Pasi Rautiainen

Java-sovelluksen ajonaikaisen muistin käytön optimointi

Tietotekniikan kandidaatintutkielma

10. kesäkuuta 2024

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Pasi Rautiainen

Yhteystiedot: pasi.j.rautiainen@student.jyu.fi

Ohjaaja: Antti-Jussi Lakanen

Työn nimi: Java-sovelluksen ajonaikaisen muistin käytön optimointi

Title in English: Thesis of optimizing memory usage in Java application

Työ: Kandidaatintutkielma

Sivumäärä: 28+0

Tiivistelmä: Selvitetään kirjallisuudesta muistinkulutuksen vähentämisen keinoja Java-sovelluksessa. Miten ajonaikaista muistinkulutusta voidaan opitimoida. Tutkielmassa selvitetään myös mitä syitä on muistinkulutukseen. Tässä tutkielmassa tutkitaan aihetta olio-ohjelmoinnin kannalta.

Avainsanat: gradu3, kandidaatintutkielmat, Java, muistinkulutus

Abstract: Topic: Reducing memory consumption in the runtime Java application. Thesis from the literature about memory consumption in Java application. This thesis give answer for quetions: What are reasons for the memory consumption? How can reduce memory consumption in the runtime application? And why this topic is good to research?

Keywords: gradu3, Bachelor’s Theses, Java, Memory consumption

Jyväskylässä 10. kesäkuuta 2024

Termiluettelo

JVM	(engl. Java Virtual Machine) Virtuaalikone
RAM	(engl. Random Access Memory) Keskusmuisti
IDA*	(engl. Iterative-Deepening Algorithm*) Rajoitettun syvyyshaun algoritmi
OOP	(engl. Object Oriented Programming) Olio-ohjelmointi
Kekomuisti	(engl. Heap) Sovelluksen käytössä oleva muistialue.
Pinomuisti	(engl. Stack) Suoritettava metodi ladataan pinomuistiin.
IoT	(Internet of Things) Sulautetut laitteet, joissa verkkoyhteys.
Sivutus	(engl. Paging) Fyysinen muisti jaetaan sivuisi, sivuja voidaan tallentaa myös kiintolevylle.
Heittovaihtotiedosto	(engl. Swap file) Fyysistä muistia jatketaan kovalevylle.
Oliot	(engl. Object) Ovat luokista syntyviä ilmentymiä, sovelluksen ajon aikana.
Roskat	(engl. Garbage) Oliot, joita ei enää käytetä.

Kuviot

Kuvio 1. JVM:n rakenne muistissa, [Oma suomennos ja grafiikka], (Grgic, Mihaljević ja Radovan 2018). . . . 4
Kuvio 2. Kirjastojen muistinkulutus, [Omat selitykset ja grafiikka], (Klohs ja Kastens 2005). . . . 12
Kuvio 3. Kekomuistinkulutus, [Omat selitykset ja grafiikka], (Kim ja Hsu 2000). . . . 18

Sisällys

1 JOHDANTO ... 1

2 MUISTIJÄRJESTELMÄ ... 3
- 2.1 JVM ... 3
- 2.2 Roskien keruu ... 4
- 2.3 Kekomuisti ja pinomuisti ... 6

3 MUISTINKULUTUS ... 8
- 3.1 Metodien ja muuttujien vaikutus ... 8
- 3.2 Roskien keräämisen vaikutus ... 11
- 3.3 Algoritmien ja tietorakenteen vaikutus ... 11
- 3.4 Testauksen vaikutus ja hyöty ... 12

4 MUISTIN SÄÄSTÄMINEN ... 14
- 4.1 Luokat, taulukot ja datan koon optimointi ... 14
- 4.2 Muistin käyttäytyminen ... 17

5 JOHTOPÄÄTÖKSET ... 19

LÄHTEET ... 21

1 Johdanto

Tutkielman tarkoitus on selvittää, että onko suorituskykyisiä keinoja vähentää muistinkulutusta sovelluksissa ja miten sen voisi tehdä ja miksi on mahdollista ”hukata” muistia. Tarkastelun kohteena on erityisesti Java-olio-ohjelmointikieli sivuten C++ OOP-kieltä (engl. Object Oriented Programming), koska Java on melko lähellä sitä. Javan erilaisuus on siihen verrattuna, että Java-koodia ei käännetä binäärikoodiksi vaan tavukoodiksi. Se käännetään dynaamisesti ”virtuaalikoneessa” tai tulkataan ajon aikana. Java on myös enemmän oliosuuntautunut kieli kuin C++, koska Javassa on pakko käyttää luokkia.

Java sovellus ajetaan virtuaalikoneessa omassa muistialueessa. Sovellus ei pääse käsiksi käyttöjärjestelmän palveluihin suoraan vaan Javan ”virtuaalikone” keskustelee isäntäkoneen kanssa, kuten tutkijat (Becerra ym. 2003) toteavat. Nykyisin myös C++ sovellus, joka on siis natiivia koodia eli binaariksi käännetty ohjelma, saa käyttöönsä rajatun muistialueen käyttöjärjestelmältä. Sovellukset eivät pääse toistensa RAM-muistialueeseen. Muistivuotoa voi kuitenkin tapahtua myös Javassa, kuten C++:ssa.

Tutkielman aiheessa mainittu ’optimointi’ tarkoittaa tehokkaita algoritmeja, tietorakenteita, ylimääräisen tiedon ja heittovaihtotiedoston (engl. Swap file) käytön minimoimista. Heittovaihtotiedosto on virtuaalista lisämuistia, jota käytetään, kun fyysinen vapaa muisti loppuu. Sivutusmekanismi tekee muistilohkoja eli sivuja fyysiseen muistiin, jos tilaa ei ole se pyytää käyttöjärjestelmältä virtuaalista muistia. Virtuaalinen muisti on kiintolevyllä oleva fyysinen tila eli heittovaihtotiedosto, josta käyttöjärjestelmä varaa sivujen tarvitseman tilan.

Kirjallisuustutkielman tavoite on löytää muistin käytön vähentämisen keinoja, niin roskien keräämistä ei ole niin paljon. Aihetta on hyödyllistä tutkia, koska muistinkulutuksella on merkitystä sovelluksissa. Optimoidut sovellukset voivat olla asiakkaiden mielestä kiinnostavampia. Asiakkaat voivat olla kiinnostuneita sovelluksen energiankulutuksesta. Muistin käytön optimointi on yksi tapa vähentää energian kulutusta. Vihreä siirtymä tarvitsee sovelluksia, jotka kuluttavat vähemmän muistia ja siten vähemmän energiaa. Muistin käytön optimointi on erityisen tärkeää mobiilisovelluksissa ja sulautetuissa laitteissa niiden rajallisen muistin takia.

Tutkielmassa käsitellään muistin käytön vähentämistä JVM:ssä (engl. Java Virtual Machine). Kirjallisuudesta selvitetään muistin käytön vähentämisen keinoja. Lopuksi kerrotaan johtopäätöksistä. Tässä tutkielmassa muistilla viitataan nimenomaan keskusmuistiin.

Seuraavassa luvussa selitetään yleistä muistijärjestelmästä. roskien keruun toiminta ja JVM-ympäristö. Sen jälkeen on muistinkulutus luku, jossa kerrotaan syitä siihen, mihin muistia kuluu. Siiten luvussa muistin säästäminen kerrotaan, miten säästää muistia sovelluksissa, miten välttää ohjelmointivirheitä ja miten muisti käyttäytyy JVM:ssä. Viimeisessä luvussa kootaan yhteen johtopäätökset muistin säästämiskeinoista.

2 Muistijärjestelmä

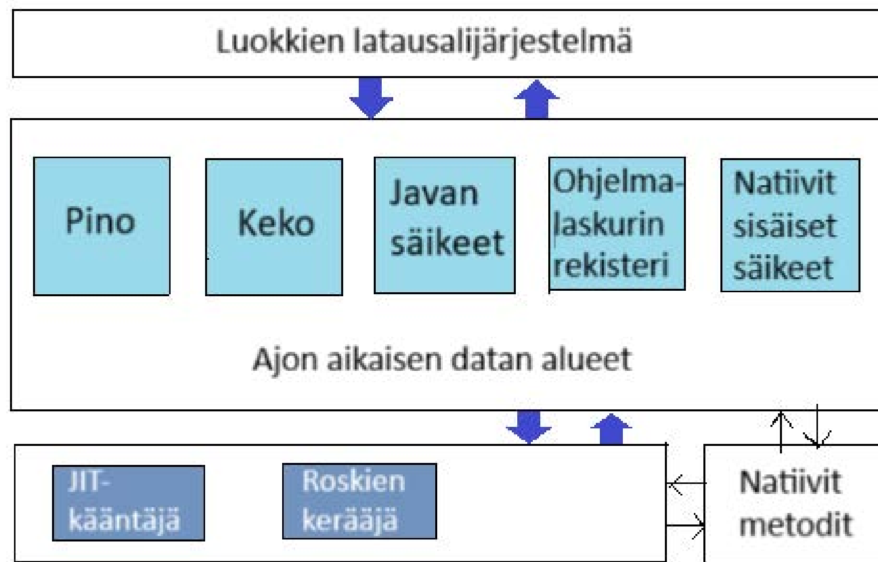
Sovellukset ladataan tietokoneen RAM eli keskusmuistiin. Käsiteltävä data ladataan myös RAM muistiin. Tietokoneissa on aika paljon muistia, mutta siitä huolimatta usein liian vähän. Kotitietokoneissa on yleensä 4-8 Gigatavua keskusmuistia. Mobiililaitteissa on huomattavasti vähemmän muistia käytössä. Sulautetuissa- ja IoT-laitteissa on hyvin vähän muistia verrattuna tietokoneisiin. Tietojen pitäminen muistissa tarvitsee myös virtaa. Mitä enemmän muistia on käytössä, sitä enemmän kuluu virtaa. On yleisesti tiedossa, että mitä isompaa datan määrää käsitellään, sitä enemmän tarvitaan muistia. Usein ajatellaan, että resursseja on rajattomasti. Ohjelmoija tekee helposti vääriä valintoja tai käyttää vääriä tapoja, jotka voivat lisätä muistinkulutusta.

Sivutusmekanismi jakaa fyysisen muistin lohkoihin kuten todettiin johdannossa. Siten sovelluksen muistiavaruuden ei tarvitse olla yhtenäinen. Kun sovellukselle varattu muisti alkaa loppumaan, käyttöjärjestelmä tallentaa muistilohkoja heittovaihtotiedostoon kiintolevylle. Se hidastaa sovelluksen toimintaa merkittävästi. Jos heittovaihtotiedosto ei ole käyttöjärjestelmä käytössä ja sovellus käyttää kaiken vapaan muistin, sovellus kaatuu eli lopettaa toimintansa. Muistin loppuminen voi johtaa koko käyttöjärjestelmän kaatumiseen eli tietokone jää jumiin.

2.1 JVM

JVM eli Javan virtuaalikone on ikään kuin tietokone, se keskustelee isäntäkoneen kanssa Kuten todettiin johdannossa 1. JVM on saatavissa eri käyttöjärjestelmille kuten esimerkiksi Windowsille ja Linuxille. Jos tarvitaan resursseja kuten esimerkiksi tiedostoja tai verkkoyhteyttä JVM pyytää niitä käyttöjärjestelmältä, sovellus ei voi pyytää niitä suoraan. JVM suorittaa Java-kielellä tehdyt sovellukset. Virtuaalikone ottaa vastaan ladattavan luokkatiedoston, kuten voidaan nähdä kaaviosta 1. Sitten se hyväksyy luokan ajoon, jos tavukoodivarmentajan luokantarkastus mene läpi. Näin sen ovat selittäneet tutkijat (Grgic, Mihaljević ja Radovan 2018).

Johdannossa 1 kerrottiin, että Java-kielisen ohjelman dynaamisesti ladattava koodi käänne-



Kuvio 1. JVM:n rakenne muistissa, [Oma suomennos ja grafiikka], (Grgic, Mihaljević ja Radovan 2018).

tään tai tulkataan virtuaalikoneessa ajon aikana. JVM varaa muistia ohjelmalle, kun sitä tarvitaan. Javassa RAM-muistia varataan myös dynaamisesti, kun sitä tarvitaan ja jossain vaiheessa JVM:n roskien keruu kerää ”roskat” muistista automaattisesti. Roskien keruusta ja milloin se tapahtuu, kerrotaan enemmän seuraavassa alaluvussa roskien keruu.

2.2 Roskien keruu

Roskien kerääminen tarkoittaa, että käyttämättömät oliot poistetaan muistista. Javassa se tehdään automaattisesti. Olio muuttuu roskaksi, kun siihen ei enää viitata. Roskien keruujärjestelmä merkitsee roskat ja poistaa ne, kun niitä on tarpeeksi tai muisti uhkaa loppua. Sen jälkeen kekomuisti järjestetään uudelleen eli oliot laitetaan vierekkäin.

Roskienkeruu ei ole ainoastaan Java-kielen ominaisuus. Se kehitettiin aiemminon esimerkiksi Lisp-kieleen (nykyisin Common Lisp), kuten kertoo Cohen (1981). C++:ssa on myös roskien keruu, mutta sekään ei ole automaattista. Se on tullut yleiseen tietoisuuteen Javan myötä, koska siinä se tehdään automaattisesti.

Java sisältää useamman roskien kerääjän. Ne on listattu alempana. Roskien kerääjä voidaan

valita Java-sovelluksen käynnistyksen yhteydessä. Kannattaa valita tarkoitukseen sopiva roskien kerääjä. Ne toimivat eri periaatteilla, kuten tutkijat (Grgic, Mihaljević ja Radovan 2018) ovat selittäneet.

Javassa roskien keräys tehdään siis automaattisesti. Automaattinen roskien keruu on prosessi. Prosessissa tarkastellaan kekomuistia, josta tunnistetaan mitkä oliot ovat käytössä ja mitkä eivät. Sen jälkeen poistetaan käyttämättömät oliot, jolloin muistialue vapautuu. Näin se kerrotaan ("Java Garbage Collection Basics" 2024) oppaassa.

Roskien keruu tapahtuu, kun varattu kekomuisti alkaa loppumaan. Vanhempi Javan versio 8 säilyttää enemmän vanhentuneita objekteja muistissa, kunnes tietyt ikäkriteerit täyttyvät. Jos roskien keräämistä on vähän, roskien keruu on suhteellisen nopea toimenpide. Roskien suurempi määrä johtaa pidempään keskeytykseen sovelluksen suorituksessa. On mahdollista, että roskien keruun pitää käyttää sivutusta, jos ohjelman kekomuisti ei riitä. Siihen tulokseen tulivat tutkijat (Grgic, Mihaljević ja Radovan 2018).

Roskien keruun alkaminen vaikuttaa suorituskykyyn, koska siitä tulee katko ohjelman toimintaan. Roskaksi merkitty olio poistetaan muistista. Kun roskat on poistettu, jäljelle jääneet oliot järjestetään toistensa lähelle. Se tapahtuu riippumatta käytettävästä roskien keruun algoritmista, mutta niissä suoritusaika, roskienkeruun määrä ja miten roskat löydetään ja poistetaan vaihtelevat. Algoritmilla on siten vaikutusta katkon pituuteen ja muistin kulutukseen. Metodin algoritmi on kompromissi suorituskyvyn ja muistin suhteen. Suurempi määrä muistin käyttöä edesauttaa olioiden pirstoutumista, jolloin voi tapahtua muistivuotoa. Toisaalta myös liian pieni keko voi johtaa olioiden pirstaloitumiseen, kuten tutkijat (Grgic, Mihaljević ja Radovan 2018) totesivat.

Java kielen vuoden 2018 ja uudemmissa versioissa JVM sisältää useita roskien kerääjiä, ne toimivat toisistaan eroavilla periaatteilla. Roskien kerääjät:

- Sarjakerääjä (engl. Serial Garbage Collector)
- Rinnakkaiskerääjä (engl. Parallel Garbage Collector)
- Samanaikaisen merkitsemisen ja poiston kerääjä (engl. Concurrent Mark Sweep (CMS) Collector)
- Roskat ensin kerääjä (engl. Garbage First Collector) (Grgic, Mihaljević ja Radovan

2018)

Sarjakerääjä käyttää yhtä säiettä sovelluksen kanssa, joten se toimii sen kanssa vuorotellen. Rinnakkaiskerääjä toimii samanaikaisesti käyttäen useampia säikeitä ja vähentää viiveitä. Se voi kuitenkin lisätä olioiden pirstoutumista. Sen tavoite on minimoida keon kokoa. Rinnakkaiskerääjä on oletuksena Java 8:ssa. "Parallel Garbage Collector" (2024). Samanaikaisen merkitsemisen ja poiston kerääjä pystyy lyhempiin roskienkeräysaikoihin, mutta tarvitsee prosessorin joutoaikaa "Concurrent Mark Sweep Collector" (2024). Roskat ensin kerääjä on suunnattu moniprosessorijärjestelmille. Se poistaa roskat heti. Se vaatii kuitenkin paljon muistia, joten se soveltuu paremmin palvelinsovelluksiin ("Garbage First Collector" 2024). Se oletuksena Java 9:stä eteenpäin.

Samassa tutkimuksessa mitattiin suoritukseen kuluvaa aikaa. Lähes aina hitain oli sarjassa toimiva roskien kerääjä, jolla kesti enimmillään 4 sekuntia. Suorituksen aika oli paras rinnakkaisella roskien kerääjällä etenkin, kun roskia oli enemmän. Se kesti jokaisessa testiajossa alle 2 sekuntia. Tutkijat (Grgic, Mihaljević ja Radovan 2018) siis mittasivat eri roskien kerääjien käyttämää aikaa prosessiin. Rinnakkainen roskien keruu oli nopein. Se on oletuksena käytössä vain Javan versiossa 8. Eli siinä roskien keruu kesti vähemmän kuin 2 sekuntia.

Maksimimääräinen muistinkäyttö on yleensä ohimenevää. Vaikka sovelluksen ja sen käsittelemä data mahtuisi muistiin, roskien kerääminen voi aiheuttaa virtuaalimuistin käyttöä eli sivutusta. Useimmat roskien kerääjät häiritsevät virtuaalisen viitehistorian muistinhallintaa.

2.3 Kekomuisti ja pinomuisti

Kekomuisti on sovelluksen käytössä oleva käyttömuisti, sinne tallennetaan kaikki luokan sisältämät oliot, esimerkiksi listat ja luokka itse eli sen olio. Kekomuistiin tallennetaan myös muut luokat ja riippuvuudet, joihin on viitattu ajettavassa luokassa. Sovelluksen kekomuisti luodaan keskusmuistiin. JVM asettaa oletuksena 1Mt:n keon sovelluksen käyttöön. Keko kasvaa asteittain 32Mt:n, jos tilaa tarvitaan lisää. Kekomuisti luodaan tietokoneen RAM-muistiin, kun sovellus käynnistetään, kuten tutkijat Grgic, Mihaljević ja Radovan (2018) ovat sen selittämeet. Jos käynnistetään toinen Java-sovellus, se saa oman kekomuistin. Keon täyttymisestä järjestelmä antaa virheilmoituksen: "Out of Memory".

Pinomuisti on yleensä prosessorin yhteydessä niin kutsutussa L2 välimuistissa. Sinne ladataan suoritettava metodi ja sen palkalliset muuttujat. Se toimii periaatteella ota päällimmäinen laita päällimmäiseksi (engl. Last in, First out). Roskien kerääjällä ei ole vaikutusta pinomuistiin. Metodit ja niiden muuttujat poistuvat pinosta, kun niiden suoritus loppuu. Pinomuisti voi täyttyä, jolloin sovellus kaatuu eli lopettaa toimintansa. Sitä ennen järjestelmä antaa virheilmoituksen: "Stack Overflow". Tutkijat (Kim ja Hsu 2000) selittivät pinomuistin toimintaa.

Seuraavaksi luvussa muistinkulutus käsitellään sitä mihin muistia kuluu. Siinä käydään läpi joitain ohjelmointivirheitä, jotka kuluttavat muistia. Siinä kerrotaan lisäksi, miten välttää ilmaistuja ohjelmointivirheitä.

3 Muistinkulutus

Tässä luvussa käsitellään muistinkulutuksen syitä ja vaikutuksia suorituskykyyn. Muistinkulutukseen vaikuttaa esim. se kuinka paljon JVM varaa muistia ohjelman käyttöön. Siihen vaikuttaa myös se, että kuinka paljon roskia pidetään muistissa. Muistinkulutukseen vaikuttaa lisäävästi sovelluksen mahdollinen välimuisti. Sovelluksen välimuisti voi olla taulukko, lista tai jokin muu tietorakenne. Tutkijat (Kim ja Hsu 2000) selittivivät JVM:n toimintaa.

Roskien keruu vaikuttaa aina myös suorituskykyyn. Tutkimuksessa (Grgic, Mihaljević ja Radovan 2018) havaittiin, että eri roskien kerääjät tuottivat lähes samat prosessorin käyttöpiikit eli prosessorin hetkellisen maksimi kuormituksen samaan aikaan.

3.1 Metodien ja muuttujien vaikutus

Muistin käyttöä metodissa lisäävät tietorakenteiden alkiot, muuttujat, oliot ja operandit. Niistä kerrotaan oppaassa ("Java Garbage Collection Basics" 2024). Kekomuistin kulutukseen vaikuttavat etenkin ohjelmakoodin määrä ja viittaukset toisiin luokkiin (eli riippuvuudet). Oliot, kuten erilaiset tietorakenteet kuluttavat myös kekomuistia. Käsiteltävä tieto täytyy tallentaa johonkin tietorakenteeseen.

Yleisiä ohjelmointivirheitä, jotka lisäävät muistivuodon mahdollisuutta ja täten muistinkulutusta:

- Resurssi, kuten tiedosto jää sulkematta
- Olioita ei hävitetä, jos tapahtuu poikkeus
- "Kuolleet oliot", eli roskat, joihin kokoelmassa (esim. lista) viittaan
- Kokoelmia ei tyhjennetä

(Ghanavati ym. 2020)

Vaikka ei tapahtuisi muistivuotoa, niin kuitenkin muistia kuluu turhaan. Tarkempi kuvaus ohjelmointivirheistä.

Resurssi tulee sulkea heti, kun se on luettu tai tallennettu muuttujaan. Jos jostain syystä

tapahtuu poikkeus (engl. Exception), resurssi voi jäädä sulkematta. Silloin se jää olemaan muistiin, koska se "käytössä". Sitä ei voi merkitä roskaksi.

Alla on pseudokoodi, jossa resurssi voisi jäädä auki ja toinen pseudokoodi, joka tuhlaa muistia ja synnyttää roskia. Alla on esitetty myös niiden korvaus paremmalla pseudokoodilla (Koodiesimerkit ovat minun).

```
// Huono tapa lukea tiedosto
List<String> list = new List<String>;
FileOpen(filename)
while(nextline) {
            list.add(ReadNextline)
}

// Parempi tapa lukea tiedosto, tässäkin voi jäädä tiedosto auki
// jos vaikka tiedoston riviä ei pysty lukemaan.
try {
FileOpen(filename)
}
catch (FileNotFoundException) {

}
finally {
while(nextline) {
             ReadNextline
}
FileClose
}
```

Jos tapahtuu jokin poikkeus, tiedosto voi jäädä sulkematta. Seuraava olisi parempi koodi.

```
// Tämä olisi paras tapa lukea koko tiedosto
```

```
if(file.exists) {
 Files.readAllBytes
}
```

Metodi avaa, lukee ja sulkee tiedoston. Se huolehtii poikkeuksista. Aina kannattaa tarkistaa onko tiedostoa olemassa.

```
// Huono tapa on tallentaa uusi tieto String-muuttujaan
String newString = "";
for (String s : result) {
 newString += s; // Joka kerta luodaan uusi String-olio
 // ja edellinen jää roskaksi
}

// Parempi tapa
StringBuilder sameString = new StringBuilder();
for (String s : result) {
 sameString.add(s); //Lisätään samaan olioon
}
// Saman voisi tehdä listalla
```

Lista voi sisältää viitauksia olioihin, joita ei enää ole. Siksi listat kannattaa tyhjentää List.Clear-toiminnolla käytön jälkeen. Vääränlaisten muuttujien / olioden käyttö lisää muistinkulutusta. Pitäisi valita tarkoitukseen sopiva muuttujan tyyppi. Long-tyypiä tulisi käyttää vain, jos käsitellään hyvin suuria lukuja. Long-tyyppi vie paljon muistia, kuten tutkijat Myalapalli ja Geloth (2015) ovat tuoneet sen esille.

Muistivuodosta johtuva muistin loppumisvirhe ilmenee, jossain vaiheessa riippumatta siitä, mikä on keon koko. Javassa muistivuotoa tapahtuu, kun prosessi ylläpitää tarpeettomia viittauksia joihinkin käyttämättömiin olioihin, kuten (Ghanavati ym. 2020) tutkimuksessa on todettu.

Paikalliset muuttujat ovat pinomuistissa suoritettavan metodin kanssa. Aiemmin kerrottiin tarkemmin mikä on pinomuisti 2.3. Kun muuttujat ja viitaukset muuttujiin sijaitsevat siellä,

niiden käyttö vie vähemmän virtaa, koska niitä ei tarvitse hakea erikseen kekomuistista kuten tutkijat (Tomar ym. 2001) ovat selittäneet.

3.2 Roskien keräämisen vaikutus

Roskien kerääminen kuluttaa muistia, koska roskien keräyksen algoritmi ja sen toiminta sekä roskat tarvitsevat tilaa kekomuistista. Kekomuisti voi tulla jopa täyteen roskien keräyksen aikana. Kekomuistin täyttyminen on yleensä ohimenevää, mutta sivutusta voi olla jo tapahtunut. Yksi roskien merkitsemiseen kehitetty algoritmi on nimeltään Bookmark Collection (BC). Keon täyttyessä, se merkitsee roskat eli poistettavat oliot. Se vähentää roskien kerääjän tarvitsemaa aikaa, koska roskat on jo merkitty. BC myös hävittää tyhjät sivutusmuistisivut kovalevyltä. BC tasoittaa muistin käyttöpiikit eli kun kekomuisti on hetkellisesti täynnä, eikä se rajoita suorituskykyä. Se on ilmaistu tutkimuksessa (Hertz, Feng ja Berger 2005, sivu 151).

3.3 Algoritmien ja tietorakenteen vaikutus

Roskien keruu algoritmilla on vaikutusta käytettävän muistin määrään. Myös algoritmi, jota aiotaan käyttää tiedon hakuun, vaikuttaa käytettävän muistin määrään. Lähin naapuri algoritmi (engl. The Nearest Neighbor Algorithm) on niin kutsuttu ahne heuristiikka algoritmi. Se on yksinkertainen, se ei kuluta paljon muistia, se on nopea ja usein myös tarkin. Algoritmi käy lähimmässä naapurissa, jossa ei vielä ole käyty ja lopuksi se palaa alkuun (aloitussolmuun). Algoritmin toimintaa ovat tutkineet (Karkory ja Abudalmola 2013, luku 2).

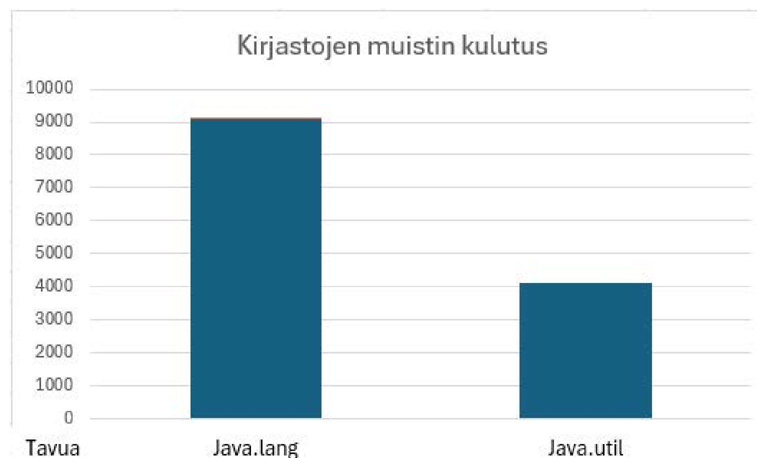
Yksi tietojen etsintää käytettävä algoritmi nimeltään IDA* kuluttaa vähemmän muistia kuin toiset. Se perustuu rajoitettuun syvyyshakuun. Algoritmi on toteutettu yksinkertaisesti eikä sen tarvitse käyttää monimutkaisia tietorakenteita. IDA*-algoritmi pitää muistissa vain yhden polun solmut ja unohtaa umpikujat. Haku lopetetaan, kun maalisolmu löytyy. IDA*:n nopeus perustuu siihen, ettei sen tarvitse ylläpitää isoa tietorakennetta, kuten (Torikka 2015) totesi.

Muistivuotoa tapahtuu, jos lisätään alkiota tietorakenteeseen, eikä niitä poisteta käytön jäl-

keen. Muistivuotoa voi tapahtua myös, jos käytetään julkisia muuttujia (engl. Public fields). Taulukoiden tai listojen iterointi vaikuttaa suorituskykyyn heikentävästi, jos osa alkioista on jo "kuollut"eli muuttunut roskaksi, kuten tutkijat (Nothaas, Beineke ja Schoettner 2019) totesivat.

Sovelluksissa olevat puskurit vievät muistia, etenkin multimediasovelluksien puskurit voivat viedä paljon muistia. Puskureiden koko kannattaa optimoida sopivaksi. Sitä mieltä ovat tutkijat (Holenderski, Bril ja Lukkien 2011).

Kirjastojen vaikutus muistin kulutukseen on melko suuri. Kuten kaaviosta 2 voidaan todeta. Täten sovellukseen ei pitäisi ottaa mukaan ylimääräisiä kirjastoja.



Kuvio 2. Kirjastojen muistinkulutus, [Omat selitykset ja grafiikka], (Klohs ja Kastens 2005).

3.4 Testauksen vaikutus ja hyöty

Testauksessa sovellusta ajetaan useita kertoja. Luokista kutsutaan yleensä joitain metodeita uudestaan. Käytetyt tietorakenteet ovat muuttuneet roskaksi, joten ne täytyy luoda uudelleen. Se lisää muistinkulutusta testin aikana. Kun testausajo kestää lyhyen ajan, roskien kerääjä ei pysty tiivistämään kekomuistia, jolloin roskat voivat pirstaloitua ja jäädä muistiin. Tutkijat (Pizlo ja Vitek 2008) havaitsivat pirstaloitumisen tutkimuksessaan.

Muistin käyttöä pitäisi testata sovelluskehityksen aikana. Testauksessa voidaan havaita muistivuotoja, testi voi epäonnistua ja saadaan jokin virheilmoitus siitä. Muistivuodoista yli 11

prosenttia havaitaan testauksen aikana. Ajonaikaisilla analyysityökaluilla, voidaan havaita muistivuotoon liittyviä vikoja. Analyysissä seurataan muistinkulutusta keossa ja pinossa, otetaan muistivedoksia ja pidetään lokitiedostoja. Seuraamalla mustinkulutusta ja suoritusaikaa voidaan havaita yli 60 % muistivuodoista, kuten tutkijat (Ghanavati ym. 2020) saivat selville.

4 Muistin säästäminen

Yksi keino vähentää muistinkulutusta, nimeen omaan Java -sovelluksessa on käyttää tavukoodi tulkkauksen sijasta dynaamista käännöstä (engl. Java HotSpot) se optimoi vain sen osan koodista, jota suoritetaan usein. Siitä seuraa suuret suorituskyvyn parannukset. Se myös tehostaa roskien keruuta. JIT (Just In Time) - kääntäminenkin on mahdollista, mutta sen haittana on kääntämisen hitaus, joka johtuu optimoinnista. Kuten tutkijat (Eskola, Kulovesi ja Säily 2005) totesivat. Se tarkoittaa koko ohjelman kääntämistä binaariseen muotoon.

FlashByte on tehty kevyeksi ratkaisuksi vähentämään muistinkulutusta keossa, sillä saavutetaan alhainen muistinkulutus. Roskien keräämisen tarve vähenee. FlashByten käytöllä voidaan vähentää muistinkulutusta jopa 25 %. FlashByte toteuttaa natiivin tallennustilan ByteBufferilla välimuistiin, johon datan muistilohkot tallennetaan. Sitä voidaan käyttää mihin tahansa hajautettuun JVM:ssä ajettavaan data-analytiikkajärjestelmään. Se voidaan ottaa käyttöön, ilman sovellusmuutoksia. FlashBytestä kertoivat tutkijat (Zhao ym. 2021).

4.1 Luokat, taulukot ja datan koon optimointi

Java-sovellukset koostuvat luokista ja viitauksista muihin luokkiin. Luokka käärii (engl. wrapped) taulukot ja muut muuttujat sekä toiminnot sisäänsä. Javassa jokaisen sovelluksessa olevat tai kutsuttavan luokan ei tarvitse olla muistissa, jos sovellus ei sitä latausvaiheessa tarvitse.

Tutkimuksessa Hartikainen, Liimatainen ja Mikkonen (2006) mittasivat joidenkin muistinsäästötekniikoiden vaikutuksia. Muistin kulutuksen vähentämiseksi, ohjelmoijan tulisi välttää pieniä luokkia, niitä on parempi yhdistää isommaksi luokaksi. Useiden sisäisten luokkien käyttöä tulisi välttää, on parempi, että yksi luokka toteuttaa useamman toiminnon. Esimerkiksi käyttöliittymän kuuntelijan ja käskyjen toteuttajan on hyvä olla samassa luokassa. Erilaiset poikkeuskäsittelijät ovat luokkia, jotka toteuttavat poikkeuksen. Kannattaa välttää useiden poikkeuskäsittelijöiden käyttöä. Muistin kulutuksen vähentämiseen auttaa myös riippuvuuksien välttäminen. Jos ei ole välttämätöntä viitata toiseen luokkaan niin, sitä ei kannattaisi tehdä.

Vektoreille ja merkkijonoille kannattaisi määrittää koko. Yleensä merkkijonojen muuttamista tulisi välttää, koska merkkijonoa muutettaessa, siitä tehdään uusi merkkijono ja vanha merkkijono muuttuu roskaksi. Jos koko on määritelty niin, että siinä on tilaa merkkijonon muuttamiseen, silloin ei luoda uutta merkkijonoa.

Jos mahdollista, niin vältä periyttämistä, koska silloin muistiin pitää ladata myös perittävä luokka eli yliluokka. Luokkien ei tulisi tarjota menetelmiä, joita sovelluksessa ei tarvita.

Rekursio (eli metodi kutsuu itseään) kasvattaa pinomuistia. Jokainen rekursio kasvattaa pinomuistia 28 tavua. Muistin säästämisen kannalta rekursiota tulisi rajoittaa tai välttää sen käyttöä. Päättymätön rekursio voi johtaa muistivuotoon pinossa.

Resurssit tulisi varata silloin, kun niitä käytetään, ei aikaisemmin. Resurssit tulisi vapauttaa heti, kun niitä ei tarvita. Silloin olioita ei pidetä turhaan muistissa. Olioiden luonti vaatii olion luokan lataamisen, jolloin JVM varaa muistia oliolle. Pitkien taulukoiden staattinen alustus voi kuluttaa paljon muistia, esimerkiksi int-taulukoille Java kääntäjä luo alustuskoodin, jonka koko on 7 tavua taulukkoa kohden. Pitkien taulukoiden sijaan voi tiedot lukea tiedostosta.

Kannattaa välttää turhien kirjastojen käyttöä, koska niissä on metatietoja ja merkkijonoja. Julkisten luokkien ja muuttujien käyttöä kannattaa välttää. Julkiset muuttujat ladataan kekomuistiin. Paikalliset muuttujat ladataan pinomuistiin. Pino on yleensä prosessorin välimuistissa, kuten todettiin luvussa 1 2. Pidempiaikaiseen säilyttämiseen kannattaa käyttää staattisia ja paikallisia muuttujia / tietorakenteita, joita säilytetään pinomuistissa metodin suorituksen ajan.

Tutkijat Myalapalli ja Geloth (2015) etsivät keinoja parantaa Javalla ohjelmointia ja he listasivat yli 50 ehdotusta. Ensimmäinen oli StringBuilderin käyttö merkkijonojen käsittelyssä. StringBuilderin käyttö parantaa tehokkuutta. Se myös vähentää muistinkulutusta, jos merkkijonoon pitää lisätä tekstiä jälkeenpäin. String-olion koko on vakio, jos siihen lisätään merkkejä niin silloin luodaan automaattisesti uusi String-olio ja edellinen jää roskaksi. Siksi StringBuilderin käyttö on suotavaa.

Tutkimuksessaan Ananian ja Rinard (2003) ovat käsitelleet tekniikoita muistin vähentämiseksi olio-ohjelmissa. Esimerkiksi datan koon optimointia muistin säätämiseksi. Kääntäjä

pyöristää muuttujien koon lähimpään tavuun. Tutkijat ehdottivat tapoja, joilla voi vähentää muistinkulutusta.

1. He tekivät kaksi luokkaa, joilla korvasivat String-luokan. String-luokkassa on 3 muuttujaa merkkijonolle. Ne ovat char-taulukko, int offset, int count. Ensimmäiseen luokkaan ei laitettu offset muuttujaa, joka on String-luokassa. Toiseen laitettiin vain offset-muuttuja, joka on osamerkkijonoa varten. Sitten he tekivät algoritmin, joka varaa niistä tarkoitukseen sopivan luokan. String-olio vie 16 tavua muistia, kun he käyttivät omia luokkia, heidän tuloksensa mukaan se säästi muistia 6 tavua per merkkijono.

2. He ehdottivat merkkijonotaulukon pituuden muuttamista parametrilla. Sovelluksen käyttäjä voisi määritellä taulukon pituuden käyttöliittymässä.

3. He totesivat, että staattiset muuttujat vähentävät muistinkulutusta, koska ne eivät muutu ajon aikana.

On selvää, että olioiden luonti lisää muistinkulutusta, niitä ei kannata luoda turhaan. Muuttujat, joita ei muuteta, kannattaa määrittää vakioiksi (engl. const) eli Javassa määritteellä final. Javassa viittausten poistaminen manuaalisesti on mahdollista, asettamalla objektin viittauksen nollaksi. Silloin aikaisempi viittaus objektiin katoaa ja muisti on siltä osin vapautettu.(Grgic, Mihaljević ja Radovan 2018).

Kirjassa Java Programming Applications G. (2020, s.8, s.46, s.118) todettiin että Java OOP-kielellä kirjoitetut koodit vievät hieman enemmän muistia. Jos tekee luokan, jossa on velvoitteita, kuten tietojen hakeminen, niin luokka voi laajentua muistissa. Jos integroi luokkaan turhaan ominaisuuksia, se kuluttaa enemmän resursseja. Datan käsittelyssä ohjelmoijan tulisi olla tietoinen datan dokumentaatiosta, jotta vältetään relevantittoman datan tuonti luokkaan. Täten tietoa, jota ei tarvita, ei tulisi ottaa käsiteltäväksi.

Yksi esimerkki on IOT-laitteiden sensoreista tuleva data, sitä tulee paljon. On järkevää rajoittaa tallennettavan datan määrää, esimerkiksi laskemalla keksiarvo mittaustuloksista. Siten tämä keskiarvo haetaan luokkaan IOT-laitteesta (oma toteamus).

Monia sovelluksia pidetään pitkään käynnissä. Tutkijoiden (Pizlo ja Vitek 2008) mukaan sellaisten sovellusten etuna on, että roskien kerääjä pystyy tiivistämään kekomuistia ja muistin

pirstaloitumista tapahtuu vähemmän. Käytännössä pirstaloituminen johtaa siihen, että roskien kerääjä ei pysty poistamaan kaikkia roskia.

Jos olio on elossa, niin roskien kerääjä ei vapauta olion käyttämää muistia. Ohjelmoijalla on vastuu vapauttaa viittaukset, jotka estävät olion merkitsemisen roskaksi. Manuaalinen koodin tarkastus on hyvä menetelmä resurssivuodon havaitsemiseksi. Resurssien vapauttamiseen voi käyttää "try-with-resources-lohkoa, niin resurssit vapautetaan automaattisesti. Koodin itsearvioinnilla voi havaita virheitä ja profilointityökalujen käytöllä voi havaita muistivuotoja.

Seuraavassa luvussa käsitellään muistin käyttäytymistä JVM:n sisällä. Siinä käsitellään myös kekomuistin koon merkitystä roskien keruuseen.

4.2 Muistin käyttäytyminen

Java-sovellusten muistikäyttäytymistä työkuormissa, ovat tutkineet Shuf ym. (2001) suoritusnopeustestillä (engl. Benchmark). Tutkimuksessa käy ilmi, että keon oletuskoko saattaa parantaa muistin suorituskykyä. Tutkimuksessa käy myös ilmi, että viittaukset objekteihin vaikuttavat muistin kulutukseen. JVM tekee niistä viitetaulukoita muistiin. Tutkimuksesta on hyvä havaita myös, että taulukot, jotka sisältävät olioviitteitä, kuluttavat huomattavasti enemmän muistia kuin primitiivisiä alkioita sisältävät taulukot.

Muistin käyttäytymistä Java-sovelluksessa tutkittiin oliotasolla ja analysoitiin muistin viitejälkiä artikkelissa (Kim ja Hsu 2000). Kuten aiemmin todettiin, JVM varaa sovellukselle kekomuistia oletusarvoisesti 1 Mt:n verran. Tutkimuksessa havaittiin, että roskien keräämisen jälkeen keko voi täyttyä nopeasti uudelleen, jolloin alkaa uusi roskien keräys. Keon oletuskoko on muistin kulutuksen kannalta parempi kuin isompi koko, mutta suorituskyvyn kannalta huonompi. Tutkimuksessa kävi myös ilmi, että roskien kerääminen vähenee, jos keon kokoa kasvatetaan. Kuten nähdään kuvasta 3.

Täten olioiden suuri määrä voi täyttää kekomuistin lähes kokonaan, kuten kaaviossa 3 oikeanpuoleinen pylväs osoittaa. Keon käyttöaste ei nouse lineaarisesti suhteessa olioiden määrään. Roskien keruulta jää myös olioita poistamatta, kun keon käyttöaste kasvaa. Olioita

Kuvio 3. Kekomuistinkulutus, [Omat selitykset ja grafiikka], (Kim ja Hsu 2000).

pirstaloituu ja viittaukset olioihin hajoavat.

5 Johtopäätökset

Tulokset voivat hyödyttää sovelluskehittäjiä säästämään muistia sovelluksissa. Kannattaa oppia valitsemaan käytänteitä ja algoritmeja, jotka vievät vähemmän muistia. Pienemmän muistikapasiteetin laitteissa muistin säästäminen on erityisen tärkeää. Sovelluksessa kannattaa luoda mahdollisimman vähän muuttujia eikä niitä kannata luoda turhaan. Julkisia muuttujia ei kannata käyttää. Samoja metodeita kannattaa käyttää uudestaan.

Katsauksen pohjalta voidaan todeta, että muistinkulutusta on mahdollista vähentää. Osa vähentämiskeinoista on helppo toteuttaa, osa niistä on melko hankalia ja aikaa vieviä ottaa käyttöön. Esimerkiksi String luokan korvaaminen omalla luokalla on melko turha ratkaisu. String olion voi korvata StringBuilderilla, jonka avulla voi säästää paljon muistia. Osa keinoista on sellaisia, että pitäisi ottaa käyttöön ylimääräinen kirjasto, jonka tutkijat ovat esitelleet. Koodin profilointityökaluilla voi löytää muistin kulutuksen syitä. On olemassa myös ajonaikaisia muistin analysaattoreita, jotka voivat havaita muistivuotoja.

Algoritmit vaikuttavat käytettävän muistin määrään lisäävästi tai vähentävästi. Jos algoritmissa on monimutkainen ja iso tietorakenne se kuluttaa enemmän muistia kuin yksinkertaisella tietorakenteilla pärjäävä algoritmi.

Metodien kutsuminen lisää muistinkulutusta pinossa, koska metodit sisältävät muuttujia. Jos metodit sisältävät viittauksia muihin luokkiin, muistinkulutus kasvaa keossa. Olioiden luonti lisää muistinkulutusta, niitä ei kannata luoda turhaan. Kannattaa välttää turhan tiedon tallentamista tietorakenteisiin. Oliot voivat sisältää tietorakenteita, jotka tarvitsevat muistia. Mitä enemmän oliota luodaan, sitä vaikeampi roskien keruun on löytää ja poistaa niitä.

Olio-viitteet kuluttavat muistia myös. Turhia luokkia ei tulisi tehdä tai ottaa sovellukseen. Pieniä luokkia kannattaa yhdistää yhdeksi isommaksi luokaksi. Jos mahdollista, tietorakenteiden koko kannattaa optimoida sopivaksi ja tyhjentää käytön jälkeen. Kaikki resurssit tulisi vapautta heti käytön jälkeen. Poikkeusten käsittely tulisi toteuttaa niin, etteivät resurssit jää vapauttamatta.

Suurempi keon koko kuin oletuskoko, kuluttaa tietysti enemmän muistia. Se voi myös johtaa

kekomuistin täyttymiseen helpommin. pitkään käytössä olevilla sovelluksilla.

Java 9:ssä oletuksena oleva roskat ensin kerääjä (G1), vapauttaa paljon muistia kerralla. Se vaatii kuitenkin paljon kekomuistia. Rinnakkaiskerääjä ei kuluta kovin paljon kekomuistia. Varsinaista tutkimusnäyttöä niiden itsensä muistin kulutuksesta ei tullut vastaan.

Tutkielmassa nostettiin esille useita keinoja muistin käytön vähentämiseksi. Ohjelmoitaessa kannattaa käyttää tutkimusten ilmaisemia tapoja, muistin käytön optimointiin. Hyviä ohjelmointiperiaatteita noudattamalla muistia säästää helposti.

Lähteet

Ananian, C Scott ja Martin Rinard. 2003. "Data size optimizations for Java programs". Teoksessa *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems,* 59–68. https://doi.org/10.1145/780732.780741.

Becerra, Yolanda, Toni Cortes, Jordi Garcia ja Nacho Navarro. 2003. "Evaluating the importance of virtual memory for java". Teoksessa *2003 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS 2003.* 101–110. IEEE. https://doi.org/10.1109/ISPASS.2003.1190237.

Cohen, Jacques. 1981. "Garbage collection of linked data structures". *ACM Computing Surveys (CSUR)* 13 (3): 341–367. https://doi.org/10.1145/356850.356854.

"Concurrent Mark Sweep Collector". 2024. Viitattu 25. huhtikuuta 2024. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.htm.

Eskola, Jukka, Kimmo Kulovesi ja Tatu Säily. 2005. "Dynaaminen kääntäminen ja Java HotSpot", viitattu 15. helmikuuta 2024. https://www.cs.helsinki.fi/u/pohjalai/k05/okk/seminar/Kulovesi-et-al-Dynamic_compilation.pdf.

G., Prudhomme. 2020. *Java Programming Applications. Arcler Press; 2020.* American Medical Assoc. ISBN: 9781774074053.

"Garbage First Collector". 2024. Viitattu 25. huhtikuuta 2024. https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-g1-garbage-collector1.html.

Ghanavati, Mohammadreza, Diego Costa, Janos Seboek, David Lo ja Artur Andrzejak. 2020. "Memory and resource leak defects and their repairs in Java projects". *Empirical Software Engineering* 25:678–718. https://doi.org/10.1007/s10664-019-09731-8.

Grgic, H, Branko Mihaljević ja Aleksander Radovan. 2018. "Comparison of garbage collectors in Java programming language". Teoksessa *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO),* 1539–1544. IEEE. https://doi.org/10.23919/MIPRO.2018.8400277.

Hartikainen, V-M, P. P Liimatainen ja T Mikkonen. 2006. “On mobile Java memory consumption”. https://doi.org/10.1109/PDP.2006.50.

Hertz, Matthew, Yi Feng ja Emery D Berger. 2005. “Garbage collection without paging”. Teoksessa *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation,* 143–153. https://doi.org/10.1145/1065010.1065028.

Holenderski, Mike, Reinder J Bril ja Johan J Lukkien. 2011. “Reducing memory requirements in a multimedia streaming application”. *IEEE transactions on consumer electronics* 57 (1): 145–152. https://doi.org/10.1109/ICCE.2011.5722608.

“Java Garbage Collection Basics”. 2024. Viitattu 15. helmikuuta 2024. https://www.oracle.com/webfolder/technetwork/Tutorials/obe/java/gc01/index.html.

Karkory, Fatma A ja Ali A Abudalmola. 2013. “Implementation of heuristics for solving travelling salesman problem using nearest neighbour and minimum spanning tree algorithms”. *International Journal of Computer and Information Engineering* 7 (10): 1524–1534. https://d1wqtxts1xzle7.cloudfront.net/91408620/17101.pdf.

Kim, Jin-Soo ja Yarsun Hsu. 2000. “Memory system behavior of Java programs: methodology and analysis”. *ACM SIGMETRICS Performance Evaluation Review* 28 (1): 264–274. https://doi.org/10.1145/345063.339422.

Klohs, Karsten ja Uwe Kastens. 2005. “Memory requirements of Java bytecode verification on limited devices”. *Electronic Notes in Theoretical Computer Science* 132 (1): 95–111. https://doi.org/10.1016/j.entcs.2005.01.031.

Myalapalli, Vamsi Krishna ja Sunitha Geloth. 2015. “Minimizing impact on JAVA virtual machine via JAVA code optimization”. Teoksessa *2015 International Conference on Energy Systems and Applications,* 19–24. IEEE. https://doi.org/10.1109/ICESA.2015.7503306.

Nothaas, Stefan, Kevin Beineke ja Michael Schoettner. 2019. “Optimized memory management for a Java-based distributed in-memory system”. Teoksessa *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID),* 668–677. IEEE. https://doi.org/10.1109/CCGRID.2019.00086.

"Parallel Garbage Collector". 2024. Viitattu 25. huhtikuuta 2024. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html.

Pizlo, Filip ja Jan Vitek. 2008. "Memory Management for Real-Time Java: State of the Art". Teoksessa *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC),* 248–254. https://doi.org/10.1109/ISORC.2008.40.

Shuf, Yefim, Mauricio J Serrano, Manish Gupta ja Jaswinder Pal Singh. 2001. "Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations". Teoksessa *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems,* 194–205. https://doi.org/10.1145/378420.378783.

Tomar, S., S. Kim, N. Vijaykrishnan, M. Kandemir ja M. J. Irwin. 2001. "Use of local memory for efficient Java execution". Teoksessa *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001,* 468–473. https://doi.org/10.1109/ICCD.2001.955067.

Torikka, Oskari. 2015. "A*-algoritmi ja siihen pohjautuvat muistirajoitetut heuristiset reitinhakualgoritmit". Tutkielma, Itä-Suomen yliopisto. https://erepo.uef.fi/bitstream/handle/123456789/14693/urn_nbn_fi_uef-20150206.pdf?sequence=1.

Zhao, Junxian, Aidi Pi, Shaoqi Wang ja Xiaobo Zhou. 2021. "Flashbyte: Improving memory efficiency with lightweight native storage". Teoksessa *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid),* 61–70. IEEE. https://doi.org/10.1109/CCGrid51090.2021.00016.