TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

VELI-MATTI KARHULA

ROSKIENKERUUN KÄYTTÖ MODERNEISSA JÄRJESTELMISSÄ

Kandidaatintyö

Tarkastaja:
Yliopistonlehtori Terhi Kilamo

Jätetty tarkastettavaksi: 15.6.2018

SISÄLLYSLUETTELO

TIIVISTELMÄ

VELI-MATTI KARHULA: Roskienkeruun käyttö moderneissa järjestelmissä

Tampereen teknillinen yliopisto

Kandidaatintyö, 25 sivua

Kesäkuu, 2018

Tietotekniikan koulutusohjelma

Pääaine: Ohjelmistotuotanto

Tarkastajat: yliopistonlehtori Terhi Kilamo

Avainsanat: Roskienkeruu, Roskienkeruumenetelmät, Viittausten laskenta, Jäljittävät menetelmät, SSD-asema, Mark-sweep, Mark-copy

Opinnäytetyö tekee yleiskatsauksen ohjelmistotekniikan roskienkeruuseen ja sen nykytilanteeseen. Huomiota kiinnitetään erityisesti erilaisiin sovelluskohteisiin ja niiden asettamiin erityishaasteisiin muistinhallinnassa ja roskienkeruussa. Työ esittelee niin aihealueen peruskäsitteet ja -menetelmät, tekee tarkempia havaintoja toteutuksesta nykyisissä kielissä ja käyttöympäristöissä ja luo myös katsauksen tuoreimpiin tutkimuksiin.

Peruskäsitteissä esitellään roskienkeruun lähtökohdat ja viittausten laskentaan ja muistialueiden jäljittämiseen perustuvat menetelmät sekä joitakin niiden variaatioita. Tämän lisäksi tutustutaan kyseisten menetelmien vahvuuksiin, heikkouksiin ja ominaispiirteisiin.

Uudemmista sovelluskohteista käsitellään esimerkiksi SSD-asemien tallennustilan hallintaa ja roskienkeruun roolia aseman tehokkuudessa ja käyttöiässä sekä palvelukeskusten, sulautettujen järjestelmien ja mobiililaitteiden asettamia haasteita virrankulutuksessa ja muistinkäytössä. Tehdään myös lyhyt katsaus esimerkiksi roskienkeruuseen nykyisissä selainohjelmissa. Kielistä perehdytään tarkemmin esimerkiksi Javan ratkaisuihin ja tutustutaan myös viittausten laskentaa laajemmin käyttäviin kieliin. Myös Haskellin kaltaisiin laiskan suorituksen kieliin tutustutaan lyhyesti.

Tulevaisuuskatsauksessa tehdään havaintoja roskienkeruumenetelmien kehityksestä jatkossa niin ohjelmisto- kuin laitteistoteknillistenkin osalta. Havaintoja on tehty esimerkiksi International Symposium of Memory Managementissa esitellyistä tutkimuksista ja innovaatioista.

1. JOHDANTO

Yksi tietokoneiden keskeisimmistä komponenteista on muisti. Se toimii prosessorin apuna väliaikaisena tallennustilana, johon voidaan tallettaa ja josta voidaan hakea tietoa huomattavalla nopeudella.

Prosessorin käsitellessä tietoa, muistin tallennuskapasiteetin rajat tulevat vastaan ennemmin tai myöhemmin. Tällöin olennaiseksi muodostuu järjestelmän muistinhallinta ja erityisesti sen kyky vapauttaa muistia takaisin omaan käyttöönsä niiltä muistialueilta, joiden tieto ei enää ole tarpeellista. Tähän haasteeseen on ohjelmistotekniikassa pääsääntöisesti kaksi lähestymistapaa. Ensimmäisessä vaihtoehdossa ohjelmoija osoittaa itse muistista vapautettavat alueet suoraan tai välillisesti ohjelmointikielen avulla. Toisessa järjestelmä pyrkii itse tunnistamaan muistin osat, jotka voidaan käyttää uudelleen menettämättä mitään jatkokäytön kannalta olennaista tietoa. Näistä jälkimmäinen tunnetaan ohjelmistotekniikassa roskienkeruuna [38] ja se on alati kehittyvä ja uusia sovelluskohteita löytävä osa ohjelmistojen ja järjestelmien kehitystä.

Roskienkeruumenetelmät ovat mielenkiintoinen ohjelmistotekniikan osa-alue. Niiden asemaan ovat vaikuttaneet aikojen saatossa niin laitekantojen ja resurssimäärien muutokset, kuin myös uusien ohjelmointikielten tarpeet [10]. Sulautettujen järjestelmien [6], älykkäiden mobiililaitteiden [40] ja SSD-asemien [14] myötä myös roskienkeruun käyttökohteet ovat laajentuneet entisestään ja niille on asetettu uudenlaisia tavoitteita ja odotuksia.

Tässä opinnäytetyössä pyritään luomaan selkeä yleiskatsaus roskienkeruun lähtökohtiin, nykytilanteeseen ja tulevaisuuteen. Mikä on roskienkeruun asema nykyisissä käyttöympäristöissä ja mihin roskienkeruun kehitys suuntautuu jatkossa?

Opinnäytetyön luvussa 2 käydään roskienkeruun historiaa, peruskäsitteitä ja muodostetaan niiden pohjalta yleiskuva aihealueesta. Luvussa 3 käsitellään roskienkeruun asemaa nykyisessä tilanteessa, johon mobiililaitteet, SSD-asemat ja uudet korkeamman tason ohjelmointikielet ovat tuoneet uusia mahdollisuuksia ja haasteita. Luvussa 4 tehdään katsaus roskienkeruun tulevaisuuteen ja tämänhetkisiin innovaatioihin. Lopuksi, luvussa 5 muodostetaan yhteenveto aikaisempien lukujen pohjalta.

2. ROSKIENKERUU

Roskienkeruun päätavoite on aina ollut selkeä: Tunnistaa ohjelman varaamat muistialueet, jotka eivät kuitenkaan enää ole tarpeellisia ohjelman suoritukselle, ja vapauttaa sitten nämä alueet ohjelman tai järjestelmän uudelleenkäytettäviksi [38]. Muistialueen tai -objektin sisältäessä ajolle tarpeellista dataa, sen sanotaan olevan elossa (lifetime, liveliness). Kun kyseistä muistivarausta ei enää tarvita, muuttuu se roskaksi (garbage) ja se tulisi palauttaa takaisin järjestelmän tai ohjelman vapaaseen käyttöön [38].

Ensimmäinen kerran ohjelmistotekniikan historiassa roskienkeruu esiteltiin vuosina 1959 ja 1960 osana Lisp-ohjelmointikielen muistiongelmien ratkaisua [27]. Toteutuksessa järjestelmän muistin tilaa kuvaavaa vapaiden muistialueiden listaa ylläpidettiin automaattisesti roskienkeruumenetelmällä sen sijaan, että ohjelmoija itse olisi joutunut päivittämään listaa. Menetelmän lähtökohtana toimi ajatus, että vain ne muistialueet joihin voitiin siirtyä ohjelman sisältä ovat olennaisia ohjelman ajolle, loppujen muistivarausten ollessa roskaa. Vastaavanlainen muistialueiden saavutettavuuden tutkiminen on lähtökohtana vielä nykyisissäkin roskienkeruumenetelmissä [38].

Roskienkeruu on läpi historiansa sitoutunut vahvasti käytettyyn ohjelmointikieleen ja sen sisäisiin ratkaisuihin, mutta myös laitteistotasolla toimivia roskienkeruumenetelmia on esitetty. Ensimmäisen erilliseen laitteistoon perustuvan roskienkeruujärjestelmän Lisp-pohjaisille järjestelmille esitteli David A. Moon 1980-luvun puolivälissä [29]. Toteutuksen hyödyt jäivät kuitenkin varsin vähäisiksi, eikä kehitystä jatkettu taloudellisesti kannattamattomana.

2.1 Roskienkeruun hyötyjä ja haittoja

Alun perin roskienkeruun kehittämistä motivoivat ohjelmoijan toteuttaman muistinhallinnan haasteet. Muistiosoitteiden eksplisiittinen varaaminen, käyttö ja vapauttaminen olivat haavoittuvaisia ohjelmoijan virheille ja aiheuttivat usein vaikeasti jäljitettäviä ongelmatilanteita ohjelmien ajossa [38]. Roskienkeruun avulla suuri osa näistä muistinhallinnan vastuualueista voidaan automatisoida osaksi ohjelmointikielen toteutusta ja siten vähentää niin ohjelmoijan työmäärää kuin myös tehtyjen virheiden määrää.

Hinta tästä helpotuksesta maksetaan useimmiten ohjelman suorituskyvyssä. Roskienkeruumenetelmät vaativat ylimääräistä laskentaa ja kirjanpitoa, mikä näkyy aina jonkinlai-

sena tallennustilan ja suorittimen käyttönä [38]. Tämän lisäksi roskienkeruu on usein vaikeasti ennakoitava operaatio, jonka vuoksi sen kuluttamat resurssit ja vaikutukset itse ohjelman suoritukseen ovat usein vaikeita arvioida luotettavasti [31] [38].

Varsinkin yksinkertaisemmat toteutukset roskienkeruusta saattavat rajoittaa ohjelmointikielessä käytettyjä tietorakenteita. Historiallisesti esimerkiksi rengasmaiset tietorakenteet ovat aiheuttaneet ongelmia roskienkeruulle [38] ja näin ollen rajoittaneet niiden käyttöä kielissä, jotka käyttävät roskienkeruuta. Nykyisin viimeistään useampien menetelmien käyttö auttaa kuitenkin välttämään nämä tietorakenteista johtuvat ongelmat [38].

Monet roskienkeruumenetelmät toimivat ainoastaan tilanteissa, joissa ne voivat pysäyttää ohjelman ajon ja suorittaa roskienkeruun ohjelman käyttämän muistin pysyessä yhdessä tilassa. Ohjelman pysäytys ei kuitenkaan sovi moniin reaaliaikaisiin järjestelmiin [18], sillä järjestelmän vaihteleva vasteaika vaikuttaa joko ihmisen käyttäjäkokemukseen tai järjestelmän kykyyn reagoida äkillisiin muutoksiin.

Roskienkeruu edellyttää myös kirjanpitoa ja analyysiä muistiin tallennetusta tiedosta [38]. Vaikka tietoja useimmiten kerätäänkin juuri roskienkeruuta varten, ei mikään estä muuta järjestelmää tai tutkijaa käyttämästä tietoja järjestelmän toiminnan parantamiseen. Roskienkeruusta varten kerätyt tiedot ovat osoittautuneet hyödyllisiksi esimerkiksi sulautettujen järjestelmien muistin virrankulutusta tutkittaessa [6].

Akkuvirran varassa toimivien mobiililaitteiden ja sulautettujen järjestelmien myötä virrankulutuksesta on tullut tärkeä ominaisuus osassa suoritusympäristöistä [12]. Roskienkeruuseen liittyvä laskenta ja sen vaikutus virrankulutukseen onkin saanut paljon lisähuomiota 2000-luvun edetessä. Toisaalta tehokas ja tiivis muistinhallinta alentaa muistin itsensä virrankulutusta ja ylläpitää järjestelmän tehokasta toimintaa, mutta toisaalta roskienkeruu itsessään kuormittaa prosessoria ja kuluttaa näin ylimääräistä virtaa. Kiinnostus virrankulutuksen minimointiin on ohjannut tutkimusta myös suurten palvelinkeskusten roskienkeruuseen [11].

2.2 Menetelmien lähtökohtia ja periaatteita

Roskienkeruun tavoite on roskaksi miellettyjen muistialueiden tunnistus, merkintä ja takaisin järjestelmän vapaaseen käyttöön palauttamien [38]. Useimmiten roskan määrittely on varsin selkeä: niin kauan kuin ohjelman sisältä on yhteys muistialueelle, se on edelleen ohjelman käytössä eikä siten ole roskaa. Muut osat taas eivät voi vaikuttaa ohjelman ajoon ja voidaan huoletta tulkita roskaksi. Valtaosa roskienkeruusta eri kielissä toimiikin yhä

tähän saavutettavuuden lähtökohtaan nojaavien menetelmien ympärillä. Menetelmät voidaan jakaa kahteen päätyyppiin: Muistin viittausten kirjanpitoon ja laskemiseen liittyviin menetelmiin sekä niin sanotut muistialuetta pyyhkiviin tai lakaiseviin menetelmiin [38].

Usein itse roskienkeruun käsitteen rajat jäävät hieman epämääräisiksi. Joskus roskienkeruuksi mielletään ainoastaan mark-sweep-tyyliset muistialueiden yli pyyhkivät menetelmät, kun taas usein - esimerkiksi tieteellisissä artikkeleissa - myös viittausten laskenta luetaan mukaan [38]. Tässä opinnäytetyössä pyritään käsittelemään roskienkeruuta ja sen muotoja mahdollisimman kattavasti, joten myös viittausten laskenta sisällytetään käsitteeseen.

2.2.1 Viittausten laskenta

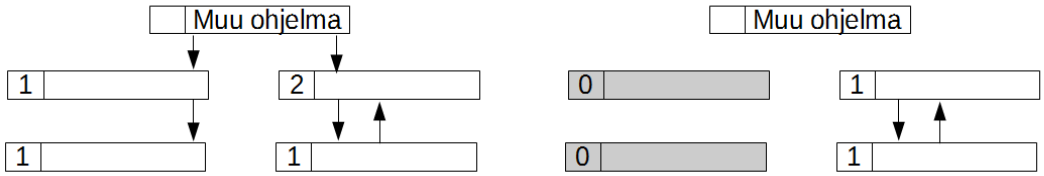
Viittausten laskenta on yksinkertaisin roskienkeruun muoto [38]. Lähtökohtana on, että aina kun muistiin tallennetaan objekti, lisätään sen mukaan myös tieto siitä, kuinka monta viittausta muualta ohjelmasta kyseiseen objektiin on. Tätä kirjanpitoa pidetään yllä aina kun objektiin lisätään viittauksia tai kun siihen liittyvät viittaukset syystä tai toisesta poistetaan. Kun kappaleeseen ei enää ole viittauksia muualta, on se ohjelman kannalta tavoittamattomissa ja voidaan siten tulkita roskaksi ja sen muistivaraus vapauttaa uuteen käyttöön [38].

Menetelmän vahvuuksina ovat näennäisen yksinkertainen perustoiminta ja pienistä, inkrementaaliisista vaiheista koostuva toteutus [38]. Nämä ominaisuudet mahdollistavat ainakin periaatteessa sulavan toiminnan muun ohjelman lomassa. Lisäksi muisti voidaan vapauttaa heti kun viittausten määrä laskee nollaan, eikä järjestelmä näin tarvitse erillisiä toimenpiteitä roskien etsimiselle ja tunnistukselle.

Käytännössä viittausten laskenta kohtaa kuitenkin ongelmia niin suurempien ohjelmistojen tehokkuuden, kuin myös monimutkaisempien tietorakenteiden kohdalla [38]. Tehokkuuden ongelmat tulevat näkyviin varsinkin ohjelmiston koko kasvaessa, sillä ohjelmiston koko vaikuttaa suoraan vaadittavan viittausten kirjanpidon määrään. Kirjanpidon kasvaessa koko roskienkeruun vaatimat resurssit kasvavat ja alkavat hidastaa koko järjestelmän suorituskykyä.

Kirjanpidon määrän lisäksi monimutkaisemmat tietorakenteet, kuten esimerkiksi rengasmaiset rakenteet ovat varsin haastavia viittaustenlaskennalle [38]. Viittaukset saattavat esimerkiksi muodostaa risti-ja takaisinviittausten verkostoja, joita ei saada purettua pelkästään naiivisti viittausmäärää tarkkailemalla. Esimerkiksi kuvan 1 esittelemässä tilanteessa kahden kappaleen viitatessa toisiinsa kumpaakaan ei periaatteessa voida purkaa

ennen toista. Usein helpoin ratkaisu viittausten laskennan sokeiden pisteiden vapauttamiseen onkin käyttää muita roskienkeruumenetelmiä täydentämään viittausten laskennan jättämiä aukkoja [38].



Kuva 1: Yksinkertainen esimerkki rengasrakenteen ja takaisinviittauksen ongelmista viittausten laskennalle. Vaikka yhteys muusta ohjelmasta katoaa, jää oikeanpuoleinen haara harmaalla merkityn roskienkeruun ulottumattomiin takaisinviittauksen takia. Pienissä laatikoissa viittausten lukumäärä.

Viittausten kirjanpidon lisäksi vapautettavaksi havaittujen muistialueiden palauttaminen järjestelmän käyttöön vaatii omia listauksiaan ja lisää ylläpitoon käytettyä resurssimäärää entisestään [38].

2.2.2 Jäljittävät menetelmät

Viittausten laskennan rinnalle toisen pääryhmän muodostavat ohjelman sisäisiä muistiviittauksia seuraavat niin sanotut jäljittävät menetelmät. Nykyisin nämä roskienkeruumenetelmät ovat hyvin yleisiä ja joskus niistä käsitellään ylimalkaisesti ainoana roskienkeruuksi miellettynä menetelmänä [38].

Jäljittävissä menetelmissä muistin käyttötapahtumat jaetaan kolmeen pääryhmään. Ohjelman itsensä - eli mutaattorin - muutokset ovat mitä tahansa roskienkeruun ulkopuolista muistin käyttöä. Itse varsinainen roskienkeruu taas jakautuu kahteen osaan: Ensin muistialueen tutkimiseen ja sen sisältämien roska-alueiden tunnistukseen ja merkintään ja tämän jälkeen pyyhkäisyyn, jossa roska-alueet vapautetaan takaisin vapaaseen käyttöön [38]. Pelkän muistin vapauttamisen lisäksi monet toteutukset pystyvät esimerkiksi järjestelemään uudelleen muistin sisältöä ja täten estämään muistin pirstaloitumista pienempiin alueisiin ja tehostamaan tulevia roskienkeruukertoja. Lopputuloksena järjestelmän suorituskykyä pystytään ylläpitämään paremmin pitkänkin ajon aikana.

Jäljityksen aikana yleisin tapa muistialueen tutkimiseen on tutkia sen sisältämiä viittauksia graafimaisena rakenteena, jossa graafin solmut ovat muistialueita ja viittaukset muodostavat yhteyksiä niiden välille [38]. Näin pystytään järjestelmällisesti käymään läpi muisti, johon ohjelman sisällä voidaan siirtyä tai jota voidaan lukea. Muut alueet eivät voi enää vaikuttaa ohjelman suoritukseen, joten ne voidaan tulkita roskaksi.

Jäljittävät menetelmät voidaan jakaa edelleen tarkempiin määrityksiin sen mukaan, miten ne käsittelevät muistialueita jäljityksen jälkeen [38]. Käsittelymenetelmät usein uudelleenjärjestelevät muistia ja ylläpitävät näin tehokasta muistinkäyttöä ja helpottavat tulevia keräyksiä.

Yksinkertaisin näistä alaluokista on niin sanottu Mark-Sweep-kerääjä [38], joka merkitsee useimmiten roska-alueet listaan ja palauttaa ne sitä kautta järjestelmän käyttöön. Tämä johtaa kuitenkin helposti muistin pirstaloitumiseen pieniin hajanaisiin varauksiin ja muutenkin johtaa usein varsin erilaisten muistivarausten sekoittumiseen samoihin osiin muistiavaruutta.

Mark-Compact-kerääjä pyrkii korjaamaan pirstaloitumista siirtämällä elävät muistiobjektit muistiin vierekkäin ja näin tiivistämään elävän muistin tiettyyn osaan muistiavaruutta [38]. Hieman samankaltaisella periaatteella toimii kopioiva kerääjä (copying collector), joka kopio vielä käytössä olevat muistiobjektit uudelle muistialueelle ja luovuttaa sen jälkeen koko vanhan kerätyn alueen roskaksi [38].

Jotta jäljittävä roskienkeruu olisi tarkka ja muistikäytön kannalta turvallinen, ei muistialue saa muuttua roskienkeruun ollessa meneillään ja koko operaatio tulisi lähtökohtaisesti suorittaa yhtenä atomisena toimenpiteenä [38]. Tämän vuoksi operaatio usein pysäyttää muun ohjelman toiminnan ja sen annetaan jatkaa suoritusta vasta kun roskienkeruu on ohi. Monissa tilanteissa tämä vaatimus järjestelmän muun ajon hetkellisestä pysäyttämisestä on kynnyskysymys jäljittävän roskienkeruun käytössä [18].

2.2.3 Menetelmien jatkokehitystä ja optimointia

Vaikka menetelmien kantavat toimintaperiaatteet ovat pysyneet samoina jo vuosikymmeniä, on niiden toiminnan pienempiä yksityiskohtia pystytty kehittämään [1]. Erilaisten optimointien myötä roskienkeruuta voidaan pyrkiä tehostamaan niin käsitellyn tiedon kuin haluttujen suorituskykytavoitteidenkin mukaan. Varsinkin jäljittävien menetelmien

toiminnassa on paljon mahdollisuuksia erilaisille säädöille ja algoritmeille, jotka säätelevät milloin ja miten mahdollisesti raskas ja aikaa vievä roskienkeruuoperaatiota suoritetaan.

Koska jäljittävät menetelmät ovat usein varsinkin suuremmille muistialuille toteutettuina työläitä ja aikaa vieviä operaatioita, on niiden tarpeellisuuden arviointi tärkeää ja aktivointiajankohta merkittävä päätös. Monissa pienemmissä ja ajoajaltaan lyhyemmissä sovelluksissa järkevin ja tehokkain vaihtoehto saattaakin olla koko menetelmän sivuuttaminen ja esimerkiksi viittaustenlaskentaan nojautuminen. Suuremmissa sovelluksissa ja esimerkiksi palvelinkäytössä muistikapasiteetin tehokas ylläpito muodostuu kuitenkin pakolliseksi, jolloin raskaampien roskienkeruuoperaatioiden suorittaminen on tarpeellista ja niiden aiheuttamia ongelmia voidaan vain pyrkiä rajoittamaan suorittamalla mahdollisimman tehokkaita ja oikea-aikaisia muistinhallintaoperaatioita [38].

Oikea-aikaisiin ja tehokkaisiin toteutuksiin voidaan pyrkiä esimerkiksi muokkaamalla roskienkeruuta sovelluskohteen mukaan. Esimerkiksi erilaisten tallennustarpeiden huomioiminen roskienkeruussa on hyvin tärkeää, sillä muistiin tallennettujen tietojen rakenne ja käyttöikä vaihtelevat erittäin paljon, mutta muodostavat samalla varsin helposti tunnistettavissa olevia ryhmiä, joita varten roskienkeruuta voidaan optimoida.

Esimerkkinä tallennustarpeiden vaihtelut huomioivasta menetelmästä on niin sanottu sukupolvellinen roskienkerääjä eli generational collector [38]. Se on jäljittävien menetelmien toteutustapa, joka pyrkii erittelemään muistiin tallennetuista tiedoista osia eri mittaisen säilyttämistarpeen mukaan ja jakamaan ne hetkellisiin ja pitkäaikaisiin muistivarauksiin. Menetelmän kantavana ajatuksena on ymmärrys siitä, että valtaosa muistivarauksista on hyvin lyhytaikaisia, mutta toisaalta pieni osa saattaa pysyä varattuna koko ohjelman ajonkin ajan. Johtuen tästä muistivarausten luonteenpiirteestä, roskienkeruu kannattaa keskittää lyhytaikaisia varauksia sisältävään muistialueeseen ja tarkastaa pitkäikäisiä muistivarauksia vain harvakseltaan.

Sukupolvellisessa keräyksessä kaikki uudet muistivaraukset sijoitetaan aluksi ensimmäiseen, usein tarkastettavaan sukupolveen [38]. Mikäli varausta ei poisteta ensimmäisillä roskienkeruukerroilla, siirretään se seuraaviin sukupolviin. Seuraavat, vanhemmat sukupolvet sijoitetaan muistialueelle, jota voidaan tarkkailla harvemmin, esimerkiksi ennalta määritellyn kokosäännön täytyttyä ja samalla siirtää pitkäikäisimmät varaukset edelleen vanhempiin sukupolviin. Näin saadaan keskitettyä roskienkeruu muistialueisiin ja -varauksiin, joiden oletettu käyttöikä on lyhyt ja roskienkeruulle tehokasta työaluetta.

Sukupolvellisen keräyksen kaltaisissa valikoivissa roskienkeruissa on omat haasteensa varmistaa, ettei harvakseltaan keräyksen kohteina olevien alueiden viittauksia epähuomiossa rikota, kun uusia alueita puhdistetaan roskasta [38]. Yksi ratkaisu tämänkaltaisiin

ongelmiin on ylläpitää erillistä osoitintaulua erillään pitkäikäisten muistiobjektien alueesta ja varmistaa sen avulla, ettei keräys pidä vanhojen objektien osoittamaa muistia roskana.

Jäljittävien menetelmien pitkäkestoisten roskienkeruuoperaatioiden aiheuttamia ongelmia voidaan pyrkiä ratkomaan esimerkiksi niin sanottujen inkrementaalisten menetelmien avulla [38]. Inkrementaalisten menetelmien avulla vältetään roskienkeruun suorittaminen yhtenä suurena tapahtumana ja sen sijaan sitä pyritään tekemään pieninä palasina muun ohjelman ajon lomassa.

Inkrementaalisia menetelmiä varten on kehitetty uusia merkintätapoja, kuten esimerkiksi kolmivärimerkintä. Sen sijaan, että muistia käsiteltäisiin vain ”mustavalkoisesti” roskana ja käytössä olevana muistina, otetaan mukaan tutkimatonta muistia kuvaava harmaa väri. Tutkimattomien osoittimien avulla päästään tilanteeseen, jossa ohjelman jo tutkitulle muistialueelle tehdyt muutokset ovat havaittavissa. Tämä lisää huomattavasti joustavuutta, jolla ohjelman tekemiä muistialueiden muutoksia ja roskienkeruuta voidaan limittää ja yhdistellä [38].

3. ROSKIENKERUUN NYKYTILANNE

Roskienkeruu on nykyisin vakiinnuttanut paikkansa miltei kaikissa korkeamman tason kielissä. Esimerkiksi hyvin laajalti käytetyn Java-kielen [36] virtuaalikoneen määritelmään kuuluu vaatimus roskienkeruusta [24] ja se on näin osana jokaista Java-pohjaista järjestelmää.

Kielellisen vakiintumisen lisäksi roskienkeruun käyttötarkoitukset -ja vaatimukset ovat moninaistuneet. Sovelluskohteesta riippuen mukaan on tullut vaatimuksia niin tietoturvasta [13], virrankulutuksesta [6] kuin rinnakkaisuudestakin. Lisäksi myös ohjelmointikielten monimuotoisuuden lisääntyminen on lisännyt myös roskienkeruulta odotettuja ominaisuuksia.

3.1 Roskienkeruu eri käyttöympäristöissä

Roskienkeruu on nykyisin levinnyt alkuperäisestä PC-tietokoneiden muistinhallinnasta myös muihin laitteisiin. Erityisen suurta mielenkiintoa ovat herättäneet sulautetut järjestelmät ja mobiililaitteet sekä SSD-asemat. Kokonaan erilaisen laitekontekstin lisäksi myös erilaiset sovelluskohtaiset roskienkeruumenetelmät ovat tulleet tarpeellisiksi.

Esimerkiksi selainohjelmien käsittelemä tietomäärä on moninkertaistunut ja monipuolistunut ja sen myötä myös niiden roskienkeruu on saanut lisähuomiota [37]. Yhden staattisen sivun sijaan käyttäjä saattaakin käsitellä hyvin vaihtelevaa ja suurta sisältömäärää lukuisissa välilehdissä ja pahimmillaan selaimen roskienkeruu alkaa muodostamaan pullonkauloja ohjelman suorituskyvylle. Omat haasteensa selainten roskienkeruulle tuovat tietoturvan takaaminen ja toisaalta myös välilehtien yhteisten resurssien jakaminen. Suosituista selaimista niin Firefox kuin Chromekin ovat päätyneet omanlaisiinsa ratkaisuihin. Chrome käsittelee välilehtiä huomattavasti erillisempinä – ja raskaampina – kokonaisuuksina, kun taas Firefox on päätynyt jakamaan saman pinon kaikille muistialueen välilehdille. Firefoxin ratkaisu mahdollistaa paremman suorituskyvyn suurilla määrillä välilehtiä, mutta aiheuttaa ongelmia niin roskienkeruulle kuin tietoturvallekin.

3.1.1 Roskienkeruu sulautetuissa- ja mobiilijärjestelmissä

Uudet suoritustehoiset äly- ja mobiililaitteet ovat luoneet PC-tietokoneista poikkeavan käyttöympäristön myös roskienkeruumenetelmille. Laitteille ominaisissa käyttökonteksteissa erityisesti rajalliset resurssit ja usein pitkät yhtäjaksoiset käynnissäoloajat ovat haastavia ja vaativat erityishuomiota roskienkeruulta.

Mobiililaitteissa rajattu akkukesto luo uudenlaisen vastakkainasettelun roskienkeruun ja virrankäytön välillä. Kiinteän virran varassa toimivat laitteet ovat rajattuja lähinnä suoritinajan ja muistinkäytön saroilla, mutta mobiililaitteissa kaikki roskienkeruuseen käytetty työ kuluttaa myös laitteen akkua. Näin ollen muistin järjesteleminen ja vapauttaminen tulisi parhaassa tapauksessa suorittaa myös mahdollisimman pienellä työmäärällä oikeaaikaisuuden ja suoritusajallisen tehokkuuden lisäksi.

Huolimatta uusista vaatimuksista, suuri osa ohjelmistoista perustuu yhä ohjelmointikieliin ja -ratkaisuihin, jotka on suunniteltu alun perin PC-ympäristöihin. Tämä johtaa välillä epäoptimaalisiin ratkaisuihin. Esimerkiksi Guangyu Chenin tutkimuksessa [6] havaittiin, että valmiit pöytätietokoneisiin suunniteltujen Java-toteutusten roskienkeruumenetelmät käyttävät usein muistia tavalla, joka haaskaa huomattavia määriä virtaa sulautettujen järjestelmien kokonaiskulutukseen nähden.

Virrankäytön minimoimiseksi roskienkeruun ja laitteiston yhteistyötä täytyy lähestyä uusilla tavoilla. On esimerkiksi havaittu, että roskienkeruun virrankulutusta voidaan vähentää huomattavasti järjestelemällä pitkäaikaisemmiksi ennustetut muistiobjektit tiettyyn osaan muistia ja sivuuttamalla se suuressa osassa roskienkeruukerroista [39]. Kyseinen alue voidaan tällöin asettaa roskienkeruun ajaksi vähän virtaa kuluttavaan tilaan ja saavuttaa huomattavia säästöjä järjestelmän virrankulutuksessa.

3.1.2 Roskienkeruu SSD-asemissa

Tavanomaisten ohjelmistoteknisten sovelluskohteiden rinnalle roskienkeruu on saanut uuden haasteen SSD-levyjen yleistymisen myötä. Kyseissä asemissa mekaaninen kovalevy on korvattu sähköiseen tallennusmenetelmään perustuvalla muistilla. Näin on mahdollista saavuttaa esimerkiksi mekaanista kovalevyä suurempi lukunopeus sekä pienempi

virrankulutus. SSD-levyjen kirjoitus- ja lukutekniikat poikkeavat perinteisen kovalevyn tavoista ja SSD-levyt hyödyntävätkin roskienkeruuta laajasti tallennustilansa järjestelyssä ja tasapuolisessa kuormituksessa [5].

Yksi SSD-aseman suorituskyvyn tärkeimmistä tekijöistä on aseman kontrolleri, jonka laiteohjelmiston tehtäviin myös roskienkeruu sisältyy. Näin ollen roskienkeruu onkin tärkeä osa suorituskykyistä ja pitkäikäistä SSD-asemaa. Kaksi erityisen tärkeää ominaisuutta ovat roskienkeruun suoritusnopeus, sekä sen aiheuttama rasitus levylle pitkäaikaisessa käytössä [5].

Erityisen tärkeäksi roskienkeruun SSD-asemille tekee asemien tapa tallentaa tietoa. SSD-asema ei nimittäin pysty tallentamaan tietoa täysin vapaasti sisältämiinsä lohkoihin, vaan kokonainen muistilohko on aina erikseen tyhjennettävä ja kirjoitettava uudestaan, kun tietoa tallennetaan. Näin ollen muistilohkojen tehokas ja hyvin ajoitettu uudelleenjärjestely onkin tehokkaasti toimivalta SSD-asemalta vaadittu ominaisuus [5].

SSD-asema asettaa levylle kirjoittamiselle ja aseman tilankäytölle omat erityisvaatimuksensa, mikä vaikuttaa myös roskienkeruun rooliin ja tehtäviin. Varsinkin hieman vanhemmat SSD-asemat pystyvät kirjoittamaan muistilohkoihin vain huomattavan vähäisen määrän kertoja ennen kuin aseman käyttö alkaa hidastumaan. Tämä rajoittaa tiedon uudelleenjärjestelyä. Lisäksi tallennetun tiedon päälle ei vain voi kirjoittaa, vaan muistialue on vapautettava erikseen kokonaisuudessaan. Johtuen SSD-aseman roolista kovalevyn korvaajana ja sen myötä tiedostorakenteiden pysyvästä luonteesta, on roskienkeruuta voitu optimoida erilaisille tiedostojärjestelmille.

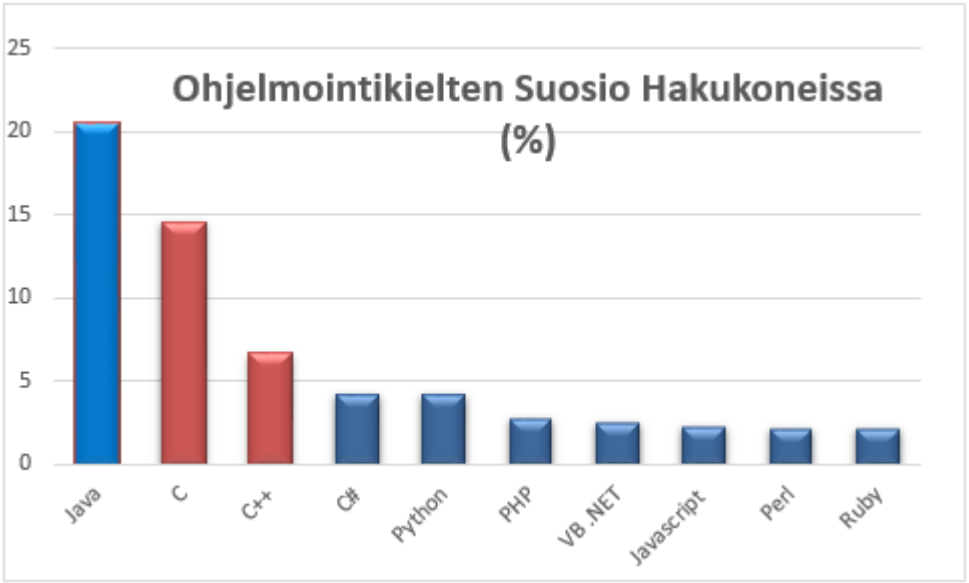
Vaikka SSD-markkinoiden alkuvuosien hidastumis- ja kestävyysongelmat ovatkin nyt miltei kokonaan historiaa, jatkuu tutkimustyö yhä. Erityisesti asemien käyttöiän ja suorituskyvyn ennakointiin esitetään vieläkin uusia malleja. Näiden avulla ongelmia pystytään ennakoimaan ennen kuin asemien huomattavasti parantuneen käyttöiän ja kuormituksen rajat pystytään käytännössä saavuttamaan. Yksittäiskäyttäjien lisäksi SSD-levyt ovat hiljalleen päätymässä myös suurin datakeskuksiin, jossa niiden kestävyydelle muodostuu uusia haasteita. Esimerkkinä mallinuksen tutkimuksesta on esimerkiksi Yongkun Lin johtama tutkimus, jossa tutkittiin asemien kulumisen matemaattista mallinnusta niin sanottujen Markovin ketjujen avulla [15].

Kuten muussakin roskienkeruussa, SSD-aseman roskienkeruuta pyritään suorittamaan niin, ettei se hidasta aseman muuten huomattavan nopeaa toimintaa. Ratkaisuksi on esitetty esimerkiksi roskienkeruumenetelmää, jossa raskaammat operaatiot suoritetaan vain aseman ollessa lepotilassa muuta järjestelmää kohtaan [18]. Näin vältetään käyttämästä levyn kriittisiä polkuja hetkinä, jolloin niiden kuormitus on jo muutenkin huomattavaa. Verrattuna 2010-luvun alun yleisesti käytettyihin tekniikoihin suorituskyky onkin parantunut huomattavasti [18].

Mielenkiintoinen osa-alue SSD-asemien toiminnassa on myös käyttöjärjestelmän ja aseman välinen yhteistyö. Nykyiset SSD-asemat sisältävät esimerkiksi TRIM-komennon, jonka avulla käyttöjärjestelmä voi merkitä levyn sisältöä roskienkeruun poimittavaksi ja täten vähentää roskienkeruuseen kulutettavia resursseja [13]. Varsin paljon roskienkeruuta muistuttavalla tavalla, TRIM-komentojen vääräaikainen käyttö teettää paljon työtä asemalle ja näin hidastaa sen toimintaa. TRIM-sarjojen puskurointi, järjestely ja työmäärältään vähäiseen kohtaan ajoittaminen on tärkeää, kun SSD-aseman roskienkeruutaakkaa pyritään keventämään [16].

3.2 Roskienkeruun toteutus nykyisissä ohjelmointikielissä

Nykyisistä laajasti käytetyistä ohjelmointikielistä suurin osa toteuttaa ainakin viittausten laskentaan perustuvan roskienkeruun. Kuvassa 2 esitetyssä ohjelmointikielten suosiota hakukoneiden avulla mittaavassa TIOBE-listauksessa maaliskuussa 2016 kymmenen suosituimman kielen joukosta ainoastaan C ja C++ eivät sisältäneet oletuksena aktiivista roskienkeruumenetelmää, vaan vaativat ohjelmoijalta muistinhallintaa [38].



Kuva 2: TIOBE-listauksen 10 suosituinta ohjelmointikieltä maaliskuussa 2016. Vain C ja C++ eivät nojaa automaattiseen roskienkeruuseen ja näistäkin C++ toteuttaa ohjelmoijan hallittavissa olevaa viittausten laskentaa älykkäiden osoittimien avulla [38].

3.2.1 Jäljittävää roskienkeruuta käyttävät kielet

Jäljittävän roskienkeruun toteuttavista järjestelmistä käyttäjämääriltään suurin on Java [38]. Sen roskienkeruun toteutus sitoutuu koodia ajavaan virtuaalikoneeseen [24]. Näin ollen mikä tahansa Java-tavukoodia tuottava kieli on myös Javan roskienkeruun piirissä. Tämän kaltaisia kieliä ovat itse Javan lisäksi myös Android-järjestelmissä käytetty Kotlin ja niin sanottuun laiskaan suoritukseen perustuva Clojure.

Vaikka Java on sinällään suunniteltu toimimaan luontevasti roskienkeruumenetelmien kanssa, aiheuttavat Javan rajapinnat ja modulaarisuus omat haasteensa esimerkiksi muistiosoittimien kommunikoinnille ohjelman eri osien välillä [15]. Tämän vuoksi Java joutuukin käyttämään omaa hajautetun roskienkeruun menetelmäänsä (DGC, Distributed Garbage Collection) hallinnoidakseen muistia joustavasti eri osa-aluiden välillä.

Javan roskienkeruun toteutuksessa tulee esiin myös roskienkeruun oikea-aikaisen suorittamisen haastavuus. Vaikka muistinhallinta periaatteessa on automaattista, tarjoaa Java myös eksplisiittisen tavan aktivoida suuren kokoluokan roskienkeruuoperaatio. Tämä menetelmä taas ei kuitenkaan sisällä juuri mitään automatisoidun roskienhallinnan käyttämistä tilannekohtaisista hienosäädöistä, joten sen tehokkuus jää miltei aina implisiittistä aktivointia heikommaksi. Näin ollen sen käyttöä ei säännönmukaisesti suositellakaan, vaan kyseessä on enemmän tilannekohtainen työkalu [15].

Javan toteutuksessa on huomattavaa myös sen monimuotoiset hienosäädöt ja asetukset. Roskienkerääjän parametrit tarjoavat mahdollisuuksia muuttaa niin itse roskienkeruumenetelmän algoritmia, kuin myös hienosäätää sen tarkempia tavoitteita. Huomattavaa on, että monet roskienkeruun asetuksista eivät suoraan vaikuta roskienkeruuseen, vaan asettavat sille tavoitteita. Näiden tavoitteiden perusteella virtuaalikone edelleen hienosäätää roskienkeruun toimintaa [31].

Yksi yleisimmistä tavoitteellisista vastakkainasetteluista on yksittäisen roskienkeruukerran aiheuttama järjestelmän käyttökatko verrattuna suhteelliseen aikaan, jonka järjestelmä toimii roskienkeruussa ja sen ulkopuolella. Useimmiten pitkä yksittäinen tauko mahdollistaa paremman roskienkeruutehokkuuden, mutta aiheuttaa samalla pitemmän käyttökatkon. Vasteaikojen ja tehokkaan ajankäytön lisäksi esimerkiksi pinon koolle voidaan asettaa tavoitteita ja rajoitteita, joihin pyritään kun järjestelmän muut tavoitteet on saavutettu.

Javan roskienkeruu huomioi myös eri suorituskyvyillä varustetut järjestelmät. Erityisen tärkeää on järjestelmän suorittimen tai suorittimien toiminta. Yksiytimisille prosessoreille

suositellaan yksittäisessä säikeessä toimivaa roskienkeruuta, kun taas moniytimisille järjestelmille vähintään pienempiä keräyksiä voidaan suorittaa samanaikaisesti muun ajon yhteydessä. Rinnakkaisuuden lisääminen vakauttaa ja alentaa järjestelmän vasteaikaa, mutta samalla kasvattaa roskienkeruuseen käytettyä kokonaisaikaa. Rinnakkaisuuden lisäksi myös järjestelmän käyttötapa täytyy huomioida roskienkerääjän valinnassa. Esimerkiksi palvelinympäristössä on syytä käyttää niin sanottua Garbage First (G1) –kerääjää, joka painottaa tehokasta rinnakkaista roskienkeruuta hetkellisten vasteaikojen epävakauksien kustannuksella [30].

Erilaisten parametrien avulla on mahdollista saada virtuaalikone kirjaamaan huomattava määrä tietoa roskienkeruun toiminnasta ja sen suorittamista toiminnoista ajon aikana. Nämä tiedot ovatkin tärkeä työkalu muistinhallinnan tilannekohtaisessa optimoinnissa [31].

Toinen käyttäjämääriltään merkittävä roskienkeruutoteutus sisältyy Microsoftin ylläpitämään .NET-ympäristöön. Sen CLR (Common Language Runtime) -ympäristö. Sen suosituin kieli on C# [38]. Ympäristö tarjoaa varsin pitkälti Javan kaltaisen sukupolvellisen roskienkeruun [28].

Uudemmista ohjelmointikielistä esimerkiksi Haskellin kaltainen laiskaan suoritukseen perustuva kieli on haasteellinen roskienkeruulle [10]. Johtuen laiskan suorituksen luonteesta Haskell-laskemat saattavat tuottaa hyvin suuria määriä väliaikaisesti tallennettavaa tietoa lyhyessäkin hetkessä. Tämän jälkeen roskienkeruun tehtäväksi jää vapauttaa suuri osa. Puhtaasti funktionaalisen Haskellin tapauksessa tietorakenteet pysyvät usein muuttumattomina, mistä johtuen vanhassa datassa ei voi olla viittauksia uudempaan dataan. Tästä johtuen uusimmat tietueet voidaan aina poistaa varmoina, ettei niihin ole viitattu muualla. Tämän lisäksi Haskell hyödyntää ”hautomoita”, 512 kilotavun kokoisia muistialueita joiden täytyttyä niistä kopioidaan vain muutama haluttu ja olennainen lopputulos päämuistialueelle [10]. Kopioinnin jälkeen koko hautomo voidaan vapauttaa. Kyseinen menetelmä toimii erityisen tehokkaasti tilanteissa, joissa roskan osuus olennaiseen tietoon verrattuna on ennakoidusti suuri

3.2.2 Muut ratkaisut kielissä

C ja C++ ovat edelleen suosituimmat järjestelmätason kielet [36]. Niiden muistinhallinta perustuu yhä joko ohjelmoijan osoittamaan varaukseen muistikeossa tai objektien elinaikoihin perustuvaan pinovaraukseen, eikä varsinaista automaattista roskienkeruuta

suoriteta. C++11-versio kuitenkin esitteli älykkäitä osoittimia sisältävän Memory-kirjaston [4], jonka avulla ohjelmoija voi itse hallita viittausten laskentaa ja päästä eräänlaiseen väliratkaisuun automaattisen roskienkeruun ja eksplisiittisen muistinhallinnan välillä.

C- ja C++-kieliin on myös mahdollista sisällyttää Mark-Sweep keräys kolmannen osapuolen kirjastojen avulla. Esimerkiksi Hans Boehmin mukaan nimetty Boehm GC tarjoaa roskienkeruun toteutuksen C:lle ja C++:lle [3]. Roskienkeruu otetaan käyttöön korvaamalla kielten omat muistinvaraukseen liittyvät komennot Boehm GC-kirjaston vastaavilla toiminnoilla. Roskienkeruun lisäksi Boehmin roskienkeruukirjastoa voidaan käyttää eksplisiittisen muistinhallinnan aiheuttamien ongelmien kuten muistivuotojen ja tuplavapautusten havaitsemiseen. Johtuen kirjaston asemasta kolmannen osapuolen toteutuksena, sen toimintaa ei kuitenkaan voida yleistää kaikkiin C ja C++-ohjelmistoihin ja -ympäristöihin.

Uusista, suosiotaan kasvattavista kielistä miltei ainoana täysmittaisen roskienkeruun välttää Rust. Se on suunniteltu haastamaan C++ suorituskykyisenä järjestelmätason kielenä ja sen muistinhallinnan ratkaisut muistuttavatkin esikuvaansa monessa mielessä. Kielen muistinhallinta perustuukin pitkälti ohjelmoijan määrittämiin varauksiin ja osoittimiin sekä objektien elinaikoihin [33]. Eräänlaisen roskienkeruun muodon kieli kuitenkin toteuttaa tarjoamallaan viittauksia laskevalla arc-osoittimella. Ainakaan tähän asti Rustin eksplisiittisempi muistienhallinta ei ole pysäyttänyt kielen leviämistä, vaan se on esimerkiksi StackOverflow-sivuston tutkimuksen mukaan yksi miellyttävimpinä pidetyistä kielistä ja myös sen käyttäjämäärät ovat kasvussa [36].

Mielenkiintoisena sivujuonteena Rust-kielen muistinhallinnassa on sen soveltuvuus uusien roskienkerääjien toteuttamiseen [23]. Kieli tarjoaa toisaalta varsin tarkan eksplisiittisen muistienhallinnan ohjelmoijan ulottuville, mutta toisaalta se on myös pyritty kehittämään turvallisemmaksi ja paremmin rinnakkaisuutta tukevaksi kuin esimerkiksi C ja C++, joilla monet muut suorituskykyisimmistä roskienkerääjistä on toteutettu. Kansainvälinen tutkijaryhmä onnistui toteuttamaan edistyneen roskienkerääjän, joka käytännössä vastaa suorituskyvyltään aikaisempaa C++ toteutusta ainakin pienemmissä testikokonaisuuksissa. Käytännön suorituskykyä pystytään mittaamaan vasta, kun toteutusta käytetään laajamittaisesti jonkin suuremman ohjelmistokokonaisuuden muistinhallinnassa.

Applen käyttöjärjestelmissä suosittu Objective-C – kieli on luopunut kokonaan roskienkeruusta ja suosii vain tiukasti ohjelmoijan hallinnoimaa viittaustenlaskentaa. Varsinaista automaattisesti aktivoituvaa pyyhkäisevää roskienkeruuprosessia ei enää käytetä Applen järjestelmissä [26].

Myös Objective-C:n seuraajaksi lanseerattu Swift-ohjelmointikieli jatkaa edeltäjänsä jalanjäljissä ja pyrkii omilla muistinhallintaratkaisuillaan muodostamaan erityisesti mobiililaitteisiin virran- ja muistinkäytöltään Java-pohjaisia Android-sovelluksia huomattavasti säästeliäämmän ja tehokkaamman automatisoidun muistinhallinnan. Sen käyttämä

viittaustenlaskenta tekee myönnytyksiä ohjelmoinnin helppouden osalta, mutta pyrkii samalla voittamaan niin akkukestoa kuin suorituskykyäkin [21].

4. ROSKIENKERUUN TULEVAISUUS

Tällä hetkellä uusia muistinhallinnan ratkaisuja esitetään esimerkiksi vuosittain järjestettävässä ISMM:ssä (International Symposium on Memory Management). Uudet julkaisut ovat usein pohjautuneet vanhempiin perusratkaisuihin, mutta kehittäneet toimintamalleja paremmin nykyiseen monimuotoiseen tietojenkäsittelyyn sopiviksi. Esimerkiksi vuoden 2014 Edinburghissa järjestetyssä ISMM:ssä pääaiheista rinnakkaisuus ja hardware sisälsivät useista julkaisuja roskienkeruumenetelmistä [14].

Parempaa ymmärrystä roskienkeruun toiminnasta pyritään saavuttamaan erilaisilla testialustoilla. Varsin monet menetelmiä tutkivat työryhmät ovat päätyneet käyttämään Java-virtuaalikoneessa toimivaa DaCapo-suorituskykytestialustaa [2]. Kyseisellä menetelmällä pystytään seuraamaan roskienkeruun vaikutusta järjestelmän suorituskykyyn, mutta esimerkiksi roskienkeruun aiheuttamaa virrankulutusta sekään ei pysty seuraamaan, vaan tutkijat ovat joutuneet kehittämään omia menetelmiään vaikutusten seuraamiseksi [2].

Tehokkaan rinnakkainen roskienkeruu on edelleen erittäin suosittu ja tärkeä kehityskohde. Sen avulla voidaan saavuttaa edistystä niin taloudellisemmassa kuin myös vähemmän ohjelman suoritusta häiritsevässäkin roskienkeruussa. Sekä suorituskykyisyys että taloudellisuus ovat erittäin tärkeitä esimerkiksi alati kasvaville palvelinkeskuksille, joiden tavoitteena on vastata palvelupyyntöihin mahdollisimman pienellä viiveellä, mutta samalla pitää huomattavat käyttökustannukset minimissä. Samaan aikaan tehokasta rinnakkaisuutta voidaan hyödyntää myös mobiililaitteiden akkukeston parantamiseen ja vasteaikojen minimoimiseen. Esimerkiksi Google tutkii edelleen tehokkaampia tapoja suorittaa rinnakkaista roskienkeruuta palvelintensa Java-virtuaalikoneissa [8]. Edistystä on tapahtunut viime aikoina esimerkiksi tehokkaammassa kuormanjaossa säikeiden välillä.

Laiskaa suoritusta hyödyntävien kielten kasvava suosio on avannut kehityskohteita myös niiden roskienhallintaan. Erityisesti muistiobjektien elinajat ja käyttökelpoisuuden ohjelmalle eivät aina noudata samaa logiikkaa kuin datapohjaisessa laskennassa. Tästä johtuen nykyisistä roskienkeruumenetelmistä tutut käytännöt eivät aina toteuta tehokkainta tai järkevintä mahdollista muistinvapautusta. Viime vuosina on esitetty esimerkiksi kehittyneempiä menetelmiä tietueiden analysointiin, minkä kautta roskienkeruuta on saatu tarkennettua ja tätä kautta sovellusten muistin käyttöä vähennettyä [20].

Ohjelmistotasolla tehokkaampaa roskienkeruuta pyritään kehittämään esimerkiksi ohjelman käyttämiin tietotyyppeihin paremmin sopeutuvilla roskienkeruumenetelmillä. Vaikka paremmalla tietotyyppien käsittelyllä pystytäänkin tehostamaan roskienkeruuta,

on menestys varsin riippuvaista myös itse ohjelmistojen toteutuksesta [7]. Näin ollen menetelmien kehitys joutuu osittain liikkumaan ohjelmistokehityksen ammattitaidon ja kulttuurin mukana ja muodostamaan toimivia yhteiskäytäntöjä pitemmällä aikavälillä.

Myös kokonaan uusia tai ainakin totutuista menetelmistä poikkeavia lähestymistapoja roskienkeruun hallintaan esitetään yhä. Esimerkiksi vuoden 2013 ISMM:ssä esiteltiin menetelmä, jossa ohjelmoija antaa koodissaan erään laisia vinkkejä vapautettavista muistiobjekteista [32]. Lähestymistapa eroaa yleisimmistä nykyisistä ohjelmoijan hallinnassa olevista muistinhallintamenetelmistä, sillä roskienkerääjä ei luota vinkkeihin suoraan vaan ainoastaan keskittää huomiota merkittyihin muistinosiin ja pyrkii varmistamaan niiden vapautumisen ohjelman käytöstä. Näin saavutetaan tilanne, jossa ohjelmoija pystyy yhä käyttämään lähes eksplisiittistä muistinhallintaa tehohyötyineen, mutta samalla myös välttämään ainakin osan eksplisiittisen muistinhallinnan virheherkkyydestä. Kyseisen menetelmän avulla ohjelmoijan virheet muistinhallinnassa heikentävät ohjelman suorituskykyä ylimääräisinä roskienkeruutarkastuksina, mutta eivät johda esimerkiksi ohjelman toiminnan vaarantumiseen samalla tavalla kuin C ja C++-muistinhallinnoissa.

Intelin uudet Haswell-prosessorit ovat laajentaneet käskykantaansa transaktionaaliseen muistiin liittyvällä käskyllä ja näin mahdollistaneet atomiset operaatiot 64- ja 128-bittisille muistialuille [33]. Laajennetun käskykannan käyttöä rinnakkaisen roskienkeruuseen on tutkittu jo jonkin verran. Ainakin tässä vaiheessa Haswell-arkkitehtuurin käyttämät ratkaisut atomiseen muistinhallintaan ovat kuitenkin niin rajoittuneita, ettei niiden avulla ole vielä saavutettu etua suorituskyvyssä muutamia yksittäisiä erikoistilanteita lukuunottamatta. Ongelmia muodostavat esimerkiksi nykyisten roskienkeruun käyttökustannukset laitetasolla. Tehokkuutta mitannut työryhmä oli kuitenkin toiveikas, että riittävällä kehitystyöllä menetelmä saattaa pystyä tarjoamaan myös nykyisiä menetelmiä tehokkaampaa roskienkeruuta [33].

Puhtaasti laitetason roskienkeruun lisäksi mobiililaitteissa virtaa voidaan jatkossa säästää hyödyntämällä laitteen eri komponentteja laajemmin ja säästeliäämmin. Vuonna 2012 tutkimusryhmä kehitti näytönohjaimen laskentaa hyödyntävän pyyhkäisevän roskienkeruun, joka hyödynsi samalle mikrosirulle sijoitetun näytönohjaimen ja prosessorin yhdistelmää [40]. Tutkimuksessa pyrittiin hyödyntämään usein vähälle kuormitukselle jäävän näytönohjaimen tehoa ja näin siirtämään roskienkeruun aiheuttamaa kuormitusta pois prosessorilta. Vaikka ajurien puutteista ja menetelmän kokeellisuudesta johtuen roskienkeruu olikin vielä prosessoritoteutuksia tehottomampaa, olivat tulokset lupaavia. Tutkimus onnistui osoittamaan roskienkeruun onnistuvan näytönohjaimen avulla ja näytönohjaimen selviytyvän roskienkeruulle tärkeistä graafin seuraamisoperaatioista nopeasti. Samaan aikaan myös laitteistot ovat luontaisesti kehittymässä suuntaan, joka mahdollistaisi myös tehokkaamman työtaakkojen jaon prosessorin ja näytönohjaimen välillä.

Vaikkei koko keskusmuistin roskienkeruuseen liittyvä laskenta vielä osoittautunutkaan tehokkaimmaksi vaihtoehdoksi, voidaan näytönohjaimen omiin toimintoihin liittyvä roskienkeruu mahdollisesti suorittaa näytönohjaimessa. Vaikka näytönohjaimen voimakkaasti rinnakkaisuuteen perustuva rakenne lisääkin omat haasteensa, olivat tulokset hyvin lupaavia ja rinnakkaisuudesta saattaakin muodostua suuri voimavara menetelmien kehittyessä [33].

ISMM 2015:ssa esiteltiin tutkimus, jossa työryhmä onnistui alentamaan roskienkeruun virrankulutusta 30 % vain 10 % suorituskyvyn menetyksellä alentamalla keruuta suorittavan ytimen kellotaajuutta [12]. Ytimen kellotaajuuden laskemisen lisäksi mobiililaitteissa on testattu niin näytönohjaimen hyödyntämistä kuin myös kokonaan erillisten hidastaajuuksisien prosessorien käyttöä.

Kehitystä yritetään saavuttaa myös kokonaan roskienkeruulle tarkoitetuilla komponenteilla [1]. Tähän asti laitteistopohjaiset menetelmät ovat osoittautuneet suorituskyvyltään tehokkaiksi, mutta samalla kömpelöiksi menetelmiksi, jotka rajoittavat käytettävien muistiobjektien rakennetta. Nyt laitteistotason toteutusta on yritetty kehittää useampaan yhdistettyyn pinorakenteeseen perustuvalla ratkaisulla, jonka on tarkoitus mahdollistaa huomattavasti monimutkaisempien muistiobjektien hallinta. Kyseinen julkaisu käsittelee laitteistotason roskienkeruuta uudelleenohjelmoitavalla FPGA-piirillä, mikä mahdollistaa menetelmän käytön myös esimerkiksi sulautetuissa järjestelmissä. Julkaisun tulokset ovat lupaavia ja osoittavat, että ainakin osa teoreettisista ongelmista ja esteistä on odotettua pienempiä. Jatkossa tutkijat aikovat integroida menetelmänsä osaksi korkeamman tason ohjelmointikieltä, mikä mahdollistaisi menetelmän jatkosovellusten tehokkaan tutkimisen ja kehittämisen.

On kuitenkin myös mielenkiintoiselta vaikuttavia tutkimuskohteita, jotka ovat ainakin vielä jääneet varsin vähälle huomiolle. Esimerkiksi laitetason rinnakkaista roskienkeruuta tutkinut ryhmä toteaa olevansa ainut aihepiiristä tutkimustuloksia hiljattain julkaissut työryhmä [1].

5. YHTEENVETO

Roskienkeruu on mielenkiintoinen ja yhä uusia sovelluskohteita löytävä ohjelmistotieteen osa-alue. Sen perusteet luotiin jo ohjelmistotekniikan alkuaikoina, mutta samanaikaisesti sen sovelluskohteet ja – ympäristöt ovat muuttuneet aikojen kuluessa huomattavasti ja jatkavat kehitystä edelleen.

Kielitasolla roskienkeruu on vakiinnuttanut asemansa miltei kaikissa korkeamman tason ohjelmointikielissä. Matalamman tason kielissä turvaudutaan usein pelkkään viittausten-laskentaan tai vaaditaan ohjelmoijalta eksplisiittisempää lähestymistapaa. Myös matalamman tason kielissä kuitenkin tapahtuu innovaatioita, kuten esimerkiksi suosiotaan kasvattava Rust-kieli osoittaa [36]. Myös esimerkiksi laiskan suorituksen kielet kuten Haskell ovat otollisia roskienkeruun kehityskohteita [20].

Erilaisista sovellusympäristöistä roskienkeruu on valloittanut itselleen niin Mobiililaitteet, palvelinkeskukset kuin SSD-asematkin ja tutkimus ja kehitys jatkuvat yhä [5]. Menetelmien kehitys ja kasvanut ymmärrys niiden toiminnasta on hiljalleen mahdollistanut ja rohkaissut roskienkeruuseen myös tiukkaa reaaliaikaisuutta järjestelmissä. Keskeisinä teemoina ovat olleet viime aikoina esimerkiksi virrankulutus, ennakoitavuus ja tietoturva.

Myös kokonaan uusia kehityskohteita tutkitaan. Jatkossa parempaa roskienkeruuta saatetaan saavuttaa esimerkiksi roskienkeruun ja rautatason paremmalla yhteistoiminnalla [25] [33].

6. LÄHTEET

[1] Bacon, David F; Cheng, Perry; Shukla, Sunil: Parallel real-time garbage collection of multiple heaps in reconfigurable hardware. 2014.

[2] Blackburn. Stephen M.; Garner, Robin; Hoffmann, Chris; Khang, Asjad M.; McKinley, Kathryn S.; Bentzur, Rotem; Diwan, Amer; Feinberg, Daniel; Frampton, Daniel; Guyer, Samuel Z.; Hirzel, Martin; Hosking, Antony; Jump, Maria; Lee, Han; Moss, J. Eliot B.; Phansalkar, Aashish; Stefanović, Darko; VanDrunen, Thomas; von Dincklage, Daniel; Wiedermann, Ben: The DaCapo Benchmarks: Java Benchmarking Development and Analysis. 2006.

[3] Boehm Garbage Collector. 2014: http://www.hboehm.info/gc/

Viitattu 6.6.2018

[4] C++ Reference: Memory

http://en.cppreference.com/w/cpp/header/memory

Viitattu 3.6.2018

[5] Chen, Feng; Koufaty, David A.;Zhang, Xiaodong: Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. 2009.

[6] Chen, Guangyu; Shetty, R.; Kandemir, Mahmut Taylan; Vijaykrishnan, Narayanan; Irwin, M. J.; Wolczko, Mario: Tuning garbage collection for reducing memory system energy in an embedded java environment. 2002.

[7] Cohen, Nachshon; Petrank, Erez: Data Structure Aware Garbage Collector. 2015

[8] Fu, Wei; Hauser, Carl: A real-time garbage collection framework for embedded systems, 2005

[9] Griffin, Paul; Srisa-an, Witawas; Chang, J. Morris: An energy efficient garbage collector for java embedded devices. SIGPLAN Not. 40, 7 (June 2005), p 230-238. 2005

[10] Haskell.org. 2014: https://wiki.haskell.org/GHC/Memory_Management viitattu: 19.4.2015

[11] Hassanein, Wessam: Understanding and Improving JVM GC Work Stealing at the Data Center Scale. 2016.

[12] Hussein, Ahmed; Hosking, Antony L.; Payer, Mathias; Vick, Christopher A. : Don't race the memory bus: taming the GC leadfoot. 2015

[13] Intel Trim Overview: https://www.intel.com/content/www/us/en/support/articles/000016148/memory-and-storage.html?wapkw=trim Viitattu 24.5.2018.

[14] International Symposium on Memory Management 2014 (ISSM 2014):

http://ismm2014.cs.tufts.edu/

Viitattu 19.4.2015

[15] Java Documentation. Garbage Collection of Remote Objects, 2010: http://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmi-arch4.html Viitattu 1.3.2016

[16] Jung, Myoungsoo; Kandemir, Mahmut: Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications, 2012

[17] Jung, Myoungsoo; Prabhakar, Ramya; Mahmut Taylan Kandemir: Taking Garbage Collection Overheads off the Critical Path in SSDs. 2014

[18] Kalibera, Thomas; Pizlo, Filip;Hosking, Anthony L.; Vitek, Jan: Scheduling Real-Time Garbage Collection on Uniprocessors, 2011

[19] Kalkov, Igor; Franke, Dominik; Schommer, John F.; Kowalewski, Stefan: A real-time extension to the Android platform. In Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES '12). p 105-114. 2012.

[20] Kumar K, Prasanna; Sanyal, Amitabha; Karkare, Amey: Liveness-Based Garbage Collection for Lazy Languages. 2016.

[21] Lattner, Chris: [swift-evolution] What about garbage collection? (8.2.2016) https://lists.swift.org/pipermail/swift-evolution/Week-of-Mon-20160208/009422.html Viitattu 1.3.2016

[22] Li, Yongkun ; Lee, Patrick P. C.; Lui, John C. S.: Stochastic modeling and optimization of garbage collection algorithms in solid-state drive systems. 2014

[23] Lin, Yi; Blackburn, Stephen M.; Hosking, Anthony L.; Norrish, Michael: Rust as a language for high performance GC implementation. 2016

[24] Lindholm, Tim; Yellin, Frank; Bracha, Gilad; Buckley, Alex: The Java Virtual Machine Specification: Java SE 7 Edition. 2013. Saatavilla: https://docs.oracle.com/javase/specs/jvms/se7/html/index.html Viitattu 5.6.2018

[25] Maas, Martin; Reames, Philip; Morlan, Jeffrey; Asanovic, Krste; Joseph, Anthony D.; Kubiatowicz, John: GPUs as an Opportunity for Offloading Garbage Collection. 2012

[26] Mac Developer Library: Transitioning to ARC, 2013: https://developer.apple.com/library/mac/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html#//apple_ref/doc/uid/TP40011226-CH1-SW17

[27] McCarthy, John: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, 1960

[28] Microsoft .NET Documentation: Fundamentals of Garbage Collection. Saatavilla: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals Viitattu 7.6.2018

[29] Moon, David A.: Architecture of the Symbolics 3600. 1985

[30] Oracle – Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide, 2016 https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html

[31] Oracle – Memory Management White Paper. 2006: http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf

[32] Reames, Philip; Necula, George: Towards Hinted Collection. 2013

[33] Ritson, Carl G.; Ugawa, Tomoharu; Jones, Richard E.:Exploring: Garbage Collection with Haswell Hardware Transactional Memory. 2014.

[34] Rust language documentation; https://doc.rust-lang.org/ viitattu 18.06.2017

[35] Siebert, Fridtjof: Real-Time Garbage-Collection. 2010.

[36] TIOBE Index, Saatavilla http://www.tiobe.com/tiobe_index Viitattu 10.3.2016

[37] Wagner, Gregor; Gal, Andreas; Wimmer, Christian; Eich, Brendan; Franz, Michael: Compartmental Memory Management in a Modern Web Browser. 2011.

[38] Wilson, Paul R: Uniprocessor Garbage Collection Techniques, 1992

[39] Velascoa, Jose Manuel; David Atienzaba; Katzalin Olcoza: Memory power optimization of Java-based embedded systems exploiting garbage collection information. 2009

[40] Veldema, Ronald; Philippsen, Michæl: Iterative data-parallel mark&sweep on a GPU. 2011