Teppo Naakka

SUORITUSKYKYONGELMIEN MINIMOINTI MONIMUTKAISISSA JAVA-POHJAISISSA SOVELLUKSISSA

Kandidaatintutkielma

JYVÄSKYLÄN YLIOPISTO
TIETOJENKÄSITTELYTIETEIDEN LAITOS
2012

TIIVISTELMÄ

Naakka, Teppo
Suorituskykyongelmien minimointi monimutkaisissa Java-pohjaisissa sovelluksissa
Jyväskylä: Jyväskylän yliopisto, 2012, 39 s.
Tietojärjestelmätiede, kandidaatintutkielma
Ohjaaja(t): Hirvonen, Pertti

Javan nouseminen teollisuuden standardiksi ohjelmistokehityskieleksi on johtanut siihen, että Java-pohjaisilla teknologioilla on tehty markkinoiden monimutkaisimmat sovellukset. Jotta monimutkaisia sovelluksia voidaan toteuttaa tehokkaasti, vaatii se erilaisten kehysjärjestelmien ja kirjastojen käyttöä. Tyypillisesti monimutkaisissa sovelluksissa on useita kirjastoja ja kehysjärjestelmiä, jotka monesti riippuvat toisistaan. Monimutkaiset sovellukset ja useat kirjastot ja kehysjärjestelmät johtavat siihen, että Java-sovellusten raportoiduin vika on suorituskykyongelma. Sovelluksen ja sen suoritusympäristön ollessa monimutkainen myös suorituskykyongelman paikantaminen on haasteellista.

Tutkielmassa käsitellään Java EE-ympäristön rakenne ja siihen kuuluvat komponentit korkealla tasolla. Sen jälkeen syvennytään suorituskykyongelmien paikantamiseen ja mittaamiseen. Lopuksi käsitellään suorituskykyongelmien hallintaa ja minimointia suunnittelun ja toteutuksen näkökulmasta. Tarkoituksena on antaa lukijalle kuva, millaisia suorituskykyongelmia Java EE-järjestelmissä on, ja miten niitä voidaan hallita.

Tutkimustuloksena esitetään, että suorituskykyongelmien paikantaminen ja poistaminen vaatii syvällistä tuntemusta kohdejärjestelmästä ja sovelluksesta sekä suorituskyvyn hallinta ennen toteutusta ja toteutuksen aikana vaatii projektia hallinnoivilta henkilöiltä lisäpanostusta, vaikka sovelluskehityksessä käytettäisiinkin ketteriä menetelmiä. Kaikkia pullonkauloja ei voida minimoida, vaan korjattavat kohteet on valittava tarkasti harkiten.

Asiasanat: ohjelmointi, suorituskyky, Java, Java Enterprise Edition

ABSTRACT

Naakka, Teppo
Minimizing software performance issues in complex J2EE software
Jyväskylä: University of Jyväskylä, 2012, 39 p.
Information Systems, Bachelor's Thesis
Supervisor(s): Hirvonen, Pertti

After Java became the industry standard programming language, the most complex pieces of software has been programmed with it. To maximize the efficiency of programmer, different types of framework systems and libraries have been developed. When complex software meets multiple and sometimes overlapping framework implementations the only outcome is software performance issues. Software performance issues are the most reported problem amongst Java Enterprise software. The underlying runtime environment and complex software combined ensures it is challenging task to pin point the piece of code degrading the overall performance of the software product in question.

This thesis discusses at high level on the features and structure of Java Enterprise runtime environment. Analyzing and finding software performance issues is discussed in more detail. The rest of the thesis includes tools for maintaining and ensuring software performance for people responsible of software project management and the last sections is about code level considerations of software performance.

As result for this study it is suggested that dealing with software performance requires deep knowledge of both, the underlying runtime environment and the software being developed. Ability to ensure software performance before and during the development process requires major extra investments from project staff, even when using agile methodologies. Careful consideration is needed when selecting parts of the software for optimization, the fixes should be always made there where it matters the most.

Keywords: software performance, metrics, Java Enterprise Edition, programming, Java

KUVIOT

KUVIO 1 Java EE -arkkitehtuuri pääpiirteissään. (Jendrock, ym., 2012) 11
KUVIO 2 Java EE-ympäristön kerrostuneisuus (Haines, 2006, 7) 12
KUVIO 3 Java EE-suoritusympäristön kutsun eteneminen ja säie- ja yhteysvarastot (Ferrari, ym., 2004) 13
KUVIO 4 Profilointiarkkitehtuuri (Liang & Viswanathan, 1999) 20
KUVIO 5 Hitaasti etenevä muistivuoto (Haines, 2006, 274) 22
KUVIO 6 Merkkijonon kopioketju (Xu, ym., 2009) 30

ESIMERKIT

ESIMERKKI 1 alitäytetty kokoelma. (Xu & Rountev, 2010) 29
ESIMERKKI 2 Muistivuodon estäminen heikolla viitteellä (Goetz, 2005) 31

SISÄLLYS

TIIVISTELMÄ 2

ABSTRACT 3

KUVIOT 4

ESIMERKIT 4

SISÄLLYS 5

1 JOHDANTO 7

2 SUORITUSYMPÄRISTÖ JA ONGELMAT 10
2.1 Java Enterprise Edition -arkkitehtuuri 10
2.2 Java Enterprise Edition -suoritusympäristö 12
2.3 Tyypillinen sovellus 13
2.4 Suorituskykyongelmat yleisesti 14
2.5 Vaatimukset suorituskyvyn parantamiselle 15

3 ANALYYSITYÖKALUT JA MITTAAMINEN 16
3.1 Ohjelmistokoodianalyysi 16
3.1.1 Ohjelmistokoodianalyysi yleisesti 16
3.1.2 FindBugs-analyysityökalu 17
3.2 Suorituksenaikainen valvonta 18
3.2.1 Instrumentointi 18
3.2.2 Profilointi 19
3.3 Suorituskykypullonkaulojen tunnistaminen 21
3.3.1 Muistivuoto 21
3.3.2 ”Kuuma piste” 22

4 SUUNNITTELUSSA HUOMIOONOTETTAVAT SEIKAT 24
4.1 Kaiken varalta suunnitteleminen 24
4.2 Arkkitehdin rooli suorituskyvyn varmistajana 25
4.3 Riskit ja ongelmat arkkitehtuurisuunnittelussa 25
4.4 Arkkitehdin työkalut 26

5 KOODITASOLLA TAPAHTUVA OPTIMOINTI 28
5.1 Tietorakenteet 28
5.1.1 Kokoelmat 29
5.1.2 Tarpeeton kloonaus 30
5.1.3 Heikko viite 31

5.2 Abstraktiot ja koodin uudelleenkäyttö 32
5.3 Lyhytikäisten olioiden laajamittainen luominen 32
5.4 Kirjastojen ja kehysten vaikutus suorituskykyyn 33
5.5 Optimointi 33
5.6 Tietoinen suorituskyvyn luovuttaminen 34

6 YHTEENVETO 35

LÄHTEET 37

1 JOHDANTO

Mooren lain perusteella mikroprosessoreiden transistoreiden määrä kaksinkertaistuu kahden vuoden välein ja transistoreiden yleisen kehityksen vuoksi niiden prosessointiteho kaksinkertaistuu 18 kuukauden aikana. Tämä kasvu on tarjonnut ohjelmistokehittäjille mahdollisuuden keskittyä ohjelmistojen ominaisuuksiin suorituskyvyn sijaan. Windows-käyttöjärjestelmä on kasvanut vuosien 1992 ja 2008 välillä 150-kertaiseksi koodirivien määrässä mitattuna. Käyttöjärjestelmän koon kasvu ylittää mikroprosessoreiden kehityksen moninkertaisesti. Ohjelmistokehittäjille on tyypillistä, että he käyttävät mieluiten aikansa uusien ominaisuuksien kehittämiseen, jonka seurauksena ohjelmistot ovat alttiita alhaiselle suorituskyvylle ja optimoinnille on kysyntää. (Arnold, Mitchell, Rountev & Sevitsky, 2010)

Viime vuosina suuret ja monimutkaiset ohjelmistot ovat siirtyneet enenevissä määrin virtuaalikoneisiin perustuvissa ympäristöissä ajettavaksi. Virtuaalikoneet tarjoavat paremmat mekanismit muun muassa ohjelmiston siirrettävyydelle ja turvallisuudelle. Virtuaalikone lisää uuden abstraktiotason käyttöjärjestelmän ja ajettavan sovelluksen väliin. Tämä abstraktiotaso helpottaa ohjelmistokehittäjän jokapäiväistä elämää ja lisää turvallisuutta, mutta samalla lisää potentiaalisten suorituskykyongelmien määrää monimutkaistamalla järjestelmän rakennetta. (Gousios & Spinellis, 2008)

André Bondi esittää artikkelissaan, että ohjelmiston suorituskyky on useimmin raportoitu vika Java-pohjaisissa sovelluksissa. Suorituskykyongelmien juuret piilevät yleensä jo ohjelmistokehitysprosessin alkuvaiheilla. Bondi kiinnittää erityistä huomiota sovellusarkkitehdin rooliin suorituskykyongelmien ehkäisijänä jo kehitysprosessin alussa. (Bondi, 2009) Suorituskykyongelmien tiedostaminen vasta projektin loppuvaiheessa aiheuttaa huomattavia ongelmia. Vaikkakin optimointi on syytä jättää vasta aivan viimeiseksi tehtäväksi, on sovelluksen suorituskykyyn vaikuttaviin päätöksiin otettava kantaa jo kehitysvaiheessa. Suorituskykyä on myös mitattava projektin edetessä, jotta ongelmiin voidaan puuttua ajoissa. (Smaalders, 2006)

Ohjelmistokehittäjät usein luulevat, että kehyksien ja kirjastojen luoma parannus tuottavuudessa kumoaa kaikki suorituskykyyn liittyvät vaikutukset. Kehyskirjastot sisältävät työkalut lähes jokaiseen ohjelmoijan kohtaamaan tilanteeseen. Kehyskirjastot itsessään luottavat myös vahvasti kirjastojen toteutuksiin. Väärinymmärrys kirjaston tarjoamassa toiminnossa tai tahaton väärinkäyttö voi aiheuttaa mittavan määrän turhaa prosessointia ja näin vaikuttaa suorituskykyyn negatiivisesti. (Mitchell, Schonberg & Sevitsky, 2010)

Java-kielessä automaattinen muistinhallinta peittää kaikki muistinkäytölliset toiminnot ohjelmistokehittäjältä. Tämä ei kuitenkaan tarkoita sitä, että Javassa ei olisi muistinhallinnallisia ongelmia. Ohjelmistokehittäjän keskittyminen ainoastaan ongelmanratkaisuun voi aiheuttaa esimerkiksi tilanteen, jossa uusia olioita luodaan aivan turhaan. Tämänkaltaiset virheet olisivat vältettävissä riittävällä tuntemuksella kielen toiminnasta. Vanhemmissa alemman tason kielissä samankaltaiset ongelmat huomattiin yleensä aiemmassa vaiheessa, koska virheellinen muistinhallina aiheuttaa välittömästi vikatilanteen ohjelmistossa. (Mitchell, ym. 2010)

Suorituskykyongelmat jaetaan yleensä kahteen ryhmään. Ajonaikaiseen suureen muistinkäyttöön (runtime bloat) ja huonon suorituskyvyn omaaviin algoritmeihin tai ohjelmakoodiin (execution bloat). Nämä ongelmat juontavat juurensa joko ohjelmoijan tai arkkitehdin tekemiin virheisiin. (Arnold, ym. 2010) Suorituskykyongelmiin kannattaa tarttua aikaisessa vaiheessa, koska kuten ohjelmistojen virheisiin, myös suorituskykyongelmiin pätee se, että mitä aikaisemmassa vaiheessa korjaus tehdään, sitä halvempaa se on. Suorituskyvyn osalta optimointi on syytä jättää vasta kehitysprosessin loppuvaiheeseen, pääosin siksi, että optimoitaisiin varmasti kohteita, jotka oikeasti tarvitsevat optimointia. Sen lisäksi, että ohjelmistokoodin optimointi liian aikaisessa vaiheessa hankaloittaa virheiden löytämistä ja heikentää ylläpidettävyyttä, voi kehitysvaiheessa olla vielä epäselvää käyttötapauksien todellinen kuormitus. Sellaisen ohjelmiston osien optimointi, joita käytetään harvoin, ei paranna ohjelmiston todellista suorituskykyä lainkaan. (Haines, 2006, 4-24)

Javan versio 1.6:n suorituskyky on jo niin lähellä C++ -ohjelmointikielen suorituskykyä, ettei enää voida puhua Java-kielen hitaudesta sen ominaispiirteenä. Joissakin operaatioissa Java on jo nopeampi kuin nopeimpana ohjelmointikielenä pidetty C++. (Felde, 2010) Tämän vuoksi Java-ohjelmistoissa esiintyvät suorituskykyongelmat ja niiden alkuperä ovat mielenkiintoisia tutkimuskohteita.

Tämän tutkielman tutkimusongelma on "Mistä monimutkaisen Java-sovellusten suorituskykyongelmat johtuvat ja miten niihin voidaan vaikuttaa?". Aihetta käsitellään reaalimaailman esimerkein ja ongelmiin pyritään antamaan tapa välttää suorituskykyongelma. Tutkimukseen kuuluu myös kuvaus useimmin käytetyistä menetelmistä ja työkaluista suorituskykyongelmien havainnoinnissa. Tutkielma on kirjallisuuskatsaus.

Tutkimusongelman jäsentämisen helpottamiseksi aihetta käsitellään kolmella erillisellä ohjelmistokehitykseen olennaisesti liittyvällä tasolla ja niihin liittyvillä tarkentavilla tutkimuskysymyksillä:

- *Millaisilla päätöksillä ohjelmiston suunnitteluvaiheessa voidaan taata lopputuotteen suorituskyky*
- *Mitä ohjelmistokehittäjän pitää ottaa huomioon toteutusvaiheessa, jottei ohjelmiston suorituskyky vaarannu*
- *Miten suorituskykyongelmat näkyvät suorituksen aikana. Miten suorituskykyongelmia voidaan tunnistaa.*

Tämä tutkielma on jaettu sisällöltään neljään osaan. Aluksi lukija perehdytetään viitekehykseen, jossa suorituskykyongelmia käsitellään. Seuraavaksi käsitellään tutkielman aihetta tarkentavien tutkimuskysymysten valossa. Tämä osio on jaettu kolmeen osaan, joista ensimmäisessä keskitytään suorituskykyongelmien havaitsemiseen tarkoitettuja työkaluja, toisessa suorituskykyongelmien minimointia suunnitteluvaiheessa ja kolmannessa osiossa suorituskykyongelmien minimointia sovelluksen toteutusvaiheessa. Tutkielman lopussa on yhteenveto, jossa tutkielman keskeiset tulokset käsitellään.

2 SUORITUSYMPÄRISTÖ JA ONGELMAT

Luvun tarkoitus on antaa lukijalle kuva, millainen tässä tutkielmassa käytettävä sovellus ja sen suoritusympäristö on. Luvussa käsitellään myös suorituskyky-ongelmia yleisellä tasolla sekä vaatimuksia, joita suorituskyvyn systemaattinen parantaminen vaatii ohjelmiston kehitykseen tai ylläpitoon osallistuvilta henkilöiltä.

2.1 Java Enterprise Edition -arkkitehtuuri

Java Enterprise Edition (Java EE) on alun perin Sun Microsystemsin suunnittelema sovellusalusta-arkkitehtuuri monimutkaisille sovelluksille ja suurille käyttäjämäärille. (Jendrock, Cervera-Navarro, Evans, Gollapudi, Haase, Oliveira & Srivathsa, 2012) Java EE-arkkitehtuuri perustuu komponenttiarkkitehtuuriin, jonka päätarkoitus on erottaa arkkitehtuurillisesti erilliset kokonaisuudet toisistaan. Koska komponentit ovat rajoitettuja ominaisuuksiltaan, tehostaa se ohjelmistokehittäjän toimintaa pilkkomalla ongelmat pienempiin kokonaisuuksiin, jotka ovat helpommin ratkaistavissa. (Richardson, 2006)

Java EE-arkkitehtuuri on jaettu kolmen tasoon: Web-tasoon, liiketoimintalogiikka-tasoon sekä tietokanta-tasoon. Web-tasolla käyttäjälle tarjotaan käyttöliittymät, joilla sovellusta käytetään. Käyttäjän tekemät muutokset ja toiminnot käsitellään liiketoimintalogiikassa, joka tekee toimintoja vastaavat muutokset tietokanta-tasolle pysyvyyskerrosta hyväksikäyttäen. (Jendrock, ym., 2012) Säiliöt sisältävät omat kehyskirjastonsa, mutta Java EE-suoritusympäristöt tarjoavat useita muita kehysjärjestelmiä käytettäväksi sovelluksissa. (Haines, 2006, 4-24)

Java EE-ympäristö eroaa normaalista Java-suoritusympäristöstä pääasiassa komponenttiarkkitehtuurin vuoksi. Suoritusympäristöön on sidottu useita muita palveluja muun muassa konfiguroitavuuden ja siirrettävyyden ylläpitämiseksi. Esimerkiksi käytettävä tietokantayhteys on määritelty joko Java-annotaationa tai XML-muotoisessa konfiguraatiotiedostossa. Yleisimmille in-

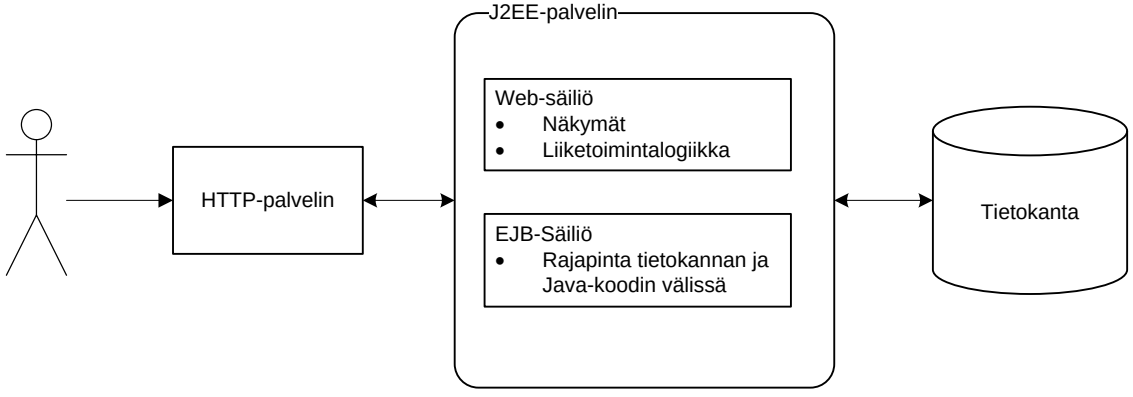
tegraatiorajapinnoille (Web Services, sähköposti ja niin edelleen) on valmis kehystoteutus Java EE-suoritusympäristössä. (Farley & Crawford, 2005, 3-24)

Java EE:n arkkitehtuuria ja järjestelmien välistä kommunikaatiota selventää kuvio 1. HTTP-palvelimella on tärkeä osa kokonaisuudessa, mutta sillä on häviävän pieni osuus suorituskykyongelmia etsittäessä. Tässä tutkielmassa keskitytään etsimään ongelmia Java EE-palvelimesta ja sen arkkitehtuurin aiheuttamista haasteista. HTTP-palvelimen oikealla konfiguraatiolla voidaan estää sovelluspalvelimen kaatuminen liiallisen käyttäjäkuorman johdosta. (Chow, 2003)

Web-säiliön tehtävänä on tarjota käyttäjälle graafinen käyttöliittymä selaimen läpi, jossa järjestelmään voidaan tehdä haluttuja muutoksia. Näkymät tulee ohjelmoida siten, että niiden ohjelmistokoodi sisältää ainoastaan käyttöliittymälogiikkaa. Sen tarkoitus on ainoastaan näyttää käyttäjälle tietoja ja työkalut muutoksien tekemiseen. Web-säiliössä voidaan käyttää ohjelmointikieliä, jotka on erityisesti tehty datan näyttämiseen, kuten JSF (JavaServer Faces) ja JSTL (JavaServer Pages Standard Tag Library). (Farley & Crawford, 2005, 3-24) Web-säiliössä voidaan myös käyttää kehyksiä, joilla käyttöliittymäkomponentteja voidaan generoida automaattisesti. Vaadin on esimerkki tällaisesta laajennuksesta. (Vaadin, 2012)

Liiketoimintalogiikka sisältää ohjelmiston varsinaisen toiminnallisuuden, se käyttää hyväkseen näkymältä saatuja syötteitä ja tekee muutoksia EJB-säiliössä määriteltyjen Enterprise JavaBean–ilmentymien kautta. Farley ja Crawford kirjoittavat, että liiketoimintalogiikka toteutetaan EJB-kerrokselle. (Farley & Crawford, 2005, 3-24) Kun liiketoimintalogiikka ja EJB-kerrokseen kuuluvan transaktionhallinnan ohjelmistokoodi yhdistetään, huononee ohjelmakoodin ylläpidettävyys. Richardson (2006) pitää tärkeänä, että liiketoimintalogiikka kirjoitetaan Java-koodiksi Web-säiliöön ja EJB-kerrokseen ei tehdä mitään ylimääräisiä muutoksia.

Enterprise JavaBean (*EJB*) –kerros huolehtii sovelluksen transaktionhallinnasta, käyttöoikeuksista ja toimii tietokannan ja sovelluksen välisenä rajapintana. EJB-säiliön pääasiallinen tarkoitus on erottaa turvallisuuteen, transaktioihin ja tietokannan toimintoihin liittyvä ohjelmistologiikka liiketoimintalogiikasta. Yleensä EJB-säiliö näkyy ohjelmistokehittäjälle ainoastaan rajapintana tietokantaan. (Farley & Crawford, 2005, 3-24)



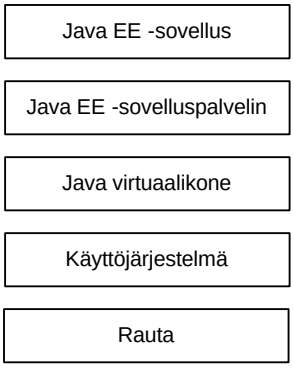
KUVIO 1 Java EE -arkkitehtuuri pääpiirteissään. (Jendrock, ym., 2012)

2.2 Java Enterprise Edition -suoritusympäristö

Tässä tutkielmassa Java EE-suoritusympäristöllä tarkoitetaan J2EE-määritykset täyttävää sovelluspalvelinta. J2EE-määritys sisältää huomattavan määrän toiminnallisia vaatimuksia, jotka on tehty ohjelmistokehittäjää ajatellen. Ohjelmistokehittäjän ei tarvitse keskittyä luomaan perustuksia sovellukselle, vaan sovelluskehityksessä voidaan keskittyä suoraan ratkaisemaan sovellukselle määritetyt ongelmat. (Chow, 2003)

Edellisessä kappaleessa esiteltyyn Java EE-arkkitehtuuriin kuuluu myös lukuisa määrä muita sovelluksen osia, jotka on määritetty kuuluvan osaksi Java EE-suoritusympäristöä. Näitä ovat muun muassa Web Service-integraatiorajapinta, JMX-rajapinta (Java Management Extension) sovelluspalvelimen suorituskyvyn seuraamiseen, JavaMail-rajapinta sähköpostiliikenteelle, JMS-rajapinta (Java Messaging Services) sanomapohjaiselle liikenteelle. Valmiit rajapinnat nostavat ohjelmistokehittäjän tehokkuutta ja nopeuttavat sovelluskehitystä, mutta monimutkaistavat sovelluspalvelinta. (Jendrock, ym., 2012) JMX-rajapintaan ja sen ominaisuuksiin palataan luvussa 3. Pugh ja Spacco (2004) tuovat esille tutkimuksessaan, että kaikilla Java EE-suoritusympäristöön kuuluvilla komponenteilla on vaikutusta sovelluspalvelimen lopulliseen suorituskykyyn.

Java-sovelluksille on tyypillistä useat abstraktiokerrokset. Java EE-sovellusympäristö lisää yhden uuden kerroksen sovelluksen ja käyttöjärjestelmän väliin. Kuvio 2 kuvaa sovellusympäristölle tyypilliset abstraktiokerrokset. Nämä kerrokset tarjoavat työkaluja esimerkiksi järjestelmän turvallisuuden hallintaan sekä laajat sovelluskirjastot, joilla yleisimmin käytössä olevat operaatiot saadaan toteutettua järjestelmään pienellä työmäärällä. (Haines, 2006, 4-24) Tämä on ollut pääsyy Java EE-sovelluspalvelimien suosion kasvuun. (Chow, 2003)



KUVIO 2 Java EE-ympäristön kerrostuneisuus (Haines, 2006, 7)

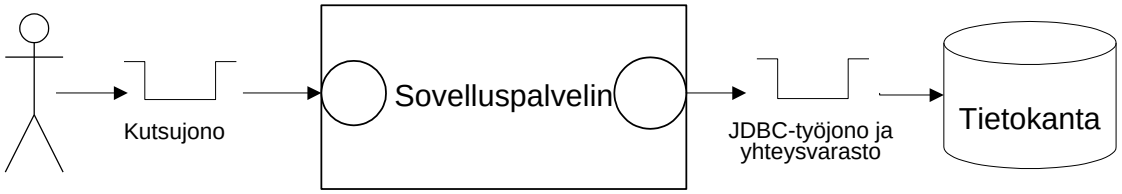
Sovelluspalvelinten suorituskykyoptimoinnissa kannattaa kiinnittää huomiota myös ajettavan sovelluksen piirteisiin. On todettu, että väärin konfiguroitu suoritusympäristö voi näyttää siltä, että itse sovelluksessa olisi huomattava suorituskykyongelma. Java EE-suoritusympäristö tarjoaa mahdollisuuden muuttaa sovelluspalvelimen tapaa käsitellä transaktioita ja tietokantaoperaati-

oita. Näiden ominaisuuksien muuttaminen vastaamaan sovelluksen tarpeita voi joissain tilanteissa poistaa tarpeen ohjelmalliselle optimoinnille. (Pugh & Spacco, 2004)

Java EE-suoritusympäristö perustuu suurelta osin työjonoihin ja erilaisiin säie- ja yhteysvarastoihin. Kuvio 3 kuvaa kutsun etenemistä ja suoritusympäristöissä yleisimmin käytettyjä varastoja. Väärin konfiguroidut varastot aiheuttavat joko tarpeetonta odotusaikaa verkkopalvelun käyttäjälle tai liiallisen kuorman sovellusympäristölle. Sovellusympäristön ylikuormituksen vaikutukset näkyvät kaikille verkkopalvelun käyttäjille, mutta liian pienien varastojen käyttö näkyy vain osalle verkkopalvelun käyttäjistä. Kummassakin tapauksessa suoritusympäristön täysi potentiaali jää hyödyntämättä. (Ferrari, Shrivastava & Ezhilchelvan, 2004; Farley & Crawford, 2005, 4-24)

Java EE-suoritusympäristön lopulliseen suorituskykyyn vaikuttaa luonnollisesti myös ajoympäristön käyttöjärjestelmä. Esimerkiksi Linux/Unix-pohjaisissa käyttöjärjestelmässä voidaan käyttöjärjestelmä optimoida sovelluksen erityispiirteiden mukaan. Käyttöjärjestelmän asetuksilla voidaan tehdä myös dramaattisia pullonkauloja järjestelmään. Esimerkiksi väärät verkkoasetukset sekä levyjärjestelmän asetukset voivat vaikuttaa ajoympäristön suorituskykyyn jopa niin paljon, että ympäristö on käyttökelvoton. (Chow, 2003)

Java EE-sovelluspalvelintoimittajia on tällä hetkellä markkinoilla lukuisia. Suurin osa tarjoaa sovelluspalvelinta kaupallisesti, mutta myös ilmaisia avoimeen lähdekoodiin perustuvia ratkaisuja löytyy. Laajan valikoima mahdollistaa sovelluspalvelintoimittajan kilpailuttamisen ja vertailun. Java EE-määritys mahdollistaa myös sovelluspalvelimen vaihtamisen, koska eri valmistajien ratkaisujen ei pitäisi vaikuttaa standardin Java EE-sovelluksen toimintaan miten-kään. (Chow, 2003)



KUVIO 3 Java EE-suoritusympäristön kutsun eteneminen ja säie- ja yhteysvarastot (Ferrari, ym., 2004)

2.3 Tyypillinen sovellus

Suurimmassa osassa tässä tutkielmassa käytetyistä lähteistä tutkimus ja sen lopputulokset perustuvat johonkin yleisesti saatavilla olevaan suorituskykytestiin. Vain pieni osa lähteistä käyttää päätelmien pohjana oikeita tuotantokäytössä olevia ympäristöjä. Niissä tutkimuksissa, joissa on käytetty niin kutsuttua oikeaa dataa, ilmenee lähes poikkeuksetta ongelmana se, ettei tutkimuksessa käytetystä ympäristöstä ja siinä ajettavasta sovelluksesta yleensä tarjota mitään yksityiskohtia.

Jotta lähteinä käytettyjen ja tämän tutkimuksen johtopäätöksiä voisi soveltaa, on kohdeympäristön toiminnasta tunnistettava tapauskohtaisesti mahdolliset suorituskykyongelmat. Sovellukset eroavat toisistaan hyvin monin tavoin ja täten jokaiselle sovellukselle tyypilliset ongelmat ovat myös sovellukselle uniikkeja.

Haines (2006) määrittelee kirjassaan tyypilliseksi Java EE-sovellukseksi sellaisen, jossa on käyttöliittymäkomponentti sekä palvelun normaaleille käyttäjille että ylläpitäjille. Esimerkkitilanteissa kuormitus on yleensä joitakin satoja yhtäaikaisia käyttäjiä ja sovellus tekee paljon muutoksia tietokantaan ja/tai kommunikoi jonkin integraatiorajapinnan kautta muihin järjestelmiin.

SPECjAppServer-suorituskykytesti on tunnetuin markkinoilla oleva tuote, jolla Java EE-suorituskykyä voidaan mitata riippumatta suoritusympäristön toimittajasta ja muista piirteistä. Suorituskykytestiä on ollut suunnittelemassa suuri osa kaupallisista Java EE-ympäristötöimittajista sekä Darmstadtin teknillinen yliopisto. Suorituskykytesti jäljittelee laajaa toiminnanohjausjärjestelmää. SPECjAppServer ja sen seuraaja SPECjEnterprise ovat alan arvostetuimmat suorituskykytestit ja niiden tuloksia voidaan pitää luotettavina. (Kounev, Weis & Buchmann, 2004) Tässä luvussa listattujen suorituskykytestien ongelmaksi kuitenkin muodostuu se, etteivät ne kykene ottamaan huomioon yksittäisen ohjelmoijan tekemiä ohjelmointiteknisiä ratkaisuja. Tämänkaltaiset suorituskykytestit auttavat löytämään konfiguraatio-ongelmat sovelluspalvelimesta, käyttöjärjestelmästä sekä mahdollisesta lähiverkosta. (Chow, 2003)

2.4 Suorituskykyongelmat yleisesti

Suorituskykyongelma on tilanne, jossa ohjelmaa käytettäessä ei ohjelmistolle asetetut suorituskykyvaatimukset täyty. Tämä vaatimus voi olla tietyn operaation läpivieminen tietyssä aikamäärässä tai sitä nopeammin tai kyky käsitellä tietty määrä yhtäaikaisia käyttäjiä. Suorituskykyongelma on ainakin joissain määrin ratkaistavissa kasvattamalla suoritusympäristön resursseja, mutta se ei välttämättä ole kustannustehokas vaihtoehto pitkässä juoksussa. (Haines, 2006, 4-24)

Suorituskykyongelmille on ominaista, etteivät ne välttämättä näy lainkaan pienillä käyttäjämäärillä. Ohjelmiston kehitysvaiheessa tehdyt testaukset eivät välttämättä tuo esille tehtyjen ratkaisujen mahdollista negatiivista vaikutusta suorituskykyyn. (Haines, 2006, 4-24) Suorituskykyongelman alkuperä on yleensä ohjelmistokoodissa, jossa jokin toimenpide on toteutettu epätehokkaasti. Nämä ongelmat ovat yleensä myös vaikeasti havaittavissa aikaisessa vaiheessa kehitysprosessia, etenkin tilanteessa, jossa ohjelmistokehityksen tukena ei käytetä suorituskykytestausta. (Smith & Williams, 2004, 10–20) Suorituskykyongelmien korjauksien kustannukset nousevat samankaltaisesti kuin normaalin ohjelmistoviankin, mitä myöhemmin ongelma huomataan, sitä kalliimmaksi korjaus tulee. (Haines, 2006, 4-24)

Javassa ja muissa Java-pohjaisissa ohjelmointikielissä yleisin suorituskyvyn alentumista aiheuttava ongelma on muistivuoto tai epätehokas muistinkäyttö. (Xu, Mitchell, Arnold, Rountev, Schonberg & Sevitsky, 2010) Olio-ohjelmointi -paradigma aiheuttaa myös ongelmia. Javaa pidetään erinomaisena kielenä olio-ohjelmointiin, mutta tutkimukset osoittavat, että myös Java kärsii suorituskyvyn heikkenemisestä tilanteissa, joissa ohjelmistokoodi sisältää paljon abstraktioita. (Xu, Arnold, Mitchell, Rountev & Sevitsky, 2009)

2.5 Vaatimukset suorituskyvyn parantamiselle

Suorituskykyongelmien tunnistamiseksi täytyy sovelluksen todellisesta käytöstä olla dataa. Tämän käyttödatan perusteella luodaan helposti toistettava suorituskykytesti, jolla kohdesovelluksen suorituskykyä voidaan tarkastella. (Chow, 2003) Sovelluspalvelimen tarjoamia työkaluja suorituskykyongelmien tunnistamiseen tarkastellaan tarkemmin luvussa 3.

Smaalders (2006) korostaa oikeiden asioiden mittaamisen tärkeyttä. Suorituskykytestiin on valittava sellaiset asiat, jotka ovat tärkeitä sovelluksen lopullisen käytön kannalta. Sovelluksen kehitysversioiden välillä on luonnollisesti eroja myös suorituskyvyssä. Kahden peräkkäisen julkaisun välinen 2 % voi tuntua vähäiseltä, mutta jos suhdanne on ollut sama pitemmän aikaa, on todennäköistä, että sovellus ei täytä enää sille asetettuja vaatimuksia.

Jos testiympäristö ja tuotantoympäristö eroavat suorituskyvyltään toisistaan, voidaan suorituskykytestin kuormitus korjata vastaamaan testiympäristön rajoitetumpia resursseja. Esimerkiksi tilanteessa, jossa tuotantoympäristö muodostuu kahdesta sovelluspalvelimesta, jotka kykenevät käsittelemään 800 yhtäaikaista käyttäjää, voidaan yhdestä sovelluspalvelimesta koostuvan testiympäristön olettaa kykenevän palvelemaan noin 400 yhtäaikaista käyttäjää. Jos käyttödata ei vastaa todellisuutta, päädytään tilanteeseen, jossa keskitytään korjaamaan ongelmia paikoista, jotka eivät todellisessa tuotantokäytössä aiheuta ongelmia. Ohjelmiston kehitysvaiheessa on luonnollista, ettei lopullista käyttödataa ole vielä saatavilla ja suorituskykytestin on tällöin perustuttava niin kutsuttuun parhaaseen arvaukseen. Kaikkein räikeimmät pullonkaulat näkyvät myös tässä vaiheessa, mutta tämänkaltaista testiä käytettäessä pitää ehdottomasti ottaa huomioon sen mahdolliset puutteet. (Haines, 2006, 25–45)

Testiympäristössä ajettavan suorituskykytestin ajaksi voidaan sovelluspalvelimen asetuksia muuttaa siten, että sovelluspalvelin tuottaa suorituskykytestin aikana dataa sovelluksen tilasta ja sen tekemistä toiminnoista. Sovelluspalvelimen tuottamasta datasta voidaan tunnistaa potentiaalisia pullonkauloja, jotka aiheutuvat ohjelmistokoodista tai sovelluspalvelimen konfiguroinnista. Suorituskykytesti tulisi ajaa siten, että ympäristöön ei kohdistu kuormitusta muilta tahoilta (esimerkiksi muut samassa palvelinympäristössä ajettavat sovellukset). Myöhemmässä vaiheessa voidaan tehdä suorituskykytestejä myös siten, että tuotantoympäristön muihin sovelluksiin kohdistuvaa kuormitusta simuloidaan kohdesovelluksen suorituskykytestiä ajettaessa. (Haines, 2006, 25–71)

3 ANALYYSITYÖKALUT JA MITTAAMINEN

Tässä luvussa käsitellään erilaisten analyysityökalujen käyttöä ohjelmistokoodin laadunvarmistuksessa ja virheiden paikallistamisessa. Työkalujen käyttötavat vaihtelevat suuresti. Luku sisältää esimerkit ohjelmistokoodianalyysista ilman suoritusympäristöä (staattinen ohjelmistokoodianalyysi) sekä kahdesta erilaisesta suorituksenaikaisesta monitoroinnista: instrumentoinnista ja profiloinnista.

Luvussa käsiteltävät analysointi- ja mittaamismenetelmien käyttöajankohta projektissa eroaa. Ohjelmistokoodia voidaan analysoida käytännössä jokaisessa ohjelmistoprojektin vaiheessa, jossa lähdekoodia on käytettävissä. Instrumentointi ja profilointi ovat suoritusympäristössä suoritettavia mittaamismenetelmiä ja niitä käytetään yleensä vasta sitten, kun ympäristöön saadaan todellista kuormitusta, joko kuormitustestillä tai palvelun lopullisilla käyttäjillä. Jonkin tasoinen ajonaikainen valvonta kuuluu sovelluksen ylläpitoprosessiin koko tuotantoajalle. (Louridas, 2006; Smith & Williams, 2002, 203–208, 310–316)

3.1 Ohjelmistokoodianalyysi

Tämä aliluku käsittelee ohjelmistokoodin analysointia. Luvussa käsitellään ensin analysointia yleisellä tasolla ja sitten esitellään työkalu, jolla ohjelmistokoodia voidaan pienellä työpanoksella päästä analysoimaan.

3.1.1 Ohjelmistokoodianalyysi yleisesti

Ohjelmistokoodianalyysi on ohjelmistokoodin läpikäymistä ja analysointia ilman, että kyseistä koodia ajetaan suoritusympäristössä. Staattisen ohjelmistokoodianalyysin perusta on C-kieleen kehitetyssä Lint-työkalussa. Työkalu perustuu yleisten inhimillisten virheiden ja huonojen koodiratkaisujen löytämiseen ohjelmistokoodista. Koodin katselmointia on yleisesti pidetty tehokkaana, mutta ihmisten suorittamana sekin sisältää mahdollisuuden virheille. Jotta kat-

selmointi olisi tehokas, on katselmointiin osallistuvilla henkilöillä oltava kokonaiskuva ohjelmistokoodista ja sen erityispiirteistä. Tämän saavuttaminen vaatii henkilöiden kouluttamista ja voi osoittautua kalliiksi toimenpiteeksi. (Louridas, 2006)

Analyysissa jokainen ohjelmistoprojektiin sisältyvä luokka käydään läpi riippumatta siitä, onko kyseinen luokka käytössä ajon aikana. (Louridas, 2006) Ohjelmistokoodianalyysi ei luota koodin suorituksenaikaiseen käyttäytymiseen, vaan analyysi suoritetaan lähdekoodista tai käännetystä tavukoodista. Analyysissa myös testien ulottumattomissa oleva koodi käydään läpi. (Hovermeyer & Pugh, 2004)

Ohjelmistokoodianalyysi perustuu kehitettyihin vikakaavoihin, joihin on kerätty tyypilliset ohjelmistoviat. Suurin osa mallin sisältämistä vioista eivät ole hankalia tai monimutkaisia, vaan yleisesti tunnettuja ongelmia ja erityispiirteitä. (Hovermeyer & Pugh, 2004) Javassa yleinen tämänkaltainen ongelma on olioviitteiden vertaamien toisiinsa niiden arvojen sijaan. Konkreettisten vikojen lisäksi työkaluilla saadaan esille myös tyylilliset ongelmat. Tyyliongelmat hankaloittavat koodin ylläpidettävyyttä ja luettavuutta. (Louridas, 2006)

Tässä luvussa käsiteltyjen yleisten periaatteiden toimintaa käytännössä käsitellään seuraavassa aliluvussa, jossa keskitytään FindBugs-analyysityokalun tarjoamiin ominaisuuksiin.

3.1.2 FindBugs-analyysityökalu

FindBugs-työkalu on Marylandin yliopiston avoimen lähdekoodin työkalu lähdekoodin analysointiin. Se on yksi suosituimmista työkaluista Checkstyle- ja PMD-työkalujen rinnalla. Checkstyle- ja PMD-työkalut käyttävät suoraa lähdekoodin analysointia, kun FindBugs käyttää analysointiin käännettyä tavukoodia. (Lauridas, 2006)

FindBugs-työkalu sisältää erilaisia strategioita, joilla lähdekoodia analysoidaan. Strategian toteuttaa ilmaisin (detector). Jokainen ilmaisin ajetaan erikseen analysoitavalle luokalle. FindBugs-työkalun valmiit strategiat sisältävät neljä eri osa-aluetta:

- Luokkahierarkia ja perintä
- Lineaarinen koodianalyysi
- Kontrollirakenteiden vuo
- Datavuo (Hovermeyer & Pugh, 2004)

Näiden osa-alueiden sisällä pystytään havaitsemaan neljään eri kategoriaan kuuluvia ongelmia:

- Yksittäisessä säikeessä ajettaessa ilmenevät ongelmat
- Säie- ja synkronointiongelmat
- Suorituskykyongelmat

- Tietoturvaongelmat (Hovermeyer & Pugh, 2004)

FindBugs-työkalu antaa virheistä lähdekoodilistauksen sekä mahdollisen lisätietolinkin ongelman korjaukseen. Virhelistauksesta voi suodattaa matalan kriittisyyden omaavat ongelmat pois, jotta kriittiset ongelmat löytyisivät helpommin. (Hovermeyer & Pugh, 2004)

Artikkeli osoittaa, että työkalulla on havaittu todellisia vikoja laajasti kaupallisessa käytössä olevasta ohjelmistosta, kuten JBoss ja Eclipse. FindBugs raportoi huomattavasti vähemmän ongelmia kuin koodityyleihin keskittyvät PMD ja CheckStyle, mutta pienemmästä määrästä on helpompi haravoida todelliset viat. Työkalun on havaittu ansaitsevan paikkansa automaattisten testien rinnalla osana laadunvarmistusprosessia. (Hovermeyer & Pugh, 2004)

3.2 Suorituksen aikainen valvonta

Kaikissa tilanteissa yksikkötestit ja staattinen sovelluksen analysointi eivät riitä sovelluksen virheenpaikantamiseen, vaan sovelluksen käyttäytymistä on seurattava suorituksen aikana. Tähän tarkoitukseen Javan virtuaalikoneeseen on tehty rajapintoja, jotka mahdollistavat hyvinkin tarkalla tasolla tapahtuvan suorituksen seuraamisen ohjelmaa suoritettaessa. (Haines, 2006, 25–45)

Suorituksen aikainen valvonta pitää sisällään instrumentoinnin ja profiloinnin. Java EE-palvelimet keräävät suorituksesta automaattisesti jonkin verran statistiikkaa. Näihin kuuluu muun muassa roskienkeruun ajotiheyden ja keston tilastointi, luokkalataimen operaatiohistoria sekä eniten suoritettujen kutsujen tilastointi. Nämä tiedot ovat käytettävissä sovelluspalvelimen JMX-rajapinnan kautta. (Haines, 2006, 25–45)

Kun tässä aliluvussa käsitellyt valvontatyökalut yhdistetään suorituskykytestaukseen tai tuotantoympäristöön, on huomattava, että ympäristön käyttäytyminen muuttuu. Pelkästään järjestelmän resurssien käytön seuraaminen nostaa järjestelmän kuormitusta ja käytännössä, mitä tarkemmat tulokset halutaan, sitä suurempi osuus järjestelmän resursseista menee suorituskykydatan keräämiseen. (Haines, 2006, 25–45) Suorituskykytesteissä valvonta nostaa kuormitusta ja kuormituksen saturaatiopiste saavutetaan aiemmin kuin kuormitettaessa ilman valvontaa. (Haines, 2006, 125–155)

3.2.1 Instrumentointi

Instrumentointi on osoittautunut äärimmäisen tärkeäksi työkaluksi vianetsinnässä. Yleisesti instrumentoinnilla tarkoitetaan sellaisten tietojen keräämistä sovelluksen ajon aikana, jotka eivät suoraan näy sovelluksen suorituksen lopputuloksesta. Sovelluksen instrumentointia on esimerkiksi suoritusparametrien tallentamista lokitiedostoon tai tietyn ohjelmakoodin suoritusmäärien mittaaminen. (Huang, 1978; Woodside, Granks & Petriu, 2007)

Instrumentoimalla ohjelmakoodiin tai tavukoodiin lisätään joko manuaalisesti tai automaattisesti niin kutsuttuja luotaimia mittaamaan määriteltyjä ohjelman tapahtumia. (Woodside, ym., 2007) Automaattisesti tapahtuva instrumentointi sisältää yleensä ainoastaan metodien alkuun ja loppuun sijoitetut kutsut, jolloin metodien suoritusaika saadaan tilastoitua. (Woodside, ym., 2007) Instrumentoinnilla on huomattava vaikutus (30–1000 %) sovelluksen suorituskykyyn, joten sitä on käytettävä varoen ympäristöissä, joissa sovelluksen suorituskyky on kriittinen. Suurin osa instrumentoinnin lisäkuormituksesta tulee kirjanpitoon ja ajanottoon liittyvistä ylimääräisistä kutsuista. (Arnold & Ryder, 2000)

Java EE-määritykseen kuuluu hallintaliittymä (*Java Management Extension, JMX*). JMX-arkkitehtuuri koostuu agentti- ja instrumentointitason hallituista pavuista (managed bean). Tiedonkeruu tapahtuu siten, että instrumentointitason papu raportoi yksittäisen tapahtuman tiedot agenttitasolle, jossa raa'asta datasta muodostetaan papuun sisäänrakennettujen sääntöjen perusteella informaatiota. Papujen mittaamia tietoja pääsee seuraamaan JMX-protokollaa tukevilla asiakasohjelmilla, www-selaimella tai muulla hallintasovelluksella. (Haines, 2006, 47–72)

JMX-määrityksen sisältämään suppeaan määrään hallittavia komponentteja tehtiin laajennus Java-ohjelmointikielen muutospyynnössä 77 (JSR77). JSR77-määritys luokittelee hallitut pavut kolmeen pääkategoriaan: tapahtumapohjaisiin, tilapohjaisiin ja suorituskykypohjaisiin papuihin. Määritys sisältää myös laajan kokoelman hallittuja papuja, jotka sovelluspalvelimen on toteutettava. Seurattaviin komponentteihin kuuluu muun muassa Java virtuaalikone, tietokantayhteydet, EJB-pavut sekä Servletit. Määritys sisältää myös yleiskäyttöiset liittymät, joita laajentamalla voidaan luoda hallittuja papuja sovelluskohtaiseen käyttöön. (Haines, 2006, 47–72)

Instrumentoinnin aiheuttaman kuormituksen minimoimiseksi on kehitetty erilaisia menetelmiä. Arnold & Ryder (2000) esittävät otantaan perustuvan instrumentoinnin käyttöä tähän tarkoitukseen. Tällöin instrumentoitu ohjelmakoodi suoritetaan ainoastaan silloin, kun instrumentointibitti on asetettu ohjelmistossa. Instrumentointibittiä ohjataan joko käyttöjärjestelmän tai rautatason ajastuksella. Otantaan perustuva instrumentointi piilottaa suurimman osan instrumentoinnin aiheuttamasta sovelluksen hidastumisesta, koska instrumentointeja suoritetaan vain osalle sovelluksesta kerrallaan. (Arnold & Ryder, 2000) Samankaltainen osa-aikainen instrumentointi on käytössä kaupallisten Java-toteutuksien JIT-kääntäjissä. (Lau, Arnold, Hind & Calder, 2006)

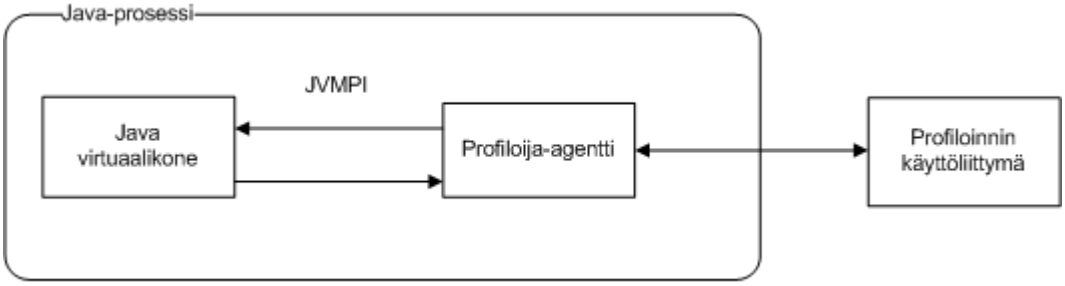
3.2.2 Profilointi

Sovelluksen profiloinnilla tarkoitetaan sen suorituksenaikaisien käskyjen ja metodikutsujen laskemista ja tilastointia. (Woodside, ym., 2007) Seurattavia asioita ovat muun muassa metodikutsut, muistiallokoinnit ja esimerkiksi luokkien lataaminen muistiin. Profiloinnilla voidaan kerätä myös tilastoja suoritusten lukumäärästä ja suoritukseen kulutetusta ajasta. (Liang & Viswanathan, 1999)

Profilointi voidaan toteuttaa kahdella erilaisella tavalla Java-pohjaisissa ympäristöissä. Profilointityökalu voidaan määrittää instrumentoimaan jokainen sovelluksen metodikutsu tavukoodissa. Tällöin profiloinnista saatu informaatio on tarkkaa, mutta haittapuolena on huomattava vaikutus suorituskykyyn. Vaihtoehto jokaisen metodikutsun instrumentoimiselle, on määritetyn intervallin välein tapahtuva kutsupinojen analysointi, eli otantapohjainen profilointi. Käytännössä tämä tarkoittaa sovelluksen täydellistä pysäyttämistä jokaisen intervallin kohdalla ja kutsupinon muodostamista. Tämä menetelmä ei ole yhtä tarkka kuin jokaisen metodikutsun instrumentointi, mutta sen keveys mahdollistaa profiloinnin käytön myös ympäristöissä, joissa suorituskyvyllä on merkitystä. Luonnollisesti profiilin tarkkuus nousee otantaan käytetyn intervallin lyhentämisellä, mutta samalla vaikutus suorituskykyyn kasvaa. (Liang & Viswanathan, 1999)

Profiloimalla pyritään löytämään ohjelmasta ongelmat, jotka näkyvät vasta suorituksen aikana. Osa näistä ongelmista voi vaatia hyvinkin pitkän yhtäjaksoisen suorittaminen ennen kuin ongelma on nähtävissä. Profiloinnin merkitys on korostunut ohjelmointikielissä, joissa on automaattinen muistinhallinta. Erityisesti muistinkäyttöön ja roskienkeruuseen liittyvät viat vaativat ohjelman pitkän yhtäjaksoisen suorittamisen. Tuotantoympäristössä tapahtuvassa profiloinnissa voidaan sovelluksesta löytää niin sanotut kuumat pisteet, sovelluksen osat, jotka joutuvat kovimmalle kuormitukselle. Näiden osien korjaamisella tai optimoinnilla saavutetaan suhteessa suurempi etu koko sovelluksen suorituskykyyn vähemmällä vaivailla kuin koko sovelluksen läpikäymisellä ja optimoinnilla. (Liang & Viswanathan, 1999; Xu, ym., 2009)

Kuvio 4 havainnollistaa Javan profilointiarkkitehtuuria. Javaan profilointi on toteutettu JVMPI-liittymän (Java Virtual Machine Profiler Interface) kautta. Virtuaalikoneeseen sisäänrakennettu profiloija (HPROF agentti) voidaan käynnistää erilaisilla asetuksilla profilointitarpeen mukaisesti. Data kerätään ajon ajalta ja tallennetaan tiedostoon ajon päätyttyä. Javan virtuaalikone lähettää profiloitavat tapahtumat JVMPI-liittymän läpi profiloinnin suorittavalle agentille. Agentin analysoima tieto on tarjolla joko reaaliaikaisen ja graafisen Jconsole-työkalun läpi tai lokitiedostona suorituksen jälkeen. (Liang & Viswanathan, 1999)



KUVIO 4 Profilointiarkkitehtuuri (Liang & Viswanathan, 1999)

Sovelluksen käyttämän muistin profiloinnilla mahdollistetaan muistivuotojen paikallistaminen sovelluksesta. Tämä tapahtuu siten, että verrataan

muistinkäyttöä ennen roskienkeruuta ja välittömästi sen jälkeen. Roskienkeruun välttäneistä olioista pyritään löytämään ne oliot, joita ei sovelluksen suorituksessa enää tarvita, mutta joihin löytyy vielä viittaus. Muistia profiloimalla löytyy sovelluksesta myös sellainen koodi, joka luo paljon lyhytaikaisia olioita. Lyhytaikaisten olioiden suuri määrä pakottaa virtuaalikoneen ajamaan roskienkeruuta liian usein, joka huonontaa sovelluksen yleistä suorituskykyä. (Haines, 2006, 125–154) Javan sisäänrakennettu profiloijalla on mahdollista saada tilastot sovelluksen osista, jotka luovat eniten uusia olioita tai käyttävät eniten muistiresursseja. Myös muistiprofiilin luominen vaatii koko virtuaalikoneen pysäyttämisen muistinalueen analysoinnin ajaksi. (Liang & Viswanathan, 1999)

3.3 Suorituskykypullonkaulojen tunnistaminen

Suorituskykypullonkaulojen tunnistaminen ja korjaaminen on pitkälti käsityötä ja vaatii valistuneen silmän sitä tekemään. Suorituskykypullonkaulojen tunnistamiseen tarvitaan tuntemusta sovelluksen toiminnasta, sovelluspalvelimen toiminnasta ja käytettävissä olevista työkaluista. (Woodside, ym., 2007)
Suorituskykydataan on yhdistettävä järjestelmässä olevat muut sovellukset ja niiden mahdollinen vaikutus järjestelmän toimintaan. Yleisesti suorituskykyongelman löytämiseen tarvitaan kaikkien edellä mainittujen taitojen yhdistäminen. (Haines, 2006, 25–45)

Tilanteessa, jossa kuormituksen vuoksi järjestelmä kaatuu 15 minuutissa, kannattaa määrittää valvontatyökaluihin riittävän tarkka raportointi päälle, vaikkakin järjestelmä kaatuu huomattavasti nopeammin silloin. Tällöin ainakin saadaan tieto, miksi järjestelmä on kaatunut. (Haines, 2006, 255–298) Monesti sovelluksen kaatumisen ajankohta on vaikeasti ennustettavissa tai se tapahtuu ajankohtana, jolloin kukaan ei ole valvomassa sovellusta. Tässä aliluvussa käsitellään myös, miten kaatumisen syytä voidaan tutkia jälkeenpäin, vaikkei reaaliaikaista valvontaa kaatumishetkellä ollutkaan käytössä. (Maxwell, Back & Ramakrishnan, 2010)

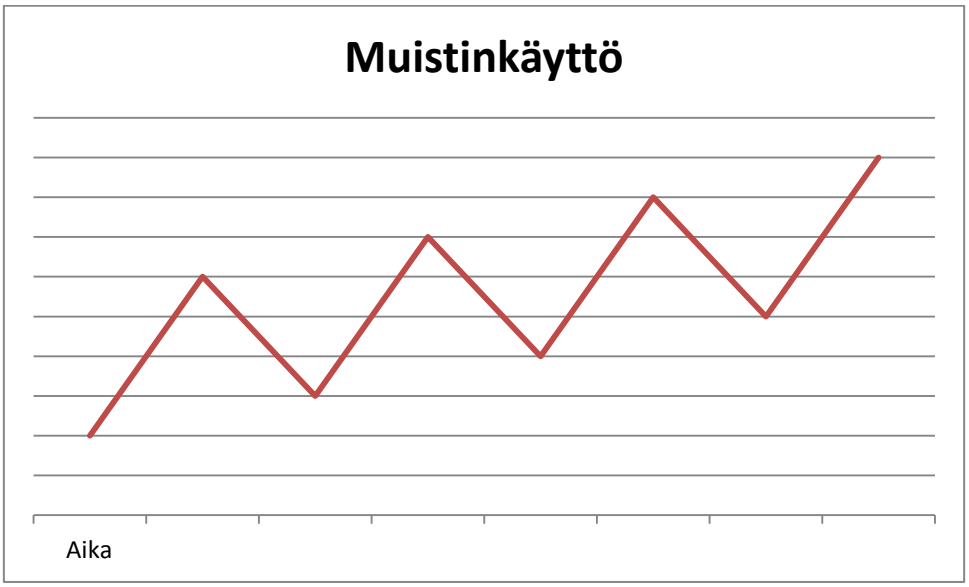
3.3.1 Muistivuoto

Muistivuodosta on tullut yleinen ongelma monimutkaisissa Java-sovelluksissa. Muistivuoto ohjelmointikielissä, joissa on automaattinen muistinkäsittely, tapahtuu siten, että sovellukseen jää aktiivinen muistiviite olioon, jota ei enää tarvita sovelluksen suorittamiseen. (Xu, ym., 2010) Muistivuoto hidastaa sovellusta kuluttamalla muistiresursseja, joka pakottaa roskienkeruun ajettavaksi niin usein, että se hidastaa sovelluksen toimintaa. (Bond & McKinley, 2009) Yleensä muistivuotojen aiheuttajana ovat erilaiset säiliö-tietotyypit, joiden sisältöä tai elinikää ei ole otettu huomioon. (Maxwell, ym., 2010)

Kuvio 5 havainnollistaa hitaan muistivuodon aiheuttamaa muistinkäytön lisäystä. Roskienkeruu ei tässä tilanteessa pääse palauttamaan muistinkäyttöä

takaisin alkutilanteeseen, vaan jokaisella ajolla enemmän muistia jää roskienkeruun ulottumattomiin. (Haines, 2006, 255–298) Kun sovellusta ajetaan pitkään ja muistiresurssit lopulta loppuvat, aiheuttaa tämänkaltainen muistivuoto sovelluksen kaatumisen. Joissain tilanteissa tämä voi vaatia jopa kuukausien yhtäjaksoisen suorittamisen. (Bond & McKinley, 2009)

Muistinkäytön profiloinnilla voidaan seurata sovelluksen tekemiä operaatioita ja tunnistaa ongelmakohtia jo sovelluksen ajon aikana. (Liang & Viswanathan, 1999) Ajonaikaisessa profiloinnissa ohjelmistokoodi, joka luo paljon olioita, voi hämätä etsimään ongelmaa väärästä paikasta. (Dufour, Ryder & Sevitsky, 2007) Muistivuodon aiheuttajaa voidaan etsiä myös analysoimalla Java-virtuaalikoneen muistivedoksia. Normaalisti muistin loppuessa Java-virtuaalikone kirjoittaa muistivedoksen pysyvään tietovarastoon ennen suorittamisen keskeyttämistä. Tästä vedoksesta voidaan erillisellä työkalulla, kuten esimerkiksi Eclipse Memory Analyzer Tool, analysoida mitä kaatumishetkellä oli virtuaalikoneen muistissa. Muistivedosten analysointi helpottaa tuotantoympäristöjen virheenpaikantamista, koska niissä ei välttämättä pystytä käyttämään tarkkaa valvontaa suorituskyvyn ja ympäristön sujuvan toiminnan takaamiseksi. (Maxwell, ym., 2010)



KUVIO 5 Hitaasti etenevä muistivuoto (Haines, 2006, 274)

3.3.2 ”Kuuma piste”

Sovelluksen niin kutsuttu kuuma piste on metodi tai luokka ohjelmistokoodissa, jota suoritetaan sovelluksen suorituksen aikana useammin kuin muita sovelluksen osia. Yhteen sovellukseen voi muodostua useita kuumia pisteitä. Määrä riippuu ainoastaan sovelluksen laajuudesta ja erilaisten suorituspolkujen määrästä. Suurimmat edut sovelluksen yleisen suorituskyvyn kannalta saadaan pyrkimällä optimoimaan keskeisiä ja lukumääräisesti useimmin suoritettavia osia sovelluksesta. (Ammons, Choi, Gupta & Swamy, 2004)

Shuf ja Steiner (2007) tuovat esille ongelman laajoja Java EE-sovelluksia profiloitaessa: järjestelmän kuormitus hajautuu tasaisesti koko koodipohjalle monista erilaisista suorituspoluista johtuen. Kun profiilista puuttuu selkeät "kuumat pisteet", on optimoitavan kohteen tunnistaminen huomattavasti hankalampaa.

Suorituskykyongelmien paikantamiseen tuodaan esille Dufourin, Ryderin ja Sevitskyn (2007) kirjoittamassa artikkelissa uusi näkökulma, jonka perusteella perinteisten profilointityökalujen avulla on huomattavan haastavaa paikallistaa suorituskykyongelmia lähinnä Java EE-sovelluksille tyypillisten päällekkäisten kirjastojen ja kehysten vuoksi.

4 SUUNNITTELUSSA HUOMIOONOTETTAVAT SEIKAT

Luvussa käsitellään suunnitteluvaiheessa tehtävien päätösten vaikutuksesta lopullisen sovelluksen suorituskykyyn ja erilaisista tavoista välttää nämä ongelmat. Vaihtoehdot pyritään käsittelemään sillä tarkkuusasteella, että lukija saa kuvan sekä vahvuuksista että heikkouksista ja pystyy tekemään päätöksen näiden seikkojen perusteella.

Luku koostuu neljästä osasta, ensin käsitellään suunnittelussa tapahtuvia virheitä sekä arkkitehdin roolia monimutkaista sovellusta kehitettäessä. Tämän jälkeen siirrytään suunnittelussa tapahtuviin riskeihin ja ongelmiin, jotka johtavat suorituskykyongelmiin ja lopuksi käsitellään arkkitehdin työkaluja suorituskykyongelmien hallintaan ja sovelluksen suorituskyvyn ennustamiseen.

4.1 Kaiken varalta suunnitteleminen

Ohjelmistoarkkitehtuurin suunnittelussa on vaarana sortua tekemään ratkaisuja, jotka tekevät jo ennestään laajoista sovelluksista vielä laajempia ja monimutkaisempia. Tämänkaltaisia ratkaisuja ovat muun muassa sellaiset, joissa kaikkien rajapintojen toteutukset suunnitellaan riippumattomiksi, vaikkei sellaiseen ole oikeata tarvetta. Arkkitehtuuri voidaan tehdä helposti laajennettavaksi ilman, että jo ensimmäisissä toteutusvaiheissa käytetään suuria työmääriä siihen, että saavutetaan riippumattomuus esimerkiksi tietokantakerroksen toimittajasta tai integraatiorajapinnan teknologiasta. Järkevällä suunnittelulla saadaan pidettyä laajennettavuuden työmäärä rajallisena, mutta ennen kaikkea järjestelmän arkkitehtuuri yksinkertaisempana. (Mitchell, ym. 2010)

4.2 Arkkitehdin rooli suorituskyvyn varmistajana

Andre Bondi (2009) tuo esille tutkimuksessaan arkkitehdin merkityksen ohjelmistokehitysprosessissa sovelluksen lopullisen suorituskyvyn varmistajana. Vaikkakin arkkitehti ei välttämättä kirjoita riviäkään itse ohjelmakoodista, ovat hänen tekemänsä päätökset suuressa roolissa. Arkkitehdin tulisi rohkaista kehittäjiä tekemään suorituskyvyn kannalta oikeita valintoja. Tämä vaatii arkkitehdiltä ymmärrystä ohjelmistojen suorituskykyyn vaikuttavista tekijöistä.

Ohjelmistoarkkitehti on myös rajapinta kehittäjien ja asiakkaan välillä ja on täten asemassa, jossa arkkitehdin on nähtävä ohjelmiston niin kutsuttu suuri kuva. (Bondi, 2009) Mitchell ja kumppanit (2010) tuovat esille tutkimuksessaan huolen suorituskyvyn laskemisesta, kun ohjelmistoa suunnitellaan ilman, että ohjelmiston todellisesta käytöstä ja sen tarpeesta on tarkkaa tietoa. Tämä lisää arkkitehdin tärkeyttä.

Lopullisen ohjelmistotuotteen suorituskykyä voidaan varmistaa liittämällä ohjelmointitehtäville suorituskykytavoite. Tällaisia tavoitteita ovat esimerkiksi sellaiset, että asiakas pystyy kirjautumaan sovellukseen ja näkemään henkilökohtaiset tietonsa viidessä sekunnissa. Näin potentiaalinen suorituskykyongelma huomataan heti ensimmäisen kehitysversion valmistuttua ja parhaassa tapauksessa ongelman korjaamisella aikaisessa vaiheessa säästetään huomattavasti resursseja ja kustannuksia. (Smith & Williams, 2002, 30–35; Woodside, ym., 2007)

4.3 Riskit ja ongelmat arkkitehtuurisuunnittelussa

Bass, Nord, Wood ja Zubrow (2007) tuovat tutkimuksessaan esille yleisimpiä riskejä, jotka liittyvät ohjelmistoarkkitehtuureihin ja ohjelmistojen toteutusprosessiin. Tutkimuksessa suurimmaksi riskiksi projektin epäonnistumiselle muodostuvat ne tehtävät, jotka jätetään arkkitehtuuriin liittyvissä tehtävissä ja prosesseissa tekemättä. Tehtävät ja toimet, jotka eivät varsinaisesti ole osa prosessia, muodostivat huomattavasti pienemmän riskin, joten huolellinen suunnittelu ja valmistelu varmistavat paremman lopputuloksen kuin arkkitehtuurisuunnittelun tehtävien karsiminen ja liiallinen keventäminen.

Arkkitehtuurisuunnittelun riskit lisääntyvät, jos toteutettavan järjestelmän vaatimukset ovat puutteelliset. Tämä puute heijastuu myös suorituskykyyn, on äärimmäisen hankala määrittää järjestelmän lopullinen käyttöprofiili, jos vaatimukset eivät ole riittävän tarkat. (Woodside, ym., 2007)

Bondi (2009) listaa tutkimuksessaan yleisimmät suunnitteluvirheet sovelluksille, jotka heikentävät ohjelmiston suorituskykyä:

- ”Jumalaluokka”, eli ohjelmiston toiminnot luottavat suurelta osin toimintoihin, jotka on ohjelmoitu yhden luokan sisälle. Tämänkal-

tainen suunnittelu aiheuttaa profiloinnissa luvussa 3.3.2 esitellyn suorituksen kuuman pisteen.
- ”Aarteen etsintä” on tilanne, jossa ohjelmistossa tehdään hakuja tauluihin vain, jotta saataisiin informaatiota informaation hakemiseksi jostain toisesta taulusta.
- ”Yksikaistainen silta” kuvaa ohjelmistoa, joka toimii mainiosti pienillä käyttäjäkuormilla, mutta siihen muodostuu pullonkauloja heti käyttäjäkuorman kasvaessa. Sovellus ei skaalaudu käyttötarpeen mukaisesti.

Ohjelmistoarkkitehdin on pyrittävä ratkaisemaan ongelmat siten, ettei lueteltuja ongelmia pääse ohjelmistoon muodostumaan. Tässä listatut suunnitteluvirheet ovat hyvin tiukasti liitoksissa koodin kirjoittamiseen, joten jotkin tämänkaltaiset virheet voivat korjaantua kokeneen ohjelmistokehittäjän toimesta ilman, että arkkitehti saa siitä tiedon. Pahimmassa tapauksessa virheet arkkitehtuurisuunnittelussa laajentuvat koko sovellukseen ja ainoastaan uudelleensuunnitteleminen ja –toteutus voivat pelastaa järjestelmän suorituskyvyn. (Hyde, 2009)

4.4 Arkkitehdin työkalut

Arkkitehdillä on mahdollista käyttää apunaan työkaluja, joilla sovelluksen lopullinen suorituskyky voidaan ennustaa tai arkkitehtuuripäätösten oikeellisuuteen voidaan ottaa kantaa etukäteen. Tässä aliluvussa käsitellään pintapuolisesti muutamaa erilaista lähestymistapaa, jolla arkkitehti voi varmistua lopullisen sovelluksen suorituskyvystä ja arkkitehtuurin oikeellisuudesta.

Suorituskyvyn ennakointi (*Software Performance Engineering, SPE*) suunnitteludokumenttien perusteella on suosittu tutkimuskohde (Haines, 2006; Smith & Williams, 2002; Woodside, ym., 2007). Pystymällä arvioimaan tarkasti järjestelmän suorituskyky jo ennen toteutusvaihetta antaisi erinomaisen kuvan lopputuotteesta ja erilaisten suunnittelupäätösten vaikutuksesta lopulliseen suorituskykyyn. Suorituskyvyn ennakointi siirtäisi sovellusarkkitehtuurin oikeellisuuden verifioinnin toteutusvaiheen lopusta suunnitteluvaiheen loppuun, joka parhaimmassa tapauksessa olisi huomattava kustannussäästö. Tällä hetkellä arkkitehtuurisuunnittelu on suunnittelijan kokemukseen perustuva valistunut arvaus. (Woodside, ym., 2007)

Ohjelmistokehitysprosessien siirtyminen kaupallisessa käytössä yhä enenevissä määrin kohti ketteriä ohjelmistokehitysmenetelmiä, on muuttanut ohjelmistoarkkitehtuurista vastaavien henkilöiden työnkuvaa, mutta arkkitehtuurisuunnittelun merkitys tuotteen lopulliseen suorituskykyyn on silti pysynyt samana. Ketterissä menetelmissä suosittu lopullisten suunnittelupäätösten venyttäminen mahdollisimman myöhäiseen vaiheeseen vaikeuttaa suorituskyvyn ennakointia. Jos ennen toteutusvaihetta pyritään minimoimaan kaikki tehtävät, on suorituskyvyn mallintaminen käytännössä mahdotonta. Kaupallisen käytön

esteenä ovat myös kustannukset. Asiakkaat eivät ole valmiita tekemään huomattavaa lisäpanostusta projektin alkuvaiheessa, jonka lisäarvo on vain arvio lopullisesta suorituskyvystä, vaikka tutkimukset osoittaisivatkin sen johtavan kustannusten säästöön pitemmällä aikavälillä. (Woodside, ym., 2007)

Tutkimuksissa yleisin ratkaisu on kohdejärjestelmän mallintaminen jo ennen toteutusvaihetta. Malli sisältää arviot eri komponenttien saamasta kuormituksesta ja niiden vuorovaikutuksesta. Etukäteisen mallintamisen kaupallisen käytön estää kuitenkin mallintamisen suuri työmäärä ja kyseenalaistettava luotettavuus. Vaikka arkkitehtuuri mallinnettaisiin kuinka tarkasti, kehitysprosessin aikana tulevat muutokset vaikuttavat aina suorituskykyyn ja täten muuttavat alkuperäistä arviota järjestelmän suorituskyvystä. (Woodside, ym., 2007)

Woodside ja kumppanit (2007) näkevät tulevaisuuden suuntauksena suorituskykytestin ja suorituskykytiedolla rikastetun arkkitehtuurimallin vuorovaikutus, jossa arkkitehtuurimallia tarkennetaan suorituskykytestin tuloksilla ja näin malli pysyisi ajan tasalla kehitysprosessin aikana. Malli olisi helposti hyödynnettävissä kehitysprosessin aikana tehtävien muutosten vaikutusten tunnistamisessa.

Suorituskykyä ei voida yhdistää ohjelmistoarkkitehtuuriin sen suunnittelun jälkeen. Jos suorituskyky on tärkeä vaatimus lopulliselle ohjelmistolle, on arkkitehtuuri suunniteltava suorituskyky huomioonottaen heti alusta. Smith ja Williams (2003) pitävät tärkeänä, että suorituskykykeskeisessä ohjelmistokehitysprosessissa myös arkkitehtuurin arviointi on olennaisessa osassa. Arkkitehtuurisilla valinnoilla on yksittäisistä aktiviteeteista suurin vaikutus ohjelmiston suorituskykyyn.

Arkkitehtuurien arviointiin on kehitetty lukuisia menetelmiä, joista tunnetuin on Architecture Tradeoff Analysis Method (ATAM). ATAM ottaa kantaa myös arkkitehtuurin suorituskykyvaatimuksiin ja niiden täyttymiseen, joten se soveltuu tässä tutkielmassa käsiteltävien ongelmien ratkaisemiseen. Huonona puolena on ATAM-prosessin raskaus, yleensä ainoastaan äärimmäisen suurissa projekteissa on resursseja toteuttaa arkkitehtuurin arviointi ATAM:ia käyttäen. Tässä tutkielmassa ei käydä läpi arkkitehtuurien arviointia syvällisemmin, mutta sen liittämisellä ohjelmistoprojektiin voi auttaa myös lopullisen suorituskyvyn säilyttämiseen. (Bass, ym., 2007)

Tässä luvussa käsiteltyjen suunnitteluun ja arkkitehtuureihin liittyvien käytänteiden liitetään seuraavassa sisältöluvussa asioita, joita ohjelmistokehittäjä voi kehityksessä käyttää apuna suorituskyvyn ylläpitämiseksi.

5 KOODITASOLLA TAPAHTUVA OPTIMOINTI

Tämä luku keskittyy ohjelmoinnissa yleisimpien suorituskykyyn negatiivisesti vaikuttavien toteutuspäätösten esilletuontiin ja niiden ratkaisuihin. Asiat pyritään esittämään siten, että ohjelmistokehittäjä saa niistä työkaluja tulevaisuuteen. Esimerkkejä ja muita havainnollistuksia käytetään mahdollisuuksien mukaan tukemaan sanallista ilmaisua. Väärien valintojen yhteydessä tuodaan esille myös, mikä suorituskyvyn hidastamisen aiheuttaa. Luvun alussa käsitellään tietorakenteiden vaikutuksia suorituskykyyn ja sen jälkeen käsitellään tarpeettomien lyhytikäisten olioiden luomista ohjelmistokoodissa. Sen jälkeen käydään läpi tilanteita, joissa kirjastoilla tai kehyksillä on vaikutus suorituskykyyn. Luvun lopussa käsitellään ohjelmistokoodin optimointia käsin ja sen tarpeellisuutta.

5.1 Tietorakenteet

Tietorakenteilla on suuri merkitys ohjelmiston lopullisen muistinkäytön kannalta. Huonosti suunnitellut ja toteutetut tietorakenteet hukkaavat ylimääräistä muistia ilman, että käytetty tietorakenne tuo sovellukselle mitään etua kevyempään rakenteeseen verrattuna. (Mitchell & Sevitsky, 2007)

Useat tutkimukset osoittavat, että erilaiset säiliöt ja niiden null-osoittimet kuormittavat Javan muistia tarpeettomasti. (Mitchell & Sevitsky, 2007; Xu, ym., 2010; Xu & Rountev, 2010) Tämänkaltaisia tietorakenteita ovat esimerkiksi käyttäjän istuntotietojen tallennus web-sovelluksessa. (Mitchell, Schonberg & Sevitsky, 2010)

Tässä aliluvussa käsitellään Javan säiliöiden tehokasta käyttöä, turhan kloonauksen aiheuttamaa kuormitusta sekä heikolla viitteellä tapahtuvaa muistivuodon ehkäisyä.

5.1.1 Kokoelmat

Kokoelmat ovat Javassa äärimmäisen suosittuja osittain helppokäyttöisyytensä vuoksi. Kokoelmat ovat myös alttiita väärinkäytölle samasta syystä. Pahin aiheutuva ongelma on muistivuoto, jos säiliön elinikä on pitkä ja sisältöä ei missään vaiheessa säiliön elinikää siivota. Lievempinä ongelmina Xu ja Rountev (2010) määrittävät säiliö-tietotyypeille kaksi ongelmatilannetta: *alikäytetty kokoelma* ja *ylitäytetty kokoelma*. Näissä tilanteissa sovellukselle ei aiheudu sellaista haittaa, että sen suoritus päättyisi ennenaikaisesti, mutta muisti- ja laskentaresursseja hukkuu. (Xu & Rountev, 2010)

Alikäytetty kokoelma on kokoelma, jossa on vähemmän arvoja kuin kokoelman oletuskoko. Kokoelman pystyisi mahdollisesti korvaamaan erillisillä muuttujilla tai olioattribuuteilla ja näin säästämään kokoelman käyttämät muistiresurssit. Esimerkki 1 sisältää esimerkin alitäytetystä kokoelmasta. Kokoelma sisältää ainoastaan kolme intervallia, joita käytetään uuden olion rakentajassa. Vector-säiliön oletuskoko on 10, joten allokointi vie turhaan muistiresursseja. Tämä toteutus olisi helposti korvattavissa lisäämällä rakentajaan parametreja, ilman että ohjelmakoodin luettavuus kärsii. (Xu & Rountev, 2010; Oracle, 2011b)

Kokoelma on ylitäytetty tilanteessa, jossa se sisältää paljon arvoja, mutta kyseiseen kokoelmaan tehdään ainoastaan yksittäisiä hakuja koodissa. Kokoelman sisältäessä arvoja, joita ei käytetä missään vaiheessa kokoelman elinikää, aiheuttavat nämä arvot ainoastaan muistiresurssien kulutusta sekä monimutkaistaa kokoelmalle tehtävää hakuoperaatiota. (Xu & Rountev, 2010)

Mitchell, Schonberg ja Sevitsky (2010) tuovat artikkelissaan esille erittäin suosittujen HashSet ja HashMap kokoelmatietotyyppien sisältävän huomattavan määrän suorituskykyä hidastavia piirteitä, kuten se, että HashSet-tietotyyppi delegoi arvojen säilyttämisen HashMap-tietotyypille. Käytettäessä näitä tietotyyppejä varastoimaan pieniä määriä dataa, aiheutuu tästä turhaa muistinkäyttöä. Lisäksi toteutuksessa tallennetaan tallennettavan olion hajautuskoodi (hashcode) turhaan. Ottaen huomioon tietotyyppien yleisen käytön, aiheutuu monimutkaiselle sovellukselle huomattava lisäkuormitus. Mitchell ja Sevitsky (2007) toteavat tutkimuksessaan, että erityisesti kokoelmat, jotka sisältävät toisen kokoelmat ovat äärimmäisen alttiita muistiresurssien hukkaamiselle, kun ne sisältävät erityisen paljon arvoja.

```
class CUP$LexParse$actions {
     RegExp makeNL(){
           Vector<Interval> list = new Vector<Interval>();
           list.addElement(new Interval('\n','\r'));
           list.addElement(new Interval('\u0085','\u0085'));
           list.addElement(new Interval('\u2028','\u2029'));
           RegExp1 c = new RegExp1(sym.CCLASS, list);
           ...
     }
}
```

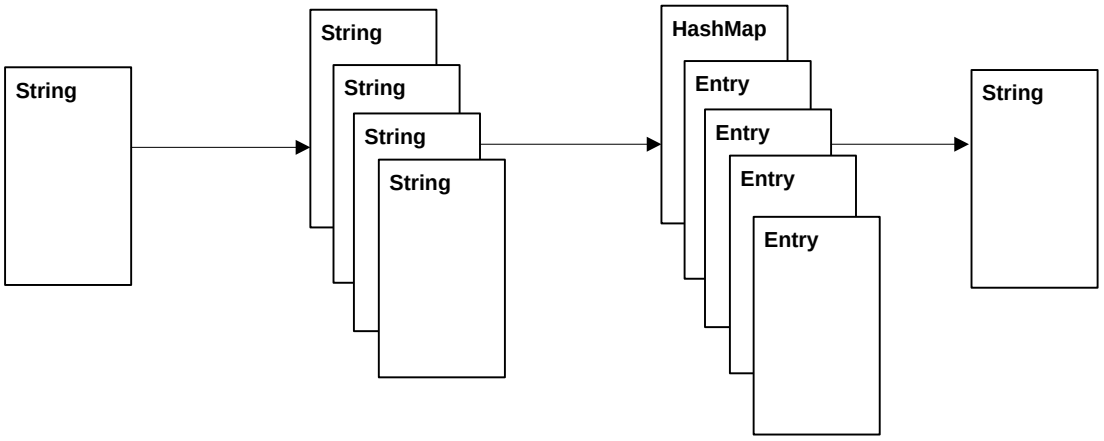
ESIMERKKI 1 alitäytetty kokoelma. (Xu & Rountev, 2010)

5.1.2 Tarpeeton kloonaus

Ohjelmakoodissa tapahtuva olioiden turha kloonaus ei näy ongelmana profiloitaessa, ja ohjelmistokehittäjät oikeuttavat tämän toiminnan ohjelmistokoodin uudelleenkäytön nimissä. Yleensä tarpeettomia kopioita syntyy tilanteessa, jossa dataa tallennetaan säiliöön tai luetaan sieltä. Oikeaoppisesti kloonauksen sijasta sovelluksessa käytettäisiin muistiviitettä ensimmäiseen olion ilmentymään aina kun mahdollista. (Xu, ym., 2009)

Kloonaus tapahtuu yleensä useammalla abstraktiotasolla ja sen huomaaminen ja korjaaminen on haasteellista. Erityistä on kiinnitettävä datan kulkuun sovelluksen sisällä. (Xu, ym., 2009)

Tarpeeton kloonaus täyttää hetkellisesti virtuaalikoneen muistia ja pakottaa näin ylimääräiseen roskienkeruuseen. Xu ja kumppanit (2009) esittelevät artikkelissaan esimerkin IBM:n dokumentinhallintajärjestelmästä, jossa evästeen (cookie) purkamisessa jokaisesta tiedosta luotiin neljä täysin erillistä kopiota virtuaalikoneen muistiin. Kuvio 6 havainnollistaa tilannetta, jossa yhdestä merkkijonosta jäsennetään useita merkkijonoja, jotka määrittävät sovelluksen asetuksia. Nämä merkkijonot tallennetaan HashMap-kokoelmaan, josta palautetaan yksi ilmentymä koodissa eteenpäin. Kun muita asetuksia tarvitaan seuraavan kerran, suoritetaan sama merkkijonon jäsentäminen uudestaan. Tämä kopioketju olisi optimoitavissa siten, että palautettaisiin suoraan haluttu merkkijono eikä luotaisi turhaa säiliötä lainkaan. (Xu, ym., 2009)



KUVIO 6 Merkkijonon kopioketju (Xu, ym., 2009)

Kloonausketjujen poistamisella Xu & kumppanit (2009) ovat kyenneet parantamaan olemassa olevien sovelluksien suorituskykyä huomattavasti (10–30 %). Huomattava osa turhasta kloonauksesta tapahtuu sovelluksen ajonaikaisten tapahtumien kirjoittamisesta lokitiedostoon.

5.1.3 Heikko viite

Osa suorituskykyongelmia ja jopa kaatumisia aiheuttavista muistivuodoista voidaan saada haltuun käyttämällä heikkoja tai pehmeitä viitteitä. Kun olioon on jäljellä ainoastaan heikkoja tai pehmeitä viitteitä, voi Java virtuaalikone tarvittaessa poistaa kyseisen olion roskienkeruussa. Heikkoa viitettä käyttävät toteutukset ovat hyödyllisiä esimerkiksi työ- tai yhteysjonotototeutuksissa sekä erilaisissa välimuisteissa, joissa toiminnon suorittamisen jälkeen olio on tarpeeton tai se ei ole myöhemmän suorituksen kannalta välttämätön. (Goetz, 2005)

Heikon ja pehmeän muistiviitteen eroaa ainoastaan roskienkeruun aikana. Heikko viite päätyy roskienkeruuseen, kun olioon on sovelluksen ajon aikana ainoastaan heikko viite. Pehmeä viitteen takana oleva olio päätyy roskienkeruuseen vasta sitten, kun olioon on ainoastaan pehmeitä tai heikkoja viitteitä ja virtuaalikoneen muisti on täyttymässä. Pehmeän viitteen omaava olio voi pysyä muistissa hyvinkin pitkään, mutta virtuaalikone siivoaa pehmeän viitteen sisältävät oliot viimeistään ennen kuin suoritus päättyy muistin täyttymisen johdosta virheeseen. (Oracle, 2011a) Monimutkaisemmissa välimuistitoteutuksissa välimuistin siivoaminen on yleensä monimutkaisempaa kuin roskienkeruun tarjoama ratkaisu. Näitä ovat esimerkiksi vähiten käytettyjen olioviitteiden poistaminen tai vanhimman viitteen poistaminen. (Bloch, 2008)

Esimerkki 2 sisältää toteutuksen esimerkiksi palvelimen sisääntuleville yhteyksille, joka ei voi aiheuttaa muistivuotoa. Jos sovelluksessa käytettäisiin perinteistä HashMap-kokoelmaa, aiheuttaisi se pidemmän suorituksen jälkeen ohjelmiston kaatumisen, koska kokoelma ei siivoa sisältöään missään vaiheessa suoritusta. (Goetz, 2005)

```
public class WeakHashMap<K,V> implements Map<K,V> {

    private static class Entry<K,V> extends WeakReference<K>
      implements Map.Entry<K,V> {
        private V value;
        private final int hash;
        private Entry<K,V> next;
        ...
    }
    public V get(Object key) {
        int hash = getHash(key);
        Entry<K,V> e = getChain(hash);
        while (e != null) {
            K eKey= e.get();
            if (e.hash == hash && (key == eKey || key.equals(eKey)))
                return e.value;
            e = e.next;
        }
        return null;
    }
```

ESIMERKKI 2 Muistivuodon estäminen heikolla viitteellä (Goetz, 2005)

5.2 Abstraktiot ja koodin uudelleenkäyttö

Java tarjoaa tehokkaat työkalut ohjelmistokoodin eristämiseen muusta koodista suorituksen aikana. Abstraktiot ovat tehokas tapa erottaa kokonaisuudet toisistaan, mutta harvoin otetaan huomioon sen mahdollista vaikutusta suorituskykyyn. (Mitchell, ym., 2010)

Ohjelmistokoodin uudelleenkäyttöä pidetään hyvänä ohjelmointitapana ja etenkin kirjaston sisällä saman koodin uudelleenkäytön pitäisi parantaa ylläpidettävyyttä. Liiallisesta uudelleenkäytöstä voi tulla huomattava ongelma ja metodikutsuketjut hidastavat ohjelman suoritusta. Pitkien kutsuketjujen mukana tulee yleensä myös paljon lyhytikäisiä olioita, jotka kuormittavat roskienkeruuta. Vääränlaisesta uudelleenkäytöstä hyvä esimerkki on Javan peruskirjaston `SimpleDateFormat`–luokka, jossa merkkijonon jäsentäminen ja rakentaminen olioksi aiheuttaa 66 metodikutsua ja 30 lyhytikäisen olion luomisen. (Mitchell, ym., 2010)

5.3 Lyhytikäisten olioiden laajamittainen luominen

Lyhytikäisten olioiden laajamittainen luominen (object churning) on tutkimuksen perusteella yleinen suorituskykyongelma Java-pohjaisissa sovelluksissa, jotka perustuvat kehyksiin. Jotta tämänkaltaisia suorituskykyongelmia pysytään havaitsemaan, täytyy kohdesovellusta pystyä kuormittamaan. Monissa tilanteissa olioiden luominen liittyy datan käsittelyyn, joten myös sovelluksen sisältämään dataan pitää kiinnittää myös huomiota. (Dufour, Ryder & Sevitsky, 2007) Lyhytikäiset oliot hidastavat sovelluksen toimintaa lisäämällä roskienkeruuajojen tiheyttä sekä täyttämällä prosessorin L1- ja L2-välimuistit. (Mitchell, ym., 2010)

Tämänkaltaisten ongelmien optimoiminen voi osoittautua mahdottomaksi tilanteessa, jossa osasyylliseksi paljastuu API-toteutus tai kirjasto, jonka lähdekoodit eivät ole saatavilla. (Shankar, Arnold & Bodík, 2008) Pahimmassa tapauksessa suorituskykyongelma johtuu yhden tai useamman kehystoteutuksen ongelmista. (Dufour, Ryder & Sevitsky, 2007)

Tulevaisuudessa voitaneen nähdä Java-virtuaalikoneessa tapahtumia automaattisia optimointeja, jotka pystyvät poistamaan sovelluksesta kohdat, jotka luovat suuria määriä lyhytikäisiä olioita turhaan. Shankar ja kumppanit (2008) tuovat tutkimuksessaan esille tavan, jossa virtuaalikone automaattisesti havaitsee paljon lyhytaikaisia olioita luovan koodin ja pyrkii optimoimaan laventamalla (inline) koodin. (Shankar, ym., 2008)

5.4 Kirjastojen ja kehysten vaikutus suorituskykyyn

Java EE-sovellusympäristö on laaja kokoelma erilaisiin tarpeisiin suunniteltuja kirjastoja ja kehystoteutuksia. Kirjastot ja kehykset lisäävät ohjelmistokehittäjän tuottavuutta huomattavasti poistamalla tarpeen tehdä perusasioita koodissa uudestaan ja uudestaan. Toisaalta päällekkäiset kehystoteutukset voivat aiheuttaa sen, että yksinkertainen operaatio suoritetaan äärimmäisen monimutkaisesti. (Dufour, Ryder & Sevitsky, 2008)

Kehystoteutusten päällekkäisyys voi aiheuttaa suorituskykyongelman pääasiassa kahdesta syystä: Suorituskykyongelma ilmenee sovelluksen luomalla suuren määrän olioita, koska operaation pilkkominen moneen vaiheeseen kehystoteutuksessa aiheuttaa samankaltaisten operaatioiden toistamista. (Dufour, Ryder & Sevitsky, 2008) Ongelma voi olla myös epätehokkaan algoritmin aiheuttama, koska kehystoteutusta ei käytetä, kuten alun perin on suunniteltu ja tarkoitettu. (Mitchell, ym., 2010)

Suurempia ongelmia voi ilmaantua, jos sovellus tai kehystoteutus on riippuvainen kirjastosta, mutta kirjastoa käytetään koodissa erilailla, kuin se, miten kirjasto on alun perin suunniteltu käytettäväksi. Mitchell ja kumppanit (2009) tuovat esille myös tilanteen, jossa `DecimalFormat`–luokan jäsennystoteutusta on käytetty päivämäärän jäsentämiseen merkkijonosta. `DecimalFormat`–luokan sisältämä koodi oli suunniteltu hyvin paljon monimutkaisempaan käyttötapaukseen ja tämän vuoksi jäsentäminen oli huomattavasti tarvittavaa raskaampi. (Mitchell, ym., 2010)

APIn tai kirjaston toteutuksen yliajaminen voi olla perusteltua joskus, mutta Bloch muistuttaa kirjassaan, että myös Javan standardikirjastot ovat jatkuvan kehityksen alaisena, joten nykyisessä versiossa sijaitseva suorituskykyongelma voi olla korjattu seuraavassa versiossa. (Bloch, 2008, 234–236)

5.5 Optimointi

Ohjelmistojen suorituskykyongelmia voidaan ratkoa optimoimalla ohjelmakoodia käsityönä. Ohjelmakoodin ylläpidettävyyden kannalta on äärimmäisen tärkeää, ettei ohjelmakoodiin tehdä ylläpitoa tai kehitystä hankaloittavaa optimointia ennen kuin ohjelmisto on tuotantokäytössä ja suorituskykyongelmasta ei päästä eroon millään muulla tavalla. Optimoinnilla voidaan piilottaa mahdollinen suunnitteluvirhe muilta ohjelmistokehittäjiltä muuttamalla ohjelmakoodi lähes lukukelvottomaksi. (Smaalders, 2006)

Hyde (2009) kirjoittaa artikkelissaan, että ohjelmoijat ovat vieraantuneet optimoinnista, vaikka se on olennainen osa ohjelmiston elinkaarta. Syynä tähän hän pitää Sir Tony Hoaren lausahdusta "Ennenaikainen optimointi on kaiken pahan alku ja juuri", joka asiayhteydestä irrotettuna kannustaa ohjelmistokehittäjää jättämään optimoinnin tekemättä. Hoare kuitenkin tarkoitti sanonnallaan, että mikro-optimointi on liian aikaisessa vaiheessa turhaa, mutta korkeammalla

tasolla suorituskyvyn huomioonottaminen on ensiarvoisen tärkeää lopputuotteen suorituskyvyn kannalta. (Hyde, 2009)

Optimointi voi vaikuttaa ohjelmiston suorituskykyyn jopa negatiivisesti, jos optimointi tehdään huolimattomasti ja kokonaisuuteen ei kiinnitetä huomiota. Yleensä riittävä suorituskykyparannus saadaan, kun optimointi suoritetaan vain yleisimmin käytetylle ohjelmistokomponentin osalle. (Smaalders, 2006)

Smith ja Williams tuovat kirjassaan esille keskittyvän periaatteen (centering principle), jossa suorituskykyongelmiin keskitytään vain niissä ohjelmiston osissa, joita käytetään eniten. Periaate on johdettu niin kutsutusta 80/20–ajattelumallista, jossa 80 % suorituskykyongelmista johtuu 20 % ohjelmistokoodista. Keskeisimmät ohjelmiston osat voidaan määrittää käyttäjädatasta tai ne voidaan analysoida käyttötapauskuvauksista. Huomioon on kuitenkin otettava myös ohjelmistossa harvemmin käytettävät operaatiot, jos niillä on huomattava vaikutus muun järjestelmän suorituskykyyn. (Smith & Williams, 2002, 245–247) Hyde (2009) esittää suoraa kritiikkiä keskittyvää periaatetta kohtaan argumentoimalla, että se 20 %, joka aiheuttaa suorituskykyongelmat, on ripoteltu pieninä osina ympäri ohjelmistoa. Sen lisäksi, että nämä ongelmakohdat on vaikea löytää, on myös todennäköistä, että ohjelmistokoodissa on laadullisia ongelmia ja lopullinen korjaaminen voi pahimmassa tapauksessa vaatia ohjelmistokomponentin uudelleenkirjoittamisen.

5.6 Tietoinen suorituskyvyn luovuttaminen

Hyde (2009) pitää äärimmäisen tärkeänä, että ohjelmistoa kehitettäessä tehtyjen päätösten vaikutus suorituskykyyn tiedostetaan. Suorituskykyongelmia muodostuu erityisesti silloin, kun ohjelmistokehittäjä päätöksellään heikentää suorituskykyä tietämättään päätöksen vaikutuksia. Hän pitää kuitenkin suotavana suorituskyvyltään epäoptimien ratkaisujen tekemisen, jos niille löytyy kunnolliset perusteet. Muun muassa koodin ylläpidettävyys ja luettavuus luetaan hyviin perusteisiin suorituskyvyltään parhaan ratkaisun käyttämättä jättämiselle. Kokemattomalle ohjelmistokehittäjälle tämänkaltaisten ratkaisujen tekeminen on haasteellista, Hyde suosittelee aloitteleville kehittäjille profilointityökalujen aktiivista käyttöä.

6 YHTEENVETO

Tutkielmassa käsiteltiin yleisellä tasolla, millainen on Java EE-sovellus ja sen suoritusympäristö, perusteet valvonnalle ja virheiden etsinnälle sekä työkaluja suorituskyvyn hallintaan suunnittelu- ja toteutusvaiheessa.

Java EE-suoritusympäristö on suunniteltu hyvin monenlaisiin tarpeisiin ja käyttötapauksiin, erityisesti monimutkaisia ja laajoja sovelluksia silmälläpitäen. Laajaa ja monimutkaista suoritusympäristöä ei oteta käyttöön tilanteessa, jossa sovelluksen tarvitsee tehdä yksinkertaisia asioita, joten myös näissä ympäristöissä ajettavat sovellukset ovat monesti äärimmäisen monimutkaisia. Suorituskykyongelma muodostuu, jos sovellus ei kykene suorittamaan sille annettua tehtävää määritetyssä ajassa. Näiden ongelmien rajoittamiseen tämä tutkielma pyrkii vastaamaan.

Suorituskykyongelman löytäminen vaatii mittaamista. Suorituskykyongelmien ratkaisussa kannattaa keskittyä sellaisten pullonkaulojen etsimiseen, jotka suoritetaan normaalin käytön aikana hyvin usein. Kaikkien sovelluksessa olevien suorituskykyongelmien paikantaminen ja korjaaminen olisi huomattava resurssihukka, koska kaikilla pullonkauloilla ei ole samanlaista vaikutusta koko sovelluksen toimintaan. Pullonkaulojen etsimiseen on tarjolla kehittyneet työkalut, mutta niiden käyttäminen vaatii määrätietoisuutta ja näkemystä, mitä kannattaa mitata.

Tutkielmassa esitetyillä neuvoilla voidaan hallita monimutkaisenkin Java EE-sovelluksen suorituskykyä, mutta todellisuus monesti ei ole yhtä helppo ja suoraviivainen kuin lista neuvoja antaa olettaa. Kehitysprosessin aikana on haastavaa pitää suunnitteluvaiheessa määritetyt suorituskykytavoitteet, varsinkin, jos toimintoihin tulee huomattavan suuria muutoksia kehitysprosessin aikana. Luvun 4 työkalut mahdollistavat suorituskyvyn hallinnan kehityksen aikana, mutta ne vaativat huomattavaa projektinhallinnallista lisäpanostusta. Näiden prosessien tuoman lisätyön integroiminen nykyisin suosittuihin ketteriin menetelmiin on haastavaa, paitsi jos sovelluksen testauskäytäntöihin liitetään myös suorituskykytestaus. Suorituskykytestauksen yhdistämisellä muihin testauskäytäntöihin, kuten automaattiseen testaukseen, voidaan järkevällä työ-

määrällä saavuttaa sovelluskehitysprosessi, jossa suorituskyvyn hallinta on ketteriäkin menetelmiä käytettäessä mahdollista.

Kiinnittämällä kehitystehtävien ohessa huomiota omaan ohjelmistokoodiin sekä sovellukseen ja sen suoritusympäristön toimintaan, esimerkiksi aktiivisella profilointityökalujen käytöllä, lisääntyy kehittäjän ymmärrys järjestelmästä. Välillisesti tämä parantaa kehitettävän järjestelmän suorituskykyä pitkällä aikavälillä sekä laajentaa kehittäjän ymmärrystä järjestelmästä. Lopullisen ohjelmakoodin tulisi olla suorituskyvyltään parasta, mitä ylläpidettävyyttä vaarantamatta voidaan kirjoittaa. Tulevaisuudessa löytynee huomattavasti enemmän ohjelmistokoodin suorituskykyongelmat huomioon ottavia analyysityökaluja kehitystyötä tukemaan.

Yleisesti tässä tutkielmassa käytetty lähdeaineisto keskittyi osoittamaan suorituskykyongelmia sillä tasolla, mihin normaalissa työympäristössä harvoin törmätään. Monet löydetyt ongelmat sijaitsivat kehysjärjestelmissä, suoritusympäristöissä ja virtuaalikone-toteutuksissa, joiden muuttaminen on normaalille kehittäjälle järkevässä ajassa mahdollisuuksien ulkopuolella. Luonnollisesti ajankäyttö ja harkinta kehysjärjestelmiä ja kirjastoja valittaessa mahdollistaa pahimpien sudenkuoppien kiertämisen ja paremman lopputuloksen. Vikojen ilmetessä kirjastoja myös päivitetään nopeaan tahtiin, mutta migraatio uuteen versioon voi aiheuttaa pahimmillaan huomattavastikin odottamatonta lisätyötä.

Tämän tutkielman aihealue on hyvin laaja ja asioita käsiteltiin äärimmäisen korkealta tasolta. Tätä tutkimusta voisi laajentaa syventämällä jossain sisältöluvussa esiteltyä aihetta tai sitten tutkimalla käyttöjärjestelmän ja Java-virtuaalikoneen vaikutusta sovelluksen lopulliseen suorituskykyyn.

LÄHTEET

Ammons, G., Choi, J.-D., Gupta, M., Swamy, N. (2004). Finding and removing performance bottlenecks in large systems. *European Conference on Object-Oriented Programming*, Springer-Verlag, 172-196.

Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G., Xu, G. (2010). Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. *Proceedings of the FSE/SDP workshop on Future of software engineering research.* ACM, 421-426.

Bass, L., Nord, R., Wood, W., Zubrow, D. (2007). Risk Themes Discovered Through Architecture Evaluations. Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture. IEEE Computer Society. 55-64.

Bloch, J., (2008). Effective Java (2. uud. painos). Boston, MA, USA: Addison-Wesley.

Bond, M. D., McKinley, K. S. (2009). Leak Pruning. *APLOS 2009*. ACM.

Bondi, A., (2009). The Software Architect as the Guardian of System Performance and Scalability. *ICSE '09 Workshop*. IEEE, 28-31.

Chow, K. (2003). Enterprise java performance: Best practices. Intel.

Dufour, B., Ryder, B., Sevitsky, G. (2008). A Scalable Technique for Characterizing the Usage of Temporaries in Framework-intensive Java Applications. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 59-70.

Dufour, B., Ryder, B. G., & Sevitsky, G. (2007, July). Blended analysis for performance understanding of framework-based applications. In Proceedings of the 2007 international symposium on Software testing and analysis (pp. 118-128). ACM.

Farley, J., Crawford, W. (2005). Java Enterprise in a Nutshell. O'Reilly.

Felde, C. (2010, 27. kesäkuuta). C++ vs. Java performance; It's a tie. Haettu 27.10.2012 osoitteesta http://blog.cfelde.com/2010/06/c-vs-java-performance/.

Ferrari, G., Shrivastava, S., Ezhilchelvan, P. (2004). An Approach to Adaptive Performance Tuning of Application Servers. *In 1st IEEE International Workshop on QoS in Application Servers, in conjunction with SRDS04*, IEEE.

Goetz, B. (2005, 22. marraskuuta). Java theory and practice: Plugging memory leaks with weak references. Haettu 27.10.2012 osoitteesta http://www.ibm.com/developerworks/java/library/j-jtp11225/.

Gousios, G., Spinellis, D. (2008). Java Performance Evaluation Using External Instrumentation. IEEE, 173-177.

Haines, S., (2006). Pro Java EE5 Performance Management and Optimization. New York, NY, USA: Apress.

Hovermeyer, D., Pugh, W. (2004). Finding Bugs is Easy. *OOPSLA 2004.* ACM, 92-106.

Hyde, R. (2009). The Fallacy of Premature Optimization. ACM Ubiquity. ACM, 3.

Jendrock, E., Cervera-Navarro, R., Evans, I., Gollapudi, D., Haase, K., Oliveira, W. M., Srivathsa, C. (2012, kesäkuu). The Java EE 6 Tutorial. Oracle. Haettu 27.10.2012 osoitteesta http://docs.oracle.com/javaee/6/tutorial/doc/index.html.

Kounev, S., Weis, B., Buchmann, A. (2004). Performance Tuning and Optimization on J2EE Applications on the JBoss Platform. *Journal of Computer Resource Management*. CMG, 113.

Oracle. (2011a). Java Platform Standard Edition 6 API: Weak Reference. Haettu 17.11.2012 osoitteesta http://docs.oracle.com/javase/6/docs/api/java/lang/ref/WeakReference.html

Oracle. (2011b). Java Platform Standard Edition 6 API: Vector. Haettu 17.11.2012 osoitteesta docs.oracle.com/javase/6/docs/api/java/util/Vector.html.

Lau, J., Arnold, M., Hind, M., & Calder, B. (2006). Online performance auditing: using hot optimizations without getting burned. In ACM SIGPLAN Notices (Vol. 41, No. 6, pp. 239-251). ACM.

Liang. S., Viswanathan, D. (1999). Comprehensive Profiling Support in the Java Virtual Machine. *5th USENIX Conference on Object-Oriented Technologies and Systems*. USENIX.

Louridas, P. (2006). Static Code Analysis. *IEEE Software*, 6, 58-61.

Maxwell, E. K., Back, G., Ramakrishnan, N. (2010). Diagnosing Memory Leaks using Graph Mining on Heap Dumps. *KDD 2010*. ACM, 115-124.

Mitchell, M., Schonberg, E., Sevitsky, G. (2010). Four Trends Leading to Java Runtime Bloat. *IEEE Software*, 1, 56-63.

Mitchell, N., Sevitsky, G. (2007). The Causes of Bloat, The Limits of Health. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. ACM, 245-260

Pawlan, M. (2001). Java 2 Enterprise Edition Technology Center. Oracle. Haettu 23.12.2011 osoitteesta http://java.sun.com/developer/technicalArticles/J2EE/Intro/#arch.

Pugh, B., Spacco, J. (2004). RUBiS Revisited: Why J2EE Benchmarking is Hard. *OOPSLA 2004*. ACM, 204-205.

Richardson, C. (2006). Untangling Enterprise Java. ACM Queue. ACM, 6, 36-44.

Shankar, A., Arnold, M., & Bodik, R. (2008). Jolt: lightweight dynamic analysis and removal of object churn. In ACM Sigplan Notices (Vol. 43, No. 10, pp. 127-142). ACM.

Smaalders, B. (2006). Performance anti-patterns. ACM Queue, 4(1), 44-50.

Smith, C., Williams, L. (2002). Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Indianapolis, IN, USA: Addison-Wesley.

Smith, C., Williams, L. (2003). Best Practices for Software Performance Engineering.

Vaadin. (2012) Haettu 11.3.2012 osoitteesta https://vaadin.com/learn.

Woodsize, M., Granks, G., Petriu, D. (2007). The Future of Software Performance Engineering. *Proceedings of the 29th International Conference on Software Engineering, Workshop on the Future of Software Engineering*. IEEE Computer Society. 171-187.

Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G. (2009). Go with the Flow: Profiling Copies to Find Runtime Bloat. *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. ACM.

Xu, G., Mitchell, N., Arnold, M., Rountev, A., Schonberg, E., Sevitsky, G. (2010). Finding Low-Utility Data Structures. *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 174-186.