Reaaliaikaisen roskankeruun tekniikat

Antti Salonen

Helsinki 16.5.2020
Pro gradu -tutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO − HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta − Fakultet – Faculty	Laitos − Institution − Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä − Författare − Author

Antti Salonen

Työn nimi − Arbetets titel − Title

Reaaliaikaisen roskankeruun tekniikat

Oppiaine − Läroämne − Subject		
Tietojenkäsittelytiede		
Työn laji − Arbetets art − Level	Aika − Datum − Month and year	Sivumäärä − Sidoantal − Number of pages
Pro gradu -tutkielma	16.5.2020	60 sivua + 9 liitesivua

Tiivistelmä − Referat − Abstract

Roskankeruulla tarkoitetaan automaattista muistinhallinnan mekanismia, jossa roskankeräin vapauttaa sovelluksen varaamat muistialueet, joihin sovellus ei enää viittaa. Keskeisiä roskankeruun perustekniikoita ovat muistiviitteiden laskenta ja jäljittävät keruutekniikat, kuten mark-sweep-keruu ja kopioiva keruu.

Reaaliaikaisissa ja interaktiivisissa sovelluksissa roskankeruusta koituvat suoritusviiveet eivät saa olla liian pitkiä. Tällaisissa sovelluksissa keruuta ei voida toteuttaa yhtenä atomisena operaationa, jonka ajaksi ohjelman suoritus keskeytyy. Sen sijaan roskankeruu voidaan kohdistaa vain osaan ohjelman muistista, tai roskankeruu toteutetaan etenemään samanaikaisesti ohjelman suorituksen kanssa. Varsinaiset reaaliaikaiset keruutekniikat vuorottavat roskankeräimen suorituksen siten, että keruusta aiheutuvat viiveet ovat tarkkaan ennakoituja.

Tutkielmassa vertailtiin Java-kielen roskankeräimiä erilaisilla työkuormilla ja erikokoisilla muistialueilla. Mittauksissa tarkasteltiin mittausajojen kestoa, roskankeruutaukojen kestoa sekä taukojen jakautumista ohjelman suorituksen ajalle. Mittauksissa löydettiin merkittäviä eroja vertailtujen keräimien välillä.

Java-kielen uusi G1-keräin suorittaa koko muistiin kohdistuvan merkintävaiheen rinnakkaisena, ja kopiointivaihe kohdistetaan kerrallaan vain pieneen osaan ohjelman muistista. G1-keräin oli suoritetuissa mittauksissa vain hieman hitaampi kuin vanha Parallel-keräin, mutta G1-keräimen keruutauot olivat huomattavasti lyhyempiä. Kun G1-keräimen keruutauoille asetettiin tavoitekesto, viiveet olivat pisimmillään vain muutamia kymmeniä millisekunteja. Vertailussa mukana olleella Shenandoah-keräimellä, joka on suunniteltu takaamaan erityisen lyhyitä suoritusviiveitä, ohjelman suoritukselle aiheutuneet viiveet olivat vain muutamia millisekunteja.

ACM Computing Classification System (CCS):
Software and its engineering → Garbage collection

Avainsanat – Nyckelord − Keywords
muistinhallinta, roskankeruu, reaaliaikajärjestelmät

Säilytyspaikka − Förvaringställe − Where deposited

Muita tietoja − Övriga uppgifter − Additional information

Sisältö

1 Johdanto **1**

2 Roskankeruun perustekniikat **3**

2.1 Viitteiden laskenta 3

2.2 Mark-sweep-keruu 6

2.3 Kopioiva keruu 8

2.4 Sukupolvimalli 10

3 Samanaikainen roskankeruu **13**

3.1 Vaiheittainen roskankeruu 13

3.2 Kolmivärimerkintä 14

3.3 Kirjoituseste 18

3.4 Lukueste 20

3.5 Rinnakkaiset keruutekniikat 22

4 Reaaliaikainen roskankeruu **26**

4.1 Reaaliaikaiset järjestelmät 26

4.2 Työperustainen vuorottaminen 27

4.3 Taukoperustainen vuorottaminen 29

4.4 Aikaperustainen vuorottaminen 30

5 Java-kielen roskankeräimet **33**

5.1 Parallel-keräin 33

5.2 CMS-keräin 34

5.3 G1-keräin 35

5.4 Shenandoah-keräin 37

6 Keräimien suorituskyvyn vertailu **40**

6.1 Testiasetelma 40

6.2 Mittaustulokset 43

6.3 Keskustelu 49

6.4 Johtopäätökset 53

7 Yhteenveto **55**

Lähteet **57**

Liite 1. H2-testin mittausajojen tulokset

Liite 2. Xalan-testin mittausajojen tulokset

Liite 3. Scrabble-testin mittausajojen tulokset

1 Johdanto

Roskankeruulla tarkoitetaan automaattista muistinhallinnan mekanismia, jossa roskankeräin vapauttaa sovelluksen varaamat muistialueet, joihin sovellus ei enää viittaa. Vapautunut muisti palaa sovelluksen uudestaan käytettäväksi. Vaihtoehtona roskankeruulle on itse ohjelmoitu muistinhallinta, jossa ohjelmoijalla on vastuu siitä, että sovellus vapauttaa varatut muistialueet, kun näitä ei enää tarvita.

Roskankeruu kehitettiin Lisp-kieleen 1950-luvun lopulla, ja sittemmin se on otettu yleiseen käyttöön muissa ohjelmointikielissä; näitä ovat esimerkiksi Smalltalk, Prolog, Haskell, Java ja useimmat skriptikielet. Esimerkiksi symbolisessa konekielessä tai C-kielessä käytetään itse ohjelmoitua muistinhallintaa. Jotkin kielet, kuten Modula-3, tukevat sekä itse ohjelmoitua muistinhallintaa että roskankeruuta (Jones, Hosking & Moss 2011, s. 161–164).

Roskankeruun keskeinen etu on, että ohjelmoijalta poistetaan vastuu varattujen muistialueiden vapauttamisesta. Muistialueiden automaattinen vapauttaminen helpottaa ohjelmointityötä, koska riskiä vapautettuihin muistialueisiin olevista viitteistä tai viittaamattomista edelleen varatuista muistialueista ei ole (Meyer 1997, s. 279–301).

Yksinkertaisempi ohjelmakoodi on helpommin ymmärrettävää ja ylläpidettävää sekä rakenteeltaan lähempänä sitä loogista ongelmaa, joka ohjelmalla yritetään ratkaista. Itse ohjelmoitua muistinhallintaa käytettäessä merkittävä osuus ohjelmistokehitykseen käytetystä ajasta kuluu muistinhallinnan toteuttamiseen tai siihen liittyvien ohjelmointivirheiden korjaamiseen (Rover 1985, s. 16).

Yksi roskankeruuseen liittyvä ongelma itse ohjelmoituun muistinhallintaan verrattuna on sen käyttämä ylimääräinen laskenta-aika. Toiseksi ohjelman tilantarve on suurempi, sillä muistialueita ei vapauteta heti, kun näihin ei enää viitata. Kolmanneksi roskankeruun suorituskertojen ajoitus ja niistä koituvat viiveet ohjelman suorituksessa eivät kaikissa keruutekniikoissa ole ohjelmoijan hallinnassa.

Edellä mainitut roskankeruuseen liittyvät ongelmat eivät ole itsenäisiä tai itsenäisesti ratkaistavissa olevia. Esimerkiksi vain vähän muistia kuluttava roskankeräin joutuu suorittamaan keruun tiheämmin ja saattaa siksi käyttää roskankeruuseen enemmän laskenta-aikaa. Joissakin toteutuksissa roskankeruusta seuraavia viiveitä voidaan harventaa antamalla ohjelman käyttöön enemmän muistia, mutta tällöin yksittäisistä keruukerroista aiheutuvat viiveet ovat pidempiä (Jones, Hosking & Moss 2011, s. 6–9).

Roskankeruusta koituvien viiveiden kestoa voidaan pienentää suorittamalla roskankeruuta vaiheittain tai rinnakkaisesti ohjelman suorituksen kanssa. Reaaliaikaisessa roskankeruussa viiveiden enimmäiskesto on hallittavissa keruutekniikan, sovelluksen ja suoritusympäristön asettamissa rajoissa. Osa reaaliaikaisista roskankeräimistä on kehitetty erityisesti reaaliaikasovellusten käyttöön, mutta yleiskäyttöiset reaaliaikaisuusvaatimuksiin kykenevät keruutekniikat ovat aktiivisen tutkimuksen kohteena (Tene, Iyengar & Wolf 2011; Flood et al. 2016).

Tutkielmassa kuvataan roskankeruun perustekniikat, samanaikaisen roskankeruun yleisimmät tekniikat ja reaaliaikaisuuden roskankeruun perusperiaatteet. Lisäksi tutkielmassa kuvataan neljä Java-kielen roskankeruutoteutusta ja tarkastellaan näiden suorituskykyä reaaliaikasovellusten näkökulmasta. Keräimiä testataan erilaisilla työkuormilla, minkä jälkeen mittaustuloksista tarkastellaan mittausajojen kestoa, roskankeruutaukojen kestoa sekä taukojen jakautumista ohjelman suorituksen ajalle.

2 Roskankeruun perustekniikat

Roskankeruun perustekniikat kehitettiin 1950- ja 1960-lukujen taitteessa varhaisia Lisp-toteutuksia varten. Näitä tekniikoita ovat viitteiden laskenta (Collins 1960), jäljittävä mark-sweep-keruu (McCarthy 1960) ja kopioiva keruu (Minsky 1963). Nykyaikaiset roskankeruun toteutukset yleisesti yhdistelevät eri keruutekniikoita, ja ne ovat yleisesti varhaisia toteutuksia tehokkaampia, mutta myös monimutkaisempia.

Roskankeräin vapauttaa sellaisia muistialueita, jotka eivät enää ole viitattuja eli aktiivisia (active, live). Kun aktiivisia muistialueita kartoitetaan, esimerkiksi globaaleista muuttujista ja suoritussäikeiden pinoista viitatut muistialueet oletetaan aktiivisiksi. Näitä muistialueita kutsutaan yhdessä juurijoukoksi (root set). Lisäksi kaikki muistialueet, joihin viitataan juurijoukosta tai muista aktiivista muistialueista, ovat myös aktiivisia. Muut muistialueet ovat passiivisia (passive, dead), ja ne vapautetaan roskankeruun yhteydessä.

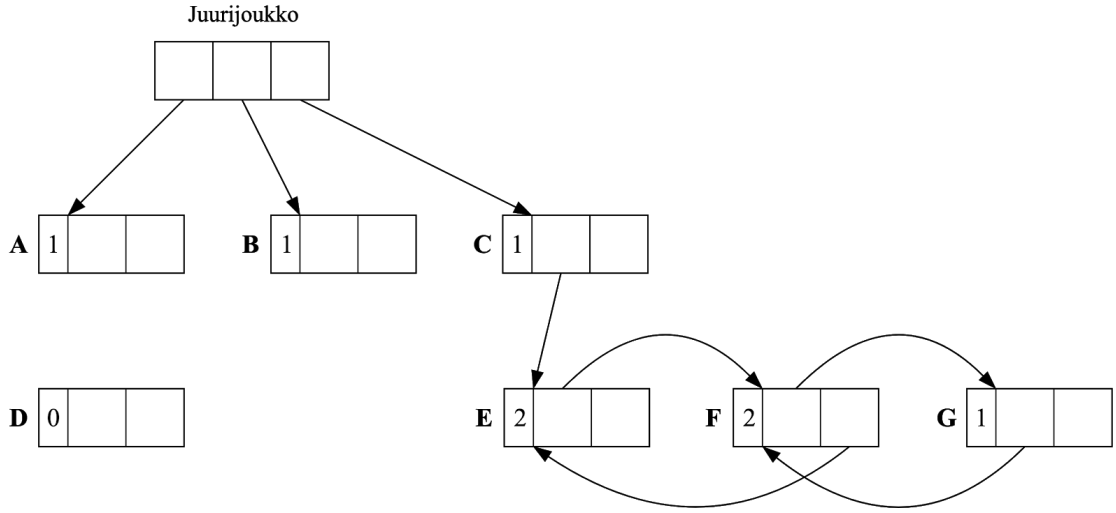
Riippumatta käytetystä tekniikasta roskankeruun on oltava turvallista (safe, sound), eli roskankeräin saa vapauttaa vain passiivisia muistialueita. Roskankeruun on hyvä olla myös täydellistä (complete, comprehensive), eli roskankeräin vapauttaa kaikki passiiviset muistialueet (Meyer 1997, s. 305–306).

2.1 Viitteiden laskenta

Viitteiden laskenta (reference counting) on perusidealtaan yksinkertainen roskankeruutekniikka, jossa muistinhallinta pitää kirjaa jokaiseen varattuun muistialueeseen osoittavien viitteiden määrästä (Jones, Hosking & Moss 2011, s. 57–58). Uuden varatun muistialueen viitelaskurin arvo on yksi. Kun muistialueeseen luodaan uusi viite, kasvatetaan viitelaskurin arvoa yhdellä, ja vastaavasti, kun viite poistetaan, pienennetään viitelaskurin arvoa yhdellä. Viitelaskurin arvon laskiessa nollaan on muistialue passiivinen ja se voidaan välittömästi vapauttaa. Koska kaikki muisti-

alueen muistiviitteet poistetaan ennen muistialueen vapautusta, voi muistialueen vapautuksesta seurata rekursiivisesti muiden muistialueiden vapautuminen.

Kuvassa 2.1 on kuvattu juurijoukko sekä seitsemän muistialuetta ja näiden viitelaskurit. Ainoa viite muistialueeseen D on poistettu, eli se voidaan vapauttaa.



Kuva 2.1: Viitteiden laskenta.

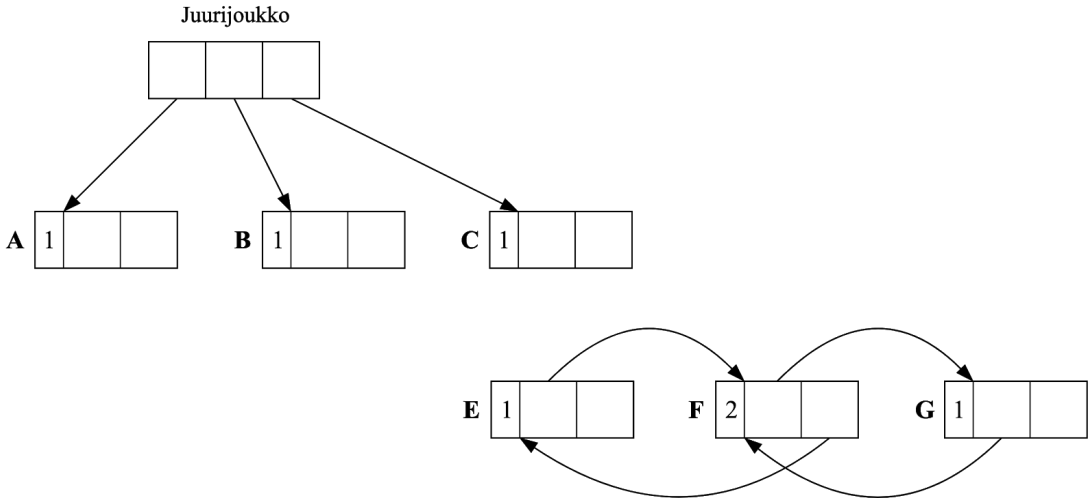
Viitteiden laskenta käyttää ohjelman muistia näennäisen tehokkaasti, sillä passiivisiksi muuttuvat muistialueet voidaan vapauttaa välittömästi. Lisäksi roskankeruuseen liittyvä laskennan tarve ei kasva, jos ohjelman muistinhallinnalla on vähän muistia käytössään (Wilson 1992, s. 6–9).

Viitteiden laskennan toinen etu on roskankeruusta aiheutuvan laskennan jakautuminen tasaisesti ohjelman suorituksen ajalle – viitteiden laskenta on luonnostaan vaiheittainen keruutekniikka (incremental collection) (Jones, Hosking & Moss 2011, s. 58–60), joita käsitellään tarkemmin luvussa 4. Viitteiden laskenta voidaan helposti toteuttaa siten, että se täyttää reaaliaikaisuusvaatimuksia; jos viitteen poistamisesta seuraa suuri joukko passiivisia muistialueita, nämä voidaan vapauttaa viivästetysti

(deferred reclamation). Viivästetyssä vapautuksessa passiiviset muisti-alueet on sijoitettu väliaikaisesti erilliseen tietorakenteeseen, jonka sisältöä käydään vaiheittain läpi ohjelman suorituksen kanssa lomitettuna (Wilson 1992, s. 6–7).

Viitteiden laskennan yksi keskeinen ongelma ovat sykliset viitteet. Jos ohjelmassa esimerkiksi poistetaan ainoa käytetty viite kaksisuuntaisesti linkitettyyn listaan, syntyy listan solmuista joukko passiivisia muisti-alueita, jotka eivät ole saavutettavissa juurijoukosta. Koska muistialueet kuitenkin edelleen viittaavat toisiinsa, ovat niiden viitelaskureiden arvot suurempia kuin nolla, eikä niitä siksi vapauteta.

Kuvassa 2.2 on esitetty tilanne, jossa muistiviite C→E on poistettu. Muistialueiden E, F ja G viitelaskurit ovat kuitenkin kaikki edelleen suurempia kuin nolla, eikä viitteiden laskentaan perustuva roskankeruu siksi ikinä vapauta näitä muistialueita. Yksi esitetty ja usein sovellettu ratkaisu on yhdistää viitteiden laskentaan jäljittävä keruu (tracing collection), kuten seuraavassa luvussa käsiteltävä mark-sweep-keruu (Jones, Hosking & Moss 2011, s. 66–72).



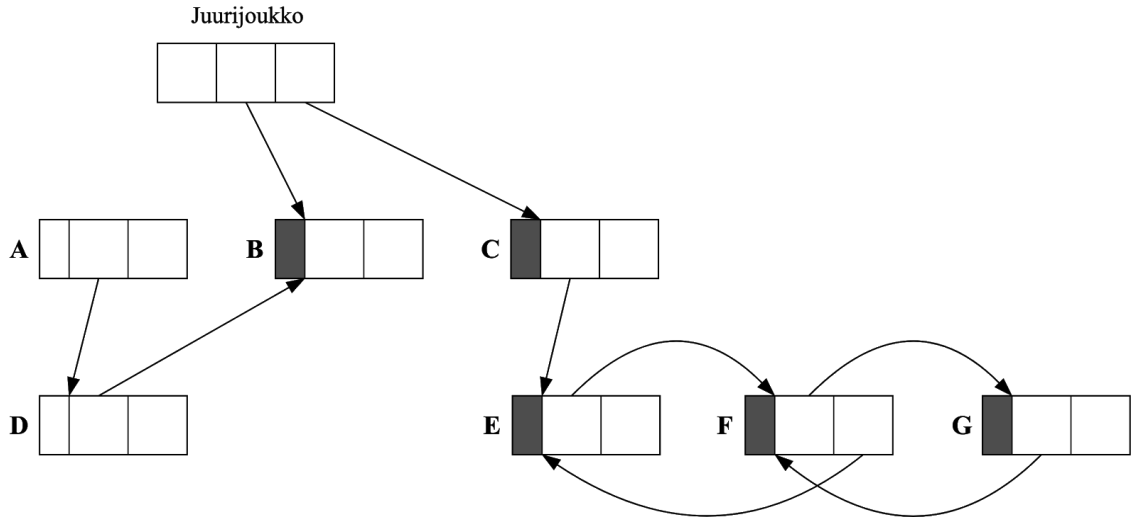
Kuva 2.2: Sykliset viitteet.

Toinen viitteiden laskentaan liittyvä ongelma on viitelaskureiden päivittämiseen kuluva laskenta-aika. Esimerkiksi listarakenteen läpikäynnin yhteydessä tyypillisesti kaikkien listan alkioiden viitelaskureiden arvoja kasvatetaan ja pienennetään. Tätä laskenta-aikaa voidaan pienentää vain välttämällä viitelaskureiden päivittämistä. Yksi esitetty tekniikka on Deutsch-Bobrow'n algoritmi, joka perustuu havaintoon, että valtaosa viitteistä talletetaan paikallisiin muuttujiin. Deutsch-Bobrow'n algoritmin viivästetyssä viitteiden laskennassa (deferred reference counting) viitelaskureita ei päivitetä, jos viite sijoitetaan paikalliseen muuttujaan. Jos muistialueen viitelaskurin arvo on nolla, se vapautetaan viivästetysti vasta, kun ensin on tarkistettu, että aktivaatiotietueet eivät sisällä näihin viitteitä (Deutsch & Bobrow 1976, s. 524).

2.2 Mark-sweep-keruu

Mark-sweep-keruu (mark-sweep collection) on jäljittävä roskankeruutekniikka, jonka Lisp-kielen kehittäjä John McCarthy julkaisi ensimmäisenä roskankeruun menetelmänä huhtikuussa 1960. Mark-sweep-keruussa passiivisia muistialueita ei pyritä tunnistamaan ja vapauttamaan heti, kun viittaukset näihin on poistettu, vaan roskankeruu suoritetaan vasta, kun ohjelman käytössä oleva muisti on loppunut (Jones, Hosking & Moss 2011, s. 18).

Mark-sweep-keruu sisältää nimensä mukaisesti kaksi työvaihetta, joista ensimmäisessä, merkintävaiheessa (mark phase, scan phase), kartoitetaan aktiiviset muistialueet käymällä ne juurijoukosta alkaen rekursiivisesti läpi. Kaikki läpikäydyt muistialueet merkitään aktiivisiksi, jolloin merkitsemättömät voidaan olettaa passiiviksi. Merkintävaihetta seuraa vapautusvaihe (sweep phase), jossa käydään läpi yksi kerrallaan kaikki juurijoukon ulkopuoliset muistialueet. Merkitsemättömät (passiiviset) muistialueet vapautetaan, ja merkityt (aktiiviset) palautetaan alkuperäiseen tilaansa seuraavaa keruukertaa varten (Jones, Hosking & Moss 2011, s. 17–20).



Kuva 2.3: Mark-sweep-keruu merkintävaiheen jälkeen.

Kuvassa 2.3 on kuvattu mark-sweep-keruun tila merkintävaiheen jälkeen. Juurijoukosta viitteiden kautta saavutetut aktiiviset muistialueet B, C, E, F ja G on merkitty. Passiivisia muistialueita A ja D ei ole saavutettu, ja ne vapautetaan, kun kaikki muistialueet käydään läpi vapautusvaiheessa.

Mark-sweep-keruu välttää viitteiden laskentaan liittyvät ongelmat; muistiviitteiden päivityksen yhteydessä ei tarvitse suorittaa muistinhallintaan liittyvää ylimääräistä laskentaa ja sykliset viitteet passiivisten muistialueiden välillä hallitaan ongelmitta, koska merkintävaiheessa saavutetaan vain aktiiviset muistialueet (Jones, Hosking & Moss 2011, s. 29–30).

Toisaalta mark-sweep-keruu käy jokaisella keruukerralla läpi ohjelman koko muistialueen, mikä voi aiheuttaa merkittäviä viiveitä ohjelman suorituksessa. Yksi esitetty parannus on käyttää koko muistialueen läpikäyvän vapautusvaiheen sijasta laiskaa vapautusta (lazy sweeping) (Hughes 1982), jossa merkintävaiheessa merkitsemättä jääneet passiiviset muistialueet vapautetaan vaiheittain vasta uusien muistialueiden varauksen yhteydessä. Tällöin yksittäisten keruukertojen kerralla kuluttama laskenta-aika rajoittuu vain aktiiviset muistialueet kattavaan merkintävaiheeseen.

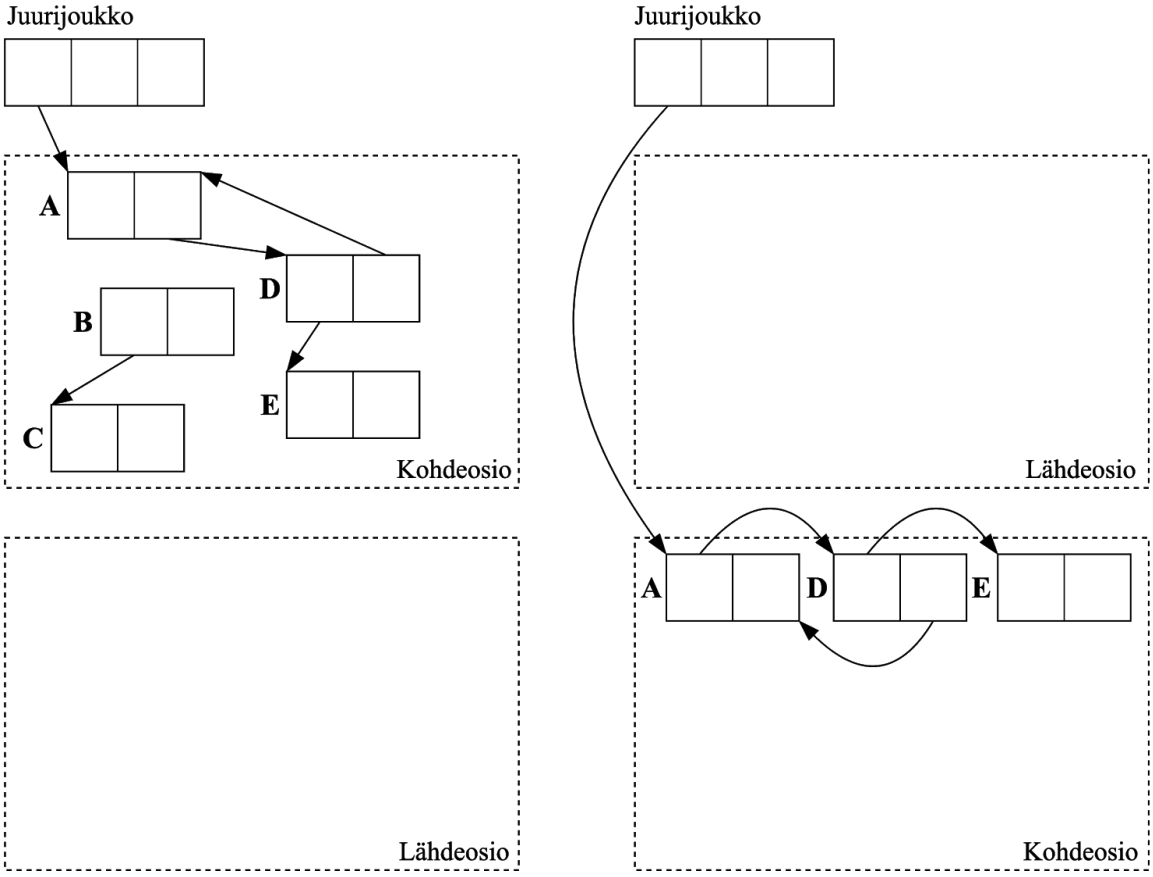
Jäljittävä roskankeruu, kuten mark-sweep-keruu, vaatii ohjelman muistinhallinnan käyttöön ylimääräistä muistia, jotta keruukerrat eivät toistuisi liian tiheään. Jos ohjelma varaa muistia tasaista tahtia, toistuvat keruukerrat sitä tiheämmin, mitä vähemmän muistia yksi keruukerta vapauttaa. Koska jokainen keruukerta käy läpi vähintään kaikki aktiiviset muistialueet, kuluttaa tiheämmin toistuva roskankeruu suhteessa suuremman osan ohjelman laskenta-ajasta. Toisaalta mitä enemmän ohjelman käytössä on muistia, sitä enemmän laskenta-aikaa jokainen keruukerta vie (Jones, Hosking & Moss 2011, s. 29–30).

Mark-sweep-keruu aiheuttaa lisäksi ohjelman muistialueen pirstaloitumista, josta seuraa muun muassa muistialueiden varaamiseen liittyvän laskenta-ajan kasvaminen. Pirstaloitumiseen on esitetty useita ratkaisuja, joissa varattuja muistialueita tiivistetään (compaction, defragmentation). Näistä ratkaisuista merkittävin on seuraavassa luvussa käsiteltävä kopioiva keruu (Jones & Lins 1996, s. 99–100, 117).

2.3 Kopioiva keruu

Kopioiva keruu on mark-sweep-keruun tavoin jäljittävä tekniikka, mutta siinä ohjelman muisti on jaettu kahteen samankokoiseen osioon (semi-space). Sovellus käyttää kerrallaan vain toista näistä muistiosioista, ja uudet varatut muistialueet lisätään tähän kohdeosioon (to-space) niin kauan, kun siinä on tyhjää tilaa. Roskankeruu suoritetaan, kun kohdeosio täyttyy muistivarauksen yhteydessä. Toinen muistiosio on lähdeosio (from-space), joka on tyhjä ennen ensimmäistä roskankeruukertaa (Jones & Lins 1996, s. 28–29).

Roskankeruun käynnistyessä ohjelman muistiosioiden roolit vaihtuvat, eli vanhasta kohdeosiosta tulee lähdeosio ja toisin päin. Roskankeräin kopioi aktiiviset muistialueet lähdeosiosta kohdeosioon, ja kohdeosioon vapautuu siten passiivisia muistialueita vastaava määrä tilaa. Kopioivan keruun suorituskerran tuloksena uudessa kohdeosiossa aktiiviset muistialueet ovat peräkkäin tiivistettynä (Jones & Lins 1996, s. 30–32).



Kuva 2.4: Kopioiva keruu – vasemmalla on muistiosioiden tilanne ennen roskankeruuta ja oikealla roskankeruun jälkeen.

Kuvassa 2.4 on esitetty muistin sisältö ennen kopioivaa keruuta ja sen jälkeen. Keruussa aktiiviset muistialueet A, D ja E on kopioitu toiselle muistiosiolle peräkkäisjärjestykseen. Passiivisia muistialueita B ja C ei kopioitu.

Mark-sweep-keruuseen verrattuna kopioiva keruu ei kärsi muistin pirstaloitumisesta, ja tästä syystä muistialueiden varaaminen on nopeampaa. Kopioiva keruu välttää luonnostaan passiivisten muistialueiden käsittelyn (Jones & Lins 1996, s. 31–32). Lisäksi Blackburn Cheng ja McKinley (2004) esittävät, että kopioiva keruu voi nopeuttaa ohjelman suoritusta, sillä toisiinsa liittyvät ja ohjelman peräkkäin käsittelevät muistialueet sijaitsevat suuremmalla todennäköisyydellä samalla virtuaalimuistin sivulla.

Kopioivan keruun ilmeinen haitta on roskankeruun suoritushetkiä lukuun ottamatta passiivinen lähdeosio, joka kaksinkertaistaa ohjelman muistintarpeen mark-sweep-keruuseen verrattuna. Samankokoisella muistialueella kopioiva keruu joutuu siis suorittamaan roskankeruun kaksi kertaa useammin (Jones & Lins 1996, s. 33).

2.4 Sukupolvimalli

Jäljittävät roskankeruutekniikat käyttävät huomattavasti laskenta-aikaa aktiivisten muistialueiden käsittelyyn. Osa näistä aktiivisista muistialueista voi olla erittäin pitkäikäisiä, jolloin esimerkiksi mark-sweep-keruu merkitsee ne jokaisella keruukerralla, vaikka osa näistä muistialueista voi säilyä aktiivisina ohjelman koko jäljellä olevan suoritusajan.

Sukupolvimallin (generational garbage collection) taustalla on kattava tutkimusnäyttö siitä, että monissa ohjelmissa valtaosa varatuista muistialueista vapautetaan hyvin nopeasti. Muistialueet saattavat olla aktiivisia yhden aliohjelmakutsun tai vain yhden toistolauseen suorituskerran ajan (Jones & Lins 1996 s. 144; Ungar 1984, s. 158).

Sukupolvimallissa ohjelman muisti on ositettu sukupolviin (generation), jotka sisältävät eri-ikäisiä muistialueita. Sukupolvien määrä vaihtelee suuresti toteutuksen mukaan (Jones & Lins 1996, s. 180–181). Uudet muistialueet sijoitetaan nuorimpaan sukupolveen, joka täyttyy siksi nopeasti ja sen roskankeruu tapahtuu hyvin tiheästi. Nuorimman sukupolven roskankeruu on kuitenkin tehokasta, koska se kohdistuu vain siihen osaan ohjelman muistia, jossa muistialueet ovat keruuhetkellä jo suurella todennäköisyydellä passiivisia. Jos nuorimmassa sukupolvessa on käytössä kopioiva keruu, ei valtaosaa tämän muistialueista käsitellä yhdenkään keruukerran yhteydessä (Wilson 1992, s. 33).

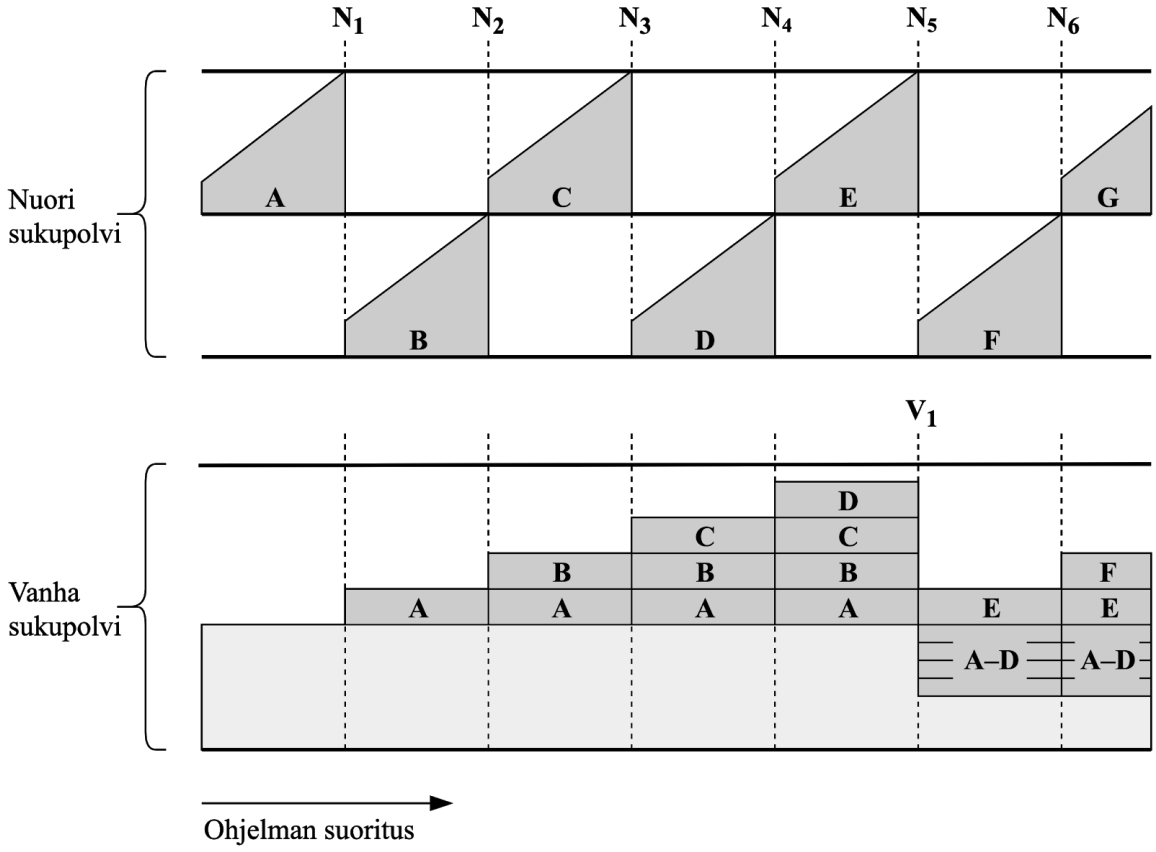
Sukupolvimallissa muistialueet siirtyvät vanhempaan sukupolveen, kun ne ovat aktiivisina selvinneet riittävän monesta roskankeruukerrasta nuoremmassa sukupolvessa. Nuoren sukupolven keruun yhteydessä osa muistialueista saattaa siis siirtyä vanhempaan sukupolveen, joka täyttyy asteittain nuoremman sukupolven roskankeruukertojen seurauksena. Nuoren sukupolven roskankeruuta saattaa siksi seurata roskankeruu myös vanhemmassa sukupolvessa. Vanhempien sukupolvien roskankeruu tapahtuu nuorempia harvemmin, mutta se on pääsääntöisesti raskaampi operaatio, koska aktiivisten muistialueiden osuus on yleensä suurempi (Jones & Lins 1996, s. 179–180).

Sukupolvimallissa roskankeruukerrat kohdistuvat vain osaan ohjelman muistista, jolloin yksittäiset keruukerrat ovat nopeampia. Lisäksi eri sukupolvien roskankeruun yhteydessä voidaan käyttää eri keruutekniikoita.

Kuvassa 2.5 on esitetty sukupolvimallin toiminta, kun ohjelman muisti on jaettu kahteen sukupolveen, joista nuoressa sukupolvessa on käytetty kopioivaa keruuta. Nuoren sukupolven muistia varataan ohjelman suorituksen edetessä, mistä seuraa tasaisin väliajoin kuusi nuoren sukupolven keruukertaa N_1–N_6. Jokaisen keruukerran yhteydessä osa muistisisällöstä vapautetaan, osa kopioidaan nuoren sukupolven toiselle muistiosiolle ja osa siirretään vanhaan sukupolveen.

Vanhan sukupolven muistialueella on vanhaa muistia (kuvassa vaaleanharmaa alue), jonka päälle siirtyy jokaisen nuoren sukupolven keruukerran jälkeen osa nuoren sukupolven muistisisällöstä – esimerkiksi keruukerran N_1 jälkeen nuoren sukupolven muistisisällöstä A osa siirtyy vanhaan sukupolveen. Sama toistuu keruukerroilla N_2–N_4, joissa vanhaan sukupolveen siirtyvät muistisisällöt B–D.

Keruukerralla N_5 nuoren sukupolven muistisisällön E vanhaan sukupolveen siirtyvä osuus ei enää mahdu vanhaan sukupolveen, jolloin ennen E:n muistisisällön siirtämistä suoritetaan ensin vanhan sukupolven roskankeruu V_1.



Kuva 2.5: Roskankeruun sukupolvimalli.

3 Samanaikainen roskankeruu

Reaaliaikaisissa ja interaktiivisissa sovelluksissa roskankeruun aiheuttamat viiveet eivät saa olla liian pitkiä. Tällaisissa sovelluksissa keruuta ei voida toteuttaa yhtenä atomisena operaationa, jonka aikana ohjelman suoritus keskeytyy, vaan roskankeräintä suoritetaan samanaikaisesti ohjelman suorituksen kanssa (concurrent garbage collection).

Yksi ratkaisu on jakaa keruukerta useaan työvaiheeseen, jotka suoritetaan lomitetusti ohjelman suorituksen kanssa. Tällöin puhutaan vaiheittaisesta roskankeruusta (incremental garbage collection), joka on mahdollista myös, kun roskankeräintä suoritetaan ohjelman kanssa samalla suoritinytimellä (Wilson 1992, s. 17). Useamman suoritinytimen ympäristöissä voidaan käyttää myös rinnakkaisia roskankeruutekniikoita (parallel garbage collection), joissa joko roskankeräin on toteutettu monisäikeisenä tai roskankeräintä suoritetaan aidosti rinnakkaisesti ohjelman suorituksen kanssa (Boehm, Demers & Shenker 1991, s. 157).

3.1 Vaiheittainen roskankeruu

Roskankeruun perustekniikoista viitteiden laskenta on luonnostaan vaiheittainen, sillä roskankeruuseen liittyvä laskenta suoritetaan muistiviitteiden päivitysten yhteydessä. Yksittäisten muistiviitteiden poistamisesta voi seurata suuren muistialuejoukon vapauttaminen, mutta tämäkin voidaan suorittaa monivaiheisesti käyttämällä luvussa 2.1 kuvattua laiskaa vapautusta. Koska viitteiden laskentaa on yleisesti pidetty laskennallisesti raskaana keruutekniikkana, on jäljittävien keruutekniikoiden kuten mark-sweep-keruun tai kopioivan keruun vaiheistamista tutkittu paljon (Wilson 1992, s. 17).

Vaiheittaisen (ja myös rinnakkaisen) roskankeruun yhteydessä sovelluksen ohjelmakoodin suoritusta kutsutaan mutaattoriksi (mutator). Nimi perustuu siihen, että vaiheittaisen roskankeräimen kannalta ohjelman suorituksella on se haitallinen sivuvaikutus, että se saattaa muuttaa niiden muisti-

alueiden tilaa, jotka roskankeräin on jo käynyt läpi (Dijkstra et al. 1978, s. 967–968). Vaiheittaisten roskankeruutekniikoiden keskeinen ongelma onkin mutaattorin ja roskankeräimen vuorovaikutusten hallinta.

Vaiheittaisten jäljittävien keruutekniikoiden yhteydessä ei ole kovin suurta merkitystä sillä, onko kyseessä esimerkiksi mark-sweep-keruu vai kopioiva keruu, sillä keskeisin ratkaistava ongelma on aktiivisten muistialueiden jäljityksen suorittaminen vaiheittain. Jos roskankeräin on jo merkinnyt jonkin muistialueen aktiiviseksi mutta mutaattori vapauttaa tämän, on roskankeräimen vaihtoehtoisesti joko havaittava tapahtunut muutos tai roskankeruu on kyseisen keruukerran osalta epätäydellistä. Roskankeruu kokonaisuudessaan täyttää kuitenkin perusvaatimuksen keruun täydellisyydestä, sillä seuraavan keruukerran alkaessa muistialue on passiivinen (Wilson 1992, s. 18).

Roskankeräimen käsitys muistialueiden aktiivisuudesta ei siis välttämättä huomioi mutaattorin viimeisimpiä muutoksia, vaan roskankeräimellä saattaa olla konservatiivinen arvio muistialueiden aktiivisuudesta; roskankeräin voi nähdä todellisuudessa jo passiivisen muistialueen aktiivisena, mutta ei koskaan aktiivista passiivisena, jotta toinen roskankeruun perusvaatimus turvallisuudesta täyttyy.

3.2 Kolmivärimerkintä

Kolmivärimerkintä (tricolor marking) on abstraktio, joka kuvaa jäljittävän roskankeruun etenemistä. Kolmivärimerkinnässä aktiivisten muistialueiden jäljitys ajatellaan verkon läpikäyntinä, jossa verkon solmut (muistialueet) ovat läpikäynnin aikana yhdessä kolmesta tilasta: valkoinen, harmaa tai musta (Dijkstra et al. 1978, s. 969–970).

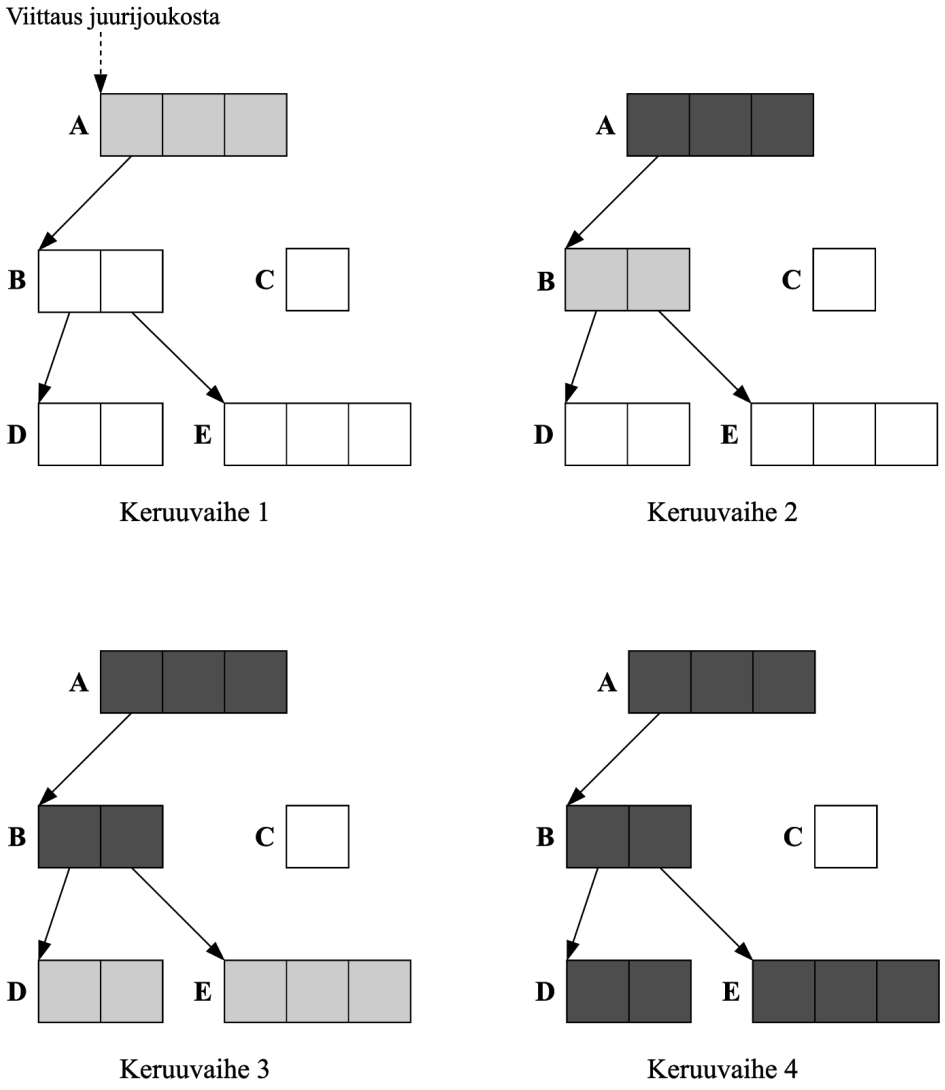
• Valkoisissa solmuissa ei ole vierailtu, eli läpikäynnin alkaessa kaikki verkon solmut ovat valkoisia.

• Harmaat solmut on ”nähty”; kun läpikäynti etenee solmuun, merkitään tästä viitatut solmut harmaiksi.

• Mustissa solmuissa on vierailtu, eli läpikäynnin lopussa kaikki saavutetut solmut ovat mustia ja loput valkoisia. Mustat solmut oletetaan aktiivisiksi.

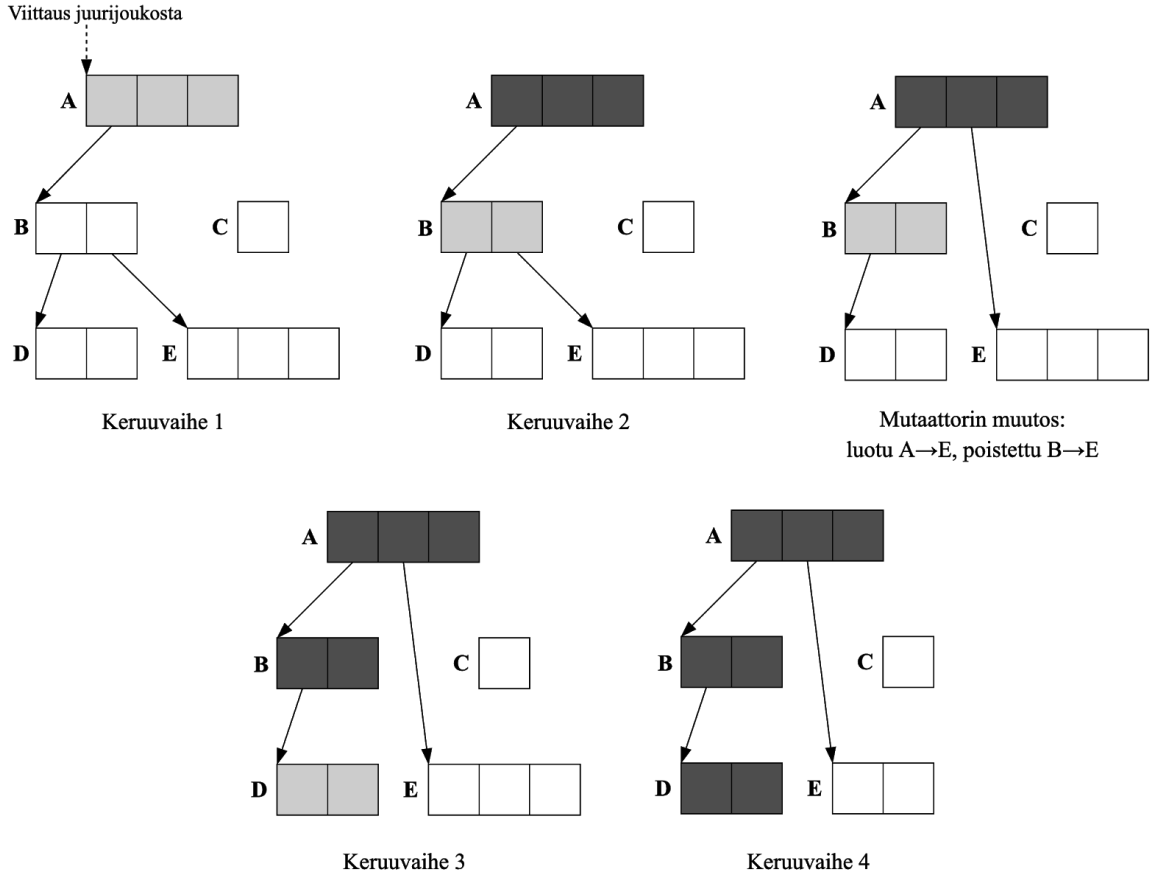
Kolmivärimerkintä on abstraktiona hyödyllinen siksi, että sen avulla voidaan tarkastella jäljittävien keruutekniikoiden oikeellisuutta (Jones Hosking & Moss 2011, s. 20–21). Läpikäynnin aikana mustia (läpikäytyjä) ja valkoisia (toistaiseksi saavuttamattomia) solmuja erottaa toisistaan yksiselitteinen harmaiden solmujen joukko (wavefront traversal). Roskankeräimen on säilytettävä se invariantti, että mustasta solmusta ei viitata suoraan valkoiseen solmuun. Vaiheittaisen roskankeruun tapauksessa roskankeräin olettaa, että mustat solmut on lopullisesti käsitelty ja verkon läpikäynti voi jatkua turvallisesti harmaista solmuista (Wilson 1992, s. 19).

Kuvassa 3.1 on esitetty roskankeruun eteneminen verkossa, jonka kaikki muistialueet ovat viitattuja ja johon ei keruun aikana kohdistu mutaattorin tekemiä muutoksia. Vaiheessa 1 juurijoukosta viitattu muistialue A on harmaa (nähty). Vaiheissa 2–4 keruu etenee siten, että edellisessä vaiheessa nähdyt muistialueet muuttuvat mustiksi (vierailluiksi) ja näistä viitatut muistialueet harmaiksi (nähdyiksi). Vaiheen 4 jälkeen kaikkia löydettyjä muistiviitteitä on seurattu, roskankeruu päättyy ja ainoa valkoinen muistialue C vapautetaan.



Kuva 3.1: Kolmivärimerkinnän eteneminen vaiheittain alkaen juurijoukosta viitatusta muistialueesta A.

Kuvassa 3.2 on esitetty roskankeruu, jonka lähtötilanne on sama kuin kuvan 3.1 esimerkissä. Nyt mutaattori kuitenkin luo vaiheen 2 jälkeen uuden viitteen A→E sekä poistaa viitteen B→E, ja kolmivärimerkinnän invariantti ei ole enää voimassa. Jos keruu jatkuisi tästä tilanteesta (vaiheet 3–4), merkittäisiin solmut B ja D lopulta oikein mustiksi. Keruu ei kuitenkaan etene solmuun E, joka jää valkoiseksi ja tulkitaan siten virheellisesti passiiviseksi. Muistialue E siis vapautetaan, vaikka siihen edelleen on viite.



Kuva 3.2: Mutaattorin suorittama kolmivärimerkinnän invariantin rike.

Roskankeräimen ja mutaattorin vuorovaikutusten hallintaan on kaksi päätekniikkaa: kirjoituseste ja lukueste. Kirjoitusestettä (write barrier) käytettäessä roskankeräin seuraa mutaattorin suorittamia muistiviitteiden kirjoitusoperaatioita ja tämän avulla säilyttää kolmivärimerkinnän invariantin voimassa. Vastaavasti lukuestettä (read barrier) käytettäessä roskankeräin seuraa mutaattorin suorittamia muistiviitteiden lukuoperaatioita (Wilson 1992, s. 19).

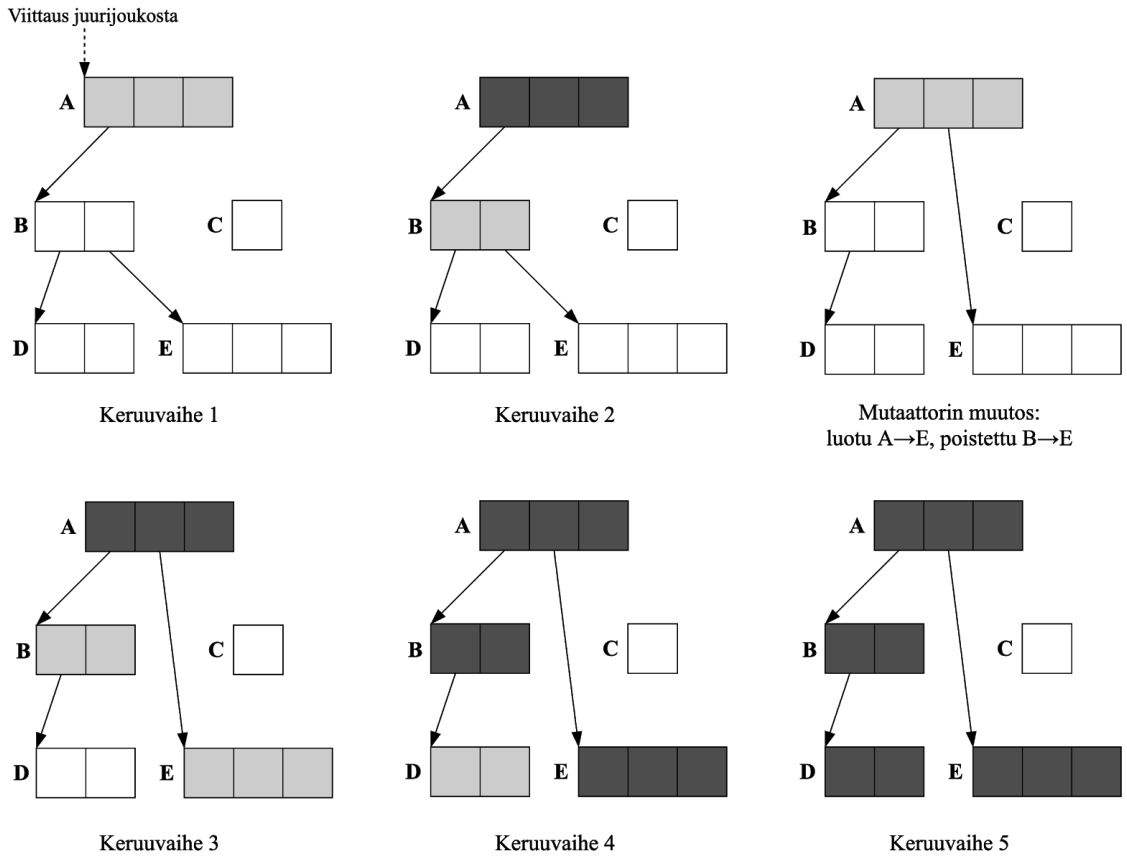
3.3 Kirjoituseste

Kirjoitusesteellä kontrolloidaan mutaattorin suorittamia muistiviitteiden kirjoitusoperaatioita eli joko muistiviitteiden luomista tai poistamista. Mutaattori voi häiritä roskankeräimen keskeneräistä läpikäyntiä suorittamalla kaksi operaatiota: sekä luomalla viitteen mustasta solmusta valkoiseen (kuvan 3.2 esimerkissä viite A→E) että poistamalla alkuperäisen viitteen samaan valkoiseen solmuun, ennen kuin sitä on seurattu (esimerkin viite B→E).

Tarkastellaan mutaattorin suorittamia operaatioita kuvan 3.2 esimerkissä: Jos viitettä A→E ei olisi luotu, toimisi roskankeruu silti oikein, sillä solmu E todella on passiivinen ja tulee vapauttaa. Jos taas viitettä B→E ei olisi poistettu, ei uudesta viitteestä A→E olisi haittaa, koska aktiivinen solmu E edelleen saavutettaisiin keruun jatkuessa. Invariantti siis palaisi voimaan keruun jatkuessa, sillä kun solmu B merkitään mustaksi, merkitään solmut D ja E harmaaksi.

Osa kirjoitusestettä käyttävistä tekniikoista havaitsee uudet muistiviitteet, jotka mutaattori luo mustasta solmusta valkoiseen. Käsitteellisesti musta solmu muuttuu tällöin uudestaan harmaaksi, jolloin siitä viitatut valkoiset solmut tavoitetaan myös siinä tapauksessa, että muut viitteet näihin olisikin poistettu (Wilson 1992, s. 19–20). Parhaiten tunnetun tämäntyyppisen roskankeruualgoritmin on kuvannut Dijkstra et al. (1978).

Kuvassa 3.3 on esitetty kolmivärimerkinnän eteneminen Dijkstran algoritmissa; muistiviitteen A→E luomisesta keruuvaiheen 2 jälkeen on nyt välitetty tieto roskankeräimelle, joka muuttaa solmun A uudestaan harmaaksi. Samalla solmusta A viitattu solmu B on muutettu uudestaan valkoiseksi. Näin roskankeruu jatkuu oikein vaiheissa 3–5 ja solmu E saavutetaan uuden muistiviitteen A→E kautta.



Kuva 3.3: Kolmivärimerkinnän eteneminen Dijkstran algoritmissa.

Toiset kirjoitusestettä käyttävät tekniikat hallitsevat muistiviitteiden poistot luomalla mallin verkon solmujen välisistä viitteistä (snapshot-at-beginning collector) sillä hetkellä, kun keruu käynnistyi. Näin keruu voidaan suorittaa verkon alkuperäisen rakenteen perusteella (Wilson 1992, s. 20). Yksi tunnettu ja yksinkertainen tämäntyyppinen roskankeruu-algoritmi on Yuasan (1990) esittämä, jossa mutaattorin korvaamat muisti-viitteet talletetaan tietorakenteeseen, josta roskankeräin löytää ne keruun jatkuessa (Wilson 1992, s. 20). Esimerkiksi kuvassa 3.3 mutaattorin poistama viite B→E talletettaisiin tähän tietorakenteeseen.

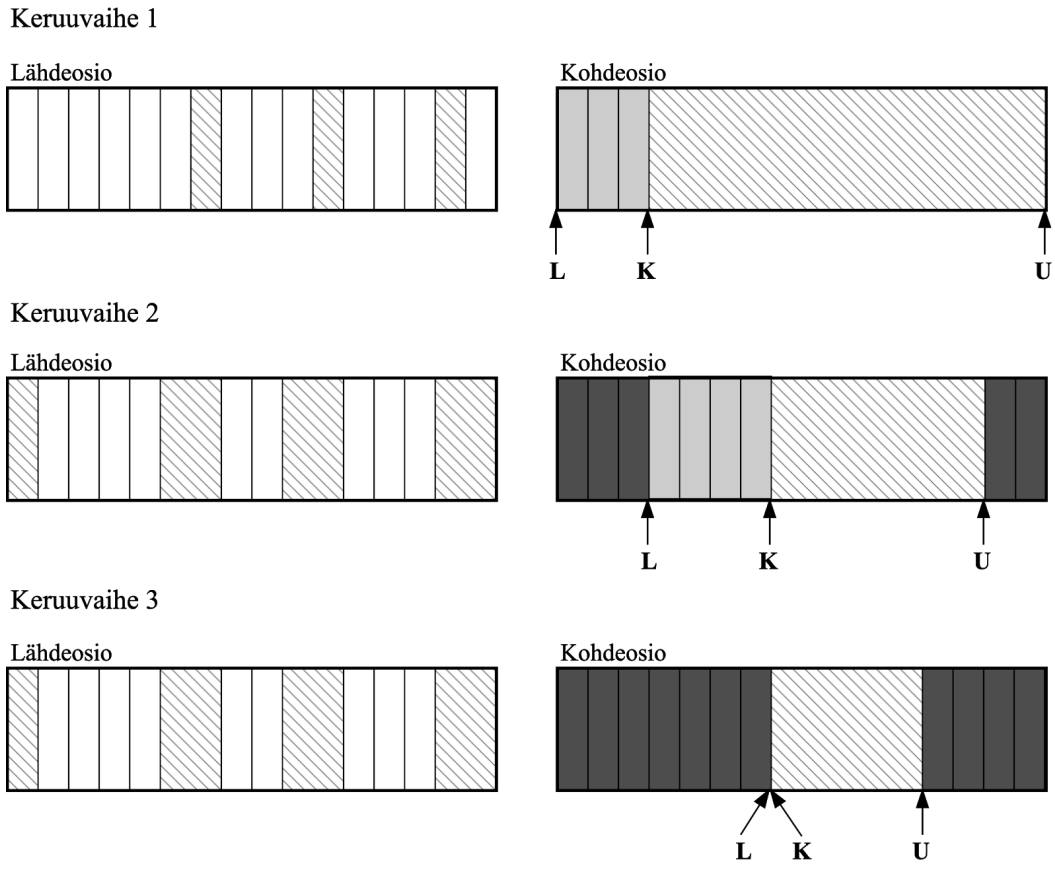
3.4 Lukueste

Lukuestettä käyttävissä tekniikoissa havaitaan mutaattorin suorittamat lukuoperaatiot viitteisiin, jotka kohdistuvat valkoisiin solmuihin. Suoritettaessa lukuoperaatio valkoinen solmu merkitään välittömästi harmaaksi, jolloin mutaattorin ei ole mahdollista luoda viitettä mustasta solmusta valkoiseen.

Parhaiten tunnettu vaiheittainen keruutekniikka on Bakerin vaiheittainen kopioiva keruu (Baker 1978). Bakerin algoritmissa keruukerta alkaa atomisella työvaiheella, jossa lähde- ja kohdeosioiden roolit vaihdetaan ja uuden kohdeosion alkuun kopioidaan lähdeosiosta kaikki juurijoukosta välittömästi viitatut muistialueet. Tämän jälkeen mutaattori saa jatkaa suoritustaan, mutta roskankeruun ollessa kesken mutaattori ei saa käsitellä lähdeosiossa olevia muistialueita; jos mutaattori lukee lähdeosioon kohdistuvan muistiviitteen, kopioidaan viitattu muistialue kohdeosioon, ennen kuin mutaattorin suoritus jatkuu (Baker 1978, s. 283).

Kuvassa 3.4 on kuvattu Bakerin vaiheittaisen kopioivan keruun eteneminen sekä lähde- ja kohdeosioiden sisältö keruun eri vaiheissa. Lähdeosiossa ovat ne muistialueet, joita ei ole vielä käyty läpi, eli ne ovat valkoisia. Mutaattorin lukuoperaatioiden seurauksena lähdeosion muistialueita kopioidaan kohdeosion alkuun osoittimen K kohdalle, ja nämä muistialueet ovat harmaita (kuvassa keruuvaihe 1).

Keruuvaiheessa 2 roskankeruu on edennyt, ja kohdeosioon kopioidut muistialueet on käyty läpi osoittimeen L asti; kolmivärimerkinnän näkökulmasta läpikäydyt muistialueet ovat mustia. Kaikki läpikäydyistä muistialueista viitatut vielä lähdeosiossa sijaitsevat muistialueet on edelleen kopioitu kohdeosioon osoittimen K kohdalle.



Kuva 3.4: Bakerin vaiheittainen kopioiva keruu.

Keruuvaiheessa 3 kaikki kohdeosion muistialueet ovat mustia ja osoittimilla L ja K on sama arvo, ja roskankeruu on näin päättynyt. Lähdeosioon jääneet valkoiset muistialueet ovat roskaa. Keruun aikana varatut muistialueet on sijoitettu kohdeosioon osoittimen U kohdalle.

Bakerin algoritmissa huomionarvoista on uusien varattujen muistialueiden käsittely roskankeruun aikana. Kuvassa 3.4 uudet muistialueet sijoitetaan kohdeosion toiseen laitaan osoittimen U kohdalle mustina; näiden siis oletetaan olevan aktiivisia käynnissä olevan roskankeruun suorituskerran aikana, ja ne kerätään aikaisintaan seuraavalla keruukerralla. Jotta kohde osio ei täyttyisi, ennen kuin kopioiva keruu on suoritettu loppuun, jatketaan kohdealueen harmaiden muistialueiden läpikäyntiä vaiheittain uusien muistialueiden varauksen yhteydessä. Uusi keruukerta käynnistyy, kun osoitin U saavuttaa osoittimet L ja K (Baker 1978, s. 283–284).

3.5 Rinnakkaiset keruutekniikat

Aidosti rinnakkaiset keruutekniikat ovat aktiivisen tutkimuksen kohteena, sillä yhden suorittimen koneelle suunnitellut keruutekniikat eivät kykene tarjoamaan lyhyitä viiveitä moderneissa laitteistoympäristöissä, joissa sovelluksilla on käytössään paljon rinnakkaista laskentakapasiteettia. Toisaalta laitteistotasolla lisääntyvä rinnakkaisuus tekee luontevaksi roskankeruun aidosti rinnakkaisen suorituksen yhdessä tai useammassa säikeessä (Barabash, Ossia & Petrank 2003, s. 1–2).

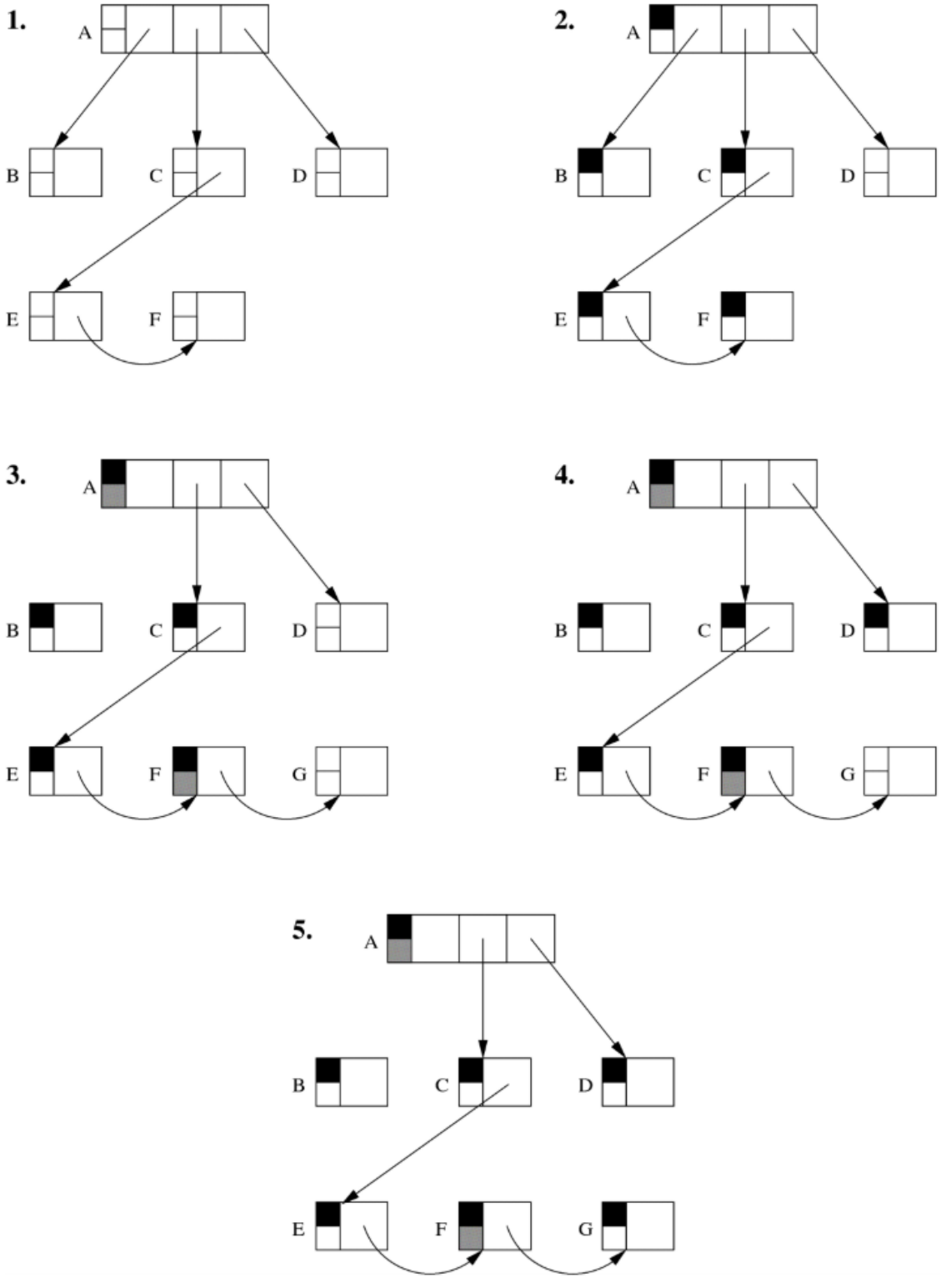
Boehm, Demers ja Shenker (1991) ovat kuvanneet ns. enimmäkseen rinnakkaisen jäljittävän keruun (mostly parallel tracing collection), jossa valtaosa roskankeräimen suorittamasta muistialueiden läpikäynnistä suoritetaan rinnakkaisesti mutaattorin kanssa. Esitetty tekniikka on yleisluontoinen ja sovellettavissa erilaisten jäljittävien keruutekniikoiden rinnakkaistamisessa; se soveltuu myös kopioiville ja sukupolvimallia käyttäville keruutekniikoille.

Boehmin, Demersin ja Shenkerin (1991) tekniikassa oletetaan, että ohjelmisto kykenee ylläpitämään tietoa siitä, mihin muistialueisiin (tai ne sisältäviin virtuaalimuistin sivuihin) on kirjoitettu. Jos muistialueisiin kirjoitetaan keruun aikana, merkitään ne muutosbitillä (dirty bit). Muutosbitit nollataan keruun alkaessa, ja roskankeräin jäljittää aktiiviset muistialueet rinnakkaisesti mutaattorin suorituksen kanssa. Jos mutaattori kirjoittaa muistialueelle roskankeruun ollessa käynnissä, kytketään muistialueen muutosbitti päälle. Rinnakkaisen jäljityksen tuloksena roskankeräimellä on käytössään arvio aktiivisten muistialueiden joukosta.

Kun rinnakkainen jäljitys on päättynyt, mutaattorit pysäytetään tilapäisesti ja roskankeräin suorittaa toisen jäljityskerran aloittaen sen niistä muistialueista, joiden muutosbitit kytkettiin päälle rinnakkaisen jäljityksen aikana. Toinen jäljityskerta merkitsee ne aktiiviset muistialueet, jotka mutaattori lisäsi rinnakkaisen jäljityksen aikana ja jotka roskankeräin oli jo ohittanut näiden lisäämishetkellä. Toisen jäljityskerran seurauksena kaikki aktiiviset muistialueet on merkitty (Boehm, Demers & Shenker 1991, s. 158).

Boehmin, Demersin ja Shenkerin (1991) tekniikka on kompromissi; se ei ole täydellisesti rinnakkainen eikä täydellisen tarkka. Toisen jäljityksen kesto riippuu niiden muistialueiden määrästä, joihin kirjoitettiin ensimmäisen, rinnakkaisen jäljityksen aikana. Jos mutaattori suorittaa rinnakkaisen jäljityksen aikana paljon kirjoitusoperaatioita, voi toinen jäljityskerta teoriassa käydä läpi kaikki aktiiviset muistialueet. Kaikkia passiivisia muistialueita ei vapauteta, sillä jos rinnakkaisen jäljityksen aikana jokin muistialue on ensin merkitty ja sen jälkeen viite tähän on poistettu, säilyy muistialue merkittynä, vaikka se todellisuudessa on jo passiivinen. Tällainen muistialue kuitenkin vapautetaan seuraavan keruukerran yhteydessä.

Kuvassa 3.5 on esitetty tekniikan toiminta tilanteessa, jossa mutaattori kirjoittaa muistialueisiin rinnakkaisen jäljityksen aikana. Vaiheessa 2 roskankeräin on syvyyssuuntaisesti merkinnyt (mustalla bitillä) muistialueet A, B, C, E ja F. Vaiheessa 3 mutaattori poistaa viitteen A→B, luo uuden muistialueen G ja luo tähän viitteen F→G. Muistialueiden A ja F muutosbitit (harmaa bitti) kytketään päälle. Vaiheessa 4 rinnakkainen jäljitys suoritetaan loppuun, jolloin myös muistialue D merkitään aktiiviseksi. Vaiheessa 5 toinen jäljityskerta merkitsee myös muistialueen G, ja roskankeruu on päättynyt.



Kuva 3.5: Boehmin, Demersin ja Shenkerin enimmäkseen rinnakkainen jäljitys.

Rinnakkaisen jäljityksen jälkeen mutaattorien suoritus pysäytetään, ja roskankeräin suorittaa toisen jäljityskerran aloittaen muistialueista A ja F, joiden muutosbitit on kytketty päälle. Tämän seurauksena myös uusi muistialue G merkitään aktiiviseksi. Toisen jäljityskerran seurauksena kaikki muistialueet on merkitty aktiivisiksi mukaan lukien todellisuudessa passiivinen muistialue B, joka vapautetaan seuraavan keruukerran yhteydessä.

Boehmin, Demersin ja Shenkerin (1991) tekniikassa on keskitytty aktiivisten muistialueiden jäljitykseen, joka on yhteinen työvaihe kaikilla jäljittävillä keruutekniikoilla. Mark-sweep-keruuta käytettäessä vapautusvaiheessa voidaan käyttää esimerkiksi laiskaa vapautusta (Hughes 1982), jolloin muistialueet vapautetaan vaiheittaisesti uusia muistialueita varattaessa. Julkaisussa on myös kuvattu rinnakkaisen jäljityksen soveltamistapa kopioivaan keruuseen, jolloin erillistä vapautusvaihetta ei ole (Boehm, Demers & Shenker 1991, s. 163).

Myös luvussa 2.1 kuvatusta viitteiden laskennasta on olemassa rinnakkaistettuja toteutuksia, kuten esimerkiksi Modula 2+ -kielen toteutus, jossa viitelaskureiden päivityksestä vastasi erillinen säie (DeTreville 1990). Jäljempänä luvussa 5 tarkasteltavat roskankeräimet ovat kuitenkin kaikki jäljittäviä keräimiä, eikä rinnakkaista viitteiden laskentaa siksi käsitellä tutkielmassa lähemmin.

4 Reaaliaikainen roskankeruu

Aiemmissa luvuissa kuvatut vaiheittaiset ja rinnakkaiset roskankeruutekniikat pyrkivät lyhentämään roskankeruusta koituvia taukoja ohjelman suorituksessa joko lomittamalla mutaattorin ja roskankeruun suoritusta samalla suorittimella tai suorittamalla näitä rinnakkaisesti eri suorittimilla. Monet näistä keruutekniikoista on kehitetty interaktiivisia sovelluksia varten, sillä pitkät tauot ohjelman suorituksessa aiheuttavat taukoja käyttöliittymän toiminnassa. Varhaisia vaiheittaisia keruutekniikoita, kuten Dijkstran et al. (1978) ja Bakerin (1978) kuvaamia, kutsuttiinkin yleisesti reaaliaikaisiksi keruutekniikoiksi.

Varhaiset vaiheittaiset keruutekniikat olivat kuitenkin reaaliaikaisia vain tiukasti rajatuilla ehdoilla – mikäli esimerkiksi muistialueiden koko ei ollut määrätyissä rajoissa, ei mutaattorin saamaa suoritinaikaa voitu varmuudella taata. Modernimman ja tiukemman määritelmän mukaan reaaliaikainen roskankeräin pystyy takaamaan enimmäispituuden niille tauoille, joita roskankeruusta koituu mutaattorin suoritukselle (Jones, Hosking & Moss 2011, s. 375–376).

4.1 Reaaliaikaiset järjestelmät

Reaaliaikaisissa järjestelmissä joillekin tehtäville on asetettu viimeinen sallittu toteutumisaika. Pehmeän reaaliaikaisissa (soft real-time) järjestelmissä aikarajoista myöhästyminen ei ole toimintavirhe, vaan myöhästyneistä tehtävistä seuraa järjestelmän tarjoaman palvelun laadun aleneminen. Printezis (2006) esittää, että pehmeän reaaliaikaisissa sovelluksissa roskankeräimelle asetettu vaatimus olisi tietty osuus määrätystä aikaikkunasta sekä se todennäköisyys, jolla tämä vaatimus pystytään täyttämään. Kovan reaaliaikaisissa (hard real-time) järjestelmissä aikarajoista myöhästyminen katsotaan toimintavirheeksi, ja siksi roskankeräimen tulisi pystyä luotettavasti takaamaan kunkin suorituskertansa enimmäiskesto.

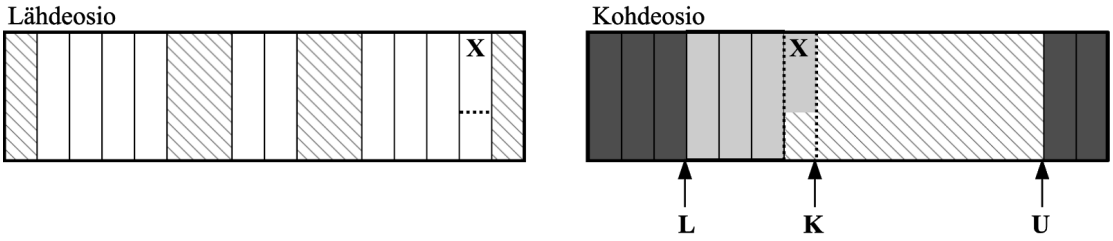
Kaikkien reaaliaikaisten keruutekniikoiden yhteinen tavoite on ajastaa roskankeräimen suoritusta siten, että siitä aiheutuvat viiveet mutaattorin suoritukselle ovat ennakoitavia. Toisaalta jotta ohjelman muistinkulutus ei kasva hallitsemattomasti, roskankeräimen on vapautettava muistia yhtä nopeasti kuin mitä mutaattori varaa. Reaaliaikaisissa keruutekniikoissa on siksi keskeinen kysymys se, milloin ja miten roskankeräimelle päätetään antaa suoritusvuoro. Eri vuorottamistekniikoihin tutustutaan tarkemmin seuraavissa luvuissa.

4.2 Työperustainen vuorottaminen

Työperustaisessa vuorottamisessa (work-based scheduling) roskankeräintä voidaan ajatella suoritettavan mutaattorin suorituksen yhteydessä "verona", eli mutaattorin saadessa suoritusaikaa saa myös roskankeräin tietyn määrän suoritusaikaa. Viitteiden laskenta on luonnostaan tällainen tekniikka. Luvussa 3.4 kuvattu Bakerin vaiheittainen kopioiva keruu oli varhainen työperustainen jäljittävä keruutekniikka; jos mutaattori luki lähdeosiolla sijaitsevia muistialueita, nämä kopioitiin ensin kohdeosiolle.

Bakerin tekniikka pystyi takaamaan ennakoitavat keruuviiveet sillä merkittävällä rajoituksella, että kopioitavien muistialueiden tuli olla kiinteän kokoisia – Bakerin tekniikan tapauksessa Lisp-kielen listarakenteiden alkioita. Mutaattorin suorittamien varausten yhteydessä käytiin läpi tietty määrä kohdeosiolla sijaitsevia harmaita alkioita ja tätä määrää säätämällä keruukerta saatiin päätökseen, ennen kuin varatut muistialueet täyttävät kohdeosion. Baker myös ehdotti tapoja vaihtelevan kokoisten taulukkorakenteiden vaiheittaiseen kopiointiin, mutta ei kuvannut näitä yksityiskohtaisesti (Baker 1978, s. 283–284).

Blelloch ja Cheng (1999) kuvaavat Bakerin työhön perustuvan monistavan kopioivan keruutekniikan (replication copying collector), jossa mutaattorin sallitaan käsitellä lähdeosiolla sijaitsevaa muistialuetta sillä välin, kun roskankeräin on kopioimassa sitä kohdeosiolle. Tämä on esitetty kuvassa 4.1. Roskankeräin on kopioinut osan muistialueesta X kohdeosiolla, ja kopioinnin ollessa kesken mutaattorin sallitaan muuttaa lähdeosiolla olevaa kopioita.



Kuva 4.1: Blellochin ja Chengin monistava kopioiva keruu.

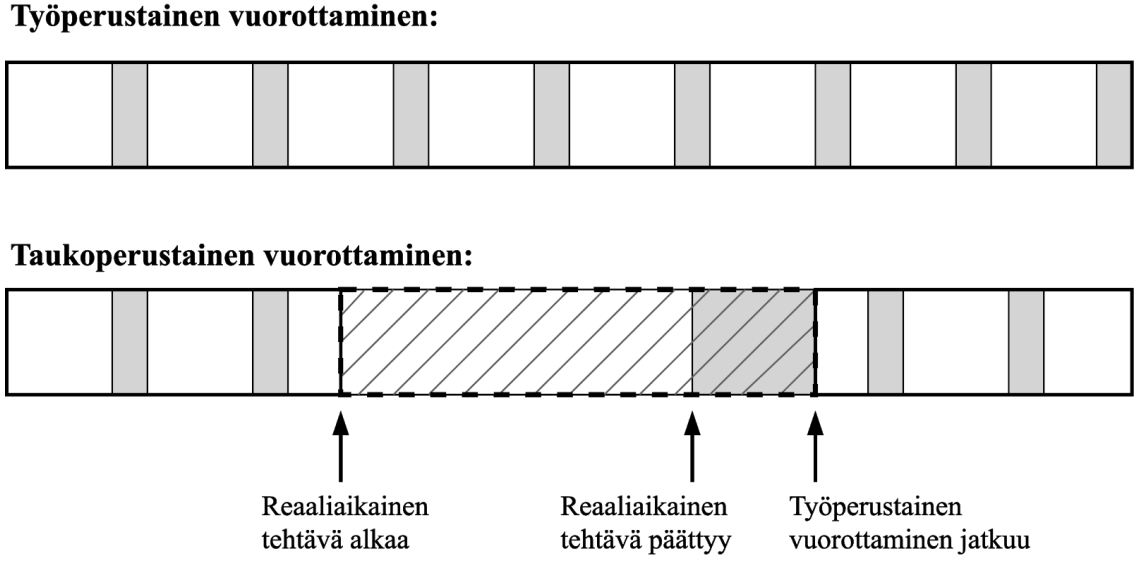
Lähdeosiolla sijaitsevaan muistialueeseen tehdyt muutokset kirjataan ylös ja ne päivitetään kohdeosiolle, kun muistialue on kokonaan kopioitu – kohdeosiolla sijaitsevan kopion sallitaan siis päivittyä viiveellä. Keruu päättyy, kun kaikki juurijoukon viitteet on käsitelty, kaikki kohdeosiolle kopioidut muistialueet on käsitelty ja kirjatut mutaattorin tekemät muutokset on käsitelty.

Blellochin ja Chengin tekniikassa roskankeräimen tila- ja aikavaatimukset ovat molemmat tiukasti rajattuja. Vaatimukset pystytään täyttämään myös suuria muistialueita kopioitaessa, sillä mutaattorin ei tarvitse kopioida näitä kokonaan yhden suoritusjaksonsa aikana (Blelloch & Cheng, 1999). Jatkotutkimuksessaan samat kirjoittajat Cheng ja Blelloch (2001) toteuttivat keräimensä ML-ohjelmointikielelle, ja roskankeruusta seuranneiden viiveiden enimmäispituudeksi mitattiin vain 3–4 millisekuntia. Toisaalta käytettyjen suorituskykytestien kesto oli keskimäärin 51 % pidempi kuin perinteistä roskankeräintä käytettäessä.

4.3 Taukoperustainen vuorottaminen

Taukoperustaisissa keruutekniikoissa (slack-based scheduling) käytetään hyväksi mutaattorin suorittamien reaaliaikaisten tehtävien säännöllisiä suoritusaikoja. Reaaliaikaisten tehtävien ja roskankeräimen suoritus pyritään vuorottamaan siten, että roskankeruu ei estä suoritusvuoroon tulevan reaaliaikaisen tehtävän käynnistymistä.

Henrikssonin (1998) kuvaamassa tekniikassa roskankeruuta ei suoriteta reaaliaikaisten tehtävien suorituksen aikana. Reaaliaikaisen tehtävän valmistuttua roskankeräin saa suoritusvuoron ja kerää reaaliaikaisen tehtävän suorituksesta syntyneen roskan. Vasta tämän jälkeen suoritusvuoron voivat saada mutaattorin muut tehtävät, joihin ei kohdistu reaaliaikaisuusvaatimuksia, ja näiden aikana roskankeräintä voidaan vuorottaa työperustaisesti. Henrikssonin taukoperustaisen keräimen vuorotus on esitetty kuvassa 4.2.



Kuva 4.2: Työperustainen vuorottaminen ja Henrikssonin (1998) taukoperustainen vuorottaminen – mutaattorin suoritusvuoro on merkitty valkoisella ja roskankeräimen harmaalla.

Henrikssonin tekniikassa keruu etenee kuten luvussa 3.4 kuvatussa Bakerin vaiheittaisessa kopioivassa keruussa, mutta reaaliaikaisen tehtävän suorituksen aikana muistialueita ei kopioida lähdeosiolta kohdeosiolle. Kohdeosiolta ainoastaan varataan tila kopioitavalle muistialueelle, ja itse kopiointi tapahtuu viivästetysti reaaliaikaisen tehtävän valmistuttua. Varatussa tilassa on viite vielä lähdeosiolla sijaitsevaan muistialueeseen.

Henrikssonin tekniikassa keräin on määritelty siten, että suoritusvuoroon tuleva reaaliaikainen tehtävä voi koska tahansa keskeyttää keräimen suorituksen. Roskankeräimen kesken jäänyt työvaihe suoritetaan uudestaan keräimen seuraavalla suorituskerralla. Koska roskankeruu ei voi valmistua reaaliaikaisen tehtävän suorituksen aikana, mutta kohdeosiolle voidaan varata uusia muistialueita, ohjelmoijan on määriteltävä jokaiselle reaaliaikaiselle tehtävälle näiden enimmillään varaaman muistin määrä.

4.4 Aikaperustainen vuorottaminen

Aikaperustaisissa (time-based scheduling) keruutekniikoissa mutaattorin käyttöastetta käytetään perusteena mutaattorin ja roskankeräimen suorituksen vuorottamiselle – mutaattorille pyritään antamaan vähintään määrätty osuus suoritusajasta määrätynpituisessa liukuvassa aikaikkunassa. Ensimmäinen tämän tyyppinen keräin oli Java-kielelle kehitetty Metronome-keräin (Bacon et al., 2003), joka on toiminut inspiraationa myöhemmin kehitetyille Javan roskankeräimille.

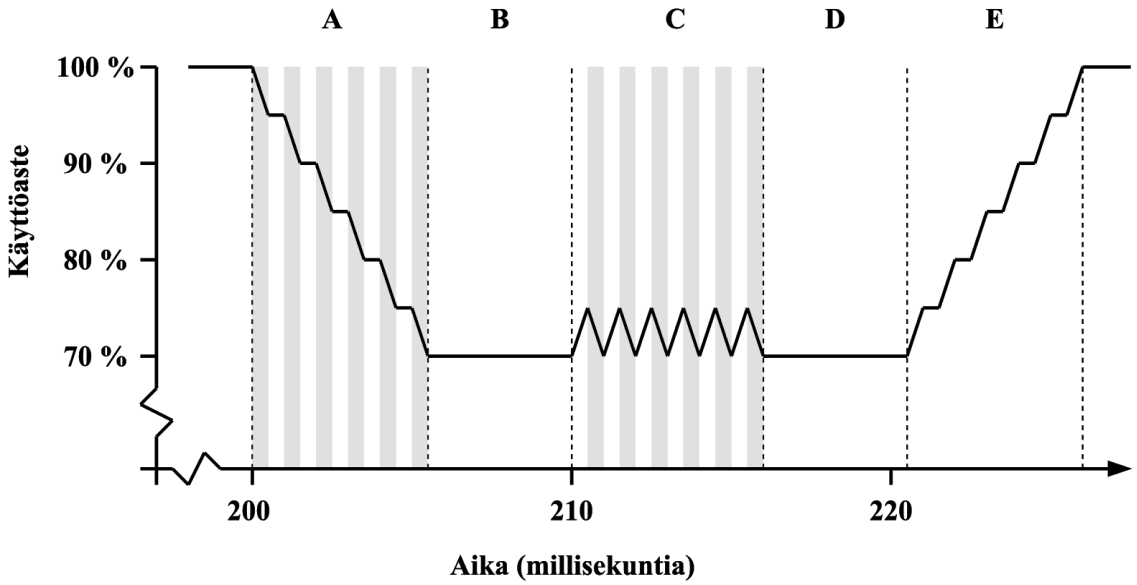
Keruukerran käynnistyessä Metronome-keräintä suoritetaan vaiheittain 500 mikrosekunnin pituisissa jaksoissa, joita seuraa aina vähintään yksi 500 mikrosekunnin pituinen mutaattorin suoritusjakso. Suoritusvuoro siis “heilahtelee” kuin metronomin neula yhden millisekunnin vaiheessa keräimen ja mutaattorin välillä.

Metronome-keräin tarkkailee myös mutaattorin käyttöastetta ja pyrkii antamaan mutaattorille vähintään 70 prosentin osuuden 10 millisekunnin pituisesta liukuvasta aikaikkunasta – 10 millisekunnin aikaikkunassa sallitaan siis enintään kuusi roskankeräimen suoritusjaksoa (yhteensä 3 millisekuntia), jolloin mutaattori saa vähintään neljätoista suoritusjaksoa (yhteensä 7 millisekuntia).

Kuvassa 4.1 on kuvattu Metronome-keräimen ja mutaattorin vuorottaminen yhden keruukerran aikana. Jaksossa A roskankeräintä ja mutaattoria suoritetaan vuorotellen, kunnes mutaattorin käyttöaste on laskenut 70 prosentin tavoitetasolle 10 millisekunnin liukuvassa aikaikkunassa. Jaksossa B mutaattorille annetaan yhtäjaksoista suoritusaikaa siihen asti, kun keruukerran alkamisesta on tullut kuluneeksi 10 millisekuntia. Jaksossa C roskankeräintä ja mutaattoria suoritetaan vuorotellen, jolloin mutaattorin käyttöaste heilahtelee 70 ja 75 prosentin välillä. Jaksossa D roskankeruu on päättynyt, mutta mutaattorin käyttöaste 10 millisekunnin aikaikkunassa säilyy vielä 70 prosentissa. Jaksossa E mutaattorin käyttöaste nousee takaisin 100 prosenttiin.

Mikäli kuvassa 4.1 kuvattu roskankeruukerta olisi kestoltaan pidempi, jaksot B ja C toistuisivat vuorotellen siihen asti, kun roskankeruukerta lopulta päättyy.

Jotta Metronome-keräin voi toimia ennustettavasti, sille annettavien parametrien tulee vastata sovelluksen muistinkäyttöä. Nämä parametrit ovat sovelluksen muistinvarausnopeus, roskankeräimen vapautusnopeus sekä mutaattorin ja roskankeräimen suoritusjaksojen pituudet. Jos esimerkiksi sovelluksen keskimääräinen muistinvarausnopeus arvioidaan liian matalaksi, ei roskankeräin ehdi vapauttaa muistia yhtä nopeasti kuin mitä mutaattori varaa.



Kuva 4.1: Mutaattorin käyttöaste Metronome-keräimen keruukerran aikana – vaaka-akselilla harmaalla merkittyinä jaksoina suoritetaan roskankeräintä ja valkoisella merkittyinä jaksoina mutaattoria.

5 Java-kielen roskankeräimet

Java-kielen virtuaalikoneeseen on määritelty roskankeruurajapinta, jonka avulla roskankeräin voi hallinnoida virtuaalikoneen kekoa. Roskankeräimen tehtävänä on sijoittaa Java-kielisen ohjelman varaamat oliot kekoon sekä löytää ja vapauttaa oliot, joihin ohjelma ei enää viittaa. Roskankeräin voi keskeyttää ohjelman suorituksen tilapäisesti, ja näiden suoritustaukojen määrä ja pituus riippuu käytetystä roskankeräimestä.

Tutkielmassa tarkastellaan lähemmin neljää Java-kielen roskankeruutoteutusta. Kaksi näistä ovat Java-kielen vanhat keräimet Parallel Collector (jäljempänä "Parallel-keräin") ja Concurrent Mark Sweep Collector (jäljempänä "CMS-keräin"). Parallel-keräin oli Javan oletuskeräin kielen versioissa 4–8, ja CMS-keräin oli tarkoitettu tämän vaihtoehdoksi tilanteissa, joissa suoritusviiveistä haluttiin lyhyempiä.

Kolmas tarkasteltava keräin on Javan uusi oletuskeräin Garbage-First Collector (jäljempänä "G1-keräin"), joka suunniteltiin takaamaan Parallel-keräintä lyhyempiä suoritusviiveitä. Neljäs keräin on Javaan versiosta 12 alkaen sisältynyt Shenandoah Collector (jäljempänä "Shenandoah-keräin"), joka on suunniteltu takaamaan vielä G1-keräintä huomattavasti lyhyempiä vasteaikoja (Flood et al. 2016).

5.1 Parallel-keräin

Javan varhaisimmissa versiossa oletuskeräimenä oli Serial-keräin (Serial Collector), joka keskeytti mutaattorin suorituksen siksi aikaa, kun roskankeräin suoritti yhdessä säikeessä keruukerran alusta loppuun. Javan versiossa 4 tämän korvasi Parallel-keräin, joka suoritti nuoren sukupolven kopioivan keruun rinnakkaisesti usealla säikeellä – oletusarvoisesti roskankeruusäikeitä oli yksi jokaista suoritusympäristön suoritinydintä kohden. Näin nuoren sukupolven keruukerrat saatiin suoritettua nopeammin.

Parallel-keräin on niin sanottu throughput-keräin, joka pyrkii käyttämään mahdollisimman vähän suoritinaikaa roskankeruuseen. Vanhan sukupolven jäljittävä keruu ei ole rinnakkainen, ja keruu suoritetaan koko vanhalle sukupolvelle. Vanhan sukupolven keruutauot ovat siksi sitä pidempiä mitä suurempaa kekoa virtuaalikone käyttää, ja näiden keruutaukojen pitkä kesto oli Parallel-keräimen suurin ongelma.

Parallel-keräin oli Javan oletuskeräin vuosina 2002–2017 versioon 8 asti ja on siksi mukana tutkielman analyysissa verrokkina.

5.2 CMS-keräin

Java-kielen versiossa 4 uuden Parallel-keräimen rinnalle tarjottiin vaihtoehdoksi CMS-keräin, joka suorittaa vanhan sukupolven jäljityksen rinnakkaisesti mutaattorin suorituksen kanssa. Mutaattorin säikeet pysäytettiin vanhan sukupolven keruun aikana kaksi kertaa, ensimmäisen kerran keruun alussa ja toisen kerran lähellä keruun puoliväliä, mutta valtaosa keruusta tapahtui rinnakkaisesti mutaattorin suorituksen kanssa.

Nuoren sukupolven roskankeruu suoritettiin samankaltaisella kopioivalla keruulla kuin Parallel-keräimessä, mutta nuoren sukupolven keruu saatettiin suorittaa myös vanhan sukupolven keruukerran ollessa kesken. Nuoren sukupolven keruu keskeytti mutaattorin suorituksen samaan tapaan kuin Parallel-keräintä käytettäessä.

CMS-keräin pyrkii saamaan vanhan sukupolven keruun valmiiksi ennen kuin vanha sukupolvi täyttyy. Jos tässä ei kuitenkaan onnistuta vaan keko täyttyy, CMS-keräin pysäyttää kaikki mutaattorin säikeet siksi aikaa, että vanhan sukupolven roskankeruu on suoritettu loppuun.

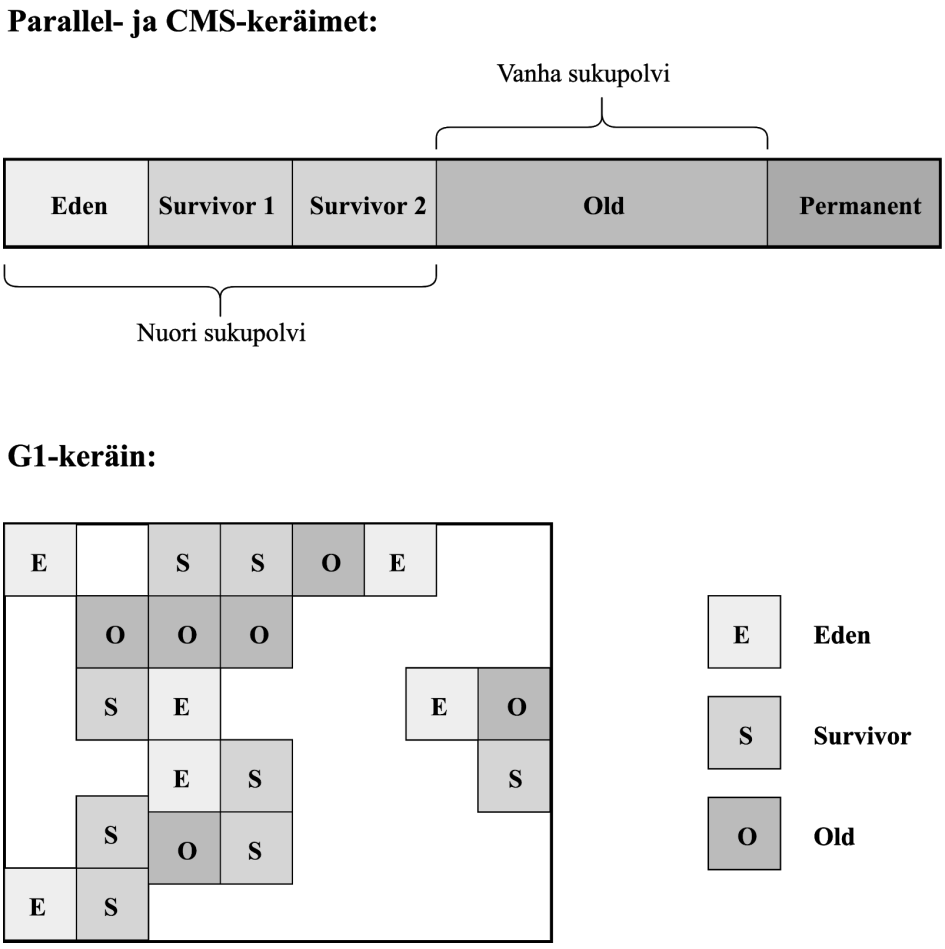
CMS-keräimen merkittäviä haittoja olivat huomattavasti Parallel-keräintä suurempi laskenta-aika sekä vanhan sukupolven pirstaloituminen, sillä vanhan sukupolven muistialuetta ei pyritty tiivistämään (Grgic, Mihajlevic & Radovan 2018).

5.3 G1-keräin

Java-kielen versiossa 9 virtuaalikoneen oletuskeräimeksi tuli uusi G1-keräin. G1-keräin suunniteltiin takaamaan suurella todennäköisyydellä huomattavasti Parallel-keräintä lyhyempiä suoritusviiveitä kuitenkin niin, että keruun tarvitsema laskenta-aika ei ole merkittävästi suurempi. G1-keräin suorittaa koko kekoon kohdistuvat työvaiheet, kuten merkinnän, rinnakkaisesti mutaattorin säikeiden kanssa. Näin keräin skaalautuu hyvin myös ympäristöihin, joissa on paljon rinnakkaista laskentakapasiteettia tai virtuaalikone käyttää hyvin suurta kekoa (Grgic, Mihajlevic & Radovan 2018).

Aiemmissa luvussa kuvatut Parallel- ja CMS-keräimet osittavat Java-virtuaalikoneen keosta yhtenäiset osiot nuoren ja vanhan sukupolven käyttöön. Näistä nuoren sukupolven muistiosio on jaettu edelleen kolmeen osioon Eden, Survivor 1 ja Survivor 2; uudet oliot varataan Eden-osioon, ja nuoren sukupolven kopioivan keruun “selvinneet” oliot ovat Survivor-osioissa, jotka vuorottelevat kopioivan keruun lähde- ja kohdeosioina. Riittävän monen kopiointikerran jälkeen olio siirtyy lopulta vanhaan sukupolveen.

G1-keräin jakaa keon suureen joukkoon keskenään yhtä suuria osioita. Osaa näistä osioista käytetään nuorena sukupolvena eli uudet oliot sijoitetaan niihin. Roskankeruun yhteydessä nuoren sukupolven osioista kopioidaan viitatut muistialueet tyhjiin osioihin, jotka kopioitavan muistialueen iästä riippuen ovat joko Survivor-osioita tai vanhan sukupolven osioita. Vanhan sukupolven osioista muistialueet kopioidaan aina vanhan sukupolven osioihin. Parallel- ja CMS-keräimien sekä G1-keräimien tavat osittaa virtuaalikoneen keko ovat esitetty kuvassa 5.1.



Kuva 5.1: Yllä Parallel- ja CMS-keräimien tapa osittaa virtuaalikoneen keko sukupolvimalli mukaan yhtenäisiin osiin, alla G1-keräimen hienojakoinen ositus.

G1-keräin suorittaa koko keon kattavan merkintävaiheen rinnakkaisesti mutaattorin suorituksen kanssa. Merkintävaiheen jälkeen G1-keräin tietää, mitkä osiot sisältävät vain vähän viitattuja muistialueita, ja se vapauttaa nämä alueet ensimäisenä – tästä tulee keräimen nimi “garbage first” eli “roskat ensin”. Mutaattorin säikeet pysäytetään kopioinnin ajaksi.

G1-keräin ei ole varsinainen reaaliaikakeräin, mutta se voi hallita kopioinnista aiheutuvaa tauon pituutta säätämällä kerrallaan kopioitavien muistiosioiden määrää. G1-keräimen roskankeruutaukojen enimmäispituudelle voidaan asettaa tavoiteaika, jolloin keräin arvioi aikaisempien keruukertojen perusteella, kuinka monta muistiosiota tavoiteajan sisällä ehditään kopioida. Luvussa 4.4 kuvatun Metronome-keräimen tapaan G1-keräimelle on myös mahdollista määrittää mutaattorin keruutaukojen välissä saaman suoritusvuoron kesto (Grgic, Mihajlevic & Radovan 2018).

5.4 Shenandoah-keräin

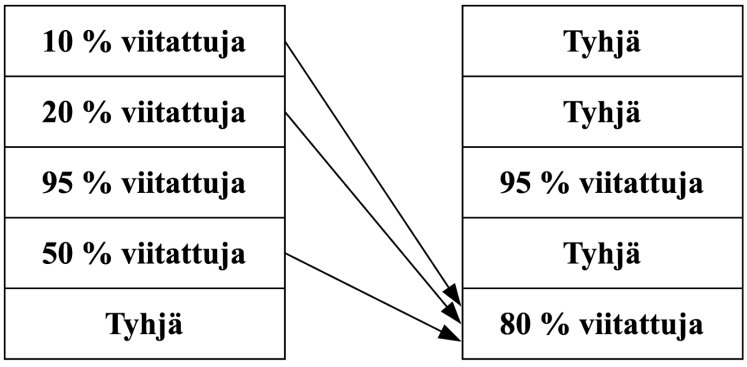
Shenandoah-keräin on Java-kieleen versiosta 12 alkaen sisältynyt reaaliaikasovelluksille suunniteltu keräin, jonka ovat kuvanneet Flood et al. (2016). Shenandoah-keräimen keruutekniikka perustuu muun muassa Brooksin (1984) versioon luvussa 3.4 kuvatusta vaiheittaisesta kopioivasta keruusta sekä luvussa 4.3 kuvattuun Metronome-keräimeen.

Shenandoah pyrkii takaamaan vielä huomattavasti lyhyempiä keruutaukoja kuin mitä G1-keräin kykenee saavuttamaan. Toisaalta Shenandoah’ta ei ole tarkoitettu yleiskeräimeksi, vaan sekä keräimen aika- että tilavaatimukset voivat olla selvästi suurempia kuin aiemmin mainituilla keräimillä.

Toisin kuin muut tutkielmassa käsitellyt roskankeräimet, Shenandoah-keräin ei käytä sukupolvimallia keon osituksen ja roskankeruun perusteena. Kuten G1-keräin, Shenandoah jakaa keon suureen määrään keskenään samankokoisia osioita, mutta näistä jokainen voi sisältää kaikenikäisiä olioita. G1-keräimen tapaan Shenandoah-keräin pyrkii luomaan ohjelman muistialueelle mahdollisimman suuria yhtenäisiä vapaita alueita.

Shenandoah-keräimen oletusasetuksilla keruu käynnistyy työperustaisesti, kun keosta on käytetty 75 prosenttia. Keräin pysäyttää mutaattorin kaksi kertaa keruukerran aikana, ja ensimmäinen keruutauko tapahtuu merkintävaiheen alussa, jolloin juurijoukon merkintä tapahtuu mutaattorin ollessa pysäytettynä. Tämän jälkeen merkintävaihe etenee rinnakkaisesti mutaattorin kanssa. Merkintävaiheen lopussa mutaattori pysäytetään lyhyesti uudestaan ja juurijoukosta kopioitaviin olioihin olevat viittaukset päivitetään osoittamaan kohdeosiolle luotuihin kopioihin.

Shenandoah-keräin valitsee kopioitavaksi keon sellaisia osioita, joissa on vain vähän viitattuja olioita, ja kopioi näillä osioilla olevat viitatut oliot tyhjälle osiolle. Jokaisella kopioinnilla on siis monta lähdeosiota ja yksi kohdeosio, ja näin jokainen keruukerta tiivistää osan keosta ja luo kekoon vapaita osioita. Tämä on esitetty kuvassa 5.2. Kopiointivaihe suoritetaan loppuun rinnakkaisesti mutaattorin kanssa, minkä jälkeen keruukerta päättyy (Flood et al. 2018).



Kuva 5.2: Keon tiivistys Shenandoah-keräimellä, mukaillen Flood et al. (2016).

Taulukossa 5.1 on yhteenveto tutkielman vertailussa mukana olevien Java-kielen roskainkeräimien piirteistä. Taulukossa on eritelty nuoren ja vanhan sukupolven keruutekniikat sukupolvimallia käyttävien keräimien kohdalla (Carpen-Amarie et al. 2015; Flood et al. 2016).

	Nuori sukupolvi				Vanha sukupolvi			
	Rinnakkainen	Tiivistävä	Samanaikainen merkintä	Samanaikainen tiivistys	Rinnakkainen	Tiivistävä	Samanaikainen merkintä	Samanaikainen tiivistys
Parallel-keräin	✓	✓				✓		
CMS-keräin	✓	✓			✓		✓	
G1-keräin	✓	✓			✓	✓	✓	
Shenandoah-keräin					✓	✓	✓	✓

Taulukko 5.1: Java-kielen roskankeräimien keruutekniikoiden piirteet.

6 Keräimien suorituskyvyn vertailu

Tutkielman kokeellisessa osuudessa vertaillaan edellisessä luvussa kuvattujen neljän Java-kielen roskankeräimen suorituskykyä. Keräimiä tarkastellaan erityisesti reaaliaikaisuuden näkökulmasta eli selvitetään, kuinka pitkiä viiveitä ne aiheuttavat mutaattorin suoritukselle.

Java-kielen versioiden 4–8 oletuksena käyttämä Parallel-keräin on suunniteltu käyttämään roskankeruuseen mahdollisimman vähän laskenta-aikaa, ja se on siksi hyvä vertailukohta muiden keräimien suoritusteholle. Toisaalta Parallel-keräimen suurin ongelma ovat vanhan sukupolven roskankeruusta seuraavat pitkät viiveet, joita muut vertailtavat keräimet pyrkivät eri tekniikoilla pienentämään.

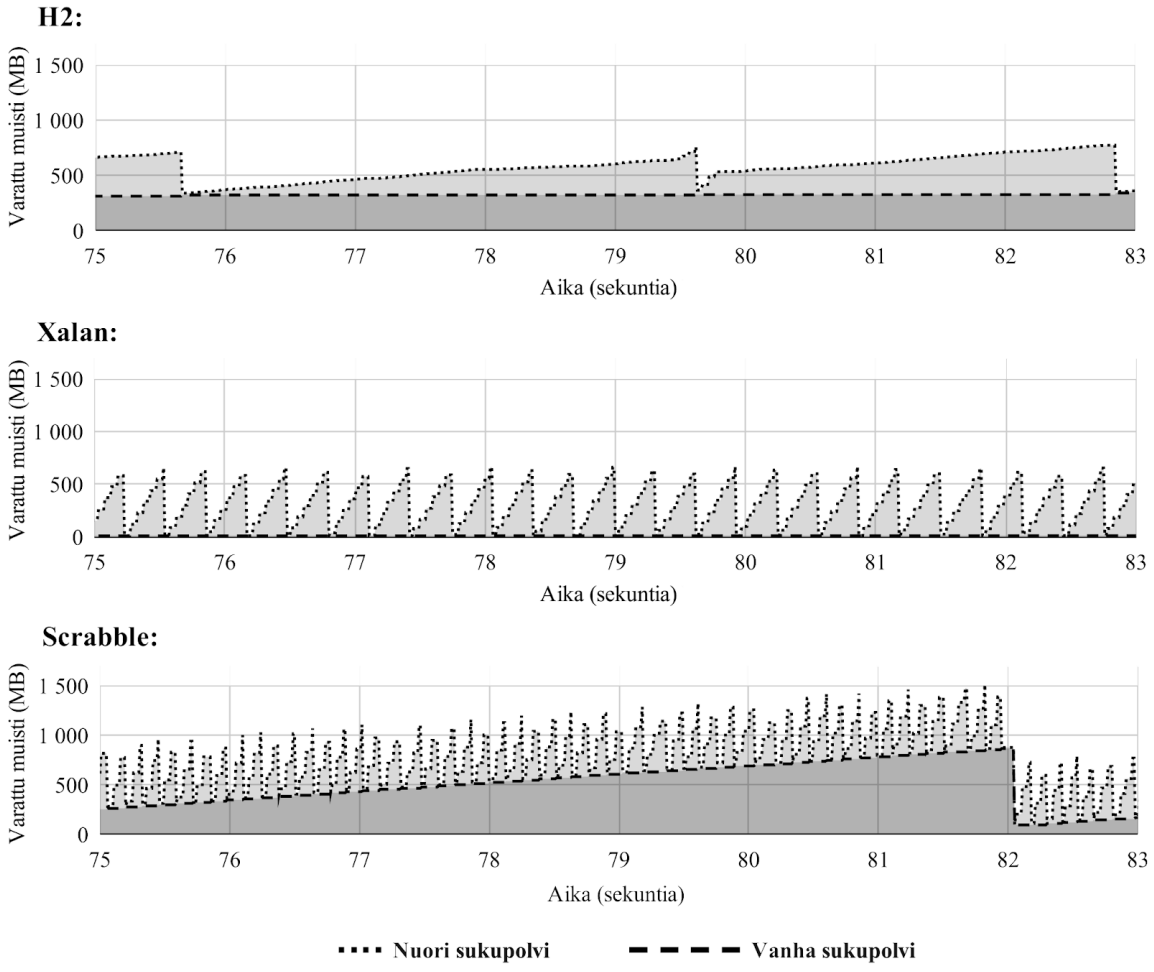
6.1 Testiasetelma

Roskankeräimien testit on suoritettu tarkoitusta varten asennetussa Ubuntu Server 19.10 GNU/Linux-ympäristössä, johon on asennettu OpenJDK:n versio 13. Käyttöjärjestelmän ajastetut ylläpitotoiminnot kytkettiin testien ajaksi pois päältä. Käytetyssä koneessa on neliytiminen Intel Core i5 -suoritin ja 24 gigatavua keskusmuistia.

Tarkasteltuja roskankeräimiä vertailtiin kolmella erilaisella testillä, jotka on suunniteltu Java-virtuaalikoneiden suorituskyvyn testaamiseen. Kaksi testeistä valittiin DaCapo-testikokoelmasta (Blackburn et al. 2006); näistä ensimmäinen suorittaa transaktioita sovelluksen muistissa sijaitsevaan H2-relaatiotietokantaan ja toinen muuntaa XML-dokumentteja HTML-dokumenteiksi Apache Xalan -kirjaston avulla. Kolmanneksi testiksi valittiin Renaissance-testikokoelmasta (Prokopec et al. 2019) testi Scrabble, joka pelaa Scrabble-sanapeliä itseään vastaan. Valittuja kolmea testiä kutsutaan jäljempänä lyhyesti nimillä “H2”, “Xalan” ja “Scrabble”.

Kaikkien kolmen testin suoritusaika on melko lyhyt – H2-testin noin 20 sekuntia, Xalan-testin noin 4 sekuntia ja Scrabble-testin vain noin 350 millisekuntia. Testejä toistettiin niin monta iteraatiota, että testiajon pituus nopeimmalla keräimellä oli noin 10 minuuttia. Testien iteraatiot ovat keskenään samanlaisia, eli ne suorittavat samat työvaiheet samalle syötteelle.

Testeissä Java-virtuaalikoneen keko (heap) asetettiin 500 megatavun, 2 gigatavun tai 8 gigatavun kokoiseksi. Keon koosta riippumatta sen vähimmäis- ja enimmäiskoko asetettiin aina samaksi. Muutoin kaikkia neljää roskankeräintä käytettiin niiden oletusparametreilla.



Kuva 6.1: Nuoren ja vanhan sukupolven muistinkulutus H2-, Xalan- ja Scrabble-testien aikana kahdeksan sekunnin pituisella jaksolla.

Testit valittiin sillä perusteella, että ne tuottaisivat roskankeräimille erityyppisiä kuormia. Kuvassa 6.1 on kuvattu nuoren ja vanhan sukupolven muistinkäyttö valituilla kolmella testillä 2 gigatavun kekoa ja Parallelkeräintä käytettäessä. Kuvaajaan on valittu mittausajoista kahdeksan sekunnin pituiset jaksot mittausajon keskeltä.

H2-testin keskimääräinen muistinvarausnopeus mitattiin hitaimmaksi, noin 100 megatavua sekunnissa. Testin edetessä tästä siirtyi vanhaan sukupolveen keskimäärin kolme megatavua sekunnissa. Testin muistinvarausnopeus ei ole kovin tasainen – esimerkiksi kuvassa 6.1 nopeus kiihtyy hetkellisesti kuvaajan puolivälissä nuoren sukupolven roskankeruukerran molemmin puolin.

Xalan-testin muistinvarausnopeus oli huomattavasti korkeampi eli noin 2,0 gigatavua sekunnissa. Nuoren sukupolven roskankeruu toistuu hyvin tasaisesti noin kolme kertaa sekunnissa. Nuoren sukupolven roskankeruu oli kuitenkin lähes täydellistä, ja vanhaan sukupolveen siirtyi vain noin 150 kilotavua sekunnissa.

Scrabble-testin muistinvarausnopeus oli korkein eli noin 4,4 gigatavua sekunnissa. Nuoren sukupolven roskankeruu toistuu 7–8 kertaa sekunnissa, ja kuvaajassa nähdään myös vanhan sukupolven nopea täyttyminen ja sen roskankeruu. Kuvaajassa käytetyllä 2 gigatavun keolla vanha sukupolvi kasvoi keskimäärin noin 80 megatavua sekunnissa.

Kunkin testiajon kesto mitattiin Linuxin komennolla `time`. Kunkin testiajon aikana suoritettujen keruukertojen määrät tarkistettiin OpenJDK:n omalla apuohjelmalla `jstat`, joka mittaa käynnissä olevan virtuaalikoneen toimintaa. Parametrilla `-gcutil` ohjelma ilmoittaa muun muassa nuoren ja vanhan sukupolven keruukerrat sekä näiden käyttämän suoritinajan. Lisäksi roskankeräimen lokitiedosto otettiin talteen virtuaalikoneen komentoriviparametrilla `-Xlog:gc*`, jotta lokitiedostosta voitiin tarkastella roskankeruun aiheuttamien suoritustaukojen pituuksia.

6.2 Mittaustulokset

Mittausajojen tulokset on listattu tutkielman liitteiden 1–3 taulukoissa. Liitteessä 1 on H2-testin tulokset, liitteessä 2 Xalan-testin tulokset ja liitteessä 3 Scrabble-testin tulokset. Kaikki kolme testiä toistettiin kaikilla neljällä keräimellä, ja lisäksi kullakin keräimellä erikseen 500 megatavun, 2 gigatavun ja 8 gigatavun keolla.

Taulukkoon 6.1 on otettu esimerkkinä liitteestä 1 H2-testin mittausten tulokset Parallel-keräimellä 500 megatavun keolla. Mittausajot toistettiin kolme kertaa, ja taulukon kahdella alimmalla rivillä on mittaustulosten keskiarvo ja suhteellinen keskihajonta. Mittausten suhteelliset keskihajonnat olivat pääsääntöisesti enintään muutaman prosentin suuruisia. Suurempia vaihteluita havaittiin lähinnä roskankeruun aiheuttamien taukojen maksimipituudessa. Jäljempänä mittausajojen tuloksia vertailtaessa käytetään tulosten keskiarvoa, ellei toisin mainita.

Taulukon ensimmäisessä sarakkeessa on mittausajon kesto sekunneissa. Seuraavissa viidessä sarakkeessa on tietoja nuoren sukupolven roskankeruusta. Sarakkeessa YGC (young garbage collection) on `jstat`-työkalun mittaama nuoren sukupolven roskankeruutaukojen määrä, ja sarakkeessa YGCT näiden taukojen kokonaiskesto. Seuraavissa sarakkeissa on roskankeräimen lokitiedostosta lasketut nuoren sukupolven keruutaukojen kestojen 90. ja 99. ja 100. persentiilit (eli maksimi).

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	576,9	457	4,9	14,5	27,5	43,8	15	2,8			188,2	346,3	386,8
2	589,1	458	4,8	14,6	28,0	56,5	16	2,8			164,6	329,1	333,3
3	582,8	461	5,1	15,9	28,7	45,4	15	2,6			163,5	263,4	439,6
ka.	**583,0**	**459**	**4,9**	**15,0**	**28,1**	**48,6**	**15**	**2,7**	-	-	**172,1**	**312,9**	**386,5**
kh.	**1,1%**	**0,5%**	**3,9%**	**5,3%**	**2,2%**	**14,2%**	**3,8%**	**3,9%**	-	-	**8,1%**	**14,0%**	**13,8%**

Taulukko 6.1: H2-testin tulokset Parallel-keräimellä ja 500 megatavun keolla.

Viimeisissä seitsemässä sarakkeessa on vastaavat tiedot vanhan sukupolven roskankeruusta. Sarakkeessa FGC (full garbage collection) on vanhan sukupolven roskankeruutaukojen määrä ja sarakkeessa FGCT näiden taukojen kokonaiskesto. Sarakkeissa CGC (concurrent garbage collection) ja CGCT on vastaavat tiedot CMS- ja G1-keräimien rinnakkaisille keruukerroille. Viimeisissä kolmessa sarakkeessa on roskankeräimen lokitiedostosta lasketut vanhan sukupolven keruutaukojen kestojen 90., 99. ja 100. persentiilit.

Huomionarvoisesti Shenandoah-keräin, joka ei käytä sukupolvimallia, raportoi kaikki keruukertansa sarakkeissa FGC ja FGCT.

H2-testin tulokset

Taulukkoon 6.2 on koostettu H2-testin tulokset kaikilla neljällä keräimellä ja kunkin keräimen kohdalla kaikilla kolmella erikokoisella keolla. Suoritusajan yhteyteen on merkitty sen pituus prosentteina suhteessa nopeimpaan aikaan, joka tässä testissä mitattiin Parallel-keräimellä 500 megatavun keolla.

Parallel-keräimellä vanhan sukupolven keruu aiheutti hyvin pitkiä taukoja 500 megatavun keolla. Suurempaa kekoa käytettäessä nuoren sukupolven keruu hidastuu jonkin verran, mutta toisaalta vanhan sukupolven keruuta ei jouduta suorittamaan kertaakaan. 2 gigatavun keolla vanhan sukupolven keruu olisi suoritettu ensimmäisen kerran 15 minuutin jälkeen ja 8 gigatavun keolla vasta noin vuorokauden jälkeen.

CMS-keräimellä mittausajon kesto kasvoi noin kymmenen prosenttia, ja nuoren sukupolven keruutaukojen kesto lyheni hieman. 500 megatavun keolla osaa vanhan sukupolven keruukerroista ei suoritettu rinnakkaisesti, ja nämä keruukerrat aiheuttivat vielä Parallel-keräintä pidempiä taukoja – CMS-keräin saattaa erikoistilanteissa pysäyttää mutaattorin, jos vanha sukupolvi täyttyy rinnakkaisen keruun ollessa vielä kesken. 2 gigatavun keolla vanhan sukupolven keruu suoritettiin kaksi kertaa, ja tästä seuranneet tauot olivat lyhyitä.

	Ajon kesto	Suhteellinen kesto	Nuoren sukupolven keruutaukojen määrä	90 %	Maksimi	Vanhan sukupolven keruutaukojen määrä	90 %	Maksimi
Parallel								
500 MB	583 s	100 %	459	15,0 ms	48,6 ms	15	172,1 ms	386,5 ms
2 GB	600 s	103 %	101	39,8 ms	87,4 ms			
8 GB	630 s	108 %	24	97,9 ms	125,4 ms			
CMS								
500 MB	648 s	111 %	405	22,7 ms	49,6 ms	44	156,4 ms	515,4 ms
2 GB	658 s	113 %	202	34,3 ms	53,2 ms	2	8,6 ms	20,4 ms
8 GB	679 s	116 %	203	54,4 ms	95,5 ms			
G1								
500 MB	615 s	105 %	408	44,5 ms	65,6 ms	237	1,6 ms	5,2 ms
2 GB	621 s	106 %	72	80,3 ms	93,7 ms	4	1,9 ms	2,4 ms
8 GB	658 s	113 %	33	140,9 ms	158,2 ms			
Shenandoah								
500 MB	731 s	125 %				2102	0,7 ms	2,2 ms
2 GB	689 s	118 %				273	0,8 ms	3,0 ms
8 GB	694 s	119 %				77	0,6 ms	1,3 ms

Taulukko 6.2: H2-testin tulokset.

G1-keräimellä mittausajon kesto kasvoi noin viisi prosenttia, mutta myös nuoren sukupolven keruutauot olivat hieman pidempiä kuin Parallel-keräimellä. Vanhan sukupolven keruutauot sen sijaan olivat hyvin lyhyitä, pisimmillään vain 5,2 millisekuntia.

Shenandoah-keräimellä keruutaukojen kestot ovat erittäin lyhyitä, pisimmillään kahden gigatavun keolla 3,0 millisekuntia. Toisin kuin muilla testatuilla keräimillä, Shenandoah-keräimellä mittausajon kesto on pisin pienintä kekoa käytettäessä. 2 gigatavun keolla kesto oli 18 prosenttia pidempi kuin Parallel-keräimellä.

Xalan-testin tulokset

Taulukkoon 6.3 on koostettu Xalan-testin tulokset. Kokonaisuudessaan keräimien väliset erot olivat pienempiä kuin H2-testissä sekä mittausajojen keston että keruutaukojen pituuksien osalta.

Xalan-testin suuremmasta muistinvarausnopeudesta johtuen nuoren sukupolven roskankeruu toistuu tiheämmin – 500 megatavun keolla yli kymmenen kertaa sekunnissa. Nuoren sukupolven roskankeruu on kuitenkin niin täydellistä, että vanhan sukupolven muistialue ei täyty millään keräimellä kertaakaan edes 500 megatavun kekoa käytettäessä.

	Ajon kesto	Suhteellinen kesto	Nuoren sukupolven keruutaukojen määrä	90 %	Maksimi	Vanhan sukupolven keruutaukojen määrä	90 %	Maksimi
Parallel								
500 MB	640 s	100 %	8308	0,7 ms	5,4 ms			
2 GB	638 s	100 %	1997	0,9 ms	4,6 ms			
8 GB	650 s	102 %	497	1,0 ms	7,0 ms			
CMS								
500 MB	660 s	103 %	10215	1,2 ms	22,5 ms			
2 GB	659 s	103 %	5126	4,2 ms	18,8 ms			
8 GB	772 s	121 %	5107	28,8 ms	52,4 ms			
G1								
500 MB	662 s	104 %	4541	1,5 ms	3,2 ms			
2 GB	675 s	106 %	1107	2,6 ms	8,1 ms			
8 GB	668 s	105 %	287	2,9 ms	6,9 ms			
Shenandoah								
500 MB	698 s	109 %				13862	0,3 ms	2,8 ms
2 GB	697 s	109 %				3288	0,3 ms	3,5 ms
8 GB	711 s	111 %				834	0,3 ms	3,9 ms

Taulukko 6.3: Xalan-testin tulokset.

Keruun täydellisyydestä johtuen nuoren sukupolven kopioiva keruu on myös Parallel- ja G1-keräimillä hyvin nopea, ja Shenandoah-keräimen keruutauot ovat vain hieman lyhyempiä. CMS-keräimellä keruutauot ovat selvästi muita keräimiä pidempiä.

Scrabble-testin tulokset

Taulukkoon 6.4 on koostettu Scrabble-testin tulokset. Scrabble-testin muistinvarausnopeus on vielä suurempi kuin Xalan-testissä, ja myös erot eri keräimien ja erikokoisten kekojen välillä ovat suurempia kuin muissa testeissä.

500 megatavun kekoa käytettäessä nuoren sukupolven keruu toistuu kymmeniä kertoja sekunnissa, ja myös vanhan sukupolven keruu toistuu tiheästi. Suuremmalla keolla vanha sukupolvi täyttyy huomattavasti hitaammin; Scrabble-testin yhdessä mittausajossa on 1700 iteraatiota, ja esimerkiksi Parallel-keräimellä 8 gigatavun keolla nuori sukupolvi täyttyy vain 1200 kertaa. Nuoren sukupolven keruu muuttuu siksi hyvin täydelliseksi.

G1-keräimellä mittausajon kesto kasvaa voimakkaasti 500 megatavun keolla, mutta suurempaa kekoa käytettäessä ero Parallel-keräimeen on noin kymmenen prosenttia. CMS-keräimellä ero on noin 30 prosenttia. Shenandoah-keräimellä mittausajon kesto kasvaa huomattavasti pientä kekoa käytettäessä – 500 megatavun keolla jopa yli kuusinkertaiseksi. 8 gigatavun keolla ero Parallel-keräimeen on edelleen 36 prosenttia.

Toisin kuin H2- ja Xalan-testeissä, Shenandoah-keräin ei pystynyt takaamaan lyhyitä, alle viiden millisekunnin keruuviiveitä, vaan pisimmät keruukerrat olivat samaa suuruusluokkaa kuin CMS- ja G1-keräimillä.

	Ajon kesto	Suhteellinen kesto	Nuoren sukupolven keruutaukojen määrä	90 %	Maksimi	Vanhan sukupolven keruutaukojen määrä	90 %	Maksimi
Parallel								
500 MB	748 s	116 %	20919	4,6 ms	35,0 ms	250	100,0 ms	114,3 ms
2 GB	707 s	110 %	4946	17,0 ms	28,4 ms	56	100,5 ms	137,6 ms
8 GB	643 s	100 %	1200	19,7 ms	38,6 ms			
CMS								
500 MB	858 s	133 %	24005	7,7 ms	17,7 ms	767	12,8 ms	18,6 ms
2 GB	827 s	129 %	12066	15,8 ms	26,1 ms	84	9,6 ms	19,9 ms
8 GB	926 s	144 %	12005	24,2 ms	56,6 ms	18	19,7 ms	34,8 ms
G1								
500 MB	961 s	149 %	12517	15,7 ms	26,8 ms	4175	1,0 ms	6,1 ms
2 GB	701 s	109 %	2657	18,3 ms	43,8 ms			
8 GB	696 s	108 %	667	19,7 ms	28,6 ms			
Shenandoah								
500 MB	4262 s	663 %				152851	0,8 ms	34,6 ms
2 GB	1416 s	220 %				24571	0,8 ms	52,3 ms
8 GB	859 s	136 %				4034	0,7 ms	41,8 ms

Taulukko 6.4: Scrabble-testin tulokset.

6.3 Keskustelu

Testeissä käytetyt virtuaalikoneen keon koot valittiin tietoisesti siten, että niistä pienin (500 megatavua) oli osalle testeistä ongelmallisen pieni. Käytettyjen kolmen testin muistin vähimmäistarve mitattiin 50 megatavun tarkkuudella, ja H2-testin vähimmäistarve oli noin 250 megatavua, Xalan-testin noin 50 megatavua ja Scrabble-testin noin 150 megatavua. Jos Java-virtuaalikoneen keko säädettiin tätä pienemmäksi, Parallel-keräintä käytettäessä testi keskeytyi muistin loppumiseen.

Esimerkiksi Hertzin ja Bergerin (2005) mukaan Java-sovellukset tarvitsevat moninkertaisesti suuremman keon kuin mikä on enimmillään viitattujen olioiden välitön muistintarve. Mitattujen muistin vähimmäistarpeiden perusteella 500 megatavun keko lienee täysin riittävä Xalan-testille, mutta H2- ja Scrabble-testeille 2 gigatavun keko on perustellumpi. Toisaalta Shenandoah-keräin saattaisi hyötyä jopa 8 gigatavua suuremmasta keosta, mutta tätä ei testattu.

Roskankeruuta ohjaavista Java-virtuaalikoneen parametreista mittausajossa käytettiin ainoastaan keon vähimmäis- ja enimmäiskokoa, jotka asetettiin samoiksi, jotta keon uudelleenmitoittamisesta ei seuraisi ylimääräisiä, suuria suoritusviiveitä. Muutoin mittausajoissa vertailtuja roskankeräimiä käytettiin niiden oletusparametreilla. Kaikissa vertailluissa keräimissä on lukuisia parametreja, joiden arvoja hienosäätämällä on mahdollista vaikuttaa epäsuorasti roskankeruun nopeuteen tai suoritusviiveisiin tiettyä sovellusta käytettäessä.

Käytetyistä kolmesta testistä Xalan ja Scrabble eivät välttämättä vastaa kovin hyvin tosielämän monimutkaisia Java-sovelluksia, sillä näiden laskenta koostui hyvin lyhyistä iteraatioista. Varausnopeus oli kummassakin testissä hyvin nopea, mutta erityisesti Xalan-testissä yhden iteraation aikana viitattujen ja vapautettujen olioiden määrä oli niin pieni, että nuoren sukupolven täyttyessä siellä oli hyvin vähän muuta kuin roskaa.

H2-testin muistintarve oli testeistä suurin, ja sen iteraatiot olivat pisimpiä, noin 20 sekunnin pituisia. Jokaisen iteraation alussa muistinvarausnopeus kiihtyi hetkellisesti, kun H2-tietokantaan luotiin testin käyttämä sisältö. Huolimatta H2-testin selvästi matalimmasta keskimääräisestä muistinvarausnopeudesta siinä tapahtui enemmän vanhan sukupolven keruita kuin Xalan-testissä. H2-testin muistinkäyttö saattaa siksi vastata muita testejä paremmin esimerkiksi interaktiivisen Java-sovelluksen muistinkäyttöä.

G1-keräimen taukojen tavoitekesto

Toisin kuin muille testatuille keräimille, G1-keräimelle voidaan asettaa tavoiteaika keruuviiveiden maksimikestolle. Keräimen oletusasetuksilla tavoitekestoa ei ole, ja suoritetuissa mittauksissa nuoren sukupolven keruuviiveet olivat yhtä pitkiä tai pidempiä kuin muilla keräimillä.

Tämän vuoksi H2-testin mittaukset päätettiin toistaa niin, että G1-keräimelle asetettiin tavoiteaika aloittaen 100 millisekuntista ja päättyen 10 millisekuntiin – keräimen oletusasetuksilla maksimiviiveeksi oli H2-testissä mitattu 93,7 millisekuntia. Mittaukset suoritettiin 2 gigatavun keolla, joka aiemmissa mittauksissa sopi hyvin kaikille keräimille. Mittaukset suoritettiin nyt pidempänä, jotta vanhan sukupolven keruu saatiin toistumaan kaikilla keräimillä. Mittausten tulokset on esitetty taulukossa 6.5.

Kun G1-keräimen keruutaukojen tavoitekestoa lyhennettiin, nuoren sukupolven keruutaukojen määrä kasvoi ja niiden maksimikesto lyheni 40–50 millisekunnin tasolle. 10 millisekunnin tavoitekestolla ei saavutettu enää lisähyötyä.

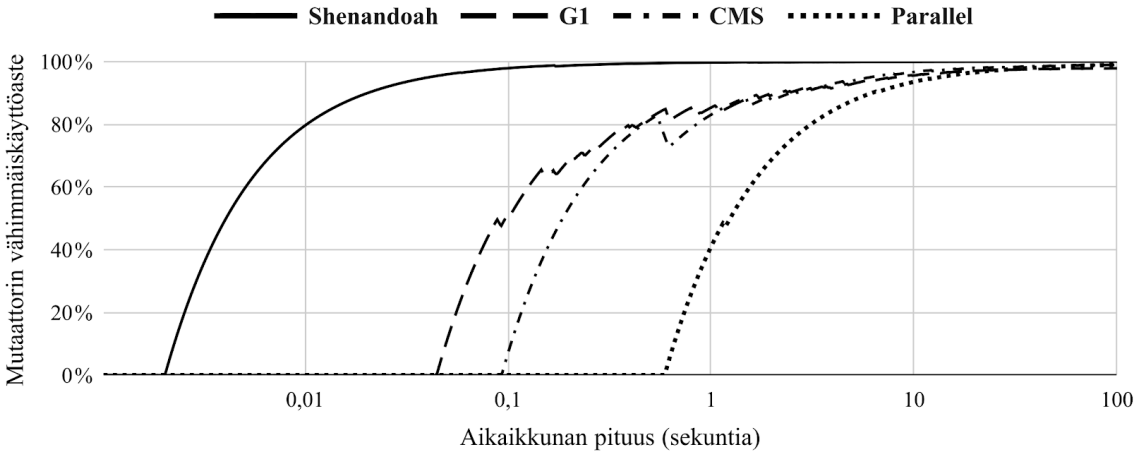
	Keruutaukojen tavoitekesto	Ajon kesto		Suhteellinen kesto	Nuoren sukupolven keruutaukojen määrä			Vanhan sukupolven keruutaukojen määrä		
					90 %	Maksimi		90 %	Maksimi	
Parallel										
2 GB		3448 s	100 %	549	34,3 ms	78,9 ms	10	193,2 ms	591,1 ms	
CMS										
2 GB		3958 s	115 %	1193	33,2 ms	76,8 ms	12	21,2 ms	26,2 ms	
G1										
2 GB		3707 s	108 %	324	77,2 ms	119,4 ms	40	2,0 ms	5,8 ms	
2 GB	100 ms	3819 s	111 %	350	80,4 ms	90,6 ms	42	2,0 ms	2,9 ms	
2 GB	80 ms	3708 s	108 %	467	68,1 ms	89,7 ms	30	1,9 ms	3,3 ms	
2 GB	60 ms	3792 s	110 %	974	51,4 ms	62,2 ms	28	2,4 ms	3,7 ms	
2 GB	40 ms	3638 s	106 %	2132	33,9 ms	47,4 ms	30	2,4 ms	3,8 ms	
2 GB	20 ms	3781 s	110 %	3406	26,7 ms	44,5 ms	32	2,3 ms	2,9 ms	
2 GB	10 ms	3635 s	105 %	3395	26,6 ms	46,6 ms	32	2,3 ms	2,8 ms	
Shenandoah										
2 GB		4090 s	119 %				1477	0,8 ms	2,0 ms	

Taulukko 6.5: H2-testin tulokset, kun G1-keräimelle on asetettu tavoiteaika keruukertojen maksimikestolle.

Mutaattorin vähimmäiskäyttöaste

Suoritusviiveiden keski- tai maksimipituudet eivät yksin kuvaa riittävän hyvin roskankeräimen soveltuvuutta reaaliaikasovelluksiin. Myös keruutaukojen jakautumisella on merkitystä, sillä mutaattorin tulee saada riittävästi suoritusaikaa taukojen välissä. Esimerkiksi aiemmin luvussa 4.4 kuvattu Metronome-keräin antaa jokaisen keruutauon jälkeen mutaattorille vähintään yhtä pitkän jakson suoritusaikaa.

Cheng ja Blelloch (2001) kuvaavat mutaattorille taattua suoritusaikaa vähimmäiskäyttöasteella (minimum mutator utilisation). Vähimmäiskäyttöaste ilmoitetaan ajan funktiona, eli mutaattorin vähintään saamana osuutena suoritinajasta tietyn mittaisessa liukuvassa aikaikkunassa.



Kuva 6.2: Mutaattorin vähimmäiskäyttöaste H2-testille toistetuissa mittauksissa.

Kuvassa 6.2 on esitetty edellisen luvun H2-testin roskankeräimien lokitiedostoista lasketut mutaattorin vähimmäiskäyttöasteet. Testi suoritettiin noin tunnin pituisena, ja myös vanhan sukupolven keruu suoritettiin kaikilla keräimillä ainakin kymmenen kertaa. G1-keräimen kohdalla mukaan on valittu mittausajo, jossa keruuaikojen tavoitekestoksi oli asetettu 20 millisekuntia.

Vaaka-akselilla on liukuvan aikaikkunan pituus ja pystyakselilla mutaattorin vähimmäiskäyttöaste. Kunkin keräimen kuvaaja kohtaa siten vaaka-akselin kohdassa, joka vastaa mittausajon pisintä keruutaukoa. Kuvaaja on rajattu oikeassa laidassa 100 sekuntiin, sillä H2-testin roskankeruu oli kokonaisuudessaan kevyttä, ja kaikki keräimet pystyivät takaamaan 100 sekunnin aikaikkunasta yli 99 prosenttia mutaattorille.

Lyhyemmissä aikaikkunoissa keräimien välillä oli kuitenkin suuria eroja. Parallel-keräimen kuvaaja laskee voimakkaasti, sillä sen pisimmät keruutauot olivat yli puolen sekunnin pituisia. G1- ja CMS-keräimet pystyivät takaamaan yhden sekunnin aikaikkunasta mutaattorille yli 80 prosenttia. Shenandoah-keräin pystyi takaamaan mutaattorille yli 80 prosenttia jopa 10 millisekunnin aikaikkunasta, sillä keruutauot olivat paitsi hyvin lyhyitä (pisimmillään 2,0 millisekuntia) myös riittävän harvaan jakautuneita.

6.4 Johtopäätökset

Mittauksissa löydettiin merkittäviä eroja testattujen roskankeräimien välillä. Xalan- ja Scrabble-testeissä myös vanha Parallel-keräin pystyi riittävän suurella keolla takaamaan lyhyitä suoritusviiveitä, sillä näissä testeissä oliot olivat keskimäärin hyvin lyhytikäisiä. Keon kokoa kasvattamalla nuoren sukupolven keruu muuttui siksi hyvin täydelliseksi. Tällaisen sovelluksen kohdalla myös Parallel-keräin voi tarjota verraten lyhyitä suoritustaukoja, ja vanhan sukupolven keruu voidaan suorittaa ajastetusti sovelluksen kannalta sopivaan aikaan – esimerkiksi kerran vuorokaudessa.

Kun G1-keräimen keruutauoille asetettiin tavoite, suoritusviiveet olivat lyhyempiä kuin CMS-keräimellä. Lisäksi G1-keräimellä mittausajojen kesto oli vain 5–10 prosenttia pidempi kuin Parallel-keräimellä, siinä missä CMS-keräintä käytettäessä ero oli suurempi, noin 15 prosenttia. Esimerkiksi Carpen-Amarie et al. (2015) ja Grgic, Mihajlevic ja Radovan (2018) ovat tehneet G1-keräimen suorituskyvystä samansuuntaisia havaintoja. CMS-keräimellä ei tästä näkökulmasta ole etua G1-keräimeen verrattuna, ja toisaalta G1-keräin vaikuttaisi puolustavan hyvin paikkaansa Javan uutena oletuskeräimenä, joka pystyy korvaamaan sekä Parallel- että CMS-keräimen.

Tutkielman vertailussa mukana ollut Shenandoah ei ole ainoa Java-kielen roskankeräin, joka on suunniteltu takaamaan hyvin lyhyitä suoritusviiveitä. OpenJDK:hon on versiosta 11 alkaen sisältynyt myös Z Garbage Collector (Lidén & Karlsson 2018), joka on vielä kokeellinen Shenandoah'n kaltainen, rinnakkaisesti kekoa tiivistävä keräin. Taatakseen lyhyitä vasteaikoja Z-keräin tulee kuitenkin parametrisoida tarkasti kullekin suoritettavalle sovellukselle, eikä se siksi ollut mukana tutkielman vertailussa.

Erityisesti reaaliaikasovelluksille suunnitelluista keräimistä voidaan mainita myös Azul Systemsin kehittämä C4-keräin (Tene, Iyengar & Wolf 2011). C4-keräin on osa Azul Systemsin kaupallista Java-suoritusympäristöä, johon sisältyy Zing-virtuaalikone ja erityisesti tälle suunniteltu laitteisto (Jones, Hosking & Moss 2011, s. 355–361).

Shenandoah käyttää enemmän laskenta-aikaa kuin muut vertailussa mukana olleet keräimet. Osa tästä erosta syntyy siitä, että Shenandoah-keräintä käytettäessä kaikki mutaattorin suorittamat olioviitteiden luku-operaatiot tehdään Brooksin (1984) tyylisen lukuesteen läpi. Flood et al. (2016) mittasivat lukuesteen osuudeksi 2–6 prosenttia eri sovellusten käyttämästä suoritinajasta. Lisäksi Shenandoah-keräimen tilantarve on muita vertailtuja keräimiä suurempi. Shenandoah-keräimellä roskankeruusta koituvat viiveet olivat kuitenkin huomattavasti lyhyempiä kuin muilla keräimillä – osassa mittauksista pisimmilläänkin vain 2 millisekunnin luokkaa.

Tutkielmassa käytetyt testit ovat Java-sovelluksina verraten yksinkertaisia, eivätkä välttämättä vastaa kovin hyvin monimutkaisten Java-sovellusten roskankeräimille tuottamaa kuormaa. Shenandoah-keräimen kehittäjät (Flood et al. 2016) vertailivat keräimiä DaCapo-testikokoelman lisäksi myös WWW-sisältöä indeksoivalla ElasticSearch-sovelluksella. Myös tässä testissä Shenandoah'n keruuviiveet olivat selvästi muita OpenJDK:n keräimiä lyhyempiä.

Javan uusi G1-keräin voi tulosten perusteella pystyä täyttämään sovelluksen reaaliaikaisuudelle asetetut vaatimukset, mikäli ne eivät ole kovin tiukkoja. Lisäksi G1-keräimen tilantarve on samaa luokkaa kuin Parallel-keräimellä, ja ero ohjelman suoritusnopeudessa on pieni. Mikäli sovelluksen reaaliaikaisuudelle asetetut vaatimukset ovat tiukempia ja mikäli virtuaalikoneen keko voidaan asettaa riittävän suureksi, Shenandoah-keräin voi pystyä takaamaan huomattavasti lyhyempiä, vain muutaman millisekunnin viiveitä.

7 Yhteenveto

Roskankeruulla tarkoitetaan automaattista muistinhallinnan mekanismia, jossa roskankeräin vapauttaa sovelluksen varaamat muistialueet, joihin sovellus ei enää viittaa. Keskeisiä roskankeruun perustekniikoita ovat muistiviitteiden laskenta ja jäljittävät keruutekniikat, kuten mark-sweep-keruu ja kopioiva keruu.

Reaaliaikaisissa ja interaktiivisissa sovelluksissa roskankeruusta koituvat suoritusviiveet eivät saa olla liian pitkiä. Tällaisissa sovelluksissa keruuta ei voida toteuttaa yhtenä atomisena operaationa, jonka ajaksi ohjelman suoritus keskeytyy. Sen sijaan roskankeruu voidaan kohdistaa vain osaan ohjelman muistista, tai roskankeruu voidaan toteuttaa etenemään samanaikaisesti ohjelman suorituksen kanssa. Varsinaiset reaaliaikaiset keruutekniikat vuorottavat roskankeräimen suorituksen siten, että keruusta aiheutuvat viiveet ovat tarkkaan ennakoituja.

Roskankeruun sukupolvimalli perustuu havaintoon, että valtaosa ohjelman varaamista muistialueista on lyhytikäisiä. Jos eri-ikäiset muistialueet sijoitetaan omiin muistiosioihinsa, voidaan roskankeruukerrat kohdistaa vain osaan ohjelman koko muistialueesta. Lisäksi eri-ikäisten muistialueiden roskankeruussa voidaan tällöin käyttää erilaisia keruutekniikoita.

Samanaikainen roskankeruu voidaan toteuttaa vaiheittaisena siten, että yksi keruukerta on jaettu useaan työvaiheeseen, jotka on lomitettu ohjelman suorituksen kanssa. Roskankeruun perustekniikoista viitteiden laskenta on luonnostaan vaiheittainen, sillä muistialueet vapautetaan muistiviitteiden päivityksen yhteydessä, mutta myös jäljittävät keruutekniikat voidaan toteuttaa vaiheittaisina. Rinnakkaiset roskankeräimet toimivat aidosti rinnakkaisesti ohjelman suorituksen kanssa siten, että ohjelman suoritus joudutaan keskeyttämään vain pieneksi osaksi roskankeräimen suoritusajasta.

Tutkielmassa vertailtiin Java-kielen roskankeräimiä erilaisilla työkuormilla ja erikokoisilla muistialueilla. Mittauksissa tarkasteltiin mittausajojen kestoa, roskankeruutaukojen kestoa sekä taukojen jakautumista ohjelman suorituksen ajalle. Mittauksissa löydettiin merkittäviä eroja vertailtujen keräimien välillä.

Java-kielen uusi G1-keräin suorittaa koko muistiin kohdistuvan merkintävaiheen rinnakkaisena, ja kopiointivaihe kohdistetaan kerrallaan vain pieneen osaan ohjelman muistista. G1-keräin oli suoritetuissa mittauksissa vain hieman hitaampi kuin vanha Parallel-keräin, mutta G1-keräimen keruutauot olivat huomattavasti lyhyempiä. Kun G1-keräimen keruutauoille asetettiin tavoitekesto, viiveet olivat pisimmillään vain muutamia kymmeniä millisekunteja.

Java-kieleen on kehitetty useita roskankeräimiä, jotka on tarkoitettu erityisesti sellaisiin sovelluksiin, joissa suoritusviiveiden tulee olla lyhyitä. Näistä keräimistä vertailussa oli mukana Shenandoah-keräin, jota käytettäessä pisimmät suoritusviiveet saattoivat olla vain muutamia millisekunteja. Lisäksi suoritusviiveiden havaittiin jakautuneen tasaisesti ohjelman suorituksen ajalle. Toisaalta Shenandoah-keräimen tilantarve on suurempi kuin muilla vertailluilla keräimillä, ja keruun laskenta-aika oli kaikissa mittauksissa vertailun suurin.

Lähteet

Baker, H 1978, *List processing in real time on a serial computer*, Communications of the ACM, 21(4):280–294.

Bacon, D, Cheng, P & Rajan, V 2003, *Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java*, LCTES 2003, 81-92.

Boehm, H-J, Demers, A & Shenker, S 1991, *Mostly parallel garbage collection*, ACM SIGPLAN Notices, 26(6):157–164.

Blackburn, S, Garner, R, Hoffmann, C, Khan, A, McKinley, K, Bentzur, R, Diwan, A, Feinberg, D, Frampton, D, Guyer, S, Hirzel, M, Hosking, A, Jump, M, Lee, H, Moss, E, Phansalkar, A, Stefanović, D, VanDrunen, T, von Dincklage, D & Wiedermann, B 2006, *The DaCapo benchmarks: Java benchmarking development and analysis*, Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications.

Blackburn, S, Cheng, P & McKinley, K 2004, *Myths and realities: The performance impact of garbage collection*, ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 25-36.

Blelloch, G & Cheng, P 1999, *On bounding time and space for multiprocessor garbage collection*, PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation, 104-117.

Barabash, K, Ossia, Y & Petrank, E 2003, *Mostly concurrent garbage collection revisited*. ACM SIGPLAN Notices, 38(11):1–14.

Brooks, R 1984, *Trading data space for reduced time and codespace in real-time garbage collection on stock hardware*, Proceedings of the 1984 ACM Symposium for Lisp and functional programming, 256-262.

Carpen-Amarie, M, Marlier, P, Felber, P & Thomas, G 2015, *A performance study of Java garbage collectors on multicore architectures*, Proceedings of 6th International Workshop on Programming Models and Applications for Multicores and Manycores, 20-29.

Cheng, P & Blelloch, G 2001, *A parallel, real-time garbage collector*, PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation, 125-136.

Collins, G 1960, *A method for overlapping and erasure of lists*, Communications of the ACM, 3(12):655–657.

Deutsch, P & Bobrow, D 1976, *An efficient incremental automatic garbage collector*, Communications of the ACM, 19(9):522–526.

DeTreville, J 1990, *Experience with concurrent garbage collectors for Modula-2+*, Technical Report 64, DEC Systems Research Center.

Dijkstra, E, Lamport, L, Martin, A, Scholten, C & Steffens, E 1978, *On-the-fly garbage collection: An exercise in cooperation*, Communications of the ACM, (21(11):966–975.

Flood, C, Kennke, R, Dinn, A, Haley, A & Westrelin, R 2016, *Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK*, Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, 8(13):1–9.

Grgic, H, Mihajlevic, B & Radovan, A 2018, *Comparison of garbage collectors in Java programming language*.

Hertz, M & Berger, E 2005, *Quantifying the performance of garbage collection vs. explicit memory management*, Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 313-326.

Henriksson, R 1998, *Scheduling garbage collection in embedded systems*, Väitöskirja, Lund Institute of Technology.

Hughes, J 1982, *A semi-incremental garbage collection algorithm*, Software Practice and Experience, 12(11):1081–1084.

Jones, R & Lins, R 1996, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley.

Jones, R, Hosking, A & Moss, E 2011, *The Garbage Collection Handbook*, Chapman & Hall.

Lidén, P & Karlsson, S 2018, *The Z Garbage Collector: An Introduction*, FOSDEM 2018: Free and Open Source Developers' European Meeting.

McCarthy, J 1960, *Recursive functions of symbolic expressions and their computation by machine, part I.*

Meyer, B 1997, *Object-oriented Software Construction – Second Edition*, Prentice-Hall.

Minsky, M 1963, *A Lisp garbage collector algorithm using serial secondary storage*, Technical Report Memo 58 (rev.), Project MAC, MIT.

Printezis, T 2006, *On measuring garbage collection responsiveness*, Science of Computer Programming, 62(2):164-183.

Prokopec, A, Rosà, A, Leopoldseder, D, Duboscq, G, Tůma, P, Studener, M, Bulej, L, Zheng, Y, Villazón, A, Simon, D, Wuerthinger, T & Binder, W 2019, *Renaissance: Benchmarking suite for parallel applications on the JVM*, PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.

Rovner, P 1985, *On adding garbage collection and runtime types to strongly-typed, statically-checked, concurrent language*, Technical Report CSL-84-7, Xerox PARC.

Tene, G, Iyengar, B, Wolf, M 2011, *C4: The continuously concurrent compacting collector*, Proceedings of the International Symposium on Memory Management 2011, 79-88.

Ungar, D 1984, *Generational scavenging: A non-disruptive high performance storage reclamation algorithm*, ACM SIGPLAN Notices, 19(5):157–167.

Wilson, P 1992, *Uniprocessor garbage collection techniques*. Proceedings of the International Workshop on Memory Management, Springer-Verlag.

Yuasa, T 1990, *Real-time garbage collection on general-purpose machines*, Journal of Systems and Software, 13(3).

Liite 1. H2-testin mittausajojen tulokset

Testikokoelma: DaCapo 9.12-MR1

Testi: h2 large, 30 iteraatiota

Taulukoissa on seuraavat sarakkeet:

- Kesto: Mittausajon kesto sekunneissa
- YGC: Nuoren sukupolven keruukertojen määrä
- YGCT: Nuoren sukupolven keruutaukojen yhteiskesto sekunneissa
- 90 / 99 / 100 %: Nuoren sukupolven keruutaukojen kestojen 90., 99. ja 100. persentiilit millisekunneissa
- FGC: Vanhan sukupolven keruukertojen määrä
- FGCT: Vanhan sukupolen keruutaukojen yhteiskesto sekunneissa
- CGC: Rinnakkaisten keruukertojen määrä
- CGCT: Rinnakkaisten keruutaukojen yhteiskesto sekunneissa
- 90 / 99 / 100 %: Vanhan sukupolven keruutaukojen kestojen 90., 99. ja 100. persentiilit millisekunneissa

Mittausajot toistettiin kolme kertaa, ja jokaisen taulukon kahdella alimmalla rivillä on tulosten keskiarvo (ka.) ja suhteellinen keskihajonta (kh.).

Parallel-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	576,9	457	4,9	14,5	27,5	43,8	15	2,8			188,2	346,3	386,8
2	589,1	458	4,8	14,6	28,0	56,5	16	2,8			164,6	329,1	333,3
3	582,8	461	5,1	15,9	28,7	45,4	15	2,6			163,5	263,4	439,6
ka.	**583,0**	**459**	**4,9**	**15,0**	**28,1**	**48,6**	**15**	**2,7**	-	-	**172,1**	**312,9**	**386,5**
kh.	**1,1%**	**0,5%**	**3,9%**	**5,3%**	**2,2%**	**14,2%**	**3,8%**	**3,9%**	-	-	**8,1%**	**14,0%**	**13,8%**

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	597,8	102	3,3	39,3	79,4	88,6							
2	617,3	101	3,2	41,0	84,6	86,6							
3	584,1	100	3,3	39,1	80,0	86,9							
ka.	**599,7**	**101**	**3,3**	**39,8**	**81,3**	**87,4**	-	-	-	-	-	-	-
kh.	**2,8%**	**1,0%**	**0,9%**	**2,5%**	**3,5%**	**1,3%**	-	-	-	-	-	-	-

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	611,8	24	1,2	92,4	126,2	127,6							
2	621,8	24	1,2	98,3	121,3	127,6							
3	657,2	24	1,2	103,0	116,8	121,1							
ka.	**630,2**	**24**	**1,2**	**97,9**	**121,4**	**125,4**	-	-	-	-	-	-	-
kh.	**3,8%**	**0,0%**	**0,2%**	**5,4%**	**3,8%**	**3,0%**	-	-	-	-	-	-	-

CMS-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	658,4	406	7,0	22,9	39,6	49,8	5	2,5	39	0,3	15,3	503,1	526,1
2	651,5	404	6,9	22,6	40,2	50,6	6	2,7	32	0,3	432,1	470,1	471,6
3	634,3	406	6,9	22,7	35,0	48,3	5	2,5	45	0,4	21,7	495,3	548,5
ka.	**648,1**	**405**	**7,0**	**22,7**	**38,2**	**49,6**	**5**	**2,5**	**39**	**0,4**	**156,4**	**489,5**	**515,4**
kh.	**1,9%**	**0,3%**	**0,3%**	**0,7%**	**7,5%**	**2,3%**	**10,8%**	**5,2%**	**16,8%**	**18,7%**	**152,7%**	**3,5%**	**7,7%**

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	639,1	¨203	5,7	33,7	39,5	53,7			2	0,0	11,8	11,8	20,0
2	669,3	202	5,8	34,4	42,9	49,9			2	0,0	11,2	11,2	20,5
3	665,4	202	5,7	34,8	40,2	55,8			2	0,0	2,9	2,9	20,7
ka.	**658,0**	**202**	**5,8**	**34,3**	**40,9**	**53,2**	-	-	**2**	**0,0**	**8,6**	**8,6**	**20,4**
kh.	**2,5%**	**0,0%**	**0,6%**	**1,6%**	**4,4%**	**5,6%**	-	-	**0,0%**	**15,7%**	**57,8%**	**57,8%**	**1,9%**

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	682,5	203	9,6	54,2	66,8	113,0							
2	693,6	203	9,7	54,2	59,4	94,4							
3	661,1	202	9,7	54,8	62,2	79,1							
ka.	**679,1**	**203**	**9,7**	**54,4**	**62,8**	**95,5**	-	-	-	-	-	-	-
kh.	**2,4%**	**0,3%**	**0,4%**	**0,7%**	**5,9%**	**17,8%**	-	-	-	-	-	-	-

G1-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	624,4	409	12,1	47,4	64,3	65,7			238	0,2	1,5	3,1	4,7
2	619,1	408	12,0	45,1	59,0	62,3			238	0,2	1,6	3,0	4,7
3	602,0	407	11,0	41,1	49,3	68,9			236	0,2	1,6	2,4	6,2
ka.	**615,2**	**408**	**11,7**	**44,5**	**57,5**	**65,6**	**-**	**-**	**237**	**0,2**	**1,6**	**2,8**	**5,2**
kh.	**1,9%**	**0,2%**	**5,0%**	**7,2%**	**13,3%**	**5,0%**	**-**	**-**	**0,5%**	**2,6%**	**2,3%**	**12,2%**	**16,9%**

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	636,7	73	4,1	76,7	88,1	89,5			4	0,0	1,8	1,8	2,8
2	613,3	71	4,1	83,1	90,2	90,9			4	0,0	2,1	2,1	2,7
3	611,6	73	4,1	81,0	90,6	100,7			4	0,0	1,8	1,8	1,8
ka.	**620,6**	**72**	**4,1**	**80,3**	**89,6**	**93,7**	**-**	**-**	**4**	**0,0**	**1,9**	**1,9**	**2,4**
kh.	**2,3%**	**1,6%**	**0,9%**	**4,1%**	**1,5%**	**6,5%**	**-**	**-**	**0,0%**	**12,4%**	**7,8%**	**7,8%**	**21,3%**

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	644,3	33	2,8	144,6	157,6	158,7							
2	664,9	33	2,7	136,5	145,7	157,9							
3	663,4	33	2,7	141,7	155,2	158,0							
ka.	**657,5**	**33**	**2,7**	**140,9**	**152,8**	**158,2**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
kh.	**1,7%**	**0,0%**	**1,8%**	**2,9%**	**4,1%**	**0,3%**	**-**	**-**	**-**	**-**	**-**	**-**	**-**

Shenandoah-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	734,7						2097	206,6			0,8	0,9	4,0
2	729,2						2088	204,7			0,7	0,9	1,1
3	728,2						2122	210,2			0,8	1,0	1,5
ka.	**730,7**	**-**	**-**	**-**	**-**	**-**	**2102**	**207,2**	**-**	**-**	**0,7**	**0,9**	**2,2**
kh.	**0,5%**	**-**	**-**	**-**	**-**	**-**	**0,8%**	**1,3%**	**-**	**-**	**4,3%**	**3,7%**	**70,9%**

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	682,0						273	31,7			0,7	1,1	1,2
2	679,5						273	31,8			0,8	1,1	4,3
3	704,9						273	32,3			0,8	1,1	3,4
ka.	**688,8**	**-**	**-**	**-**	**-**	**-**	**273**	**32,0**	**-**	**-**	**0,8**	**1,1**	**3,0**
kh.	**2,0%**	**-**	**-**	**-**	**-**	**-**	**0,0%**	**1,1%**	**-**	**-**	**7,8%**	**2,5%**	**53,2%**

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	689,8						77	9,7			0,6	0,9	1,2
2	692,6						77	9,6			0,7	0,7	1,3
3	698,7						77	9,7			0,6	1,0	1,5
ka.	**693,7**	**-**	**-**	**-**	**-**	**-**	**77**	**9,6**	**-**	**-**	**0,6**	**0,8**	**1,3**
kh.	**0,7%**	**-**	**-**	**-**	**-**	**-**	**0,0%**	**0,4%**	**-**	**-**	**7,8%**	**14,2%**	**11,3%**

Liite 2. Xalan-testin mittausajojen tulokset

Testikokoelma: DaCapo 9.12-MR1

Testi: xalan large, 150 iteraatiota

Taulukoissa on seuraavat sarakkeet:

- Kesto: Mittausajon kesto sekunneissa
- YGC: Nuoren sukupolven keruukertojen määrä
- YGCT: Nuoren sukupolven keruutaukojen yhteiskesto sekunneissa
- 90 / 99 / 100 %: Nuoren sukupolven keruutaukojen kestojen 90., 99. ja 100. persentiilit millisekunneissa
- FGC: Vanhan sukupolven keruukertojen määrä
- FGCT: Vanhan sukupolen keruutaukojen yhteiskesto sekunneissa
- CGC: Rinnakkaisten keruukertojen määrä
- CGCT: Rinnakkaisten keruutaukojen yhteiskesto sekunneissa
- 90 / 99 / 100 %: Vanhan sukupolven keruutaukojen kestojen 90., 99. ja 100. persentiilit millisekunneissa

Mittausajot toistettiin kolme kertaa, ja jokaisen taulukon kahdella alimmalla rivillä on tulosten keskiarvo (ka.) ja suhteellinen keskihajonta (kh.).

Parallel-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	646,4	8311	4,7	0,7	1,1	5,1							
2	621,1	8298	4,6	0,7	1,1	6,5							
3	652,1	8316	4,7	0,7	1,1	4,6							
ka.	**639,8**	**8308**	**4,7**	**0,7**	**1,1**	**5,4**	-	-	-	-	-	-	-
kh.	**2,6%**	**0,1%**	**0,7%**	**1,5%**	**1,0%**	**18,0%**	-	-	-	-	-	-	-

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	638,0	1996	1,2	0,9	1,2	4,0							
2	637,7	2000	1,2	0,9	1,2	5,2							
3	638,0	1994	1,2	0,9	1,2	4,6							
ka.	**637,9**	**1997**	**1,2**	**0,9**	**1,2**	**4,6**	-	-	-	-	-	-	-
kh.	**0,0%**	**0,2%**	**0,5%**	**0,8%**	**0,6%**	**12,8%**	-	-	-	-	-	-	-

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	649,7	497	0,4	1,0	3,4	4,7							
2	657,0	497	0,4	1,0	2,8	4,2							
3	643,9	498	0,4	1,0	3,0	12,2							
ka.	**650,2**	**497**	**0,4**	**1,0**	**3,1**	**7,0**	-	-	-	-	-	-	-
kh.	**1,0%**	**0,1%**	**1,6%**	**0,5%**	**8,8%**	**64,0%**	-	-	-	-	-	-	-

CMS-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	670,1	10208	9,9	1,2	1,9	36,6							
2	663,5	10202	9,8	1,2	1,9	15,5							
3	646,9	10234	9,9	1,2	1,8	15,3							
ka.	**660,1**	**10215**	**9,9**	**1,2**	**1,8**	**22,5**	-	-	-	-	-	-	-
kh.	**1,8%**	**0,2%**	**0,8%**	**1,4%**	**3,2%**	**54,5%**	-	-	-	-	-	-	-

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	661,7	5130	17,6	4,1	4,3	17,9							
2	659,8	5128	17,8	4,1	4,4	19,0							
3	654,1	5121	18,6	4,3	4,6	19,4							
ka.	**658,5**	**5126**	**18,0**	**4,2**	**4,5**	**18,8**	-	-	-	-	-	-	-
kh.	**0,6%**	**0,1%**	**3,0%**	**2,6%**	**3,4%**	**4,2%**	-	-	-	-	-	-	-

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	768,7	5109	140,4	30,4	34,1	59,1							
2	786,7	5110	128,5	27,7	31,4	45,5							
3	761,2	5101	129,1	28,2	30,9	52,7							
ka.	**772,2**	**5107**	**132,7**	**28,8**	**32,1**	**52,4**	-	-	-	-	-	-	-
kh.	**1,7%**	**0,1%**	**5,1%**	**5,0%**	**5,3%**	**13,0%**	-	-	-	-	-	-	-

G1-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	665,7	4542	4,8	1,5	1,8	2,7							
2	650,7	4541	5,0	1,6	1,8	3,4							
3	669,5	4539	5,0	1,6	1,8	3,4							
ka.	**662,0**	**4541**	**4,9**	**1,5**	**1,8**	**3,2**	-	-	-	-	-	-	-
kh.	**1,5%**	**0,0%**	**2,7%**	**1,4%**	**1,3%**	**13,1%**	-	-	-	-	-	-	-

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	682,6	1107	2,6	2,7	4,6	7,0							
2	677,2	1103	2,4	2,5	4,2	8,1							
3	664,8	1110	2,6	2,7	3,2	9,1							
ka.	**674,8**	**1107**	**2,5**	**2,6**	**4,0**	**8,1**	-	-	-	-	-	-	-
kh.	**1,3%**	**0,3%**	**4,3%**	**3,9%**	**17,5%**	**13,1%**	-	-	-	-	-	-	-

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	677,3	287	0,8	3,0	5,5	7,2							
2	671,0	287	0,7	2,9	6,0	6,5							
3	657,1	287	0,7	2,9	6,5	7,1							
ka.	**668,4**	**287**	**0,7**	**2,9**	**6,0**	**6,9**	-	-	-	-	-	-	-
kh.	**1,5%**	**0,0%**	**0,8%**	**1,8%**	**8,1%**	**5,2%**	-	-	-	-	-	-	-

Shenandoah-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	692,0						13853	41,6			0,3	0,4	2,9
2	696,1						13905	42,3			0,3	0,4	2,7
3	706,9						13829	41,5			0,3	0,4	2,7
ka.	**698,3**	-	-	-	-	-	**13862**	**41,8**	-	-	**0,3**	**0,4**	**2,8**
kh.	**1,1%**	-	-	-	-	-	**0,3%**	**1,0%**	-	-	**1,3%**	**1,4%**	**4,9%**

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	695,1						3306	10,1			0,3	0,4	3,7
2	703,3						3285	10,1			0,3	0,4	3,4
3	692,7						3273	10,1			0,3	0,4	3,4
ka.	**697,0**	-	-	-	-	-	**3288**	**10,1**	-	-	**0,3**	**0,4**	**3,5**
kh.	**0,8%**	-	-	-	-	-	**0,5%**	**0,1%**	-	-	**1,8%**	**0,9%**	**5,4%**

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	720,1						833	2,8			0,3	0,4	7,0
2	700,4						833	2,8			0,3	0,4	2,9
3	712,3						837	2,9			0,3	0,5	1,7
ka.	**710,9**	-	-	-	-	-	**834**	**2,8**	-	-	**0,3**	**0,4**	**3,9**
kh.	**1,4%**	-	-	-	-	-	**0,3%**	**0,6%**	-	-	**1,5%**	**7,6%**	**72,1%**

Liite 3. Scrabble-testin mittausajojen tulokset

Testikokoelma: Renaissance 0.10.0

Testi: scrabble, 1700 iteraatiota

Taulukoissa on seuraavat sarakkeet:

- Kesto: Mittausajon kesto sekunneissa
- YGC: Nuoren sukupolven keruukertojen määrä
- YGCT: Nuoren sukupolven keruutaukojen yhteiskesto sekunneissa
- 90 / 99 / 100 %: Nuoren sukupolven keruutaukojen kestojen 90., 99. ja 100. persentiilit millisekunneissa
- FGC: Vanhan sukupolven keruukertojen määrä
- FGCT: Vanhan sukupolen keruutaukojen yhteiskesto sekunneissa
- CGC: Rinnakkaisten keruukertojen määrä
- CGCT: Rinnakkaisten keruutaukojen yhteiskesto sekunneissa
- 90 / 99 / 100 %: Vanhan sukupolven keruutaukojen kestojen 90., 99. ja 100. persentiilit millisekunneissa

Mittausajot toistettiin kolme kertaa, ja jokaisen taulukon kahdella alimmalla rivillä on tulosten keskiarvo (ka.) ja suhteellinen keskihajonta (kh.).

Parallel-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	735,7	20715	84,6	4,7	5,5	34,6	249	21,0			99,6	105,4	113,5
2	746,9	21023	84,4	4,6	5,4	35,0	249	20,7			99,3	105,9	112,3
3	760,0	21019	84,0	4,6	5,4	35,3	253	21,8			101,1	106,4	117,2
ka.	**747,5**	**20919**	**84,4**	**4,6**	**5,4**	**35,0**	**250**	**21,2**	-	-	**100,0**	**105,9**	**114,3**
kh.	**1,6%**	**0,8%**	**0,3%**	**0,9%**	**0,9%**	**1,0%**	**0,9%**	**2,6%**	-	-	**0,9%**	**0,5%**	**2,2%**

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	693,8	4905	75,2	17,3	20,0	26,8	56	5,3			100,0	136,0	141,5
2	711,1	4954	73,5	16,8	19,4	27,5	56	5,3			99,6	136,7	137,1
3	715,9	4978	73,8	16,8	19,6	31,0	56	5,3			101,8	134,1	134,3
ka.	**706,9**	**4946**	**74,2**	**17,0**	**19,6**	**28,4**	**56**	**5,3**	-	-	**100,5**	**135,6**	**137,6**
kh.	**1,6%**	**0,8%**	**1,3%**	**1,8%**	**1,5%**	**8,0%**	**0,0%**	**0,2%**	-	-	**1,1%**	**1,0%**	**2,7%**

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	635,7	1200	13,4	19,6	26,4	39,0	1	0,2					168,8
2	648,3	1200	13,2	19,4	22,5	38,8	1	0,2					190,6
3	645,3	1200	13,7	20,1	25,9	38,1	1	0,2					186,4
ka.	**643,1**	**1200**	**13,4**	**19,7**	**25,0**	**38,6**	**1**	**0,2**	-	-	-	-	**181,9**
kh.	**1,0%**	**0,0%**	**1,8%**	**2,0%**	**8,4%**	**1,3%**	**0,0%**	**6,7%**	-	-	-	-	**6,4%**

CMS-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	858,2	23965	159,6	7,7	8,9	17,2			766	4,3	12,7	15,3	19,4
2	853,0	23842	159,4	7,8	9,0	18,4			776	4,2	12,5	14,9	17,9
3	861,7	24208	160,2	7,7	8,9	17,3			760	4,3	13,2	15,3	18,4
ka.	**857,6**	**24005**	**159,7**	**7,7**	**9,0**	**17,7**	-	-	**767**	**4,3**	**12,8**	**15,1**	**18,6**
kh.	**0,5%**	**0,8%**	**0,2%**	**0,5%**	**0,6%**	**3,8%**	-	-	**1,1%**	**1,0%**	**2,9%**	**1,7%**	**4,2%**

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	838,4	12126	169,9	15,9	17,7	25,7			84	0,6	9,4	15,2	16,2
2	818,5	11944	167,6	16,0	17,7	26,4			84	0,5	9,3	16,9	17,0
3	825,0	12127	165,4	15,5	17,3	26,0			84	0,5	10,2	15,3	26,7
ka.	**827,3**	**12066**	**167,6**	**15,8**	**17,6**	**26,1**	-	-	**84**	**0,5**	**9,6**	**15,8**	**19,9**
kh.	**1,2%**	**0,9%**	**1,4%**	**1,6%**	**1,4%**	**1,3%**	-	-	**0,0%**	**3,2%**	**5,4%**	**5,9%**	**29,3%**

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	914,7	11944	255,2	24,3	35,0	57,7			18	0,2	21,0	25,7	41,3
2	925,1	11944	258,2	24,4	35,5	56,0			18	0,2	21,0	22,2	32,1
3	939,2	12126	258,9	24,1	35,1	56,2			18	0,2	17,0	21,1	31,1
ka.	**926,3**	**12005**	**257,4**	**24,2**	**35,2**	**56,6**	-	-	**18**	**0,2**	**19,7**	**23,0**	**34,8**
kh.	**1,3%**	**0,9%**	**0,8%**	**0,6%**	**0,8%**	**1,6%**	-	-	**0,0%**	**5,9%**	**11,8%**	**10,5%**	**16,1%**

G1-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	960,3	12559	156,3	15,6	19,6	28,2			4206	2,4	1,0	1,8	5,3
2	979,0	12489	156,2	15,7	19,9	26,6			4220	2,3	1,0	1,2	6,5
3	944,1	12503	155,8	15,7	20,0	25,7			4098	2,3	1,0	1,8	6,5
ka.	**961,1**	**12517**	**156,1**	**15,7**	**19,9**	**26,8**	-	-	**4175**	**2,3**	**1,0**	**1,6**	**6,1**
kh.	**1,8%**	**0,3%**	**0,1%**	**0,5%**	**1,1%**	**4,7%**	-	-	**1,6%**	**1,5%**	**0,4%**	**21,6%**	**11,0%**

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	717,2	2666	30,8	18,1	23,4	38,9							
2	701,1	2681	31,8	18,3	23,9	47,6							
3	684,8	2624	31,1	18,3	23,9	44,9							
ka.	**701,0**	**2657**	**31,2**	**18,3**	**23,7**	**43,8**	-	-	-	-	-	-	-
kh.	**2,3%**	**1,1%**	**1,6%**	**0,7%**	**1,4%**	**10,1%**	-	-	-	-	-	-	-

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	689,0	669	7,9	19,7	23,9	29,2							
2	701,6	666	7,9	19,4	24,8	27,9							
3	698,9	666	8,1	20,0	26,4	28,6							
ka.	**696,5**	**667**	**8,0**	**19,7**	**25,0**	**28,6**	-	-	-	-	-	-	-
kh.	**1,0%**	**0,3%**	**0,9%**	**1,4%**	**4,9%**	**2,3%**	-	-	-	-	-	-	-

Shenandoah-keräin

500 megatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	4317,0						153011	7112,7			0,8	2,6	34,1
2	4199,0						152792	6867,6			0,8	3,7	34,5
3	4268,0						152749	6981,5			0,8	4,1	35,1
ka.	**4261,3**	-	-	-	-	-	**152851**	**6987,3**	-	-	**0,8**	**3,5**	**34,6**
kh.	**1,4%**	-	-	-	-	-	**0,1%**	**1,8%**	-	-	**3,5%**	**22,7%**	**1,5%**

2 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	1420,0						24709	1554,9			0,8	2,6	54,0
2	1419,6						24531	1543,8			0,8	2,3	51,5
3	1407,1						24473	1532,5			0,7	2,1	51,5
ka.	**1415,6**	-	-	-	-	-	**24571**	**1543,7**	-	-	**0,8**	**2,3**	**52,3**
kh.	**0,5%**	-	-	-	-	-	**0,5%**	**0,7%**	-	-	**5,1%**	**12,1%**	**2,7%**

8 gigatavun keko

	Kesto	YGC	YGCT	90 %	99 %	100 %	FGC	FGCT	CGC	CGCT	90 %	99 %	100 %
1	881,0						4070	508,6			0,7	12,1	44,2
2	846,9						4017	478,7			0,7	10,9	43,8
3	850,3						4015	476,3			0,7	8,8	37,3
ka.	**859,4**	-	-	-	-	-	**4034**	**487,9**	-	-	**0,7**	**10,6**	**41,8**
kh.	**2,2%**	-	-	-	-	-	**0,8%**	**3,7%**	-	-	**4,6%**	**16,0%**	**9,3%**