiin perustuvat kielet selaimista. Tämä ei olisi mahdollista ilman virtuaalikonekehittäjien panosta suorituskyvyn parantamiseksi.

Googlen innovatiivinen työ V8:n kanssa on kannustanut muita virtuaalikoneiden kehittäjiä parantamaan virtuaalikoneidensa suorituskyyä. Siirtyminen pelkästä tulkista useisiin JIT-kääntäjiin on parantanut suorituskyyä huomattavasti aikaisempaan arkkitehtuuriin verrattuna.

Tämänhetkisten optimointimenetelmien riippuvuus staattisesta käytöksestä vähentää niiden hyödyllisyyttä todellisissa sovelluksissa. Ainakin Google kertoo siirtävänsä huomionsa raa'asta suorituskyyvystä yleisiin käyttötapauksiin ja sovelluskehyksiin. Onkin tärkeää opetella käyttämään työkaluja, joilla oman sovelluksen suorituskyyä voi mitata, sen sijaan, että opettelisi ulkoa optimointikikkoja. Virtuaalikoneet muuttuvat niin nopeaa tahtia, että se mikä vielä eilen oli hidasta voi olla huomenna jo nopeaa.

Virtuaalikoneiden kehittäjät julkaisevat jatkuvasti blogiviestejä uusista ominaisuuksista ja optimointimenetelmistä. Microsoft kertoi juuri avaavansa oman toteutuksensa avoimeksi lähdekoodiksi ja näin auttaa kaikkia toteutuksia jakamalla ideoitaan. JavaScriptin tulevaisuus vaikuttaa kirkkaalta ja Brendan Eichin, JavaScriptin luoja, sanoihin on hyvä päättää: "Always bet on JS" [5].

Lähteet

1. Ahn, W., Choi, J., Shull, T., Garzarán, M.J. ja Torrellas, J.: *Improving JavaScript Performance by Deconstructing the Type System*. SIGPLAN Not., 49(6):496–507, kesäkuu 2014, ISSN 0362-1340. <http://doi.acm.org/10.1145/2666356.2594332>.
2. ECMA International: *ECMAScript® 2015 language specification*., kesäkuu 2015. <http://www.ecma-international.org/ecma-262/6.0/>.
3. Egorov, Vyacheslav ja Corry, Erik: *A game changer for interactive performance*. <https://blog.chromium.org/2011/11/game-changer-for-interactive.html>, vierailtu 22.11.2015.
4. Eich, B.: *New JavaScript engine module owner*. <https://brendaneich.com/2011/06/new-javascript-engine-module-owner/>, vierailtu 3.10.2015.
5. Eich, Brendan: *Always bet on JS*. <http://alwaysbetonjs.com>, vierailtu 7.12.2015.
6. Eich, Brendan: *From ASM.JS to WebAssembly*. <https://brendaneich.com/2015/06/from-asm-js-to-webassembly/>, vierailtu 5.12.2015.
7. Fink, Stephen J. ja Qian, Feng: *Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement*. Teoksessa *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, sivut 241–252, Washington, DC, USA, 2003. IEEE Computer Society, ISBN 0-7695-1913-X. <http://dl.acm.org/citation.cfm?id=776261.776288>.

8. GitHub: *Atom — A hackable text editor for the 21st century*. <https://atom.io/>, vierailtu 16.9.2015.
9. Goethe University Frankfurt: *SEPL Goethe University Frankfurt — YouTube*. https://www.youtube.com/channel/UCpSoGwyH5yHHvQut3x6c_2g, vierailtu 28.11.2015.
10. Google: *Chrome V8 design elements*. <https://developers.google.com/v8/design>, vierailtu 1.10.2015.
11. Google: *Ignition: V8 Interpreter*. <https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMhOuouUZLdyrtmMoL44/edit>, vierailtu 1.12.2015.
12. Google: *Google Chrome: a new take on the browser*. Press Release. Google Inc, 2008. http://googlepress.blogspot.fi/2008/09/google-chrome-new-take-on-browser_02.html, vierailtu 4.10.2015.
13. Miadowicz, Andrew: *Advances in JavaScript Performance in IE10 and Windows 8*. <http://blogs.msdn.com/b/ie/archive/2012/06/13/advances-in-javascript-performance-in-ie10-and-windows-8.aspx>, vierailtu 23.11.2015.
14. Microsoft: *Microsoft Edge's JavaScript engine to go open-source*. <https://blogs.windows.com/msedgedev/2015/12/05/open-source-chakra-core/>, vierailtu 6.12.2015.
15. Mozilla: *asm.js*. <http://asmjs.org/>, vierailtu 4.12.2015.
16. Mozilla: *SpiderMonkey Internals*. <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals>, vierailtu 3.10.2015.

17. Paolini, G: *Netscape and Sun announce JavaScript, the open cross-platform object scripting language for enterprise networks and the Internet*. Press Release. Sun Microsystems Inc, 1995.
18. Payer, Hannes ja McIlroy, Ross: *Getting Garbage Collection for Free*. <http://v8project.blogspot.fi/2015/08/getting-garbage-collection-for-free.html>, vierailtu 22.11.2015.
19. Pizlo, Filip: *Introducing the WebKit FTL JIT*. <https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/>, vierailtu 7.12.2015.
20. Ravenbrook Limited.: *Bibliography – Memory Management Reference 4.0 documentation*. <http://www.memorymanagement.org/bib.html>, vierailtu 22.11.2015.
21. Richards, Gregor, Lebesne, Sylvain, Burg, Brian ja Vitek, Jan: *An Analysis of the Dynamic Behavior of JavaScript Programs*. Teoksessa *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10*, sivut 1–12, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0019-3. <http://dl.acm.org/citation.cfm?doid=1806596.1806598>.
22. Smith, J.E. ja Nair, R.: *The architecture of virtual machines*. Computer, 38(5):32–38, May 2005, ISSN 0018-9162.
23. Terlson, Brian: *Chakra: The JavaScript Engine that powers Microsoft Edge*. <https://channel9.msdn.com/Events/WebPlatformSummit/2015/Chakra-The-JavaScript-Engine-that-powers-Microsoft-Edge>, vierailtu 2.12.2015.
24. Titzer, Ben L.: *Revving up JavaScript performance with TurboFan*. <https://blog.chromium.org/2015/07/>

- revving-up-javascript-performance-with.html, vierailtu
16.10.2015.
25. Verschore, Hannes: *The monkeys in 2013*. <https://blog.mozilla.org/javascript/2014/01/23/the-monkeys-in-2013/>, vierailtu
4.12.2015.
26. Vijayan, Kannan: *The Baseline Compiler Has Landed*.
[https://blog.mozilla.org/javascript/2013/04/05/
the-baseline-compiler-has-landed/](https://blog.mozilla.org/javascript/2013/04/05/the-baseline-compiler-has-landed/), vierailtu 4.12.2015.
27. Wingo, A.: *inside javascriptcore's low-level interpreter*. [https://wingolog.org/archives/2012/06/27/
inside-javascriptcores-low-level-interpreter](https://wingolog.org/archives/2012/06/27/inside-javascriptcores-low-level-interpreter), vierailtu
1.12.2015.
28. Wingo, A.: *JavaScriptCore, the WebKit JS implementation*. [https://wingolog.org/archives/2011/10/28/
javascriptcore-the-webkit-js-implementation](https://wingolog.org/archives/2011/10/28/javascriptcore-the-webkit-js-implementation), vierailtu
5.10.2015.
29. Wingo, A.: *on-stack replacement in v8*. [https://wingolog.org/
archives/2011/06/20/on-stack-replacement-in-v8](https://wingolog.org/archives/2011/06/20/on-stack-replacement-in-v8), vierailtu
28.11.2015.
30. Wingo, A.: *V8: a tale of two compilers*. [https://wingolog.org/
archives/2011/07/05/v8-a-tale-of-two-compilers](https://wingolog.org/archives/2011/07/05/v8-a-tale-of-two-compilers), vierailtu
4.10.2015.