JavaScript ja virtuaalikoneet

Ville Lahdenvuo

Aine
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 16. lokakuuta 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
Aine	16. lokakuuta 2015	12

Tiivistelmä — Referat — Abstract

JavaScript-virtuaalikoneet ovat perinteisesti toimineet kääntämällä ohjelman tavukoodiksi ja tulkkaamalla sitä. Modernit virtuaalikoneet ovat ottaneet käyttöön erilaisia suoritusaikaisia kääntäjiä eli JIT-kääntäjiä tai jättäneet tulkin kokonaan pois. JavaScriptin dynaamisuuden takia virtuaalikoneiden kehittäjät ovat joutuneet toteuttamaan monimutkaisia menetelmiä tehokkaan konekoodin tuottamiseksi käyttäen hyväksi oletusta, että sovellukset käyttäytyvät melko staattisesti, dynaamisesta kielestä huolimatta.

Automaattista muistinhallintaa on jouduttu parantamaan sovellusten vaatimusten kasvaessa. Selaimet eivät voi pysäyttää ohjelman suoritusta useaksi sadaksi millisekunniksi roskienkeräyksen takia, sillä tämä huonontaa käyttökokemusta varsinkin interaktiivisissa sovelluksissa.

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 [9].

Selainten suosio sovellusalustana on kasvanut ja sen ansiosta JavaScriptin käyttö on lisääntynyt. Kasvua siivittää se, että 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 standardisoitu. 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 [4].

JavaScript on dynaaminen oliopohjainen kieli, mutta se tukee myös imperatiivista ja funktionaalista ohjelmointityyliä. JavaScript tarjoaa siis monia tapoja toteuttaa samoja toiminnallisuuksia [2, 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].

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

asiat kannattaa toteuttaa. Jos hyödyntää dynaamisuutta liikaa, voi helposti tehdä koodia, jota virtuaalikone ei pysty optimoimaan.

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

JavaScriptin standardisoiminen, sen käytön lisääntyminen ja selainvalmistajien keskinäinen kilpailu suorituskyvystä on parantanut kielen asemaa ja mainetta. JavaScriptin rooli on muuttunut skriptikielestä yleiskäyttöiseksi ohjelmointikieleksi [2, 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 optista asemaa käyttämällä isännän tiedostojärjestelmää hyväksi, jolloin virtuaalikoneessa suoritettava ohjelma luulee esimerkiksi lukevansa optista levyä, kun todellisuudessa tieto tulee kiintolevyltä.

2.1 Kahdenlaisia virtuaalikoneita

Virtuaalikoneita on kahdenlaisia, *järjestelmä-* ja *prosessivirtuaalikoneita* [10, s. 33]. Järjestelmävirtuaalikone tarjoaa kokonaisen käyttöjärjestelmän palvelut toisin kuin prosessivirtuaalikone, joka tarjoaa vaan yhden prosessin suorittamista varten tarvittavat palvelut. Tässä tutkielmassa virtuaalikoneella

tarkoitetaan JavaScriptillä toteutettuja ohjelmia suorittavaa prosessivirtuaalikonetta.

Yksi suurimmista virtuaalikoneiden höydyistä on se, että ohjelma tarvitsee kääntää usean alustan sijaan yhdelle virtuaalikoneelle, jolloin se 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 [10, s. 36]. Pahantahtoisen ohjelman on siis löydettävä haavoittuvuus sekä virtuaalikoneesta että isännästä voidakseen aiheuttaa ongelmia.

2.2 JavaScript-virtuaalikoneen anatomia

Ensimmäisen JavaScript-virtuaalikoneen nimi on SpiderMonkey [3]. 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* [8]. Tämän arkkitehtuurin lisäksi on kuitenkin toteutettu muita järjestelmiä suorituskyvyn parantamiseksi.

Kääntäjä huolehtii koodin *jäsentämisestä* (parsing) ja kääntämisestä *tavukoodiksi*. Virtuaalikoneen tavukoodi on verrattavissa todellisen koneen konekoodiin. Tavukoodia on helpompi käsitellä ohjelmallisesti kuin tekstimuotoista ohjelmakoodia, sillä se on yksinkertaisempaa syntaksiltaan, mutta runsassanaisempaa.

Tulkin tehtävä on suorittaa tavukoodia. Tulkki siis lukee tavukoodia käsky kerrallaan ja kutsuu tarvittavia palveluja omassa koodissaan sekä isäntäjärjestelmässään. Tulkista on siis oltava oma versionsa jokaista eri alustaa varten. Yksi tulkin eduista on, että se on joustavampi kuin todellinen prosessori ja siihen voi toteuttaa monimutkaisempia tavukoodikäskyjä.

Roskienkerääjän tehtävä on yksinkertaisesti poistaa muistista muuttujat ja oliot, joihin ohjelmassa ei enää viitata. Roskienkerääjän ansiosta ohjelmoijan ei tarvitse vapauttaa muistia itse, vaan järjestelmä hoitaa muistinhallinnan automaattisesti. Automaattinen muistinhallinta vähentää virheiden määrää, kuten muistivuotoja, mutta ei poista kaikkia ongelmia. Muistivuodot ovat silti mahdollisia.

2.3 Suoritusaikainen kääntäminen

Valitettavasti tulkit ovat hitaita, tai ainakin hitaampia kuin haluttaisiin. Riippumatta toteutustavasta, tulkki joutuu aina tekemään useita konekäskyjä yhden tavukoodikäskyn suorittamiseksi [10, s. 35]

Vuonna 2008 Google julkaisi uuden selaimen, Google Chromen, jonka oli tarkoitus parantaa verkkosovellusten käyttökokemusta [7]. Googlen kiinnostus käyttökokemuksen ja ennen kaikkea suorituskyvyn parantamisesta on ymmärrettävää, sillä yhtiöllä on paljon verkkopalveluita, jotka hyötyvät hyvästä suorituskyvystä. Näistä syistä Google päätyi toteuttamaan Chrome-selaintaan varten oman virtuaalikoneen nimeltään V8.

Mielenkiintoisen V8:sta tekee se, että siinä ei ole lainkaan tulkkia. Sen sijaan V8 kääntää koodin nopeasti suoraan konekoodiksi ennen suorittamista niin sanotulla *lähtötilannekääntäjällä* (baseline compiler) [5]. Nopean kääntämisen saavuttamiseksi se ei tee monimutkaisia optimointeja vielä tässä ensimmäisessä käännösvaiheessa.

V8:n innoittamana muut virtuaalikonetoteutukset ovat muuttaneet toimintaansa siten, että tulkkia käytetään vain suorituksen alkuvaiheessa ja koodi 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 erillistä optimoivaa JIT-kääntäjää, joka on hitaampi kuin lähtöti-

lannekääntäjä, mutta tuottaa suorituskykyisempää konekoodia. JavaScript-koodin optimointi osoittautuu kuitenkin hankalaksi.

2.4 Optimoinnin ongelmat

Dynaamisesti tyypitetyllä ohjelmointikielellä toteutetusta ohjelmasta generoitu konekoodi vaatii paljon tyyppitarkastuksia ja poikkeustapauksia muuttujien tyypeille. On kuitenkin huomattu, että ohjelmat käyttäytyvät melko ennustettavalla tavalla. Etenkin usein kutsuttuja funktiota kutsutaan usein samankaltaisilla parametreilla.

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 se suoritetaan vain muutamia kertoja.

Virtuaalikoneet keräävät tietoa ohjelman käyttäytymisestä sen suorituksen aikana. Tiedon kerääminen hoidetaan usein tulkissa ja V8:n tapauksessa lähtötilannekääntäjä lisää generoituun konekoodiin käskyjä keräämään tietoa ohjelman käyttämistä tyypeistä [12]. Tämän profiloinnin ansiosta on helpompi 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 he kutsuvat nimellä *DFG-JIT*. Lyhenne DFG, joka tulee sanoista *Data Flow Graph*, kuvaa ohjelman suoritusaikaisen tyyppitiedon tallentavaa tietorakennetta. Siis muiden virtuaalikoneiden tapaan JavaScriptCore kerää ensin tyyppitietoa ja generoi sen avulla optimoitua konekoodia [11].

Automaattinen roskienkeräys on todella kätevä toiminnallisuus ohjelmoijan kannalta, mutta sen toteuttaminen hyvin on haastavaa. 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

Konekoodiksi kääntäminen tuo ongelmaksi hitaan käännösvaiheen. Tämän takia modernit virtuaalikoneet sisältävät useamman kuin yhden kerroksen JIT-kääntäjiä eritasoisilla optimoinneilla. Tässä luvussa esitellään muutamia keinoja, joilla JavaScript-virtuaalikoneet ovat parantaneet tuottamansa konekoodin suorituskykyä.

3.1 Piiloluokat

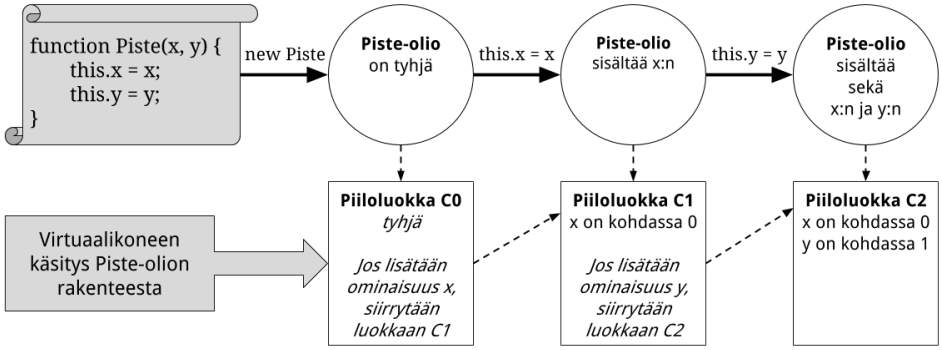
JavaScriptin oliot käyttävät prototyyppiperintää. Tämä tarkoittaa, että oliolla voi olla toinen olio prototyyppinään, jolla on toinen olio prototyyppinään ja niin edelleen. Kun oliolta pyydetään jonkin *ominaisuuden* (property) arvoa, etsitään sitä ensin itse oliosta. Jos sitä ei löydy, käydään olion prototyyppiketjua läpi, kunnes ominaisuus löytyy tai kaikki prototyypit on käyty läpi, jolloin arvo on ”undefined“. Tämä tarkoittaa, että ominaisuuksien arvot saattavat olla muistissa kaukana toisistaan ja niiden löytämiseen tarvitaan hidasta assosiatiivista hakurakennetta.

V8 esitteli ominaisuuden nimeltään *piiloluokka* (hidden class) [5]. Piiloluokat ovat staattisia eli *muuttumattomia* (immutable) luokkia, joita virtuaalikone käyttää dynaamisten JavaScript-olioiden rakenteen kuvaamiseen. Piiloluokka on tavallaan siis olion tyyppi, vaikka JavaScriptissä muuttujilla ei ole tyyppejä. Olioita voi käyttää hyvin dynaamisesti, mutta virtuaalikone olettaa, että useimmissa tapauksissa niitä käytetään melko staattisesti. Käyttämällä piiloluokkaa ikään kuin olion tyyppinä virtuaalikoneet voivat

hyödyntää tehokkaita kääntämistekniikoita, joita käytetään staattisesti tyypitetyissä kielissä.

Tämä onnistuu siten, että kun olio luodaan, luodaan sille piiloluokka. Aina kun oliota muutetaan lisäämällä siihen ominaisuus, luodaan sitä vastaamaan uusi piiloluokka. Luokkaa ei voi tehdä etukäteen, koska dynaamisuuden takia on vaikea tietää ennalta millainen olio tulee rakenteeltaan olemaan.

Kuvassa 1 näkyy esimerkki Piste-nimisen olion ja sen piiloluokan muodostus. Kun konstruktorifunktiota kutsutaan, luodaan tyhjä olio, jota virtuaalikone kuvaa tyhjällä piiloluokalla **C0**. Luokkaa C0 käytetään kuvaamaan kaikkia tyhjiä olioita, joilla on sama prototyyppi. Konstruktorissa olioon liitetään ominaisuus x, jolloin virtuaalikone luo uuden piiloluokan **C1** ja lopulta y:n lisäyksen jälkeen **C2**:n. Piiloluokkiin tallennetaan tieto mihin luokkaan pitää siirtyä tietyn ominaisuuden lisäämisen jälkeen.



Kuva 1: Esimerkki piiloluokkien toimintaperiaatteesta.

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 tavallaan samantyyppisiä.

Kun Piste-oliolta pyydetään esimerkiksi ominaisuuden x arvoa, virtuaalikone tietää, että olio on luokan **C2** ”tyyppinen“ ja arvo löytyy yhdellä konekäskyllä muistipaikasta 0 ilman, että täytyy tehdä hidasta hakua asso-

siatiivisesta hakurakenteesta. Virtuaalikone voi siis generoida tehokkaampaa konekoodia ja muistiviittaukset nopeutuvat huomattavasti.

Jos oliota muokataan uudestaan luomisen jälkeen, luodaan sille uusi piiloluokka ja optimoinnit täytyy suorittaa tälle uudelle tyypille uudestaan. Jos kaikkia ominaisuuksia ei alusteta aina samassa järjestyksessä tai niitä alustetaan ehdollisesti, se tuottaa ongelmia piiloluokkien toiminnan kannalta. Ongelmaksi muodostuu myös se, että piiloluokkaan sisällytetään koko prototyyppiketju [1, s. 499], jolloin muutokset johonkin prototyyppiketjussa ylempänä olevaan objektiin pakottavat uuden piiloluokan luomisen.

3.2 Sisällytetty välimuisti

Yleinen olettamus on, että samalla *hakupaikalla* (access site) oliot ovat usein samantyyppisiä. Tämän takia on alettu käyttämään *sisällytettyä välimuistia* (inline cache) [1, s. 498]. Nimi tulee siitä, että välimuisti tulee osaksi generoitua konekoodia.

Esimerkiksi Piste-oliolla voi olla metodi getX, joka palauttaa x:n arvon. Keräämällä tietoa suoritusaikana virtuaalikone voi päätellä, että tässä kohtaa olio on usein Piste-olio. Virtuaalikone muokkaa konekoodia ja lisää tarkistuksen "onko olion piiloluokka sama kuin Piste-olion?". Jos oliolla on sama piiloluokka, x:n arvon lukemiseksi riittää katsoa piiloluokasta sen *siirtymä* (offset) muistissa ja suorittaa yksi muistioperaatio.

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 olion metodin kutsumista ja se on samankaltainen lataamisen kanssa, sillä ensin pitää ladata metodi, joka on JavaScriptissä viittaus funktio-olioon.

Jos välimuistista tulee huti, V8 lisää uuden olion piiloluokan tarkistuksen sisällytettyyn välimuistiin. Ylimääräiset tarkistukset tekevät koodista hitaampaa, mutta tutkimusten mukaan [1, s. 498] suoritus välimuistin avulla voi

olla parhaimmillaan monta suuruusluokkaa nopeampi kuin ilman välimuistia, joten tässä tapauksessa vaihtokauppa on kannattava.

3.3 Roskienkeräys

Roskienkeräys (garbage collection) on perinteisesti toiminut melko yksinkertaisella algoritmilla nimeltään *merkitse ja pyyhkäise* (mark-and-sweep). Se aloittaa merkitsemisvaiheella ja lähtee liikkeelle niin sanotusta juurisetistä olioita. Algoritmi seuraa muistiviitteitä merkiten vastaan tulleet oliot kunnes kaikki on käyty läpi. Sen jälkeen tulee pyyhkäisyvaihe. Algoritmi käy läpi koko muistin ja vapauttaa kaikki oliot, jotka eivät ole merkittyjä. Samalla algoritmi poistaa merkit seuraavaa kierrosta varten.

Ohjelman suoritus ei voi jatkua roskienkeräyksen aikana ja koko muistialueen merkitseminen ja läpikäyminen on hidasta. Tämä on suuri ongelma varsinkin animaatioiden ja interaktiivisten sovellusten tapauksessa, sillä pitkä roskienkeräystauko haittaa käytettävyyttä ja saa sovelluksen tuntumaan hitaalta vaikka muu suoritus olisikin todella nopeaa.

Ongelman ratkaisemiseksi on kehitetty erilaisia heuristiikkoja ja optimointeja. Esimerkiksi *sukupolvittainen roskienkeräys* (generational garbage collection) [5] perustuu oletukseen, että suurin osa luoduista muistivarauksista on lyhytaikaisia. Jakamalla muistialue kahteen erikokoiseen sukupolveen, nuoreen ja vanhaan, voidaan allokoida uudet oliot pienemmälle nuorelle alueelle. Nuoren alueen voi pienempänä käsitellä nopeammin ja useammin. Jos olio selviää nuorella alueella roskienkeräyksestä, se on todennäköisesti pitkäikäisempi olio ja se siirretään vanhalle alueelle, jota käsitellään harvemmin.

4 Yhteenveto

Vaikka JavaScriptin suunnittelijoilla ei voinut olla käsitystä mihin kaikkeen JavaScriptia tultaisiin käyttämään, he onnistuivat luomaan hitin. Alkuvai-

heessa kukaan ei varmaankaan osannut ennustaa kielen tulevaa menestystä, eikä menestys olisi ollut mahdollinen ilman toteuttajien innovaatioita.

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

Nykyinen trendi käyttää JIT-kääntäjiä ja olettaa ohjelmien staattinen käyttäytyminen voi kuitenkin olla huono idea pidemmällä tähtäimellä. Vaikka virtuaalikoneet näyttävät nopeilta suorituskykytesteissä, voi verkkopalveluiden todelliset käyttäytymismallit olla dynaamisempia. Korkean tason ohjelmointikielen ohjelmoijan ei pitäisi tarvita tietää virtuaalikoneen sisäisestä toteutuksesta pystyäkseen kirjoittamaan tehokasta koodia.

Virtuaalikoneiden kehittäjät tuntuvat jatkuvasti julkaisevan uusia viestejä blogeissaan, joissa he kertovat kuinka he ovat taas keksineet tai toteuttaneet uusia tapoja optimoida virtuaalikonettaan. Esimerkiksi V8:n kehittäjät kertovat blogissaan toteuttavansa uutta optimoivaa kääntäjää [6], joka pystyy optimoimaan enemmän erikoistapauksia kuin nykyinen ja mahdollistaa helpomman jatkokehityksen.

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. 6. versio*, kesäkuu 2015. `http://www.ecma-international.org/ecma-262/6.0/`.

3. Eich, B.: *New JavaScript engine module owner*. `https://brendaneich.com/2011/06/new-javascript-engine-module-owner/`, vierailtu 3.10.2015 .

4. GitHub: *Atom — A hackable text editor for the 21st century*. `https://atom.io/`, vierailtu 16.9.2015 .

5. Google: *Chrome V8 design elements*. `https://developers.google.com/v8/design`, vierailtu 1.10.2015 .

6. Google: *Revving up JavaScript performance with TurboFan*. `https://blog.chromium.org/2015/07/revving-up-javascript-performance-with.html`, vierailtu 16.10.2015 .

7. 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 .

8. Mozilla: *SpiderMonkey Internals*. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals`, vierailtu 3.10.2015 .

9. 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.

10. Smith, J.E. ja Nair, R.: *The architecture of virtual machines.* Computer, 38(5):32–38, May 2005, ISSN 0018-9162.

11. Wingo, A: *JavaScriptCore, the WebKit JS implementation.* `https://wingolog.org/archives/2011/10/28/javascriptcore-the-webkit-js-implementation`, vierailtu 5.10.2015 .

12. 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 .