JavaScript ja virtuaalikoneet

Ville Lahdenvuo

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 7. joulukuuta 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

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	7. joulukuuta 2015	22

Tiivistelmä — Referat — Abstract

JavaScript-virtuaalikoneet ovat perinteisesti toimineet tulkkaamalla lähdekoodista muodostettua abstraktia syntaksipuuta tai käännettyä tavukoodia. Virtuaalikoneissa on otettu käyttöön monenlaisia JIT-kääntäjiä ja joissain tapauksissa tulkki on jätetty kokonaan pois. JavaScriptin dynaamisuuden takia virtuaalikoneiden kehittäjät ovat joutuneet toteuttamaan monimutkaisia menetelmiä tehokkaan konekoodin tuottamiseksi.

Monet optimoinnit käyttävät hyväksi oletusta, että sovellukset käyttäytyvät melko staattisesti, kielen dynaamisuudesta huolimatta. Tämän oletuksen nojalla on pystytty hyödyntämään optimointimenetelmiä, joita tyypillisesti käytetään staattisesti tyypitettyjen kielten kanssa.

Tutkimuksissa on kuitenkin havaittu, että oletus staattisesta käytöksestä saattaa olla virheellinen. Vaikka yleisesti käytössä olevat suorituskykytestit käyttäytyvät varsin staattisesti, todelliset sovellukset hyödyntävät kielen dynaamisuutta enemmän, mikä vähentää optimoinneista saatavia hyötyjä.

ACM Computing Classification System (CCS):
Software and its engineering → Virtual machines
Software and its engineering → Very high level languages

Avainsanat — Nyckelord — Keywords
JavaScript, virtuaalikone, suorituskyky

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

JavaScript on ohjelmointikieli, joka suunniteltiin ensisijaisesti verkkosivujen tekijöille. Sen tavoite oli täydentää Java-ohjelmointikieltä ja HTML-merkkauskieltä. 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 kuluttajatietokoneissa 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 tyypitettyjä, mikä helpottaa ohjelmakoodin kirjoittamista. Dynaamisuudesta seuraa kuitenkin myös ongelmia, sillä virtuaalikoneiden on vaikea ennustaa dynaamisia muutoksia ja tehdä järkeviä optimointeja, joilla suorituskykyä voitaisiin parantaa [1, s. 497].

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

konekoodia. Optimoitavuus edellyttää, että ohjelmoija tietää miten asiat kannattaa toteuttaa. Jos hyödyntää dynaamisuutta liikaa, voi helposti tehdä koodia, jota virtuaalikone ei osaa vielä 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 suorituskyky [1, s. 497].

JavaScriptin standardoiminen, sen käytön lisääntyminen ja selainvalmistajien keskinäinen kilpailu suorituskyvystä on parantanut kielen asemaa ja mainetta. JavaScriptin rooli on muuttunut skriptikielestä yleiskäyttöiseksi ohjelmointikieleksi [2, luku 4.]. Kielen tulevaisuus näyttää lupaavalta. Siihen on tullut paljon ohjelmointia helpottavia ominaisuuksia ja tapoja välttää yleisiä ”sudenkuoppia”, joihin varsinkin aloittelevat ohjelmoijat usein törmäävät.

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 isännän 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 vaan yhden prosessin suorittamista varten tarvittavat palvelut. Tässä tutkielmassa virtuaalikoneella tarkoitetaan JavaScript-ohjelmia suorittavaa prosessivirtuaalikonetta.

Yksi suurimmista virtuaalikoneiden hyödyistä on, että ohjelma tarvitsee kääntää usean alustan sijaan yhdelle 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 helpompi 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 sitä ylläpitää Mozilla ja sitä käytetään muun muassa Mozillan Firefox-selaimessa. Nykyinen SpiderMonkey on kehittynyt paljon, mutta se koostuu yhä kolmesta fundamentaalisesta komponentista: *kääntäjä*, *tulkki* ja *roskienkerääjä* [16]. Tämän arkkitehtuurin lisäksi on toteutettu myös muita menetelmiä suorituskyvyn parantamiseksi.

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 heti, vaan koodilohkoja käännetään osissa sitä mukaa kun suoritus etenee. Esimerkiksi suuret kirjastot eivät siis kasvata käännösaikaa, sillä niistä käännetään vain ne osat, joita oikeasti käytetään.

Tulkin tehtävä on suorittaa ohjelmaa. Tulkki siis lukee tavukoodia käsky kerrallaan tai käy läpi abstraktia syntaksipuuta ja kutsuu tarvittavia palveluja isäntäjärjestelmästään. Tulkista 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 huomaamattaan jättää viittauksia olioihin, joita ei enää käytä.

2.3 Suoritusaikainen kääntäminen

Riippumatta toteutustavasta tulkki joutuu aina tekemään useita konekäskyjä suorittaakseen yhden tavukoodikäskyn [22, s. 35]. Tämä johtuu siitä, että tavukoodi toimii ohjelmistotasolla, kun konekoodi suoritetaan suoraan laitteistolla. Valitettavasti 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 se hidastaisi ohjelman käynnistystä. 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 *suoritusaikaisella 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 suorituskykyisempää konekoodia.

2.4 Optimoinnin ongelmat

Dynaamisesti tyypitetyllä ohjelmointikielellä toteutetusta ohjelmasta generoitu konekoodi vaatii paljon tyyppitarkastuksia ja poikkeustapauksia muuttujien tyypeille. On kuitenkin tehty oletuksia ohjelmien käyttäytymisestä. Esimerkiksi oletetaan, 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]. Tämän 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 sekä Googlen V8:n innoittamana 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 animaation 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 erityspiirteiden toteuttamiseen. Suuri osa menetelmistä hyödyntää virtuaalikoneen suoritusaikana keräämää profilointitietoa. Frankfurtin Goethe-yliopiston YouTube-kanavalta löytyy videoluentoja, joissa selitetään juurta jaksain 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 kuitenkin muokkaamaan epästandardin `__proto__`-ominaisuuden kautta tai standardiin myöhemmin 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.

V8:n kehittäjät esittelivät ominaisuuden nimeltään *piiloluokka* (hidden

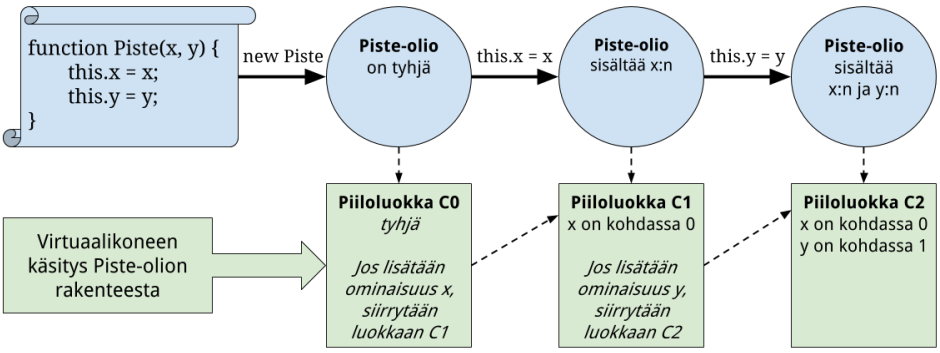
class) [10] ja sittemmin myös muita virtuaalikoneiden kehittäjiä on ottanut samanlaisen mallin käyttöön. Piiloluokat ovat *muuttumattomia* (immutable) luokkia, joita virtuaalikone käyttää JavaScript-olioiden rakenteen kuvaamiseen. Piiloluokka on tavallaan siis olion tyyppi, vaikka JavaScriptissä olioilla ei ole varsinaisesti tyyppejä, paitsi kieleen sisäänrakennetut perustyypit, kuten `String` ja `Number`. Käyttämällä piiloluokkaa olion tyyppinä virtuaalikoneet voivat hyödyntää tehokkaita kääntämistekniikoita, joita käytetään yleisesti staattisesti tyypitetyissä kielissä.

Piiloluokat toimivat siten, että oliota luodessa ensimmäistä kertaa, virtuaalikone luo sille samalla piiloluokan. Aina kun oliota muutetaan lisäämällä siihen ominaisuus, 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 1 näkyy, miten V8 luo piiloluokkia luodessaan Piste-olion ensimmäisen kerran. Jokainen kaavion askel vastaa yhden koodirivin suorittamista. 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 olioon liitetään ominaisuus `x`, jolloin virtuaalikone luo uuden piiloluokan `C1` ja lopulta `y`:n lisäyksen jälkeen `C2`:n. Piiloluokkiin tallennetaan myös tieto mihin luokkaan pitää siirtyä tietyn ominaisuuden lisäämisen jälkeen sekä viittaus olion prototyyppiin.

Piiloluokkia ei tarvitse luoda uudestaan joka kerta, kun halutaan luoda uusi Piste-olio. Riittää, että seurataan niihin sisällytettyjä 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, virtu-



Kuva 1: Esimerkki piiloluokkien toimintaperiaatteesta.

aalikone 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 Pisteen, jolla on tasokoordinaattien lisäksi `z`-ominaisuus. Tällöin piiloluokka `C2` sisältää tiedon siirtymästä molemmissa, ympyrän ja kolmiulotteisen pisteen, tapauksissa.

Wonsun Ahn kumppaneineen on tutkinut piiloluokkien hyötyjä ja haittoja todellisissa verkkosovelluksissa [1]. Tutkimuksessa on löytynyt muutamia heikkouksia, jotka vähentävät piiloluokkien hyötyjä todellisissa sovelluksissa ja heikkouksiin ehdotetaan parannuksia. Siinä myös moititaan kehittäjien 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 optimointien,

kuten sisällytetyn välimuistin, toimivuutta.

3.2 Sisällytetty välimuisti

Hakupaikaksi (access site) kutsutaan kohtaa koodissa, jossa viitataan olion ominaisuuteen. Koska muuttujilla ei ole tyyppiä, virtuaalikone ei voi tietää ennalta löytyykö edes kyseistä ominaisuutta. Yleinen olettamus on, että samalla hakupaikalla oliot ovat usein samantyyppisiä. 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 tässä hakupaikassa `obj` on 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 (haePiiloluokka(obj) == 'C2') return obj[C2_X_OFFSET];
2 else return teeHidasHaku(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 olioon 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`-funktiota ympyräolioilla, V8 voi lisätä sisällytettyyn välimuistiin tarkistuksen: ”Onko `obj`:n piiloluokka `C3`?”. Tällöin arvon voi ladata taas nopeasti tunnetusta siirtymästä pseudokoodilla:

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

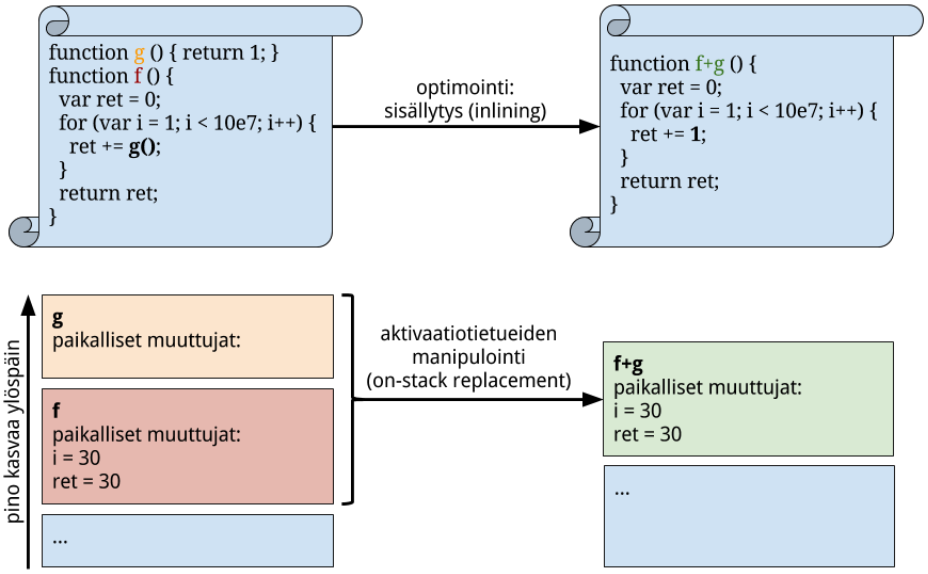
Ylimääräiset tarkistukset tekevät suorituksesta tietenkin hitaampaa. Tutkimuksen mukaan suoritus välimuistin avulla voi olla parhaimmillaan monta kertaluokkaa nopeampaa kuin ilman välimuistia [1, s. 498], joten tulokset tukevat sisällytetyn välimuistin käyttöä.

3.3 Aktivaatiotietueiden 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.

Tätä varten on kehitetty menetelmä, jonka englanninkielinen nimi on ”on-stack replacement” [7]. Suomeksi menetelmää voi kutsua aktivaatiotietueiden manipuloinniksi. Kuvassa 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 aikaa ja funktio pitäisi optimoida. Virtuaalikoneen optimointiheuristiikka päättää sisällyttää funktion `g` funktion `f` sisään eliminoiden turhan funktiokutsun. Esimerkissä tämä tapahtuu 30. toiston kohdalla.



Kuva 2: Esimerkki aktivaatiotietueiden manipuloinnista [29].

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` aktivaatiotietuetta pinossa, siten että se sisältää myös funktion `g` mahdolliset paikalliset muuttujat. Tässä esimerkissä sellaisia ei ole, mutta todellisessa tapauksessa virtuaalikoneen täytyy luoda kuvaus molemmista aktivaatiotietueista yhdistetyn funktion `f+g` aktivaatiotietueeksi.

Aktivaatiotietueiden manipuloinnin täytyy toimia myös toiseen suuntaan, eli jos koodissa tulee vastaan tapaus, jota optimointi ei ottanut huomioon, täytyy optimointi pystyä peruuttamaan, jolloin aktivaatiotietue pitää pilkkoa osiin taas. Menetelmään liittyy paljon teknisiä yksityiskohtia, kuten miten pysäyttää funktion suoritus ja miten manipuloida aktivaatiotietueita oikein. Tästä huolimatta suurin 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, ja koko muistialueen käsitteleminen on hidasta. Tämä on suuri ongelma varsinkin animaatioiden ja interaktiivisten sovellusten tapauksessa, sillä 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 tärkeimpiä menetelmiä ja ideoita, 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 korkealla tasolla tekee koodille suorittaakseen sitä. Virtuaalikoneet eivät ole hylänneet tulkkejaan, vaikka niihin on lisätty JIT-kääntäjiä. Esimerkiksi JavaScriptCoren tulkki on kirjoitettu kokonaan uudelleen paljon edeltäjäänsä 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. 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. Uusimmassa Chakra-virtuaalikoneessa 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 aktivaatiotietueiden manipulointia, joten optimoitu koodi otetaan käyttöön vasta kun funktiota kutsutaan uudestaan. Chakran kehittäjät ovat pyrkineet hyödyntämään laitteistoa mahdollisimman paljon rinnakkaistamalla käännös- ja roskienkeruuoperaatioita. Myös V8-virtuaalikone suorittaa käännösprosessia rinnakkaisesti suorituksen kanssa.

SpiderMonkeyn kehittäjät ovat olleet myös ahkeria ja virtuaalikone on nähnyt jo viisi erilaista JIT-kääntäjää: TraceMonkey, JägerMonkey, IonMonkey, OdinMonkey ja alkutilannekääntäjä (Baseline compiler) [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 ainut 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 taustalla ja päättelevät 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.

Myös Chakra käyttää sisällytettyjä välimuisteja, mutta 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, vaikka niillä on eri piiloluokat, koska niiden rakenne on tarpeeksi lähellä toisiaan eli tässä käyttötapauksessa ne ovat ekvivalentit tyypeiltään, koska molemmilla on ominaisuus `x` samassa siirtymässä.

5 Tulevaisuus

JavaScriptistä on tullut, jo kliseen omaisesti, Webin konekieli [6]. JavaScriptiä 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 kääntäjäohjelmoijien hyödynnettäväksi.

Asm.js [15] on epävirallinen standardi osajoukosta JavaScriptiä, joka on mahdollista kääntää tehokkaaksi konekoodiksi. Sen idea on olla toimivaa JavaScriptia, mutta mahdollistaa tehokas kääntäminen konekoodiksi muun muassa tyyppivinkeillä. Esimerkiksi kokonaislukuparametri merkitään tekemällä funktion alussa bittitason operaatio: `myIntParam = myIntParam|0`, joka pakottaa muuttujan arvon kokonaisluvuksi. Myös V8-virtuaalikoneen kehittäjät suunnittelevat tukea asm.js-muotoisen koodin optimoinnille uuden TurboFan JIT-kääntäjän avulla.

Asm.js:n tueksi JavaScriptiin on tuotu lisää suorituskykyä parantavia toimintoja, kuten *SIMD-käskyt*. SIMD tulee sanoista *Single Instruction Multiple Data*, joka tarkoittaa suomeksi: ”Yksi käsky, monta data-alkiota”. SIMD-käskyjen avulla pystyy hyödyntämään prosessorien mahdollisuutta käsitellä monta data-alkiota yhdellä konekäskyllä.

Lisäksi suunnitteilla on parantaa tukea rinnakkaisohjelmoinnille. Kielessä on jo Web Worker -rajapinta, joka mahdollistaa ohjelman jakamisen rinnakkaisiin ”työläisiin”. Työläisten kommunikaatio tapahtuu viestinvälityksellä, joka on melko hidasta. Tämä vuoksi kieleen ollaan tuomassa *SharedArrayBuffer*-rajapinta, eli jaettu taulukkopuskuri, ja atomiset operaatiot.

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. WebAssembly on tiiviissä

binäärimuodossa, joten sitä ei tarvitse purkaa, kuten pakattua JavaScript-koodia, eikä jäsentää uudelleen selaimessa. WebAssemblyn tavoite ei ole korvata JavaScript-koodia ja nykyistä kehitystapaa, vaan tarjota parempi tuki myös käännetyille ohjelmille, jotka aikaisemmin ovat toimineet epäturvallisina 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 suorituskykyä. Siirtyminen pelkästä tulkista useisiin JIT-kääntäjiin on parantanut suorituskykyä huomattavasti aikaisempaan arkkitehtuuriin verrattuna.

Tämänhetkisten optimointimenetelmien riippuvuus staattisesta käytöksestä vähentää niiden hyödyllisyyttä todellisissa sovelluksissa. Ainakin Google kertoo siirtävänsä huomionsa raa'asta suorituskyvystä yleisiin käyttötapauksiin ja sovelluskehyksiin. Onkin tärkeää opetella käyttämään työkaluja, joilla oman sovelluksen suorituskykyä 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 luojan, 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.libproxy.helsinki.fi/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/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/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, Lebresne, 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://doi.acm.org.libproxy.helsinki.fi/10.1145/1806596.1806598`.

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/, 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, vierailtu 1.12.2015.

28. Wingo, A.: *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, 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, vierailtu 4.10.2015.