OULUN YLIOPISTO
UNIVERSITY of OULU

Java-pohjaiset ohjelmalliset transaktiomuistit

Oulun Yliopisto
Tietojenkäsittelytieteiden laitos
Pro gradu -tutkielma
Antti Ollila
2015

Tiivistelmä

Erilaiset järjestelmät suorittavat nykypäivänä yhä enemmän rinnakkaista laskentaa prosessorien ja niiden laskentaytimien lukumäärän kasvaessa. Tämän seurauksena perinteiset lukkoihin perustuvat rinnakkaisuuden hallintamenetelmät tulevat koko ajan monimutkaisemmiksi käyttää ja implementoida. Ohjelmalliset transaktiomuistit ovat vaihtoehto perinteisille lukkoihin perustuville rinnakkaisuuden hallintamenetelmille. Ne lupaavat estää yhden perinteisten lukkojen pahimmista haittapuolista, eli lukkiutumien tapahtumisen, sekä yksinkertaistaa rinnakkain suoritettavan koodin kirjoittamista.

Tässä pro gradu -tutkielmassa käytettiin tutkimusmenetelmänä teknologiavertailua, jonka tavoitteena oli selvittää ovatko ohjelmalliset transaktiomuistit varteenotettava vaihtoehto perinteisille lukkoihin perustuville ratkaisuille. Tutkielmassa kahteen valittuun synkronointiongelmaan, eli tuottaja-kuluttaja- ja lukija-kirjoittaja-ongelmaan, luotiin ratkaisut käyttäen yleiskatsauksessa valittuja erilaisia Javaan pohjautuvia ohjelmallisia transaktiomuistitoteutusvaihtoehtoja. Vertailun vuoksi synkronointiongelmiin tehtiin myös perinteisiin lukkoihin perustuvat toteutukset. Luotuja toteutuksia mitattiin ja vertailtiin tämän jälkeen käyttäen valittuja mittausparametreja.

Saatujen mittaustulosten perusteella tutkimukseen valitut ohjelmalliset transaktiomuistit erosivat huomattavasti toisistansa sekä helppokäyttöisyyden että suorituskyvyn osalta. Deuce STM on huomattavasti helppokäyttöisempi kuin ScalaSTM, mutta kääntöpuolena sen suorituskyky on useissa tilanteissa huomattavasti heikompi. Myös molempien ohjelmallisten transaktiomuistien käyttämät vuonohjausmekanismit eroavat huomattavasti toisistansa. Lukkopohjainen toteutus suoriutui kuitenkin yleensä ottaen molempia ohjelmallisia transaktiomuistitoteutuksia paremmin.

Tutkimuksen perusteella vaikuttaisi, että käytettäessä Java-ohjelmointikieltä, perinteisiin lukkoihin perustuva rinnakkaisuudenhallinta on vielä toistaiseksi ohjelmallisia transaktiomuisteja parempi vaihtoehto. Jos ratkaistava synkronointiongelma on hyvin monimutkainen ja sen ratkaisemisen tueksi ei löydy Java-ohjelmointikielestä valmiita mekanismeja, nousee hyvin optimoitu ohjelmallinen transaktiomuisti, kuten ScalaSTM, varteenotettavaksi vaihtoehdoksi. Lisäksi jos synkronointiongelmaan kuuluu huomattavasti enemmän lukuoperaatioita suhteessa kirjoitusoperaatioihin, voi ohjelmallinen transaktiomuistitoteutus tarjota mahdollisesti jopa suorituskyvyn lisäystä verrattuna perinteisiin lukkopohjaisiin ratkaisuihin.

Tutkielmassa keskityttiin tarkastelemaan kahta eri synkronointiongelmaa käyttäen kahta eri Java-pohjaista ohjelmallista transaktiomuistitoteutusta sekä yhtä lukkoihin perustuvaa toteutusta. Tämä rajoittaa tutkielmassa saatujen tuloksien yleistettävyyttä.

Avainsanat
ohjelmallinen transaktiomuisti, rinnakkaisuus, rinnakkainen ohjelmointi, lukkiutuma

Ohjaajat
Yliopistonlehtori, Ari Vesanen
Yliopistonlehtori, Antti Siirtola

Sisällysluettelo

1. Johdanto

Tämän tutkielman tavoitteena on vertailla ja arvioida erilaisia ohjelmallisia transaktiomuistitoteutuksia. Ohjelmallinen transaktiomuisti on vaihtoehtoinen lähestymistapa rinnakkaisuuden hallintaan järjestelmissä. Tässä luvussa esitellään lyhyesti aiheeseen liittyvät käsitteet ja niiden merkitykset.

1.1 Säikeet, rinnakkaisuus ja samanaikaisuus

Säikeet (threads), rinnakkaisuus (concurrency) ja samanaikaisuus (parallel) ovat keskeisiä käsitteitä moniajossa sekä siihen liittyvässä lukkiutumisien hallinnassa. Tässä tutkielmassa edellä mainituilla käsitteillä tarkoitetaan seuraavaa. *Samanaikaisuudella* tarkoitetaan tässä yhteydessä, että useaa säiettä suoritetaan eteenpäin samaan aikaan. *Säikeellä* tarkoitetaan ohjelman osaa, joka suoritetaan joko näennäisesti tai todellisesti rinnakkain järjestelmän muiden mahdollisten säikeiden kanssa. Useita säikeitä voidaan suorittaa *näennäisesti rinnakkaisesti* yksiprosessoriympäristöissä, eli järjestelmissä jossa on vain yksi laskentayksikkö, jolloin järjestelmä jakaa pieniä siivuja laskenta-ajasta kullekin säikeelle. Tässä tapauksessa säikeitä sanotaan suoritettavan näennäisesti rinnakkaisesti, koska säikeiden suoritus vaikuttaa käyttäjän näkökulmasta samanaikaiselta, mutta käytännössä järjestelmä vaihtelee säikeiden suorittamista niiden kesken. Toisin sanoen tässä tapauksessa useiden säikeiden käyttäminen ei voi pienentää järjestelmältä vaadittua suoritusaikaa tietyn työmäärän suorittamiseen (Kugler, 2013.)

Järjestelmissä joissa on useita prosessoreja tai järjestelmän prosessori pitää sisällään useita laskentaytimiä (core), säikeiden suoritus voi tapahtua *todella rinnakkaisesti*. Useampia säikeitä suoritetaan siis samanaikaisesti yhtä aikaa eteenpäin. Tällöin tietyn työmäärän suorittamiseen kuluvaa aikaa voidaan pienentää, edellyttäen, että useiden säikeiden suorittaminen on tehokkaasti toteutettu kyseisessä järjestelmässä (Kugler, 2013.)

1.2 Haasteet

Useiden säikeiden rinnakkainen suorittaminen järjestelmissä tarjoaa monenlaisia haasteita ohjelmien suunnittelijoille. Nämä haasteet liittyvät keskeisesti resurssien jakamiseen säikeiden kesken. Jos useampi säie yrittää muokata samaa resurssia samanaikaisesti, on mahdollista, että niiden välillä tapahtuu konflikti. Tämä tarkoittaa käytännössä, että resurssin tila ei säikeiden suorittamisen jälkeen ole se mikä sen loogisesti pitäisi olla (Kugler, 2013.) Voimme ottaa tästä esimerkkinä resurssin A, jonka alkuarvo on 0. Oletamme, että on kaksi säiettä, jotka kummatkin muokkaavat kyseistä resurssia A kasvattamalla sen arvoa yhdellä. Kummankin säikeen suorittamisen jälkeen on siis resurssin A arvo loogisesti oltava 2, koska molemmat säikeet kasvattivat sen arvoa yhdellä. Ilman lukkojen käyttöä resurssin A arvo voi kuitenkin olla säikeiden suorittamisen jälkeen joko 1 tai 2. Kuten aiemmin mainittiin, tämä johtuu siitä, että ohjelmoija ei pysty hallitsemaan missä järjestyksessä yksittäisiä säikeitä suoritetaan osa-osalta eteenpäin. Molemmat säikeet voivat siis lukea resurssin A arvon 0 samaan aikaan, kasvattavat luettuja arvoja yhdellä, jonka jälkeen molemmat säikeet kirjoittavat saadun tuloksen eli arvon 1 resurssiin A. Toisaalta, jos säikeiden suoritus sattuu tapahtumaan peräjälkeen, on lopputuloksena resurssin A arvo 2. Tällöin ensimmäinen

säie lukee resurssin A arvon 0, kasvattaa luettua arvoa yhdellä, ja lopuksi kirjoittaa arvon 1 resurssiin A. Tämän jälkeen toinen säie vastaavasti lukee resurssin A arvon 1, kasvattaa luettua arvoa yhdellä, ja lopuksi kirjoittaa arvon 2 resurssiin A. Voidaan siis sanoa, että kummatkin säikeet kilpailevat resurssiin A pääsystä. Tätä ilmiöitä kutsutaankin näin ollen *kilpailutilanteeksi* (race condition).

Kilpailutilanteet ovat näin ollen vaarallisia, koska ne tekevät järjestelmästä potentiaalisesti arvaamattoman. Kilpailutilanteet voivat lisäksi toteutua myös hyvin harvoin, jolloin niiden olemassa oloa on vaikeata huomata järjestelmää suunniteltaessa ja testattaessa. Järjestelmä voi toimia pitkänkin aikaa normaalisti, kunnes kilpailutilanteen toteutuminen voi tehdä järjestelmästä haavoittuvan (Marshall, 2005.) Kilpailutilanteiden toteutuminen on perinteisesti estetty käyttämällä hyväksi *poissulkevia lukkoja* (mutual-exclusion locks), jolloin resurssiin pääsee käsiksi vain yksi säie kerrallaan. Poissulkevien lukkojen käyttämisestä aiheutuu kuitenkin uusia potentiaalisia ongelmia järjestelmälle. Näihin ongelmiin voidaan sanoa kuuluvaksi lukkiutumat (deadlocks), elolukot (livelocks), nälkiintyminen (starvation), prioriteettien kääntymiset (priority inversion) ja aikatauluttajan (scheduler) toimintaan liittyvät ongelmat.

Lukkiutumalla tarkoitetaan tilannetta, jossa säikeet jäävät pysyvästi odottamaan toistensa lukitsemia resursseja. Tällöin molempien säikeiden suoritus estyy kokonaan. *Elolukolla* tarkoitetaan tilannetta, jossa säikeiden tilat muuttuvat jatkuvasti, mutta niiden suoritus ei etene koska ne reagoivat toistensa tekemiin muutoksiin. Tämä muistuttaa tilannetta, jossa kaksi käytävällä olevaa henkilöä väistävät aina toistensa eteen, jolloin he eivät pääse toistensa ohitse (Marshall, 2005; Oracle-sivusto, 2015e.) Lukkiutumat ovat siis hyvin epätoivottava ilmiö missä tahansa järjestelmässä. Jotta tämä ilmiö voidaan estää tai välttää, on järjestelmän käyttämät lukkomekanismit suunniteltava niin hyvin, että lukkiutumia ei pääse missään tilanteissa tapahtumaan. Vaihtoehtoisesti järjestelmään on rakennettava mekanismit lukkiutumien havaitsemiseen ja purkamiseen (Stallings, 1998, sivut 253-279; Silberschatz ja Galvin, 1998, sivut 245-268.) Molemmat toteutustavat voivat olla hyvinkin haasteellisia riippuen rakennettavan järjestelmän koosta ja monimutkaisuudesta. *Nälkiintymisellä* tarkoitetaan tilannetta, jossa säie ei pääse käsiksi haluamaansa resurssiin, eikä näin ollen voi jatkaa suoritustaan (Oracle-sivusto, 2015e). Tämä voi tapahtua esimerkiksi tilanteessa jossa useat korkean prioriteetin säikeet estävät jatkuvasti alhaisemman prioriteetin säikeen pääsyyn jaettuun resurssiin.

Koska poissulkevia lukkoja käytettäessä useampi säie voi odottaa samanaikaisesti saman resurssin vapautumista, on usein tarpeellista, että eri säikeille asetetaan eritasoiset prioriteetit. Prioriteetilla tarkoitetaan kuinka tärkeää yksittäisen säikeen suorittaminen on järjestelmän toiminnan kannalta. Korkeamman prioriteetin säikeet siis pyritään aina suorittamaan ensin ennen alhaisemman prioriteetin säikeiden suorittamista. On kuitenkin mahdollista, että alhaisemman prioriteetin säie ehtii lukita resurssin käyttöönsä, juuri ennen korkeamman prioriteetin säikeen suorittamisen aloittamista. Tällöin kyseinen alhaisemman prioriteetin säie estää korkeamman prioriteetin säikeen suorituksen aloittamisen, kunnes se vapauttaa varaamansa lukon. Tätä ilmiötä kutsutaan *prioriteetin kääntymiseksi* (priority inversion) (Fraser, 2004.) Jos alhaisemman prioriteetin säikeen suorittamisessa kestää huomattavan kauan, voi tästä seurata huomattavia ongelmia esimerkiksi aikakriittisissä järjestelmissä.

Aiemmin mainittiin, että ohjelmoija ei voi vaikuttaa siihen missä järjestyksessä yksittäisiä säikeitä suoritetaan eteenpäin. Mekanismia joka hallitsee tätä säikeiden suorittamisen edistymistä, kutsutaan aikatauluttajaksi. Aikatauluttaja siis määrää milloin kutakin säiettä suoritetaan hieman eteenpäin. Tämä aikatauluttaja voi kuitenkin muodostaa ongelman poissulkevia lukkoja käytettäessä. Kuten aiemmin mainittiin, on

mahdollista, että useampi säie voi odottaa samanaikaisesti tietyn resurssin vapautumista. Säie joka on lukinnut kyseisen resurssin käyttöönsä voi kuitenkin joutua tietyissä tilanteissa aikatauluttajan pysäyttämäksi, jolloin se antaa suoritusaikaa muille säikeille. Nämä muut säikeet voivat kuitenkin kaikki odottaa kyseisen resurssin vapautumista, jolloin yksikään niistä ei pääse suoriutumaan eteenpäin, kunnes aikatauluttaja antaa jälleen suoritusaikaa resurssin lukinneelle säikeelle (Marshall, 2005.)

Perinteisiin lukkoihin pohjautuva rinnakkaisuuden hallinta tuo siis mukanaan monenlaisia haasteita sekä ongelmia. Tässä tutkielmassa tutustummekin vaihtoehtoiseen tapaan, eli ohjelmalliseen transaktiomuistiin, toteuttaa kyseinen rinnakkaisuuden hallinta. Seuraavaksi käydäänkin läpi transaktioihin liittyvät perusideat ja periaatteet.

1.3 Transaktiot

Transaktiot ovat tuttuja operaatioita tietokantaohjelmistoista. Transaktiolla tarkoitetaan tässä yhteydessä tapahtumaa, jonka tekemät muutokset on mahdollista peruuttaa jolloin kyseiset muutokset eivät näy transaktion ulkopuolelle. Transaktio koostuu siis useista askelista, jotka voivat muuttaa järjestelmän jaettujen resurssien tilaa, mutta nämä resurssien muutokset eivät ole näkyvissä transaktion ulkopuolelle ennen kuin transaktio on suoritettu kokonaan loppuun asti. Transaktion lopussa transaktio voidaan joko käynnistää uudelleen (retry), peruuttaa (abort) tai varmistaa (commit). Uudelleen käynnistämisen tai peruuttamisen tapauksessa transaktion järjestelmän jaettuihin resursseihin tekemiä muutoksia ei tallenneta. Uudelleen käynnistetyn transaktion suoritus aloitetaan uudestaan alusta, ja peruutetun transaktion suoritus keskeytetään. Vain transaktion varmistamisen tapauksessa sen tekemät muutokset tallennetaan järjestelmän jaettuihin resursseihin, jolloin ne ovat näkyvissä koko järjestelmälle. Transaktio varmistetaan vain, jos mikään muu transaktio ei ole muuttanut transaktion käyttämien järjestelmän jaettujen resurssien tilaa sen suorituksen aikana. Transaktion voidaan näin ollen ajatella olevan järjestelmästä eristetty tapahtuma, jonka tekemät muutokset tulevat järjestelmän näkyville yhtä aikaa tiettynä ajan hetkenä.

Tietokantaohjelmistoissa transaktioiden idea perustuu seuraaviin ominaisuuksiin (Harris, Larus ja Rajwar, 2010, luku 1; Marshall, 2005.):

- *Atomisuus* (atomicity) tarkoittaa, että järjestelmän näkökulmasta transaktio suoritetaan aina kokonaisuudessaan tai sitä ei suoriteta ollenkaan. Epäonnistunut transaktio, eli transaktio jota ei suoritettu loppuun asti, ei saa vaikuttaa järjestelmän kokonaistilaan.

- *Eheys* (consistency) tarkoittaa, että transaktioiden käsittelemien resurssien tilat pysyvät eheinä sekä transaktion alussa että sen lopussa, riippumatta siitä, mitä transaktion suorituksen aikana tapahtui. Mitä eheydellä käytännössä tarkoitetaan, riippuu käytettävästä järjestelmästä sekä itse transaktion käsittelemästä resurssista. Esimerkkinä transaktion käsittelemästä resurssista voidaan pitää taulukkoa, johon säilötään 1-2 objektia, joiden arvo voi olla 1-10. Kyseisen taulukon (resurssin) voidaan sanoa pysyvän eheinä, jos taulukossa on aina transaktion suorituksen jälkeen vähintään yksi ja korkeintaan kaksi objektia, joiden arvot ovat välillä 1-10.

- *Eristyneisyys* (isolation) tarkoittaa, että transaktio ei häiritse järjestelmää tai muita transaktiota sen suorituksen aikana. Toisin sanoen mikään transaktioiden käsittelemä resurssi ei näy transaktion ulkopuolelle osittain suoritettuna. Jos transaktiota ei ole suoritettu kokonaan loppuun asti, näkyy muille transaktioille ja järjestelmälle kyseinen resurssi siinä tilassa kuin se oli ennen transaktion

aloittamista, edellyttäen, että mikään muu transaktio ei ole onnistuneesti ehtinyt jo muuttamaan kyseisen resurssin tilaa.

- *Pysyvyys* (durability) tarkoittaa, että varmennettujen transaktioiden resursseihin tekemät muutokset ovat pysyviä ja näkyvissä järjestelmälle sekä muille transaktioille.

1.4 Ohjelmalliset transaktiomuistit

Ohjelmalliset transaktiomuistit ovat vaihtoehtoinen lähestymistapa edellä läpikäydyille perinteisille lukoille. Niiden idea on varsin yksinkertainen siinä mielessä, että ne tarjoavat ohjelmoijalle jonkinasteisen rajapinnan rinnakkaisten luku- ja kirjoitusoperaatioiden suorittamista varten ohjelman jaettuihin resursseihin eli muuttujiin. Ohjelmallinen transaktiomuisti varmistaa, että jaettujen resurssien tilat pysyvät eheinä ohjelman suorituksen aikana. Ohjelmallisia transaktiomuisteja käytettäessä vastuu rinnakkaisuuden hallinnasta siirretään siis ohjelmoijalta käytettävällä ohjelmalliselle transaktiomuistille. Perinteisiä lukkoja käytettäessä ohjelmoijan täytyisi käyttää matalan tason rinnakkaisuudenhallintamekanismeja kuten semaforeja rinnakkaisuuden hallintaan. Korkeamman tason kielissä kuten Javassa ja C++:ssa voidaan käyttää matalan tason rinnakkaisuudenhallintamekanismien pohjalta rakennettuja monitoreja tähän rinnakkaisuuden hallintaan. Joka tapauksessa näitä mekanismeja on hyvin vaikeata käyttää oikein tuntematta täysin kehitettävän ohjelman rakennetta (Harris ja muut, 2010, luku 1.)

Ohjelmallisessa transaktiomuistissa säikeiden tekemät operaatiot jaettuihin resursseihin eli muuttujiin voidaan suorittaa transaktioiden sisäpuolelta. Transaktioiden rakenne vaihtelee hieman riippuen siitä miten ne on toteutettu eri ohjelmallisissa transaktiomuisteissa. Atomisuus varmistaa, että jokainen transaktio suoritetaan onnistuneesti loppuun asti tai mahdollisesti keskeytetään, jos operaatiota ei saada onnistuneesti suoritettua useankaan yrityksen jälkeen. Eristyneisyys varmistaa, että transaktiot tuottavat aina samat tulokset riippumatta kuinka monta rinnakkaista transaktiota ohjelmassa suoritetaan samanaikaisesti. Eheys takaa, että transaktioiden käsittelemien jaettujen resurssien eli muuttujien tilat pysyvät eheinä riippumatta siitä, missä järjestyksessä transaktioita suoritetaan eteenpäin tai mitä transaktioiden suorituksen aikana tapahtuu (uudelleen käynnistys, peruutus tai varmistus) (Harris ja muut, 2010, luku 1.)

Transaktiomuisti voidaan toteuttaa laitteisto- tai ohjelmapohjaisena sekä näiden eriasteisina hybrideinä. Erilaisia ohjelmallisia transaktiomuistitoteutuksia on tänä päivänä valtava määrä eri ohjelmointikielille. Nämä ohjelmalliset transaktiomuistitoteutukset voidaan tavallisesti jakaa kolmeen eri tyyppiin. Erilliseen kirjastoon pohjautuviin ohjelmallisiin transaktiomuisteihin, joissa transaktiot on toteutettu kirjaston tarjoamilla metodeilla. Koodimanipulaation perustuviin ohjelmallisiin transaktiomuisteihin, joissa koodia manipuloidaan jossain vaiheessa koodin kääntämistä tai ajamista siten, että transaktiot saadaan toteutettua. Transaktiomuisti voi olla myös integroitu itse käytettävään ohjelmointikieleen (WikiSTM-sivusto, 2015.) Ohjelmallinen transaktiomuistituki on esimerkiksi integroitu Clojure-ohjelmointikieleen (Emerick, Carper ja Grand, 2012, sivut 159-226). Intel on myös julkaissut prototyyppiversion ohjelmallisesta transaktiomuistikääntäjästä C++-ohjelmointikielelle, jonka kehitys on tosin jo keskeytetty (Geva, 2012).

1.5 Tutkielman tavoite ja rakenne

Tämän tutkielman tavoitteena on selvittää teknologiavertailua käyttämällä ovatko ohjelmalliset transaktiomuistit varteenotettava vaihtoehto perinteiselle lukkoihin perustuvalle rinnakkaisuudenhallinnalle. Tutkimusongelma ja -menetelmä esitellään tarkemmin luvussa 3. Saatujen tulosten perusteella Java-pohjaiset ohjelmalliset transaktiomuistit eivät ole vielä varteenotettava vaihtoehto kuin tietyissä erityisissä tilanteissa, mutta on hyvin todennäköistä, että lähitulevaisuudessa tilanne muuttuu. Yksityiskohtaiset tulokset ja pohdinta esitellään tarkemmin luvuissa 6 ja 7.

Tutkielman rakenne on seuraava. Luvussa 1 tehdään lyhyt johdanto ohjelmallisiin transaktiomuisteihin ja niihin liittyviin tekijöihin. Luvassa 2 suoritetaan kompakti kirjallisuuskatsaus liittyen ohjelmallisiin transaktiomuisteihin ja niiden historiaan. Luvussa 3 esitellään tutkimusongelma, -kysymykset, -menetelmät, rajaukset, synkronointiongelmat sekä käytetyt mittausparametrit. Luvussa 4 tehdään aluksi yleiskatsaus, jonka perusteella tutkielmassa käytetyt Java-pohjaiset ohjelmalliset transaktiomuistit valittiin. Tämän jälkeen tutkielmaan valitut ohjelmalliset transaktiomuistit käydään tarkemmin läpi. Luvussa 5 esitellään luodut ohjelmalliset transaktiomuistitoteutukset ja niiden toiminta. Luvussa 6 esitellään saadut mittaustulokset sekä niiden analyysi. Lopuksi pohdinta ja yhteenveto suoritetaan luvussa 7.

2. Tutkimuksen tausta

Resursseja jaettaessa rinnakkaisuuden hallinta asettaa järjestelmille monenlaisia haasteita sekä rajoitteita. Tässä luvussa suoritetaan kirjallisuuskatsaus, jossa keskitytään lukkoihin ja ohjelmallisiin transaktiomuisteihin perustuvien toteutusten asettamiin haasteisiin, rajoitteisiin sekä näiden eroavaisuuksien tarkasteluun. Kirjallisuuden hakemisessa käytettiin Scopus, Web of Science, ProQuest ja ACM Digital Library tietokantoja sekä Google Scholar työkalua. Hakusanoina käytettiin termejä software, transactional memory, deadlock, parallel computing ja multithreading.

Kuten aiemmin totesimme, transaktiot ovat olleet jo pitkän aikaa tunnettuja operaatioita tietokantaohjelmistoissa. Lomet esitti vuonna 1977, että tietokantojen transaktioiden tapaiset abstraktiot voisivat toimia hyvänä mekanismina eri ohjelmointikielissä jaettujen resurssien eheyden varmistamisessa (Lomet, 1977). Knight esitteli vuonna 1986 alustavan idean laitteistopohjaisen tuen lisäämisestä transaktioihin (Knight, 1986). Herlihy ja Moss esittelivät vuonna 1993 oman ideansa laitteistopohjaisesta transaktiomuistista (Herlihy ja Moss, 1993). Samana vuonna Stone, Heidelberger ja Turek (1993) esittelivät heidän versionsa transaktiomuistista, joka mahdollisti atomisten päivitysten tekemisen ohjelman jaettuihin muuttujiin, sekä yksinkertaisti niihin liittyvää rinnakkaisuuden hallintaa. Shavit ja Touitou esittelivät vuonna 1995 idean puhtaasti koodiin perustuvista ohjelmallisista transaktiomuisteista (Shavit ja Touitou, 1997).

Haskell-ohjelmointikielen kääntäjään GHC:hen (Glasgow Haskell Compiler) ohjelmallinen transaktiomuistituki lisättiin vuonna 2005 julkaistussa versiossa 6.4 (GHC compiler version 6.4, 2005). Vuonna 2007 Sun Microsystems kehitteli Rock-prosessoria, johon oli määrä tulla laitteistopohjainen transaktiomuistituki. Projekti kuitenkin peruutettiin vuonna 2009 (Wiki Rock Processor, 2015.) Vuonna 2007 ilmestyneessä Rick Hickeyn suunnittelemassa Clojure-ohjelmointikielessä ohjelmallinen transaktiomuistituki oli integroituna itse ohjelmointikieleen (WikiClojure-sivusto, 2015). Ensimmäinen kaupallinen järjestelmä, joka sisälsi laitteistopohjaisen transaktiomuistituen, oli vuonna 2011 julkaistu Blue Gene/Q supertietokone. Kyseisessä supertietokoneessa on 16 laskentaydintä, jotka toimivat 1.6GHz:n kellotaajuudella (Wiki Blue Gene, 2015.) Ilmaisella GNU-ohjelmointilisenssillä (General Public License) levitettävään GCC-kääntäjään (GNU Compiler Collection) kokeellinen ohjelmallinen transaktiomuistituki lisättiin vuonna 2012 julkaistussa versiossa 4.7. Kyseinen julkaisu sisälti myös erillisen kirjaston ohjelmallisen transaktiomuistin käyttöä varten (GCC 4.7 Release, 2012.) Vuonna 2012 Intel julkaisi TSX laajennuksen (Transactional Synchronization Extensions), joka lisäsi laitteistopohjaisen transaktiomuistituen x86-arkkitehtuuria käyttäviin Haswell-prosessoreihin. Laajennuksen avulla ohjelmoijat voivat merkitä tietyn osan koodia suoritettavan transaktiona (Wiki TSX, 2015.) Viime vuosiin asti sekä laitteisto-, että ohjelmapohjaisten transaktiomuistien tutkiminen ja kehittäminen on ollut siten hyvin suosittua.

Perinteisesti järjestelmät käyttävät lukkoihin pohjautuvia ratkaisuja järjestelmän jaettujen resurssien rinnakkaisuuden hallinnassa. Lukkoihin perustuvat ratkaisut tuovat kuitenkin mukanaan monia haittapuolia. Lukkoihin perustuvat järjestelmät ovat usein huonosti skaalautuvia sekä ylläpidettäviä. Lukkojen oikean määrän valinta voi olla hyvin haasteellista järjestelmää suunniteltaessa, mikä aiheuttaa rajoitteita sen skaalautuvuudelle ja ylläpidettävyydelle. Lisäksi näitä lukkoja on hallittava

oikeaoppisesti, eli ne täytyy lukita ja avata oikeassa järjestyksessä, jotta lukkiutumia ei pääse tapahtumaan (Weimerskirch, 2008.)

Ohjelmalliset transaktiomuistit tarjoavat monia etuja lukkoihin perustuviin toteutuksiin nähden. Ohjelmoijan ei tarvitse huolehtia erikseen rinnakkaisuudesta koodia kirjoitettaessa, koska käytetty ohjelmallinen transaktiomuistitoteutus hoitaa automaattisesti koodin oikeaoppisen rinnakkaisen suorituksen (Weimerskirch, 2008). Ohjelmoija voi siis kirjoittaa rinnakkain suoritettavan koodiosan samalla tavalla kuin kirjoitettaessa normaalia peräkkäisesti suoritettavaa koodia, ja lopuksi merkitä kyseisen koodiosan suoritettavaksi transaktiona. Lukkoihin pohjautuvassa ratkaisussa ohjelmoijan täytyisi hallita säikeiden pääsyä kyseiseen osaan koodia oikeaoppisesti käyttäen järjestelmän sekä ohjelmointikielen tarjoamia hallintamekanismeja. Tällaiseen ohjelmointiin liittyvä samanaikaisuus ja epädeterministisyys kasvattavat huomattavasti ohjelmoijan työtaakkaa. Tämä johtuu siitä, että ohjelmoijan on kyettävä hahmottamaan ja seuraamaan miten eri lukkomekanismit toimivat keskenään (Harris ja muut, 2010, luku 1).

Toinen transaktiomuistien tarjoama etu liittyy itse transaktioiden toteutusperiaatteeseen. Ohjelmallisia transaktiomuisteja käytettäessä ohjelman muuttujia ajatellaan järjestelmän jaettuina resursseina. Näiden muuttujien arvoja käsitellään ohjelmallisen transaktiomuistin määrittelemässä transaktiossa eli niin sanotussa ohjelman *atomisessa osuudessa* (atomic block). Kun muuttujia käsitellään atomisen osuuden sisäpuolelta käytetty ohjelmallinen transaktiomuisti varmistaa, että konflikteja tai lukkiutumia ei pääse tapahtumaan rinnakkain suoritettavien transaktioiden kesken. Atomisessa osuudessa suoritettavaa koodia suoritetaan niin kauan kunnes se saadaan suoritettua ilman muuttujien välisiä konflikteja. Perinteisissä lukkopohjaisissa ratkaisuissa vastaavia atomisia osuuksia kutsuttaan *kriittisiksi alueiksi* (critical section). Kriittinen alue tarkoittaa sitä osaa koodia, jonne säikeiden pääsyä on rajoitettava ennalta määritellyllä tavalla, jotta konflikteja ei pääse tapahtumaan. Perinteisissä lukkopohjaisissa ratkaisuissa ongelmana on, että jokaiselle kriittiselle alueelle on määriteltävä erikseen mitä muuttujaa tai muuttujajoukkoa se suojaa. Ohjelmallisten transaktiomuistien atomisia osuuksia käytettäessä näitä muuttujia ei tarvitse erikseen määritellä jokaiselle transaktiolle. Tästä seuraa, että transaktioita käyttävät ohjelmat ovat yleensä paremmin koostettavia. Paremmalla *koostettavuudella* (composability) tarkoitetaan, että monia atomisia operaatiota voidaan ajaa rinnakkain ja yhdistellä tuloksen pysyessä atomisena. Perinteisiä lukkoja käytettäessä eri operaatioita on huomattavasti vaikeampi yhdistellä muokkaamatta lukkojen sisäistä rakennetta (Harris ja muut, 2010, luku 2; Jones, 2007, 385-406.)

Paremman koostettavuuden lisäksi ohjelmallisten transaktioiden käyttäminen auttaa ohjelmoijaa välttämään monia lukkojen hallitsemiseen liittyviä ongelmia. Kolmas ohjelmallisten transaktiomuistien tarjoama etu liittyy läheisesti ohjelmoijan tarpeeseen hallita säikeiden pääsyä jaettuihin resursseihin lukkoihin perustuvissa järjestelmissä. Tällöin on mahdollista, että huonosti toteutettu lukkopohjainen ratkaisu voi estää säikeiden suoritusta tarpeettomasti tai jopa lukita tiettyjen säikeiden suorituksen kokonaan. Ohjelmoija voi esimerkiksi sisällyttää ohjelmaan liian vähän lukkoja, jolloin kilpailutilanne voi toteutua ja ohjelmassa tapahtuu konflikti. Liian monien lukkojen käyttäminen voi parhaassa tapauksessa tarpeettomasti estää säikeiden suorittamista, ja pahimmassa tapauksessa johtaa lukkiutumaan. Lukkoja voi myös varata väärässä järjestyksessä, jolloin lukkiutumisien todennäköisyys kasvaa huomattavasti. Virheistä toipuminen voi olla lukkoja käytettäessä hyvin vaikeata, koska ohjelmoijan on taattava, että virheen jälkeen ohjelma on yhä eheässä tilassa. Hukatut säikeiden väliset herätykset (wake-ups) ja virheelliset uudelleen yritykset (erroneous retries) voivat myös aiheuttaa ongelmia rinnakkain suoritettavia säikeitä käytettäessä (Jones, 2007, 385-406.)

Ohjelmallisia transaktiomuisteja käytettäessä edellä mainittuja säikeiden tarpeetonta estämistä tai lukkiutumista ei pääse tapahtumaan (Weimerskirch, 2008.)

Vaikka ohjelmalliset transaktiomuistit estävät lukkiutumien tapahtumisen, eivät ne välttämättä takaa, että järjestelmässä ei voisi esiintyä elolukkoja. Yksi ohjelmallinen transaktiomuisti voi taataa, että elolukkoja ei pääse tapahtumaan, kun taas toinen ohjelmallinen transaktiomuisti voi jättää elolukoista huolehtimisen ohjelmoijan vastuulle (Harris ja muut, 2010, luku 2.) Ohjelmallisissa transaktiomuisteissa elolukkoja ja nälkiintymistä kontrolloidaan niin sanottujen kilpailunhallitsijoiden (contention managers) avulla. Se kuinka kilpailunhallitsijat on toteutettu, vaihtelee riippuen siitä mikä ohjelmallinen transaktiomuisti on kyseessä. Ohjelmallisten transaktiomuistien käyttämät kilpailunhallitsijat voivat erota toisistansa huomattavasti, eikä vielä ole päästy yksimielisyyteen siitä millainen kilpailunhallitsija olisi paras (Ennals, 2006; Scherer III ja Scott, 2004; Spear, Dalessandro, Marathe ja Scott, 2009.) Nälkiintymisen välttämiseksi on esimerkiksi ehdotettu protokollaa, joka antaisi korkeamman prioriteetin sellaisille transaktioille, joita on jouduttu uudelleen yrittämään useita kertoja (Waliullah ja Stenstrom, 2007).

Vaikka ohjelmalliset transaktiomuistit helpottavat rinnakkaisuuden hallinnan toteuttamista järjestelmissä, ne eivät välttämättä ole paras vaihtoehto kaikissa tapauksissa. Ohjelmalliset transaktiomuistit voivat vaikuttaa järjestelmän suorituskykyyn negatiivisesti, jolloin monimutkaisemmat lukkopohjaiset ratkaisut voivat olla tehokkaampia. Tämä tilanne voi esiintyä esimerkiksi tilanteissa joissa transaktioita joudutaan uusimaan moneen kertaan suorituksen aikana (Weimerskirch, 2008.) Transaktioiden liiallinen uusiminen syö järjestelmän laskenta-aikaa tarpeettomasti ja voi tietyissä tilanteissa johtaa ohjelman suorituskyvyn laskemiseen (Spear, Marathe, Scherer III ja Scott, 2006). Vaikka transaktioita ei joutuisikaan uusimaan useita kertoja, ovat ne yleensä keskimäärin raskaampia kuin vastaavat lukkoihin perustuvat ratkaisut. Tämä johtuu siitä, että suoritettaessa luku- ja kirjoitusoperaatioita käyttäen ohjelmallista transaktiomuistia, sen on yleensä suoritettava useampia laskutoimituksia kuin käytettäessä perinteisiä lukkoja. Tämä luonnollisesti vähentää ohjelmallisen transaktiomuistin suorituskykyä. Järjestelmissä, joissa on useita laskentaytimiä, tämä suorituskyvyn väheneminen kuitenkin lieventyy, koska laskutoimituksia jaetaan useiden laskentaytimien kesken. Ohjelmallisilla transaktiomuisteilla voi olla myös yhteensopivuusongelmia kolmannen osapuolten kirjastojen ja valmiiksi käännettyjen ohjelman osien kanssa (Harris ja muut, 2010, luku 4.)

Järjestelmän oikeaoppinen toteuttaminen näin ollen helpottuu käytettäessä ohjelmallisia transaktiomuisteja, mutta kääntöpuolena niiden käyttö voi mahdollisesti heikentää järjestelmän suorituskykyä. Se, kuinka paljon suorituskyky voi laskea, riippuu toteutettavasta järjestelmästä. Tämän arvioiminen voi olla hyvinkin haastavaa, ja koska rinnakkaisuudella haetaan yleensä suorituskyvyn lisäystä, on ohjelmoijan kyettävä määrittelemään järjestelmää suunniteltaessa tarjoaako ohjelmallisten transaktiomuistien käyttö riittävästi etuja lukkopohjaisiin ratkaisuihin verrattuna (Harris ja muut, 2010, luku 1). Toinen rajoite ohjelmallisten transaktiomuistien käytössä on, että kaikkia operaatioita ei voi suorittaa transaktiona. Weimerskirch (2008) mainitsee, että tiettyjä I/O-operaatioita kuten esimerkiksi lokitiedostoihin kirjoittamista ei voi sisällyttää transaktioihin.

Ohjelmalliset transaktiomuistit siis helpottavat rinnakkaisesti suoritettavan koodin kirjoittamista sekä välttävät säikeiden tarpeettoman estämisen ja lukkiutumisen. Tästä huolimatta ohjelmalliset transaktiomuistit eivät missään nimessä ole mikään ihmeratkaisu rinnakkaisessa ohjelmoinnissa. Ohjelmoijan täytyy yhä jakaa tehtävä osiin (transaktioihin), joiden suoritus jaetaan järjestelmän prosessorien kesken. Lisäksi

transaktioiden toteutuksessa voidaan tehdä virheitä samalla tavalla kuin lukkopohjaisissa toteutuksissa. Transaktion toteutuksessa voidaan esimerkiksi tehdä looginen virhe, kuten unohtaa hyväksyä transaktio oikeaan aikaan (Harris ja muut, 2010, luku 1.)

Aiemmin mainittiin, että ohjelmalliset transaktiomuistit eivät välttämättä ole paras vaihtoehto kaikissa tapauksissa. Missä tapauksissa ohjelmallisia transaktiomuisteja siis kannattaisi käyttää? Harris ja muut (2010) sanovat, että ohjelmallisia transaktiomuisteja kannattaisi käyttää niissä tapauksissa, jossa lukkopohjainen toteutusvaihtoehto aiheuttaisi vakavia rajoitteita järjestelmän skaalautuvuudelle. Esimerkkinä näistä tapauksista Harris ja muut (2010) mainitsevat graafien algoritmit sekä Quake-pelipalvelimen transaktiomuistipohjaisen toteutuksen. Graafien algoritmien tapauksessa koko graafi on lukittava yksittäisen säikeen käyttöön, vaikka se muokkaisi vain pientä osaa koko graafista. Vaihtoehtoisesti, jos kyseistä graafia haluaa muokata useampi säie kerrallaan, on ohjelmoijan implementoitava hienorakeinen säikeiden hallintamekanismi jokaiselle yksittäiselle osalle graafia, jota erilliset säikeet haluavat muokata. Ohjelmallista transaktiomuistia käytettäessä kutakin yksittäistä osaa graafista voitaisiin muokata säikeiden suorittamilla transaktiolla välittämättä siitä mitä muut transaktiot parhaillaan tekevät kyseiselle graafille.

Quake-pelipalvelimen lukkopohjaisessa toteutuksessa sen skaalautuvuutta rajoittaa pelin suoritukseen liittyvä simulaatio (Zyulkyarov, Gajinov, Unsal, Cristal, Ayguadé, Harris ja Valero, 2009). Lukkopohjaisen pelipalvelimen täytyy määritellä simuloinnin avulla mitkä pelielementeistä täytyy lukita kun pelaaja tekee toimenpiteen. Tämän simuloinnin suorituksen jälkeen pelipalvelin lukitsee kyseiset pelielementit ja tarkistaa onko kyseinen pelaajan tekemä toimenpide yhä hyväksyttävä, jos on, niin peli suorittaa pelaajaan tekemän toimenpiteen. Ohjelmallista transaktiomuistia käytettäessä edellä mainittua simulointiosuutta ei tarvitse tehdä, mikä tekee pelipalvelimen koodista paljon paremmin skaalautuvan.

Ohjelmalliset transaktiomuistit ovat myös vastaanottaneet huomattavaa kritiikkiä monilta tahoilta. Transaktiomuistit tuovat mukanaan monia ohjelmointiongelmia joita ei esiinny perinteisissä lukkopohjaisissa ratkaisuissa. Nämä ongelmat vaikeuttavat huomattavasti yhtenäisen transaktiosemantiikan luomista, mistä johtuu, että eri transaktiomuistitoteutukset voivat erota hyvin paljon toisistansa. Suurin osa näistä ongelmista liittyy siihen miten transaktiomuistit toimivat olemassa olevien ohjelmointikielten mekanismien kanssa. On haasteellista määritellä miten ohjelmallisen transaktiomuistin tulisi toimia, jos muuttujien arvoja tarvitsisi jostain syystä muuttaa transaktioiden ulkopuolelta. Lisäksi perinteisten lukkojen ja transaktioiden käyttäminen sekaisin on hyvin vaikeata koska transaktiot ovat eristettyjä toisistansa, eli lukkojen tilat eivät näy muille transaktioille. Poikkeusten käsittely transaktioiden sisäpuolella on myös haasteellista, koska poikkeusten käsittelyn oikeata järjestystä on vaikeata taata transaktioiden luonteen takia. On myös ongelmallista määritellä yksiselitteisesti miten ohjelmallisen transaktiomuistin tulisi toimia sellaisten operaatioiden tapauksessa joita ei voida peruuttaa, eikä myöskään näin ollen suorittaa transaktiona. Lopuksi kaikkien transaktioiden suorituksen etenemisen takaaminen on vaikeata kun konflikteja tapahtuu paljon (Cascaval, Blundell, Michael, Cain, Wu, Chiras ja Chatterjee, 2008.)

Muut ohjelmallisten transaktiomuistien käyttöönottoon liittyvistä rajoitteista sekä haasteista liittyvät läheisesti siihen, että monet ohjelmalliset transaktiomuistitoteutukset ovat vielä kehitysasteella. Ensinnäkin, jotta ohjelmallisia transaktiomuistitoteutuksia voitaisiin alkaa käyttää laajemmin, on niiden pystyttävä toimimaan rinnakkain nykyisten lukkopohjaisten järjestelmien kanssa. Ohjelmalliset transaktiomuistit tulevat tuskin lyömään itseänsä läpi, jos niiden käyttöönotto edellyttää aiempien järjestelmien täydellistä korvaamista. Näin ollen jonkin asteinen yhteensopivuus on varmistettava

nykyisten järjestelmien kanssa (Adl-Tabatabai, Kozyrakis ja Saha, 2006.) Toiseksi ohjelmallisia transaktiomuistitoteutuksia tukevia kehitystyökaluja on parannettava, jotta suurempi osa ohjelmistojen kehittäjistä hyväksyisi ohjelmallisten transaktiomuistien käyttöönoton. Rinnakkaisesti toimivat järjestelmät voivat olla hyvin virhealttiita, joten luotettava ja vakaa kehitystyökalu esimerkiksi virheiden etsintään on välttämätön edellytys ohjelmallisten transaktiomuistien käyttöönotolle (Weimerskirch, 2008.)

3. Tutkimusmenetelmä

Tässä gradussa tutkimusmenetelmänä käytetään teknologiavertailua, jossa vertaillaan luotuja ohjelmallisia transaktiomuistitoteutuksia perinteiseen lukkopohjaiseen toteutukseen. Tutkimusmenetelmä muistuttaa kontrolloitua koetta, jossa hallitaan siihen liittyviä tekijöitä sekä muuttujia (Järvinen, 1999, luku 3.1). Tutkimuksen tavoitteena on ensin luoda ratkaisuja erityyppisiin synkronointiongelmiin käyttäen erilaisia Javaan pohjautuvia ohjelmallisia transaktiomuistitoteutuksia. Tämän jälkeen kyseisiä toteutuksia arvioidaan ja vertaillaan käyttäen valittuja mittausparametreja. Myös mahdollisia toteutusten tekemisen aikana ilmi tulleita transaktiomuisteihin liittyviä seikkoja arvioidaan. Tässä luvussa ensin esitellään tutkimusongelma ja tutkimuskysymykset sekä tutkimuksen rajaukset. Tämän jälkeen käytettävät synkronointiongelmat kuvaillaan. Lopuksi esitellään käytettävät mittausparametrit sekä syyt niiden valinnalle.

3.1 Tutkimusongelma ja -kysymykset

Tutkimusongelmana on selvittää onko ohjelmallisten transaktiomuistien käyttö varteenotettava vaihtoehto perinteisille lukkoihin perustuville ratkaisuille.

Tutkimusongelmaa lähdetään tässä tutkielmassa ratkaisemaan luomalla ohjelmallisiin transaktiomuisteihin perustuvia ratkaisuja valittuihin synkronointiongelmiin, valita niiden arviointiin sopivat mittausparametrit, sekä lopuksi vertailla näitä käyttäen saatuja mittaustuloksia. Luotuja ohjelmallisia transaktiomuistiratkaisuja verrataan myös vastaavaan lukkoihin perustuvaan ratkaisuun.

Tutkimuskysymykset ovat seuraavat:

- Mitä etuja/haittoja ohjelmallisella transaktiomuistitoteutuksella on perinteiseen lukkiutumisien hallintaan?
- Onko eri transaktiomuistitoteutuksien suorituskykyjen välillä huomattavia eroja?
- Onko transaktiomuistitoteutuksen käyttöönotto suorituskyvyn kannalta varteenotettava vaihtoehto jo olemassa oleviin sovelluksiin, jotka pohjautuvat perinteiseen lukkiutumisen hallintaan?

3.2 Tutkimuksen rajaukset

Transaktiomuisteista on olemassa laitteistopohjaisia ja ohjelmallisia ratkaisuja, sekä näiden eriasteisia hybridejä. Tässä gradussa keskitytään kuitenkin tarkastelemaan pelkästään ohjelmallisia transaktiomuistitoteutuksia. Tämä tutkimus ei ota huomioon sellaisia ohjelmallisia toteutuksia, jotka perustuvat tavalla tai toisella muokattuun Javan virtuaalikoneeseen (Java Virtual Machine). Toisin sanoen ohjelmallisen transaktiomuistitoteutuksen on toimittava muokkaamattomassa Javan virtuaalikoneympäristössä. Laitteistopohjaiset ja muokattuun Javan virtuaalikoneeseen perustuvat toteutukset on rajattu tämän tutkielman ulkopuolelle, koska niiden käyttäminen vaatisi, että ohjelman suoritusalustaa pitäisi muokata, jotta kyseisen

transaktiomuistitoteutus saataisiin otettua käyttöön. Tutkielmassa ei myöskään huomioida hajautettua laskentaan käyttäviä ohjelmallisia transaktiomuisteja. Tämän lisäksi keskitytään tarkastelemaan vain toteutuksia, jotka ovat toteutettavissa Java-ohjelmointikielellä. Java on suosittu ohjelmointikieli ja sille on olemassa useita erilaisia transaktiomuistitoteutuksia. Lisäksi Java-ohjelmointikieleen pohjautuviin ohjelmallisiin transaktiomuisteihin keskittyminen antaa tälle tutkielmalle selkeän ja helposti ymmärrettävän rajauksen sekä fokuksen.

Tutkimuksen laajuus rajoittuu esiteltyihin synkronointiongelmiin, sekä niihin liittyvien ratkaisujen toteutukseen, analysointiin sekä vertailuun.

3.3 Synkronointiongelmat

Tässä kappaleessa esitellään tarkemmin tutkimuksessa käytettävät synkronointiongelmat, eli tuottaja-kuluttaja- ja lukija-kirjoittaja-ongelma. Tuottaja-kuluttaja-ongelma valittiin tähän tutkielmaan edustamaan tilanteita, joiden ratkaisemiseen käytetään poissulkevia lukkoja. Lukija-kirjoittaja-ongelma edustaa vastaavasti tilanteita, joiden ratkaisemiseen tarvitaan kategorista poissulkevuutta. Käytettävät synkronointiongelmat sekä niiden tarkat kuvaukset on lainattu Downeyn (2005) kirjasta nimeltä “ The Little Book of Semaphores”. Synkronointiongelmia on myös monia muita, kuten esimerkiksi klassinen ruokailevien filosofien ongelma (dining philosophers problem) ja tupakanpolttajien ongelma (cigarette smokers problem), mutta valitut synkronointiongelmat antavat jo riittävän yleiskuvan rinnakkaisuuden hallinnasta tutkielmaa varten.

3.3.1 Tuottaja-kuluttaja-ongelma

Tuottaja-kuluttaja-ongelma (producer-consumer problem) on klassinen esimerkki synkronointiongelmasta, joka esiintyy rinnakkaisissa järjestelmissä. Ongelma koostuu kahdenlaisista säikeistä, tuottajista ja kuluttajista, sekä puskurista jota nämä säikeet käsittelevät. Tuottajasäikeen tehtävänä on lisätä dataa puskuriin eli toisin sanoen tuottaa objekteja siihen. Kuluttajasäikeen tehtävänä on vastaavasti poistaa dataa puskurista eli toisin sanoen kuluttaa objekteja siitä. Puskuri määritellään tässä ongelmassa rajatun kokoiseksi sekä säieturvattomaksi. Säieturvattomuudella tarkoitetaan tässä yhteydessä, että puskuria ei voi muokata useampi säie (tässä tapauksessa tuottaja tai kuluttaja) samanaikaisesti ilman, että suorituksen turvallisuus vaarantuu. Tässä tapauksessa puskurin voidaan sanoa sijaitsevan niin sanotussa kriittisessä alueessa (critical section). Kriittinen alue tarkoittaa sitä osaa koodista, jonne säikeiden pääsyä on ohjattava siten, että kyseisen synkronointiongelman vaatimukset täyttyvät.

Tuottaja-kuluttaja-ongelma voidaan jakaa seuraaviin osiin (Downey, 2005):

- Tuottaja ei saa yrittää lisätä dataa puskuriin, jos puskuri on täynnä. Vajaaseen puskuriin tuottajan täytyy voida onnistuneesti lisätä dataa.

- Kuluttaja ei saa yrittää poistaa dataa puskurista, jos puskuri on tyhjä. Ei-tyhjästä puskurista kuluttajan täytyy voida onnistuneesti poistaa dataa.

- Puskuria saa muokata vain yksi tuottaja (lisätä dataa) tai kuluttaja kerrallaan (poistaa data).

Tuottaja-kuluttaja-ongelma valittiin tähän tutkielmaan, koska se on hyvä esimerkki resurssista eli tässä tapauksessa puskurista, johon pääsyä on perinteisesti hallittu

käyttämällä poissulkevia lukkoja, jolloin vain yksi säie kerrallaan voi siis muokata puskurin sisältöä. Voimme siis arvioida miten eri ohjelmalliset transaktiomuistitoteutukset suoriutuvat poissulkevien lukitusmekanismien korvaamisessa. Kyseinen synkronointiongelma on myös hyvin kirjoitusraskas, koska sekä tuottaja- että kuluttajasäikeet muokkaavat puskuria, eli kirjoittavat siihen. Näin ollen tämän synkronointiongelman avulla voidaan tarkastella miten eri ohjelmalliset transaktiomuistitoteutukset selviytyvät, kun kirjoitusoperaatioita jaettuun resurssiin on paljon. Lopuksi tässä synkronointiongelmassa on määriteltävä miten ohjelmallinen transaktiomuistitoteutus käyttäytyy kun puskuri on täynnä tai tyhjä. Eli miten estetään tuottajaa yrittämästä lisätä objektia täyteen puskuriin tai kuluttajaa yrittämästä lukea objektia tyhjästä puskurista. Näin ollen voimme arvioida minkälaisia mekanismeja yksittäiset ohjelmalliset transaktiomuistit tarjoavat näiden tilanteiden ratkaisemista varten.

3.3.2 Lukija-kirjoittaja-ongelma

Lukija-kirjoittaja-ongelma on toinen klassinen esimerkki synkronointiongelmasta joka esiintyy lukuisissa erilaisissa tietorakenteissa kuten esimerkiksi tietokannoissa ja tiedostojärjestelmissä (Downey, 2005). Ongelmassa kuvaillaan kaksi erityyppistä säiettä. Nämä ovat lukijasäikeet eli ns. lukijat, jotka lukevat jaettua resurssia, sekä kirjoittajasäikeet eli ns. kirjoittajat, jotka vastaavasti kirjoittavat jaettuun resurssin. Kirjoitettava/luettava resurssi määritellään myös tässä tapauksessa säieturvattomaksi. Toisin sanoen kyseistä resurssia voi lukea useampi lukija yhtä aikaa, mutta siihen voi kirjoittaa kerrallaan vain yksi kirjoittaja (jolloin resurssin lukeminen ei ole sallittua). Tässä synkronointiongelmassa kirjoitettavaa/luettavaa resurssia voidaan pitää kriittisenä alueena.

Lukija-kirjoittaja-ongelma voidaan jakaa seuraaviin osiin (Downey, 2005):

- Yksi tai useampi lukija voi lukea jaettua resurssia samaan aikaan.
- Vain yksi kirjoittaja kerrallaan voi muokata jaettua resurssia, jolloin kyseistä resurssia ei saa lukea.

Lukija-kirjoittaja-ongelma eroaa edellisestä tuottaja-kuluttaja-ongelmasta siltä osin, että siinä käytetään kategorista poissulkevuutta (categorical mutual exclusion). Kategorisella poissulkevuudella tarkoitetaan tässä yhteydessä, että lukija- ja kirjoittajasäikeiden pääsyä jaettuun resurssiin hallitaan eri tavalla, riippuen kumpi säie on kyseessä. Perinteisissä toteutuksissa lukija- ja kirjoittajasäikeille käytetään siis erillisiä lukkoja, joita kutsutaan kirjoitus- ja lukemislukoiksi. Lukija-kirjoittajalukon käyttäminen perinteisissä toteutuksissa voi potentiaalisesti parantaa suorituskykyä verrattua pelkän poissulkevan lukon käyttämiseen. Tämä riippuu kuitenkin siitä kuinka usein jaettua resurssia luetaan ja siihen kirjoitetaan, kuinka pitkän aikaa näihin luku- ja kirjoitusoperaatioihin menee, sekä kuinka monta lukija- ja kirjoittajasäiettä yrittää päästä jaettuun resurssin käsiksi samaan aikaan. Koska lukija-kirjoittajalukko on huomattavasti monimutkaisempi kuin pelkkä poissulkeva lukko, on mahdollista, että sitä käytettäessä suorituskyky voi jopa laskea, jos lukuoperaatioiden kesto on hyvin lyhyt (Oracle-sivusto, 2015d.) Näin ollen tämä synkronointiongelma valittiin, jotta voimme arvioida miten eri ohjelmalliset transaktiomuistit selviytyvät tällaisen lukija-kirjoittajalukon korvaamisessa käytettäessä eri määriä lukija- ja kirjoittajasäikeitä sekä luku- ja kirjoitusoperaatioita.

3.4 Mittausparametrien valinta

Tuottaja-kuluttaja-toteutuksien mittauksia varten parametreiksi valittiin:

- tuottajasäikeiden lukumäärä (tuottaja_lkm),
- kuluttajasäikeiden lukumäärä (kuluttaja_lkm),
- käsiteltävien objektien kokonaislukumäärä (obj_lkm),
- puskurin koko (puskuri_kok).

Tuottaja-kuluttaja-toteutuksien mittauksissa mitataan niiden suoritusaikaa (t).

Tuottaja- ja kuluttajasäikeiden lukumäärien suhdetta muuttamalla voidaan mitata miten se vaikuttaa eri toteutusten suoritusaikaan. Käsiteltävien objektien kokonaismäärällä tarkoitetaan kuinka monta objektia yhteensä tuottajasäikeet tuottavat ja kuluttajasäikeet kuluttavat toteutuksen mittauksen aikana. Kokonaislukumäärä asetetaan vakioksi mittausten tekemistä varten niin, että ohjelman suoritukseen kuluva aika on mittausten kannalta sopiva, eli ei siis liian lyhyt tai liian pitkä. Puskurin kokoa voidaan muutella tai se voidaan asettaa vakioksi, riippuen onko koon muuttamisella merkittävää vaikutusta suorituskykyyn.

Suoritusajalla tarkoitetaan aikaa joka kuluu ohjelman suorittamiseen, eli kuinka kauan tiettyjen objektien kokonaismäärän käsittelemiseen menee.

Lukija-kirjoittaja-toteutuksien mittauksia varten parametreiksi valittiin:

- kirjoittajasäikeiden lukumäärä (kirjoittaja_lkm),
- lukijasäikeiden lukumäärä (lukija_lkm),
- kirjoittajasäikeiden tekemien kirjoitusoperaatioiden lukumäärä (kirjoitusten_lkm),
- lukijasäikeiden tekemien lukuoperaatioiden lukumäärä (lukujen_lkm).

Myös lukija-kirjoittaja-toteutuksien mittauksissa mitataan niiden suoritusaikaa (t).

Mittauksessa tarkastellaan miten eri lukija-kirjoittaja-toteutukset suoriutuvat käytettäessä erimääriä lukija- ja kirjoittajasäikeitä, sekä muuttamalla niiden luku- ja kirjoitusoperaatioiden määrä.

4. Ohjelmallisten transaktiomuistitoteutusten valinta

Tässä luvussa käydään ensin läpi yleiskatsaus, jonka perusteella tutkielmassa käytettäviin transaktiomuistitoteutuksiin päädyttiin. Tämän jälkeen valittuihin transaktiomuistitoteutuksiin ja niiden tarjoamiin ominaisuuksiin tutustutaan tarkemmin.

4.1 Yleiskatsaus

Kuten aiemmin mainittiin, tässä tutkimuksessa keskitytään ohjelmallisiin transaktiomuistitoteutuksiin, jotka ovat toteutettavissa Java-ohjelmointikielellä. Alustavan haun jälkeen seuraavat mahdolliset Java-pohjaiset ohjelmalliset transaktiomuistitoteutusvaihtoehdot nousivat esille: AtomJava, JVSTM, Deuce, Multiverse, TMJava sekä ScalaSTM.

Transaktiomuistitoteutusvaihtoehtojen valintakriteerit ovat seuraavat. Ohjelmallinen transaktiomuistitoteutus ei saa olla alustavassa kehitysvaiheessa. Toisin sanoen kyseisestä transaktiomuistitoteutuksesta täytyy olla olemassa toimiva julkaisuversio tai vähintään hyvin lähellä julkaisua oleva versio. Näin voidaan pienentää riskiä, että tehdyt transaktiomuistitoteutukset toimisivat puutteellisesti sen takia, että niitä ei ole vielä optimoitu riittävän hyvin tai ne sisältäisivät kriittisiä virheitä. Kehitysvaiheessa oleva ohjelmallinen transaktiomuisti ei myöskään olisi varteenotettava vaihtoehto lukkopohjaisen toteutuksen korvaamiseen, koska sen toimivuutta ei ole vielä taattu millään tavalla. Lopuksi transaktiomuistitoteutuksen on tarjottava riittävä dokumentaatio koskien kyseisen toteutuksen käyttöönottoa sekä käyttämistä. Muista mahdollisista hylkäysperusteista mainitaan erikseen tässä kappaleessa kyseisen transaktiomuistitoteutusvaihtoehdon läpikäynnin yhteydessä.

AtomJava on Polygot-kehyksen (framework) lisäosa, joka lisää tuen ohjelmallisille transaktioille lähdekoodin kääntämisen yhteydessä. AtomJava -sivuston (2015) mukaan se on kuitenkin kehitysasteella (versio 0.1), jonka takia sitä ei sisällytetä tähän tutkielmaan.

Java Versioned STM (JVSTM) on erilliseen Java-kirjastoon perustuva toteutus ohjelmallisesta transaktiomuistista. Kyseinen toteutus perustuu versioituihin laatikoihin (versioned boxes), jotka ovat transaktioiden käsittelemiä resursseja. JVSTM-sivuston (2015) mukaan se toimii varsin samalla tavalla verrattuna muihin kirjastopohjaisiin ohjelmallisiin transaktiomuistitoteutuksiin, kuitenkin sillä erotuksella, että versioidut laatikot säilövät historiatietoja liittyen niitä muokanneisiin transaktioihin. JVSTM ei kuitenkaan tarjonnut riittävää dokumentaatiota sen käyttöönottoon tai käyttämiseen tämän tutkielman tekemisen ajankohtana, joten sitä ei sisällytetä tähän tutkielmaan.

Deuce STM-sivuston (2015) mukaan Deuce STM on koodimanipulaatioon perustuva toteutus ohjelmallisesta transaktiomuistista. Toisin sanoen tavukoodin ajamisen aikana kyseistä koodia muutetaan niin, että ohjelmallinen transaktiomuistituki saadaan sisälletyksi ajettavaan ohjelmaan. Deuce STM:n viimeisin julkaisuversio on 1.3, joka on julkaistu vuoden 2010 maaliskuussa. Deuce STM-sivusto tarjoaa riittävän

dokumentaation käyttöönotolle sekä käyttämiselle. Näin ollen Deuce valitaan yhdeksi toteutusvaihtoehdoksi tämän tutkielman tekemistä varten.

Multiverse on toinen erilliseen Java-kirjastoon perustuva toteutus ohjelmallisesta transaktiomuistista. Multiverse-sivuston (2015) mukaan Multiversen kaksi päätavoitetta ovat olla kieliriippumaton sekä toimia kehyksenä (framework) muille mahdollisille ohjelmallisille transaktiomuistitoteutuksille. Multiversen viimeisin julkaisu versio on 0.7, joka on julkaistu vuoden 2012 huhtikuussa. Multiverse ei kuitenkaan tarjonnut riittävän selkeätä dokumentaatiota sen käyttöönottoon tai käyttämiseen tämän tutkielman tekemisen ajankohtana, joten sitä ei sisällytetä tähän tutkielmaan.

TMJava on aiemmin käsiteltyyn Deuceen perustuva toteutus ohjelmallisesta transaktiomuistista. TMJava-sivuston (2015) mukaan TMJava on hyvin varhaisessa kehitysasteessa (viimeisin versio on 0.1), lisäksi se perustuu suurelta osin jo toteutusvaihtoehdoksi valittuun Deuceen. Näiden tietojen perusteella TMJavaan pohjautuvaa toteutusta ei sisällytetä tähän tutkielmaan.

ScalaSTM on alun perin Scala-ohjelmointikielelle tehty erilliseen kirjastoon perustuva ohjelmallinen transaktiomuistitoteutus. ScalaSTM-sivuston (2015a) mukaan ScalaSTM tarjoaa kuitenkin varsin kompaktin sovellusrajapinnan (application programming interface) Java-ohjelmointikieltä varten. ScalaSTM:n viimeisin julkaisuversio on 0.7, joka on julkaistu vuoden 2012 joulukuussa. ScalaSTM-sivusto tarjoaa lisäksi riittävän dokumentaation sen käyttöönotolle. Näin ollen ScalaSTM valitaan yhdeksi toteutusvaihtoehdoksi tätä tutkielmaa varten.

Edeltävän yleiskatsauksen perusteella toteutettaviksi ohjelmallisiksi transaktiomuisteiksi valittiin Deuce STM sekä ScalaSTM. Lisäksi vertailun vuoksi tähän tutkielmaan sisällytetään perinteisiin lukkoihin perustuvat toteutukset käytettävistä synkronointiongelmista, jotka on toteutettu käyttäen Javan tarjoamaa java.util.concurrent-pakettia (package). Toteutuksia tullaan siis tässä tutkielmassa tekemään 6 erilaista. Kolme eri toteutusta tuottaja-kuluttaja synkronointiongelmasta, yksi lukkoihin perustuva ja kaksi eri ohjelmallista transaktiomuistitoteutusta (Deuce STM ja ScalaSTM). Vastaavasti lukija-kirjoittaja synkronointiongelmasta tehdään kolme eri toteutusta.

4.2 Deuce STM

Tässä kappaleessa tutustutaan tarkemmin Deuce STM:n toimintaperiaatteeseen, käyttöönottoon sekä muihin sen tarjoamiin ominaisuuksiin.

4.2.1 Toimintaperiaate

Deuce STM:n tavoitteena on ollut rakentaa sellainen ohjelmallinen transaktiomuisti, joka vaatii mahdollisimman vähän muutoksia sekä lisäyksiä toteutettavan ohjelman koodiin. Se ei myöskään vaadi minkäänlaisia muutoksia Javan virtuaalikoneeseen eli ohjelman suoritusalustaan. Ohjelmoijan tarvitsee ainoastaan määritellä mitkä osat suoritettavasta koodista tulisi suorittaa atomisesti eli transaktiona. Tähän määrittelyyn Deuce STM käyttää Java-ohjelmointikielen huomautuksia (annotations). Huomautusten avulla ohjelmoijat voivat merkitä ohjelman käyttämät metodit metadatalla, jota voidaan tarkastella kun ohjelman luokkia ladataan muistiin. Kun ohjelmoija haluaa, että tietty metodi suoritetaan transaktiona, tarvitsee hänen vain merkitä kyseinen metodi Deuce STM:n määrittelemällä tavalla (Korland, Shavit, Felber, 2010.)

Transaktiona suoritettaviksi merkityt metodit instrumentoidaan Deuce STM:ää käytettäessä seuraavalla tavalla. Kaikista transaktiona suoritettavaksi merkityistä metodeista tehdään niin sanotut transaktiokopiot, jotka eroavat alkuperäisistä siten, että metodille suoritettavat kutsut tehdään alkuperäisen metodin sijasta transaktiokopiolle. Jokainen transaktiona suoritettavaksi merkitty metodi korvataan lisäksi uudelleenyritys-silmukalla (retry-loop), jossa metodin transaktiokopiota suoritetaan niin kauan, että se saadaan suoritettua ilman konflikteja (Afek, Korland ja Zilberstein, 2011.)

4.2.2 Käyttöönotto

Deuce STM tarjoaa kaksi erilaista instrumentointimenetelmää sen käyttöönottamiseksi (Korland ja muut, 2010). Nämä ovat reaaliaikainen (online) ja ei-reaaliaikainen (offline) instrumentaatio. Reaaliaikaisen instrumentaation tapauksessa sovellus instrumentoidaan tavukoodin ajon aikana ennen sen lataamista muistiin suoritusta varten. Reaaliaikainen instrumentaatio otetaan käyttöön lisäämällä Javan komentoriville vipu -javaagent:bin/deuceAgent.jar.

Reaaliaikainen instrumentaatio voidaan suorittaa esimerkiksi seuraavalla tavalla:

```
java -javaagent:bin/deuceAgent.jar -cp my.jar myMain
```

Edellä esitetty komento ajaa ohjelman myMain instrumentoiden sen Deucella.

Deuce STM tarjoaa myös ei-reaaliaikaisen vaihtoehdon jonka avulla voidaan luoda valmiiksi instrumentoitu versio käytetystä JAR-tiedostosta. . JAR-tiedosto on Javan arkistointitiedosto (Java Archive File) johon voidaan niputtaa yhteen useita tiedostoja, joista ohjelma koostuu. Ei-reaaliaikaista instrumentointia voidaan käyttää niissä tapauksissa, joissa reaaliaikaisen instrumentoinnin käyttäminen ei ole mahdollista jonkin rajoitteen takia

Ei-reaaliaikainen instrumentaatio voidaan suorittaa esimerkiksi seuraavalla tavalla:

```
java -jar deuceAgent.jar my.jar out_my.jar
```

Tämä komento instrumentoi my.jar tiedoston ja luo uuden valmiiksi instrumentoidun version kyseisestä tiedostosta nimeltä out_my.jar.

4.2.3 Transaktion rakenne

Deuce STM-toteutuksessa transaktio muodostetaan lisäämällä huomautus (annotation) `@Atomic` sellaisten metodien eteen, jotka halutaan suoritettavan transaktiona. Seuraavaksi otamme lyhyen esimerkin tästä.

```
import org.deuce.Atomic;

@Atomic
public static void exampleTransactionMethod(){
        //transaction being run
}
```

Tässä tapauksessa metodin exampleTransactionMethod sisältö suoritetaan siis transaktiona. Transaktion rakenne on varsin yksinkertainen verrattuna Scalan vastaavaan toteutuksen, jonka käymme läpi kappaleessa 4.3.

4.2.4 Transaktion vuonohjaus

Kuten aiemmin totesimme, transaktion keskeisenä ideana on, että sitä ajetaan uudestaan kunnes se saadaan suoritettua onnistuneesti loppuun asti. Käytettävä ohjelmallinen transaktiomuistitoteutus hoitaa yleensä nämä operaatiot automaattisesti. Tietyissä tapauksissa on kuitenkin tarpeellista, että ohjelmoija voi itse vaikuttaa siihen kuinka transaktion kulku etenee. Esimerkiksi määritellä kuinka transaktio toimii poikkeustapauksissa ja erityistilanteissa. Vuonohjauksella tarkoitetaan näin ollen niitä toimintoja, joilla ohjelmoija voi vaikuttaa transaktion kulun etenemiseen sen suorituksen aikana.

Deuce STM tarjoaa kaksi erilaista tapaa ohjelmoijalle hallita transaktioiden kulkua, nämä on toteutettu käyttäen poikkeuksia. Transaktion sisältä voidaan heittää poikkeukset `TransactionException` ja `AbortTransactionException`. Poikkeuksen `TransactionException` heittäminen aiheuttaa, että transaktion tekemät muutokset hylätään, ja transaktion suoritus aloitetaan uudestaan välittömästi. Poikkeuksen `AbortTransactionException` heittäminen puolestaan aiheuttaa, että transaktion tekemät muutokset hylätään, mutta transaktion suoritusta ei uusita. Transaktion suoritus siis keskeytetään.

4.3 ScalaSTM

Tässä kappaleessa tutustutaan tarkemmin ScalaSTM:n toimintaperiaatteeseen, käyttöönottoon sekä muihin sen tarjoamiin ominaisuuksiin.

4.3.1 Toimintaperiaate

ScalaSTM on ohjelmallinen transaktiomuisti, joka on toteutettu erillisenä kirjastona. ScalaSTM:n käyttämiseksi ohjelmoijan on sisällytettävä kyseinen kirjasto ohjelmaansa. Kirjasto tarjoaa tämän jälkeen rajapinnan, jonka avulla ohjelmallisia transaktiomuistiominaisuuksia voidaan käyttää hyväksi. ScalaSTM on alun perin tarkoitettu Scala-ohjelmointikielelle, mutta se tarjoaa myös rajapinnan Java-pohjaiselle toteutukselle.

Käytettäessä ScalaSTM:ää se valvoo, että määriteltyjen atomisten osuuksien sisäpuolella tapahtuvien säikeiden suorittamien luku- ja kirjoitusoperaatioiden välillä ei tapahdu konflikteja. Konfliktin tapauksessa atomisessa osuudessa suoritetut kirjoitusoperaatiot perutaan, ja kyseisen atomisen osuuden suoritusta yritetään uudestaan. Jos konfliktia ei ole tapahtunut, atomisessa osuudessa suoritetut säikeiden tekemät kirjoitusoperaatiot varmistetaan. Muut säikeet näkevät ohjelman suorituksen aikana vain varmistetut kirjoitusoperaatiot (ScalaSTM-sivusto, 2015a.)

4.3.2 Käyttöönotto

Kuten aiemmin mainittiin, ScalaSTM on alun perin Scala-ohjelmointikielelle suunniteltu ohjelmallinen transaktiomuistitoteutus, joka on toteutettu erillisenä kirjastona. Se kuitenkin tarjoaa sovellusrajapinnan (application programming interface) Java-pohjaiselle toteutukselle. ScalaSTM-kirjaston voi ottaa käyttöön lataamalla sen arkistointitiedoston (JAR) ScalaSTM-sivustolta (2015a) ja sisällyttämällä sen toteutettavaan ohjelmointiprojektiin. ScalaSTM:n voi myös ottaa käyttöön Java-

projekteissa käyttämällä avoimeen lähdekoodiin perustuvaa Maven2-rakennustyökalua (build tool).

Arkistointitiedoston ohjelmointiprojektiin sisällyttämisen jälkeen ScalaSTM asettaa kaksi vaatimusta ohjelmoijalle, jotta sen tarjoamia ominaisuuksia voidaan käyttää. Ensinnäkin järjestelmän jaetut resurssit on toteutettava tai korvattava ScalaSTM tarjoamilla transaktioviitteillä (Ref). Toiseksi kyseisiä transaktioviitteitä saa käyttää vain transaktio-operaatioiden, eli atomisten osuuksien (atomic blocks), sisäpuolelta.

4.3.3 Transaktion rakenne

ScalaSTM-toteutuksessa transaktio muodostetaan STM-luokan metodilla `atomic()`, joka ottaa argumenttina Javan Runnable- tai Callable-luokalla tehdyn olion. Seuraavaksi otamme lyhyet esimerkit näiden käytöstä.

```
import scala.concurrent.stm.japi.STM;

STM.atomic(new Runnable() {
        public void run() {
                //transaction being run
        }
});
```

Tämä koodi luo uuden transaktion Runnable-luokasta tehdyn olion pohjalta. Metodi `run()` muodostaa tämän transaktion atomisen osuuden.

```
import java.util.concurrent.Callable;
import scala.concurrent.stm.japi.STM;

boolean returnValue = STM.atomic(new Callable<Boolean>() {
        public Boolean call() {
                //transaction being run
                return true;
        }
});
```

Tämä koodi luo uuden transaktion Callable-luokasta tehdyn olion pohjalta. Metodi `call()` muodostaa tämän transaktion atomisen osuuden. Erona edelliseen Runnable-toteutukseen on, että tämä transaktio pystyy palauttamaan tuloksen transaktion suorituksen lopuksi. Tässä esimerkissä palautettava arvo on tyyppiä Boolean.

4.3.4 Transaktioiden käyttämien transaktioviitteiden rakenne

ScalaSTM:n transaktioiden käyttämiseksi ohjelman jaetut resurssit eli muuttujat täytyy korvata niitä vastaavilla transaktioviitteillä. Scalassa transaktioviite voidaan luoda ja sitä voidaan operoida esimerkiksi seuraavalla tavalla.

```
private final Ref.View<Integer> variable = STM.newRef(0);

public void readWriteMethod(){
        STM.atomic(new Runnable() {
                public void run() {
                        variable.set(1);
                        variable.get();
                }
        });
}
```

Tässä koodiesimerkissä luodaan transaktioviite *variable* kokonaislukumuuttujan (Integer) pohjalta käyttämällä luokan STM metodia `newRef()`. Säikeen käyttämässä metodissa `readWriteMethod()` olevassa transaktiossa kyseistä transaktioviitettä *variable* voidaan operoida. Transaktioviitteeseen *variable* voidaan kirjoittaa käyttämällä metodia `set()`, ja sitä voidaan lukea käyttämällä metodia `get()`. Ohjelmoijan ei siis tarvitse erikseen hallita useiden säikeiden pääsyä kyseiseen metodiin `readWriteMethod()`, vaan ScalaSTM hoitaa tähän liittyvän rinnakkaisuuden hallinnan.

4.3.5 Transaktion vuonohjaus

Vuonohjauksella tarkoitetaan niitä toimintoja, joilla ohjelmoija voi vaikuttaa transaktion kulun etenemiseen sen suorituksen aikana. ScalaSTM tarjoaa kaksi eri metodia vuonohjausta varten, nämä ovat `retry()` sekä `retryFor()`. Metodin `retry()` kutsuminen transaktion suorituksen aikana aiheuttaa transaktion suorituksen peruuttamisen (palaamisen alkuun), jonka jälkeen transaktio jää odottamaan, että yksi transaktion käsittelemistä transaktioviitteistä muuttuu. Metodi `retryFor()` toimii kuin metodi `retry()` kuitenkin sillä erotuksella, että ohjelmoija voi rajoittaa kuinka kauan transaktio korkeintaan odottaa transaktion käsittelemien transaktioviitteiden muuttumista.

5. Ohjelmalliset transaktiomuistitoteutukset

Tässä luvussa esitellään tehdyt transaktiomuistitoteutukset sekä niiden toiminta. Aiemmin esitellyt synkronointiongelmat pyritään siis ratkaisemaan käyttämällä eri transaktiomuistitoteutusten tarjoamia ominaisuuksia, joissa ei käytetä perinteiseen rinnakkaisuuden hallintaan pohjautuvia menetelmiä. Näihin menetelmiin voidaan laskea kuuluvaksi esimerkiksi semaforit (semaphores) ja monitorit (monitors), kuten esimerkiksi Javan synkronoidut metodit (synchronized methods). Toteutusten vertailua varten kustakin synkronointiongelmasta tehdään myös perinteisiin lukkoihin pohjautuvat toteutukset käyttäen Javan java.util.concurrent-pakettia.

Toteutusten läpikäynnissä esitellään vain toteutusten keskeisimmät osat. Toteutusten täydet lähdekoodit ovat saatavilla osoitteesta:

http://goo.gl/RHRfOY

5.1 Tuottaja-kuluttaja-toteutukset

Tuottaja-kuluttaja-toteutuksissa luodaan eri määriä tuottaja- ja kuluttajasäikeitä. Toteutus tuottaa aina äärellisen kokonaismäärän objekteja jaettuun puskuriin, objektien tuottaminen jaetaan tasaisesti tuottajasäikeiden kesken. Kun tuottajasäikeet ovat tuottaneet kaikki objektinsa, ne sulkevat itsensä. Kuluttajasäikeet kuluttavat objekteja puskurista kunnes puskuri on tyhjä ja kaikki tuottajasäikeet ovat sulkeneet itsensä, jonka jälkeen myös ne sulkevat itsensä. Tämän jälkeen ohjelman suoritus päättyy.

Otetaan esimerkkinä tapaus jossa on 5 tuottaja- ja kuluttajasäiettä, ja tuotettavien objektien kokonaismäärä on 100. Tällöin jokainen tuottajasäie tuottaa 20 objektia jaettuun puskuriin, jonka jälkeen ne sulkevat itsensä. Kuluttajasäikeet kuluttavat objekteja puskurista kunnes ne ovat kuluttaneet yhteensä 100 objektia, eli kunnes kaikki tuottajat ovat sulkeneet itsensä ja jaettu puskuri on tyhjä. Yksittäisten kuluttajien kuluttama objektien lukumäärä voi näin ollen vaihdella, mutta niiden yhteismäärä on aina sama. Kuluttajien kuluttama lukumäärä voi esimerkiksi olla $k_1(25) + k_2(15) + k_3(30) + k_4(10) + k_5(20) = k_{kok}(100)$, missä k_i on yksittäisen kuluttajan i kuluttama määrä, ja k_{kok} kulutettujen objektien yhteismäärä.

5.1.1 Lukkoihin perustuva toteutus

Lukkoihin perustuva toteutus on tehty käyttäen Javan java.util.concurrent-paketin tarjoamaa rajapintaa BlockingQueue sekä luokkaa ArrayBlockingQueue. Tässä kappaleessa esitellään vain toteutuksen keskeisimmät osat.

Lukkoihin perustuvassa toteutuksessa puskuri on tehty seuraavalla tavalla.

```
public final int CAPACITY = 10;
private BlockingQueue<CustomObject> que = new
ArrayBlockingQueue<CustomObject>(CAPACITY);
```

Puskurina toimii siis luokan ArrayBlockingQueue pohjalta luotu olio *que*, johon voidaan säilöä puskurin kapasiteetin verran luokan CustomObject-tyyppisiä olioita. Olio

que on äärellisen kokoinen syklinen puskuri, johon lisättävät CustomObject-oliot järjestetään FIFO-periaatteella (First-In First-Out). CustomObject-olioita lisätään kyseisen puskurin häntään (tail) ja niitä poistetaan puskurin päästä (head). Puskurin hännässä sijaitsee aina viimeisin puskuriin lisätty CustomObject-olio, eli olio joka on ollut puskurissa lyhyimmän ajan. Puskurin päässä sijaitsee aina seuraavaksi poistettava CustomObject-olio, eli olio joka on ollut puskurissa pisimmän ajan (Oracle-sivusto, 2015a.)

Tuottajasäikeet lisäävät objekteja puskuriin käyttäen metodia `putToBuffer()`.

```
public void putToBuffer(final CustomObject obj){
        try {
                que.put(obj);
        } catch (InterruptedException e) {
                e.printStackTrace();
        }

}
```

Metodi `que.put()` lisää tuottajan tarjoaman objektin puskuriin, jos puskurissa on tilaa. Siinä tapauksessa, että puskuri on täynnä, kyseinen metodi odottaa, että puskuriin vapautuu tilaa, jonka jälkeen se lisää objektin puskuriin. Metodi `que.put()` heittää poikkeuksen `InterruptedException()`, jos täyttä puskuria odottava tuottajasäie keskeytetään sen odotuksen aikana. Koska tässä toteutuksessa kuluttajasäikeet sulkevat itsensä vasta sen jälkeen kun kaikki tuottajasäikeet ovat sulkeneet itsensä, ei tässä toteutuksessa ole kuitenkaan tarvetta keskeytellä tuottajasäikeitä normaalin suorituksen aikana.

Kuluttajasäikeet kuluttavat objekteja puskurista käyttäen metodia `removeFromBuffer()`.

```
public CustomObject removeFromBuffer(){
        try {
                return que.take();
        } catch (InterruptedException e) {
                return null;
        }
}
```

Metodi `que.take()` ottaa objektin puskurista. Jos puskuri on tyhjä, metodi odottaa, että objekti tulee saataville puskuriin. Myös tämä metodi heittää poikkeuksen `InterruptedException()`, jos kuluttajasäie keskeytetään, kun se odottaa tyhjää puskuria. Kun tämä keskeytys tapahtuu, palauttaa metodi `removeFromBuffer()` null-osoittimen, mikä kertoo kuluttajasäikeelle, että se voi sulkea itsensä. Metodi `que.take()` keskeytetään vain siinä tapauksessa, että kaikki tuottajasäikeet ovat sulkeneet itsensä ja puskuri on tyhjä.

Kuluttajasäikeet keskeytetään käyttäen metodia `setallProducersFinished()`, sen jälkeen kun kaikki tuottajasäikeet ovat sulkeneet itsensä. Kyseistä metodia kutsutaan vain kerran ohjelman suorituksen aikana, ja sen tehtävänä on ilmoittaa kuluttajasäikeille, että ne voivat sulkea itsensä sen jälkeen kun puskuri on tyhjä.

```
public static void setallProducersFinished(Thread[] consumerArray){
        while(true){
                if(que.size() == 0)
                        break;
        }

        int i = 0;
        while(i < consumerArray.length){
                if(consumerArray[i].getState() ==
Thread.State.WAITING){
                        consumerArray[i].interrupt();
                        i++;
                }
        }
}
```

Ensimmäisessä while-silmukassa odotetaan, että puskuri on tyhjä, eli kuluttajasäikeet ovat ehtineet kuluttaa kaikki objektit puskurista. Taulukko consumerArray sisältää osoittimet kaikkiin kuluttajasäikeisiin, joita käytetään toisessa while-silmukassa. Toisessa while-silmukassa käydään läpi yksitellen kaikki kuluttajasäikeet. Jos kuluttajasäie odottaa tyhjää puskuria, se keskeytetään, ja siirrytään seuraavaan kuluttajasäikeeseen kunnes kaikki kuluttajasäikeet on keskeytetty.

Paketin java.util.concurrent tarjoama rajapinta BlockingQueue on tarkoitettu erityisesti tuottaja-kuluttaja-tyyppisten synkronointiongelmien ratkaisemiseen, kun käytetään Java-ohjelmointikieltä. Näin ollen tämä toteutus toimii mainiona vertailukohtana ohjelmallisiin transaktiomuistitoteutuksiin niiden suorituskykyä vertaillessa.

5.1.2 ScalaSTM-toteutus

Tässä kappaleessa esitellään ScalaSTM-toteutuksen keskeisimmät osat, eli kuinka puskuri ja transaktiot ovat toteutettu, sekä kuinka ne toimivat.

ScalaSTM-toteutuksessa puskuri on tehty seuraavalla tavalla.

```
public final int CAPACITY = 10;
private TArray.View<CustomObject> que = STM.newTArray(CAPACITY);
private Ref.View<Integer> front = STM.newRef(0);
private Ref.View<Integer> back = STM.newRef(0);
private Ref.View<Boolean> allProducersFinished = STM.newRef(false);
```

Kuten aiemmin mainittiin, jotta ScalaSTM-toteutuksen tarjoamia ominaisuuksia voidaan käyttää, on transaktioiden käsittelemät resurssit eli muuttujat korvattavat niitä vastaavilla transaktioviitteillä. Puskurina toimii luokan TArray pohjalta luotu *que*, joka pitää sisällään puskurin kapasiteetin verran CustomObject-transaktioviitteitä. Luokkaa TArray voidaan ajatella taulukkona, joka pitää sisällään rajatun lukumäärä tietyn tyyppisiä transaktioviitteitä. Transaktioviitteitä *front* ja *back*, jotka on luotu kokonaislukumuuttujien (Integer) pohjalta, käytetään seuraamaan puskurin tämän hetkistä tilaa. Transaktioviite *back* osoittaa aina seuraavaan puskurin vapaaseen paikkaan, johon uusi CustomObject-olio voidaan lisätä. Transaktioviite *front* puolestaan osoittaa aina siihen puskuriin paikkaan, jossa sijaitsee seuraava puskurista poistettava CustomObject-olio. Olio *que* on siis syklinen puskuri, johon CustomObject-olioita lisätään transaktioviitteen back osoittamaan paikkaan, ja poistetaan transaktioviitteen front osoittamasta paikasta. Transaktioviitettä *allProducersFinished*, joka on luotu totuusarvomuuttujan (Boolean) pohjalta, käytetään ilmoittamaan kuluttajasäikeille koska ne voivat sulkea itsensä.

Puskurin erityistilat, eli tilat jolloin se on täynnä tai tyhjä, määritellään käyttäen seuraavia metodeja.

```
private boolean bufferIsFull() {
        int difference = back.get() - front.get();
        if(difference == -1 || difference == (CAPACITY-1))
                return true;
        else
                return false;
}

private boolean bufferIsEmpty() {
        if(front.get() == back.get())
                return true;
        else
                return false;
}
```

Puskuri on täynnä silloin kun transaktioviitteiden *back* ja *front* välinen erotus on joko -1 tai CAPACITY-1. Tilanteessa, jossa erotus on -1, transaktioviite *back* on saavuttanut transaktioviitteen *front* (esimerkiksi: *back* = 2, *front*=3, *back-front*=2-3=-1). Erityistilanteessa jossa erotus on CAPACITY-1, transaktioviite *back* osoittaa puskurin viimeisimpään paikkaan ja transaktioviite *front* ensimmäiseen paikkaan (esimerkiksi: CAPACITY=10, *back*=9, *front*=0, *back-front*=9-0=CAPACITY-1). Puskuri on tyhjä silloin kun transaktioviitteiden *front* ja *back* arvot ovat samat, eli puskuriin ei ole lisätty yhtään CustomObject oliota.

Tässä toteutuksessa tuottajasäikeet lisäävät objekteja puskuriin käyttäen metodia `putToBuffer()`.

```
public void putToBuffer(final CustomObject obj) {
        STM.atomic(new Runnable() {
                public void run() {
                        if(bufferIsFull())
                                STM.retry();
                        else {
                                que.update(back.get(), obj);
                                back.set((back.get() + 1) % CAPACITY);
                        }
                }
        });
}
```

Kyseisen metodin `putToBuffer()` transaktio toimii pääpiirteittäin seuraavalla tavalla. Transaktio suoritetaan onnistuneesti loppuun asti vain, jos se saa lisättyä tuottajan tarjoaman objektin puskuriin. Muussa tapauksessa transaktio käynnistetään uudelleen kunnes se saadaan suoritettua onnistuneesti loppuun asti.

Tällä transaktiolla on kaksi vaihtoehtoista suoritusreittiä. Ensimmäinen suoritusreitti toteutuu jos puskuri ei ole täynnä. Tällöin transaktio ensin lisää tuottajasäikeen tarjoaman objektin transaktioviitteen *back* osoittamaan puskurin paikkaan. Tämän jälkeen transaktio kasvattaa transaktioviitteen *back* arvoa yhdellä osoittamaan seuraavaan puskurin vapaaseen paikkaan, jonka jälkeen transaktio on suoritettu onnistuneesti loppuun asti. Tämä olettaen, että mikään muu transaktio ei ole ehtinyt jo muokkaamaan transaktioviitteiden *front* ja *back* arvoja, eli joko lisännyt objektin tai poistanut objektin puskurista. Jos toinen transaktio on jo ehtinyt lisätä tai poistaa objektin puskurista, hylätään transaktion tekemät muutokset, ja transaktion suoritus aloitetaan alusta.

Toinen vaihtoehtoinen suoritusreitti toteutuu jos puskuri on täynnä, eli metodi `bufferIsFull()` palauttaa arvon tosi. Tällöin transaktio kutsuu metodia `STM.retry()`. Tämä aiheuttaa sen, että transaktion tekemät muutokset perutaan ja sen suoritus palautuu alkuun, ja transaktio jää odottamaan sen käsittelemien transaktioviitteiden muuttumista, jonka jälkeen transaktiota yritetään suorittaa uudestaan alusta. Tässä toteutuksessa transaktio ei tee muutoksia joita tarvittaisiin perua, mutta se lukee transaktioviitteiden front ja back arvot metodissa `bufferIsFull()`. Näin ollen transaktio jää siis odottamaan kunnes toinen transaktio muuttaa transaktioviitteen front tai back arvoa, jonka jälkeen transaktion suoritusta yritetään uudestaan.

Kuluttajasäikeet kuluttavat objekteja puskurista käyttäen metodia `removeFromBuffer()`.

```
public CustomObject removeFromBuffer(){
        CustomObject objectTakenFromQue = STM.atomic(new
        Callable<CustomObject>() {
                public CustomObject call() {
                        CustomObject obj=null;
                        if(allProducersFinished.get() == true)
                                return null;

                        if(bufferIsEmpty())
                                STM.retry();
                        else {
                                obj = que.apply(front.get());
                                que.update(front.get(), null);
                                front.set((front.get() + 1) %
                                CAPACITY);
                        }
                        return obj;
                }
        });
        return objectTakenFromQue;
}
```

Tämä metodin `removeFromBuffer()` transaktio toimii pääpiirteittäin seuraavalla tavalla. Kun transaktio suoritetaan onnistuneesti loppuun asti, palauttaa se puskurista otetun objektin. Muussa tapauksessa transaktiota yritetään uudestaan kunnes transaktio saa onnistuneesti otettua objektin puskurista. Poikkeuksena tähän on tilanne, jossa kaikki tuottajasäikeet ovat sulkeneet itsensä ja puskuri on tyhjä. Tällöin transaktio palauttaa null-osoittimen, mikä kertoo kuluttajasäikeelle, että se voi sulkea itsensä.

Tällä transaktiolla on kolme vaihtoehtoista suoritusreittiä. Ensimmäinen suoritusreitti toteutuu jos transaktioviitteen *allProducersFinished* arvo on tosi. Tällöin transaktio palauttaa null-osoittimen, mikä kertoo kuluttajasäikeelle, että se voi sulkea itsensä. Transaktioviite *allProducersFinished* asetetaan todeksi vain silloin, kun kaikki tuottajasäikeet ovat sulkeneet itsensä ja puskuri on tyhjä.

Toinen suoritusreitti toteutuu jos puskuri ei ole tyhjä. Tällöin transaktio ottaa objektin tämän hetkisestä taulukon *que* paikasta, ja kasvattaa transaktioviitteen *front* arvoa osoittamaan seuraavaan otettavaan objektiin. Lopuksi transaktio palauttaa puskurista otetun objektin, jolloin transaktio on suoritettu onnistuneesti loppuun asti. Kuten aiemmin, tämä pätee vain siinä tapauksessa, että mikään muut transaktio ei ole jo ehtinyt muokkaaman transaktioviitteiden *front ja back* arvoja. Muussa tapauksessa transaktion suoritus aloitetaan alusta.

Kolmas vaihtoehtoinen suoritusreitti toteutuu, jos puskuri on tyhjä, mutta kaikki tuottajasäikeet eivät ole vielä sulkeneet itseänsä, eli transaktioviitteen

allProducersFinished arvo on epätosi. Tällöin kutsutaan jälleen metodia `STM.retry()`, joka aiheuttaa sen, että transaktion tekemät muutokset hylätään ja transaktio jää odottamaan transaktioviitteiden *front* ja *back* arvojen muuttumista, koska ne luetaan metodissa `bufferIsEmpty()`.

Kuluttajasäikeen transaktiossa näimme transaktioviitteen *allProducersFinished*, tämä transaktioviite päivitetään käyttäen metodia `setallProducersFinished()`.

```
public void setallProducersFinished(){
        STM.atomic(new Runnable() {
                public void run() {
                        if(bufferIsEmpty())
                                allProducersFinished.set(true);
                        else
                                STM.retry();
                }
        });
}
```

Metodia `setallProducersFinished()` kutsutaan, kun kaikki tuottajasäikeet ovat sulkeneet itsensä. Metodin sisältämää transaktiota suoritetaan kunnes puskuri on tyhjä (metodi bufferIsEmpty palauttaa arvon tosi), jolloin se asettaa transaktioviitteen *allProducersFinished* todeksi, jonka jälkeen transaktio on suoritettu onnistuneesti loppuun asti. Jos puskuri ei ole tyhjä, kutsutaan jälleen metodia `STM.retry()`. Tällöin jää transaktio odottamaan transaktioviitteiden *front* ja *back* muuttumista, koska niiden arvot luetaan metodissa `bufferIsEmpty()`, jonka jälkeen transaktion suoritus käynnistetään uudelleen alusta.

ScalaSTM-toteutus täyttää näin ollen kaikki tuottaja-kuluttaja -synkronointiongelman vaatimukset. Tuottajasäikeet eivät yritä lisätä objekteja täyteen puskuriin, eivätkä kuluttajasäikeet yritä poistaa objekteja tyhjästä puskurista. Puskuria voi lisäksi muokata vain yksi kuluttaja tai tuottaja kerrallaan, jolloin muiden mahdollisten säikeiden tekemät muutokset hylätään. Tätä ScalaSTM-toteutusta voidaan pitää osittain estävänä (partially blocking), koska niissä tapauksissa joissa puskuri on joko tyhjä tai täynnä, säikeiden suorittama transaktio jää odottamaan sen lukemien transaktioviitteiden muuttumista, eikä jatkuvasti yritä suorittaa itseänsä uudestaan.

5.1.3 Deuce STM-toteutus

Deuce STM-toteutus on tehty pääpiirteittäin samalla periaatteella kuin ScalaSTM-toteutus. Näin ollen jokaista tämän toteutuksen osiota ei käydä läpi yhtä yksityiskohtaisesti kuin ScalaSTM-toteutuksesta. Deuce STM-toteutuksella on kuitenkin useita eroavaisuuksia Scalan vastaavaan toteutukseen ja näihin eroavaisuuksiin kiinnitetään erityistä huomiota tässä kappaleessa.

Deuce STM-toteutuksen puskuri on tehty seuraavalla tavalla.

```
public final int CAPACITY = 10;
private CustomObject[] que = new CustomObject[CAPACITY];
private int front = 0;
private int back = 0;
private Boolean allProducersFinished = false;
```

Voimme huomata, että tässä toteutuksessa jaettuja resursseja, eli puskuria *que* ja muuttujia *front*, *back* sekä *allProducersFinished*, ei tarvitse korvata erikseen transaktioviitteillä. Näin ollen voimme käyttää puskurin toteutuksen Javan

peruselementtejä kuten taulukkoja ja kokonaislukumuuttujia. Puskurina toimii taulukko *que*, joka pitää sisällään puskurin kapasiteetin verran luokan CustomObject-olioita. Samalla tavalla kuin vastaavassa ScalaSTM-toteutuksessa, kokonaislukumuuttujia *front* ja *back* käytetään seuraamaan puskurin tämän hetkistä tilaa.

Puskurin erityistilat, eli tilat jolloin se on täynnä tai tyhjä, määritellään käyttäen seuraavia metodeja, joiden toimintaperiaate on sama kuin ScalaSTM-toteutuksessa. Metodien rakenne kuitenkin hieman eroaa ScalaSTM-toteutuksessa käytetyistä vastaavista metodeista.

```
private boolean bufferIsFull() {
        int difference = back - front;
        if(difference == -1 || difference == (CAPACITY-1))
                return true;
        else
                return false;
}

private boolean bufferIsEmpty() {
        if(front == back)
                return true;
        else
                return false;
}
```

Erona vastaavaan ScalaSTM-toteutukseen on siis, että muuttujien *front* ja *back* arvot voidaan lukea suoraan, koska niitä ei tarvitse korvata transaktioviitteillä tässä toteutuksessa.

Deuce STM-toteutuksessa tuottajasäikeet lisäävät objekteja puskuriin käyttäen metodia `putToBuffer()`.

```
@Atomic
public void putToBuffer(final CustomObject obj){
        if(bufferIsFull())
                throw new TransactionException();
        else {
                que[back] = obj;
                back = (back + 1) % CAPACITY;
        }
}
```

Metodin `putToBuffer()` transaktio toimii samalla periaatteella kuin ScalaSTM-toteutuksen vastaava transaktio. Transaktio suoritetaan siis onnistuneesti loppuun asti vain siinä tapauksessa, että se saa onnistuneesti lisättyä tuottajasäikeen tarjoaman objektin puskuriin, muussa tapauksessa transaktiota suoritetaan uudelleen kunnes se saadaan suoritettua onnistuneesti loppuun asti. Erona ScalaSTM-toteutukseen on jälleen se, että muuttujien arvoja voidaan käsitellä suoraan, koska niitä ei tarvitse korvata transaktioviitteillä. Voimme lisäksi huomata, että tässä toteutuksessa meidän ei tarvitse luoda transaktiota luokkien Runnable tai Callable pohjalta luotujen olioiden avulla. Vaan riittää, että lisäämme huomautuksen (annotation) `@Atomic` metodin `putToBuffer()` alkuun osoittamaan, että sen sisältö suoritetaan transaktiona.

Tällä transaktiolla on kaksi mahdollista suoritusreittiä samalla tavalla kuin ScalaSTM-toteutuksessa. Ensimmäinen suoritusreitti, jolloin puskuri ei ole täynnä, toimii samalla tavalla kuin ScalaSTM:n vastaavassa toteutuksessa. Toinen mahdollinen suoritusreitti, jolloin puskuri on täynnä, eroaa kuitenkin ScalaSTM-toteutuksesta. Jos puskuri on täynnä, heittää metodi `putToBuffer()` transaktiopoikkeuksen

`TransactionException()`. Kyseinen poikkeus on DeuceSTM:n tapa hallita transaktion suorituksen etenemistä eli ns. vuonohjausta. Se aiheuttaa, että transaktion tekemät muutokset hylätään ja transaktion suoritus aloitetaan uudestaan välittömästi. Transaktio ei siis jää ScalaSTM-toteutuksen tavoin odottamaan kokonaislukumuuttujien *front* ja *back* muuttumista, vaan käynnistää transaktion suorituksen välittömästi uudestaan.

ScalaSTM-toteutuksen tavoin Deuce STM-toteutuksessa kuluttajasäikeet kuluttavat objekteja puskurista käyttäen metodia `removeFromBuffer()`.

```
@Atomic
public CustomObject removeFromBuffer(){
        CustomObject obj=null;
        if(allProducersFinished == true)
                return null;

        if(bufferIsEmpty())
                throw new TransactionException();
        else {
                obj = que[front];
                que[front] = null;
                front = (front + 1) % CAPACITY;
        }
        return obj;
}
```

Kuten edellisen tuottajatransaktion tapauksessa, tämän metodin `removeFromBuffer()` transaktion toimintaperiaate on sama kuin vastaavassa ScalaSTM-toteutuksessa. Transaktio suoritetaan siis onnistuneesti loppuun asti vain jos se saa otettua objektin puskurista. Muussa tapauksessa transaktiota yritetään uudelleen kunnes se saadaan onnistuneesti suoritettua. Poikkeuksena ollen tietenkin tilanne jossa puskuri on tyhjä ja kaikki tuottajasäikeet ovat sulkeneet itsensä eli muuttuja allProducersFinished arvo on tosi, jolloin transaktio palauttaa null-osoittimen.

Transaktion ensimmäinen ja toinen suoritusreitti ovat samat kuin vastaavassa ScalaSTM-toteutuksessa. Kolmas mahdollinen suoritusreitti, jolloin puskuri on tyhjä ja kaikki tuottajasäikeet eivät ole vielä sulkeneet itseänsä, eroaa vastaavasta ScalaSTM-toteutuksesta myös siltä osin, että transaktio aloittaa suorituksensa välittömästi uudestaan, kun poikkeus TransactionException() heitetään, eikä jää vastaavan ScalaSTM-toteutuksen tavoin odottamaan kokonaislukumuuttujien *front* ja *back* arvojen muuttumista.

Muuttuja *allProducersFinished* päivitetään tässä toteutuksessa käyttäen seuraavaa metodia `setallProducersFinished()`.

```
@Atomic
public void setallProducersFinished(){
        if(bufferIsEmpty())
                allProducersFinished = true;
        else
                throw new TransactionException();
}
```

Transaktion toimintaperiaate on jälleen sama kuin vastaavalla ScalaSTM-toteutuksella. Erona toteutuksessa on jälleen, että transaktio ei jää odottamaan metodissa `bufferIsEmpty()` luettujen kokonaislukumuuttujien *front* ja *back* arvojen muuttumista, vaan aloittaa suorituksen välittömästi uudestaan, jos puskuri ei ole tyhjä.

Deuce STM-toteutus täyttää ScalaSTM-toteutuksen tavoin kaikki tuottaja-kuluttaja -synkronointiongelman vaatimukset. Tätä toteutusta voidaan pitää hyvin optimistisena siinä mielessä, että missään vaiheessa säikeiden transaktioiden suoritusta ei estetä.

5.2 Lukija-kirjoittaja-toteutukset

Lukija-kirjoittaja-toteutuksissa luodaan eri määriä lukija- ja kirjoittajasäikeitä. Kukin lukijasäie lukee jaettua resurssia äärellisen monta kertaa. Vastaavasti kukin kirjoittajasäie kirjoittaa jaettuun resurssiin äärellisen monta kertaa. Kun lukijasäikeet ovat lukeneet jaettua resurssia määritetyn määrän, ne sulkevat itsensä. Vastaavasti kun kirjoittajasäikeet ovat kirjoittaneet jaettuun resurssiin määrätyn monta kertaa, ne myös sulkevat itsensä. Kun kaikki lukija- ja kirjoittajasäikeet ovat sulkeneet itsensä, eli lukeneet/kirjoittaneet jaettua resurssia tarvitun määrän, ohjelman suoritus päättyy.

5.2.1 Lukkoihin perustuva toteutus

Lukkoihin perustuva lukija-kirjoittaja-toteutus on tehty käyttäen Javan java.util.concurrent-paketin tarjoamaa luokkaa ReentrantReadWriteLock. Tässä kappaleessa esitellään vain toteutuksen keskeisimmät osat.

Tässä lukkoihin perustuvassa toteutuksessa jaettu resurssi on tehty seuraavalla tavalla.

```
private String readWriteResource = "initial";
private final ReentrantReadWriteLock rwLock = new
ReentrantReadWriteLock();
private final Lock readLock = rwLock.readLock();
private final Lock writeLock = rwLock.writeLock();
```

Jaettuna resurssina toimii tässä toteutuksessa merkkijonomuuttuja *readWriteResource*. Säikeiden pääsyä kyseiseen resurssiin käsiksi hallitaan luokan ReentrantReadWriteLock pohjalta tehdyn olion *rwLock* avulla. Lukemislukkoa *readLock* käytetään hallitsemaan lukijasäikeiden pääsyä jaettuun resurssiin. Vastaavasti kirjoittajalukkoa *writeLock* käytetään hallitsemaan kirjoittajasäikeiden pääsyä jaettuun resurssiin.

Kirjoittajasäikeet kirjoittavat jaettuun resurssiin käyttäen metodia `write()`.

```
public void write(final String toWrite) {
        writeLock.lock();
        readWriteResource = toWrite;
        writeLock.unlock();
}
```

Kirjoittajasäie yrittää ensin varata kirjoituslukon (*writeLock*) käyttöönsä kutsumalla metodia `writeLock.lock()`. Jos mikään muu säie ei ole ehtinyt varaamaan käyttöönsä lukemislukkoa (*readLock)* tai kirjoittajalukkoa (*writeLock)*, varaa kyseinen kirjoittajasäie kirjoituslukon (writeLock) omaan käyttöönsä. Seuraavaksi kyseisen kirjoittajasäikeen tarjoama merkkijono *toWrite* kirjoitetaan jaettuun resurssiin *readWriteResource*. Tämän jälkeen kirjoittajasäie vapauttaa varaamansa kirjoituslukon (*writeLock*) kutsumalla metodia `writeLock.unlock()`. Siinä tapauksessa, että toinen säie on ehtinyt jo varaamaan jommankumman lukoista, kirjoittajasäie jää odottamaan kunnes se saa kirjoituslukon (*writeLock*) käyttöönsä. Kirjoituslukon (*writeLock*) voi siis varata käyttöönsä vain yksi kirjoittajasäie kerrallaan (Oracle-sivusto, 2015b.)

Lukijasäikeet lukevat jaettua resurssia käyttäen metodia `read()`.

```
public String read() {
        String readMade;
        readLock.lock();
        readMade = readWriteResource;
        readLock.unlock();
        return readMade;
}
```

Vastaavalla tavalla lukijasäie yrittää ensin varata lukemislukon (*readLock*) käyttöönsä kutsumalla metodia `readLock.lock()`. Jos mikään muu säie ei ole ehtinyt varaamaan käyttöönsä kirjoituslukkoa (*writeLock*), varaa kyseinen lukijasäie lukemislukon (*readLock*) omaan käyttöönsä. Seuraavaksi kyseinen lukijasäie lukee jaetun resurssin *ReadWriteResource* sisällön merkkijonomuuttujaan *readMade*, jonka jälkeen se vapauttaa varaamansa lukemislukon (*readLock*) kutsumalla metodia `readLock.unlock()`. Lopuksi metodi `read()` palauttaa lukemansa jaetun resurssin arvon. Vastaavasti siinä tapauksessa, että toinen säie on ehtinyt jo varaamaan kirjoituslukon käyttöönsä, lukijasäie jää odottamaan kunnes se saa lukemislukon (*readLock*) käyttöönsä. On hyvä huomata, että lukemislukon (*readLock*) voi varata käyttöönsä useampi lukijasäie kerrallaan, niin kauan kuin mikään toinen säie ei varaa kirjoituslukkoa (*writeLock*) käyttöönsä (Oracle-sivusto, 2015c.)

5.2.2 ScalaSTM-toteutus

Tässä kappaleessa esitellään lukija-kirjoittaja ScalaSTM-toteutuksen keskeisimmät osat, eli millainen jaettu resurssi on, sekä kuinka sitä luetaan ja siihen kirjoitetaan.

ScalaSTM-toteutuksessa jaettu resurssi on tehty seuraavalla tavalla.

```
private final Ref.View<String> readWriteResource =
STM.newRef("initial");
```

Jaettuna resurssina tässä toteutuksessa toimii transaktioviite *readWriteResource*, joka luotu merkkijonomuuttujan (String) pohjalta. Jaettuun resurssiin käsiksi pääsyä varten ei tarvitse määritellä erillisiä mekanismeja kuten lukkoihin perustuvassa vastaavassa toteutuksessa, koska käytetty ohjelmallinen transaktiomuisti hoitaa jaetun resurssin pääsyn hallinnan.

Kirjoittajasäikeet kirjoittavat jaettuun resurssiin käyttäen metodia `write()`.

```
public void write(final String toWrite) {
        STM.atomic(new Runnable() {
                public void run() {
                        readWriteResource.set(toWrite);
                }
        });
}
```

Metodin `write()` suorittamassa transaktiossa kirjoitetaan kirjoittajasäikeen tarjoama merkkijono *toWrite* jaettuun resurssiin eli transaktioviitteeseen *readWriteResource* käyttämällä metodia `set()`. Metodi `set()` suorittaa transaktioviitteelle *readWriteResource* transaktionaalisen kirjoitusoperaation (transactional write), mikä tarkoittaa, että kyseinen kirjoitusoperaatio ei tule muiden säikeiden näkyville ennen transaktion hyväksymistä (commit). Transaktion suoritus hyväksytään jos mikään toinen kirjoittajasäie ei ole jo ehtinyt kirjoittamaan kyseiseen jaettuun resurssiin ennen

kyseistä kirjoittajasäiettä. Muussa tapauksessa transaktiota suoritetaan uudestaan alusta kunnes se saadaan suoritettua onnistuneesti loppuun asti (ScalaSTM-sivusto, 2015b.)

Lukijasäikeet lukevat jaettua resurssia käyttäen metodia `read()`.

```
public String read() {
        String readMade = STM.atomic(new Callable<String>() {
                public String call() {
                        return readWriteResource.get();
                }
        });
        return readMade;
}
```

Metodin `read()` suorittamassa transaktiossa luetaan jaetun resurssin eli transaktioviitteen *readWriteResource* arvo käyttämällä metodia `get()`. Lukemisen jälkeen transaktio palauttaa luetun resurssin arvon metodille `read()`, joka puolestaan palauttaa sen lukijasäikeelle (ScalaSTM-sivusto, 2015b.)

5.2.3 Deuce STM-toteutus

Tässä kappaleessa esitellään lukija-kirjoittaja Deuce STM-toteutuksen keskeisimmät osat, eli millainen jaettu resurssi on, sekä kuinka sitä luetaan ja siihen kirjoitetaan.

Deuce STM-toteutuksessa jaettu resurssi on tehty seuraavalla tavalla.

```
private String readWriteResource = "initial";
```

Jaettuna resurssina tässä toteutuksessa toimii merkkijonomuuttuja *readWriteResource*. Voimme huomata, että jaettua resurssia ei tarvitse muokata millään tavalla tätä toteutusta käytettäessä.

Lukija- ja kirjoittajasäikeet lukevat/kirjoittavat jaettuun resurssiin käyttäen metodeja `read()` ja `write()`.

```
@Atomic
public void write(final String toWrite) {
        readWriteResource = toWrite;
}

@Atomic
public String read() {
        return readWriteResource;
}
```

Metodien `read()` ja `write()` rakenne on samanlainen kuin kirjoitettaessa vastaavaa peräkkäin suoritettavaa koodia. Erona ollen, että transaktiona suoritettavat metodit on merkittävä käyttäen huomatusta `@Atomic`.

6. Mittaustulokset

Tässä luvussa esitellään aluksi kuinka mittaukset suoritettiin, jonka jälkeen esitellään saadut mittaustulokset ja niiden analysointi. Mittauksissa mitattiin ohjelman suoritusaikaa siitä ajankohdasta kun ensimmäinen säie käynnistetään, siihen ajankohtaan asti kun kaikki säikeet ovat sulkeneet itsensä. Lisäksi säikeiden käyttäytymistä tarkasteltiin käyttämällä VisualVM-profilointityökalua (VisualVM-sivusto, 2015). Kyseisellä työkalulla voidaan tarkastella säikeiden suorituksen etenemistä sen tuottaman visuaalisen aikajanan avulla. Aikajanan avulla voidaan seurata yksittäisten säikeiden aktiivisuutta sekä suoritusaikaa. Tässä tutkielmassa kyseistä työkalua käytettiin eri toteutusten tekemisen ja mittausten aikana varmistamaan, että ne toimivat halutulla tavalla. Kuten esimerkiksi varmistamaan, että säikeitä suoritettiin oikeita määriä ja niiden suoritusajat vastasivat mittauksissa saatuja tuloksia.

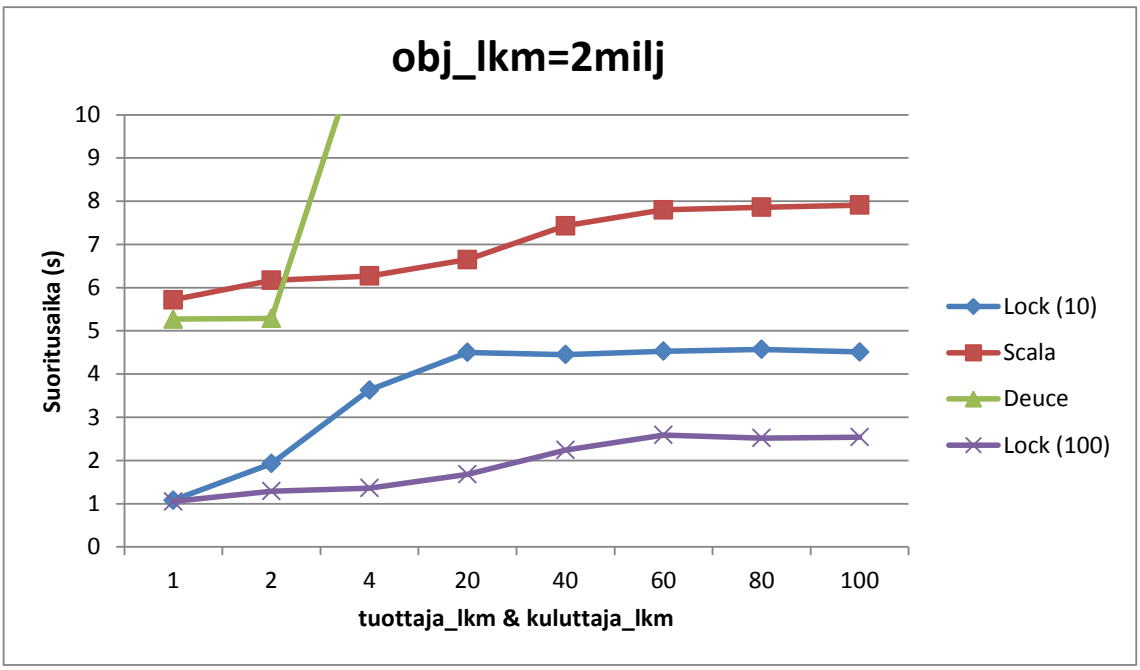
Mittaukset suoritettiin tietokoneella jossa oli 3,3GHz:n kellotaajuudella toimiva neliytiminen i5-2500K prosessori. Keskusmuistia tietokoneessa oli 8GB ja käyttöjärjestelmänä toimi Windows 7. Toteutusten koodit käännettiin käyttämällä Oraclen standardia Java-kääntäjää.

6.1 Tuottaja-kuluttaja-toteutuksien mittaustulokset

Tässä kappaleessa esitetään graafiset esitykset tuottaja-kuluttaja-toteutuksista saaduista mittaustuloksista. Tuottaja-kuluttaja-toteutusten mittaustulosten numeeriset arvot ovat nähtävillä liitteessä A.

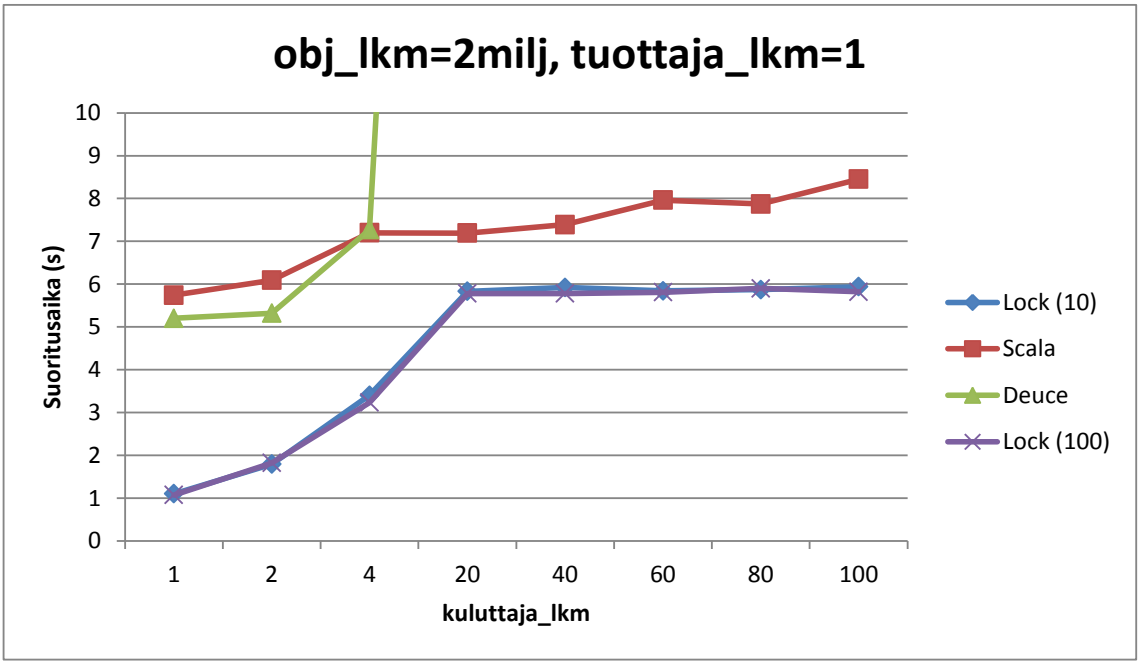
Alustavissa tuottaja-kuluttaja-toteutuksien mittauksissa huomattiin, että puskurin koon muuttaminen ei vaikuttanut merkittävästi Deuce STM- ja ScalaSTM-toteutusten suoritusaikaan. Lukkopohjaisen toteutuksen mittauksissa ilmeni kuitenkin, että sen suoritusaika parantui kun puskurin kokoa kasvatettiin kymmenestä sataan (kuva 1). Lukkopohjaisen toteutuksen puskurin koon kasvattaminen edelleen sadasta suuremmaksi ei vaikuttanut merkittävästi sen suoritusaikaan. Eri toteutusten puskurin koko seuraavissa mittauksissa on 10 (puskuri_kok). Lisäksi lukkopohjaista toteutusta mitattiin puskurin koon arvolla 100.

Tuottaja-kuluttaja-toteutuksien mittauksissa tehtiin kolme mittausta. Kaikissa kolmessa mittauksessa käsiteltävien objektien lukumäärä (obj_lkm) on kaksi miljoonaa. Käsiteltävien objektien lukumäärä asetettiin vakioksi kahteen miljoonan, koska tällöin suoritusajat ovat mittausten tekemisen kannalta sopivan pituiset. Kuvissa näkyvät suoritusajat ovat kymmenen mittauksen pohjalta laskettu keskiarvo. Mittauksia ei enää suoritettu, jos yksittäisen mittauksen ajamiseen kului yli yksi minuutti. Ensimmäisessä mittauksessa mitataan miten eri toteutukset suoriutuvat kun tuottaja- ja kuluttajasäikeiden lukumäärää (tuottaja_lkm & kuluttaja_lkm) kasvatetaan samanaikaisesti. Mittauksissa käsiteltävien objektien lukumäärä jaetaan tasaisesti tuottajasäikeiden kesken, eli 20 tuottajasäikeen mittauksessa kukin tuottajasäie tuottaa 100 000 objektia, ja vastaavasti 40 tuottajasäikeen mittauksessa kukin tuottajasäie tuottaa 50 000 objektia. Ensimmäisen mittauksen tulokset ovat näkyvillä kuvassa 1.



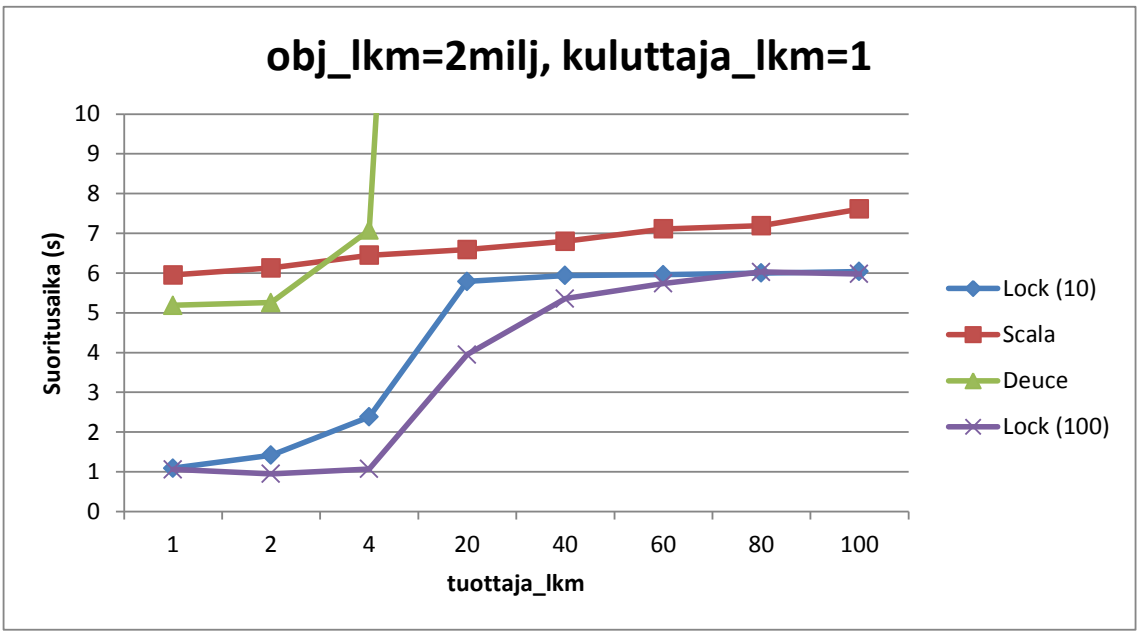
Kuva 1. Tuottajien ja kuluttajien lukumäärät ovat samat.

Toisessa mittauksessa mitataan miten eri toteutukset suoriutuvat kun tuottajasäikeiden lukumäärä on alhainen (mittauksessa yksi) ja kuluttajasäikeiden lukumäärää kasvatetaan. Toisen mittauksen tulokset ovat näkyvillä kuvassa 2.



Kuva 2. Tuottajien lukumäärä on alhainen.

Kolmannessa mittauksessa mitataan miten eri toteutukset suoriutuvat kun kuluttajasäikeiden lukumäärä on alhainen (mittauksessa yksi) ja tuottajasäikeiden lukumäärää kasvatetaan. Kolmannen mittauksen tulokset ovat näkyvillä kuvassa 3.



Kuva 3. Kuluttajien lukumäärä on alhainen.

6.2 Tuottaja-kuluttaja-toteutusten mittaustulosten analysointi

Tuottaja-kuluttaja-toteutusten mittauksien perusteella lukkopohjaisen toteutuksen suorituskyky on kaikissa tapauksissa ohjelmallisia transaktiomuistitoteutuksia parempi (kuvat 1-3). ScalaSTM-toteutus suoriutuu seuraavaksi parhaiten ohjelmallisista transaktiomuistitoteutuksista. Säikeiden lukumäärän ollessa pieni ScalaSTM-toteutuksen ero lukkopohjaiseen toteutukseen on huomattava (noin 5s), mutta säikeiden lukumäärän kasvaessa ero kyseisten toteutusten välillä pienenee (noin 1-3s). Deuce STM-toteutuksen suorituskyky oli kaikista huonoin. Sen suorituskyky vastaa suurin piirtein ScalaSTM-toteutuksen suorituskykyä siihen asti kunnes käytettyjen säikeiden yhteislukumäärä on neljä. Kun käytettyjen säikeiden yhteislukumäärä kasvaa yli neljän, Deuce STM-toteutuksen suorituskyky on huomattavasti muita toteutuksia huonompi. Mittauksissa käytettiin neliytimistä prosessoria, mikä voi selittää miksi Deuce STM-toteutuksen suorituskyky alkaa pienentyä huomattavasti juuri tässä pisteessä, jolloin yksittäisiä säikeitä ei voi enää suorittaa todellisesti rinnakkain omissa ytimissään vaan niiden suoritusta on vaihdeltava ytimien kesken. Toisena selityksenä huonoon suorituskykyyn voi olla Deuce STM:n vuonohjausmekanismi eli poikkeukset. Poikkeukset ovat laskennallisesti hyvin raskaita operaatioita ja niiden liiallinen käyttämien voi mahdollisesti olla yksi syy huonoon suorituskykyyn.

Ohjelmallisten transaktiomuistitoteutuksia käytettäessä puskurin koko ei myöskään vaikuttanut niiden suoritusaikaan merkittävästi. Mutta lukkopohjaisen toteutuksen tapauksessa ilmeni huomattava ero silloin kun tuottaja- ja kuluttajasäikeiden lukumäärät olivat samat (kuva 1). Kun tuottaja- ja kuluttajasäikeiden lukumäärää kasvatettiin yhtä aikaa, lukkopohjaisen toteutuksen suorituskyky parantui kun puskurin kokoa kasvatettiin kymmenestä sataan. Silloin kun kuluttajasäikeitä oli huomattavasti tuottajasäikeitä enemmän, puskurin koolla ei ollut vaikutusta lukkopohjaisen toteutuksen suorituskykyyn (kuva 2). Ero ei myöskään ollut yhtä merkittävä silloin kuin tuottajasäikeitä oli huomattavasti lukijasäikeitä enemmän (kuva 3).

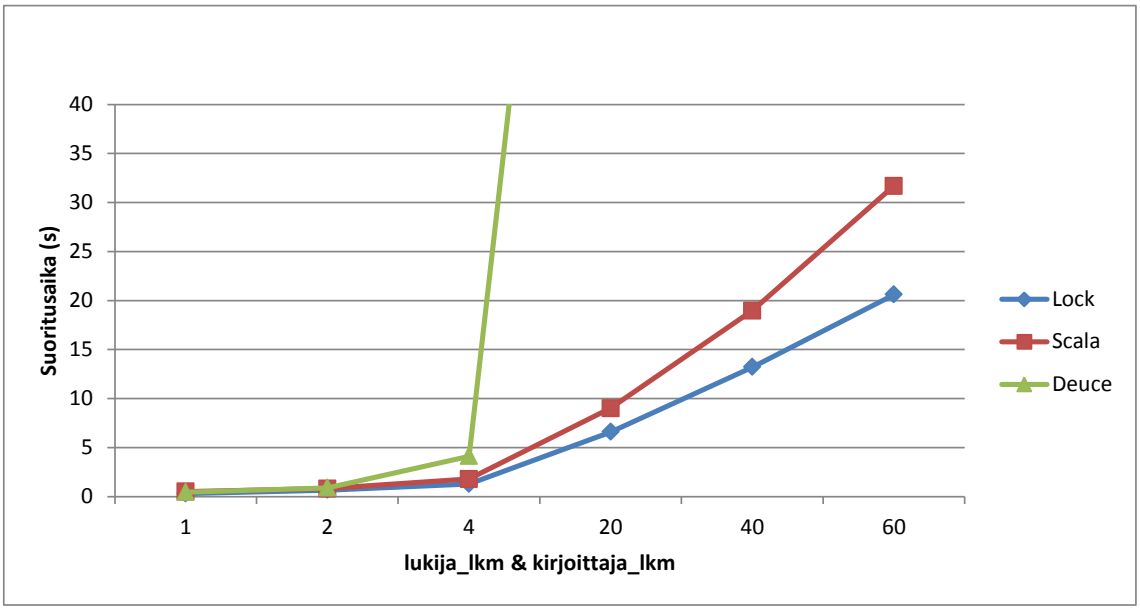
6.3 Lukija-kirjoittaja-toteutuksien mittaustulokset

Tässä kappaleessa esitetään graafiset esitykset lukija-kirjoittaja-toteutuksista saaduista mittaustuloksista. Lukija-kirjoittaja-toteutusten mittaustulosten numeeriset arvot ovat nähtävillä liitteissä B ja C.

Lukija-kirjoittaja-toteutuksista tehtiin tässä tutkielmassa kaksi eri variaatiota. Ensimmäisessä variaatiossa lukija- ja kirjoittajasäikeet lukevat/kirjoittavat aina yhteen samaan jaettuun resurssiin. Toisessa variaatiossa lukija- ja kirjoittajasäikeet lukevat/kirjoittavat satunnaisesti yhteen kymmenestä jaetusta resurssista.

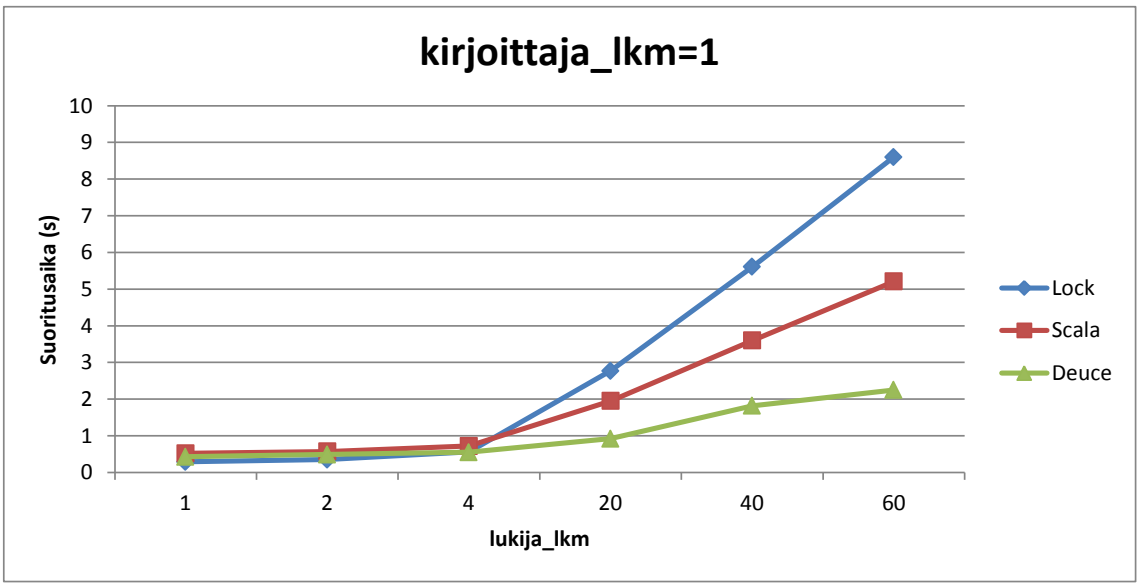
6.3.1 Yhden resurssin lukeminen ja kirjoittaminen

Tässä kappaleessa esitetyissä mittauksissa lukija- ja kirjoittajasäikeet lukevat/kirjoittavat samaan jaettuun resurssiin. Lukija-kirjoittaja-toteutuksien mittauksissa tehtiin aluksi kolme eri mittausta. Kaikissa kolmessa mittauksessa kukin yksittäisen lukijasäie tekee miljoona lukuoperaatiota. Vastaavasti kukin yksittäinen kirjoittajasäie tekee miljoona kirjoitusoperaatiota. Kuvissa näkyvät suoritusajat ovat kymmenen mittauksen pohjalta laskettu keskiarvo. Mittauksia ei enää suoritettu, jos yksittäisen mittauksen ajamiseen kului yli yksi minuutti. Ensimmäisessä mittauksessa mitataan miten eri toteutukset suoriutuvat kun luku- ja kirjoitusoperaatioita on saman verran, eli lukija- ja kirjoittajasäikeiden lukumäärää (lukija_lkm & kirjoittaja_lkm) kasvatetaan samanaikaisesti. Yhden lukija- ja kirjoittajasäikeen mittauksessa siis tehdään siis miljoona luku- ja kirjoitusoperaatiota. Vastaavasti 20 lukija- ja kirjoittajasäikeen mittauksessa tehdään 20 miljoonaa luku- ja kirjoitusoperaatiota. Ensimmäisen mittauksen tulokset ovat näkyvillä kuvassa 4.



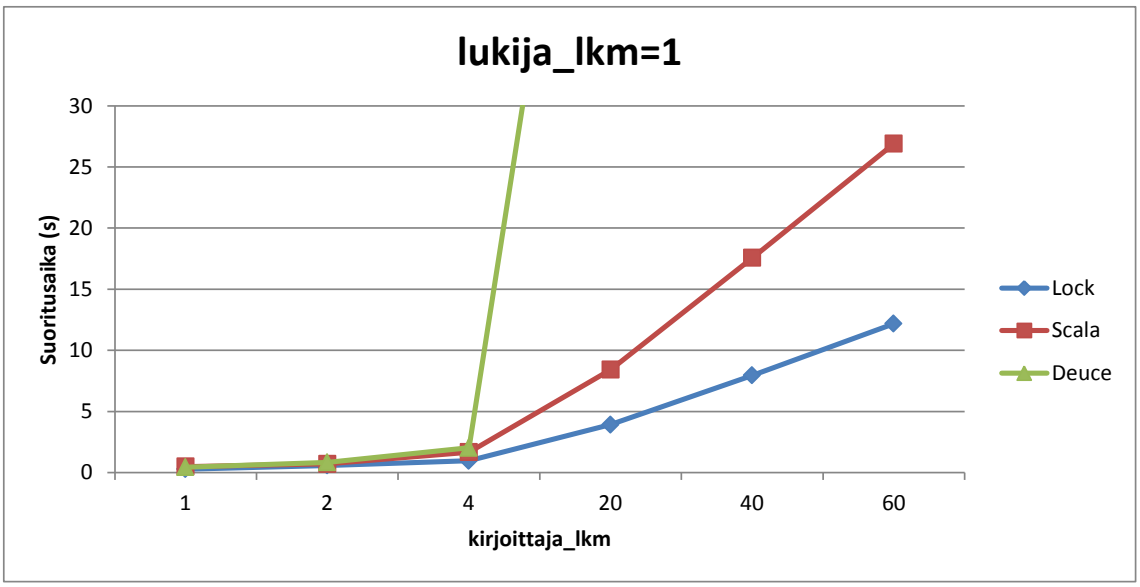
Kuva 4. Yksi jaettu resurssi, lukija- ja kirjoittajasäikeitä on saman verran.

Toisessa mittauksessa mitataan miten eri toteutukset suoriutuvat kun lukuoperaatioita on huomattavasti enemmän kuin kirjoitusoperaatioita. Tässä mittauksessa kirjoittajasäikeiden lukumäärä on yksi (kirjoittaja_lkm) ja lukijasäikeiden lukumäärää kasvatetaan. Toisen mittauksen tulokset ovat näkyvillä kuvassa 5.



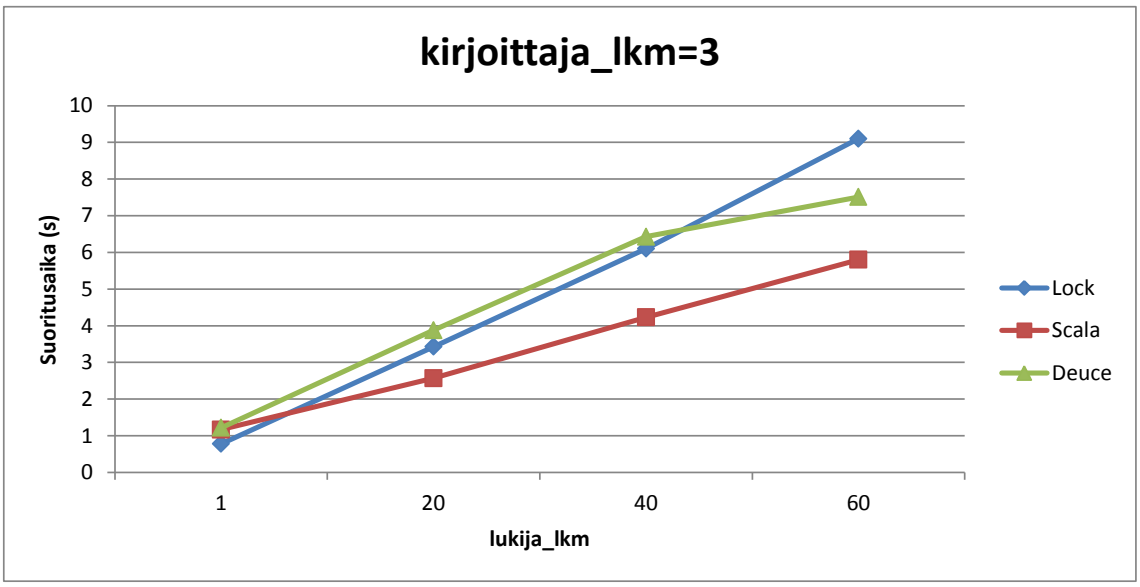
Kuva 5. Yksi jaettu resurssi, lukijasäikeitä on huomattavasti kirjoittajasäikeitä enemmän.

Kolmannessa mittauksessa mitataan miten eri toteutukset suoriutuvat kun kirjoitusoperaatioita on huomattavasti enemmän kuin lukuoperaatioita. Tässä mittauksessa lukijasäikeiden lukumäärä on yksi (lukija_lkm) ja kirjoittajasäikeiden lukumäärää kasvatetaan. Kolmannen mittauksen tulokset ovat näkyvillä kuvassa 6.



Kuva 6. Yksi jaettu resurssi, kirjoittajasäikeitä on huomattavasti lukijasäikeitä enemmän.

Toisessa mittauksessa (kuva 5) voimme huomata, että Deuce STM-toteutuksen suoritusaika on parempi kuin muilla toteutuksilla silloin kun lukijasäikeitä on huomattavasti enemmän kuin kirjoittajasäikeitä. Muissa mittauksissa Deuce STM-toteutuksen suoritusaika on kuitenkin huomattavasti huonompi kuin muilla toteutuksilla (kuvat 4 ja 6). Tämän takia suoritettiin neljäs mittaus, jossa mitattiin missä tilanteessa Deuce STM-toteutuksen suoritusaika vastaa suurin piirtein lukkopohjaisen toteutuksen suoritusaikaa. Mittausten mukaan tämä tapahtuu kun kirjoittajasäikeiden lukumäärä on kolme (kirjoittaja_lkm). Neljännen mittauksen tulokset ovat näkyvillä kuvassa 7.



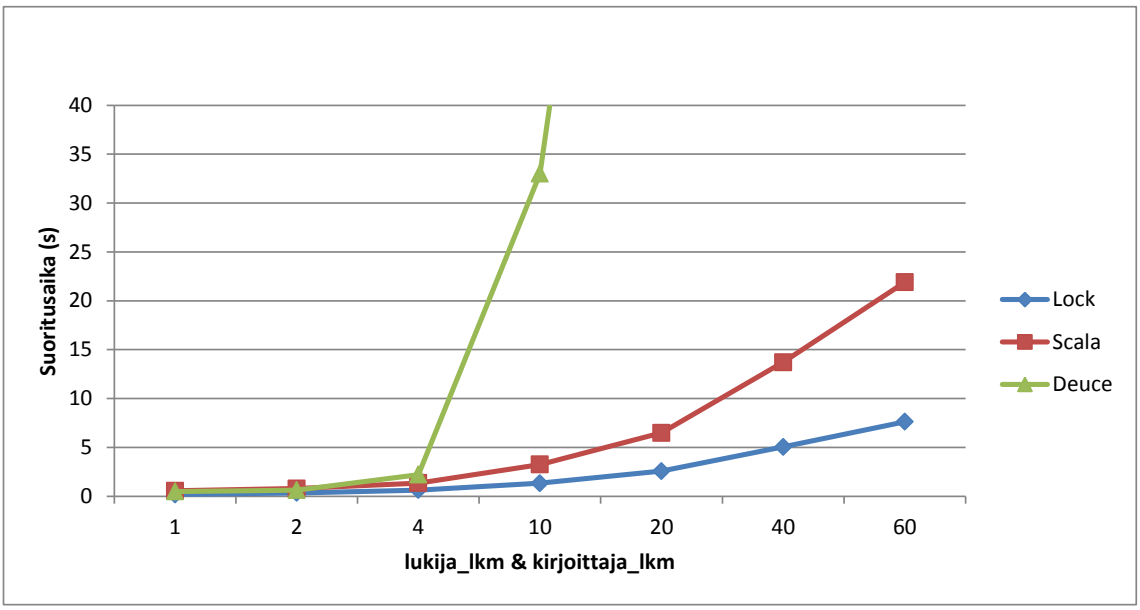
Kuva 7. Yksi jaettu resurssi, kirjoittajasäikeitä on kolme ja lukijasäikeiden lukumäärää kasvatetaan.

Neljännessä mittauksessa (kuva 7) Deuce STM-toteutuksen ja lukkopohjaisen toteutuksen keskimääräiset suoritusajat vastasivat suurin piirtein toisiansa. Deuce STM-toteutuksen mittauksissa yksittäiset suoritusajat kuitenkin vaihtelivat jonkin verran muiden toteutuksien suoritusaikoja enemmän.

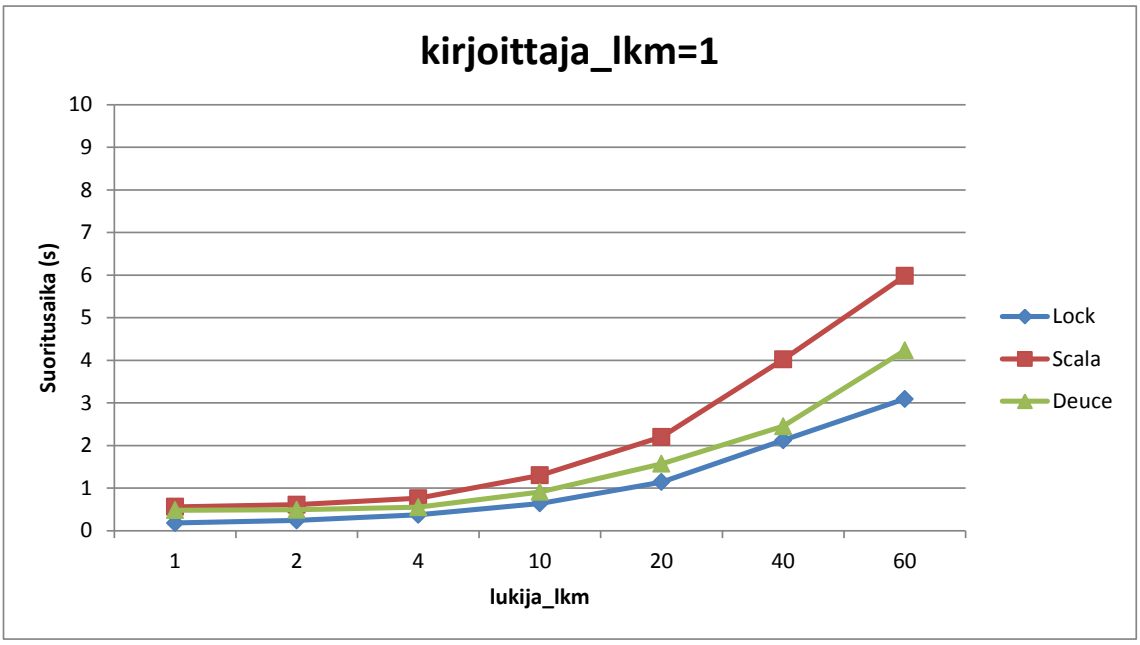
6.3.2 Kymmenen resurssin lukeminen ja kirjoittaminen

Edellisissä lukija-kirjoittaja mittauksissa oli vain yksi jaettu resurssi, jota lukija- ja kirjoittajasäikeet käsittelivät. Seuraavissa mittauksissa jaettujen resurssien määrää kasvatettiin kymmeneen. Lukija- ja kirjoittajasäikeet valitsevat jokaisella kirjoitus- ja lukukerralla satunnaisesti yhden näistä kymmenestä jaetusta resurssista, johon ne kirjoittavat tai jota ne lukevat. Jokaisella jaetulla resurssilla on yhtä suuri todennäköisyys tulla valituksi. Tällä mittauksella pyritään arvioimaan onko perinteisen lukkopohjaisen toteutuksen ja ohjelmallisten transaktiomuistitoteutusten välillä eroavaisuuksia tilanteessa, jossa jaettuja resursseja on useita ja jokaisella jaetulla resurssilla on oma lukko perinteisessä lukkopohjaisessa toteutuksessa. Jaettujen resurssien lukumääräksi valittiin tähän mittaukseen kymmenen. Resurssien määrän kasvattaminen kymmenestä ylöspäin ei alustavien mittausten perusteella aiheuttanut merkittäviä eroja eri toteutusten mittaustuloksien välille.

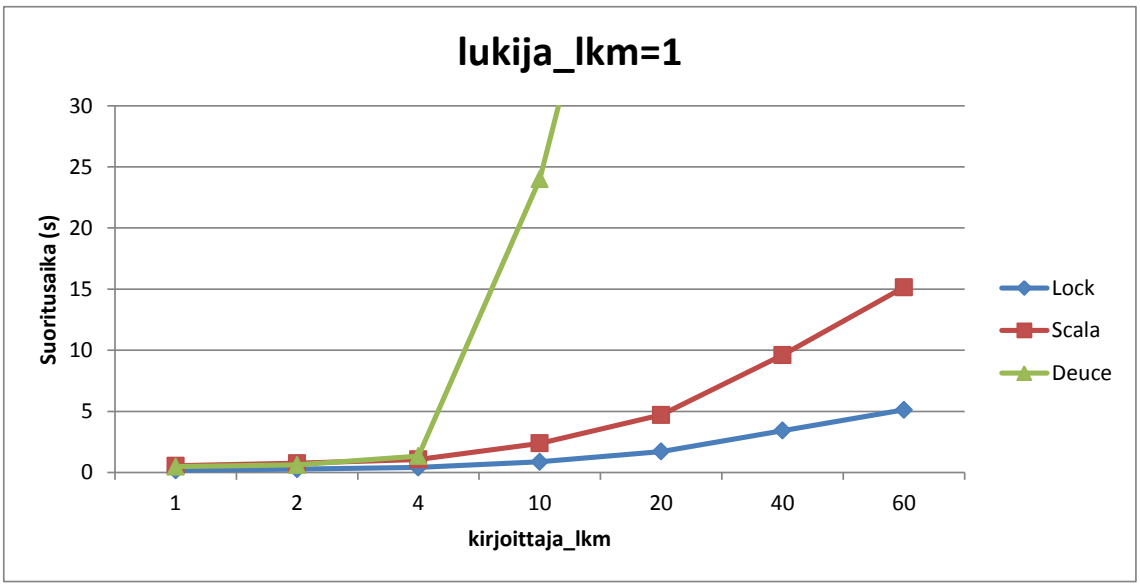
Mittaukset suoritettiin samalla tavalla kuin edellisissä lukija-kirjoittaja-mittauksissa. Kukin yksittäisen lukijasäie tekee siis miljoona lukuoperaatiota. Vastaavasti kukin yksittäinen kirjoittajasäie tekee miljoona kirjoitusoperaatiota. Kuvissa näkyvät suoritusajat ovat kymmenen mittauksen pohjalta laskettu keskiarvo. Oleellisena erona on siis, että lukija- ja kirjoittajasäikeet valitsevat satunnaisesti yhden kymmenestä jaetusta resurssista, johon ne kirjoittavat tai jota ne lukevat, sen sijaan, että kaikki säikeet käsittelisivät aina yhtä samaa jaettua resurssia. Mittausten tulokset ovat näkyvillä kuvista 8-11.



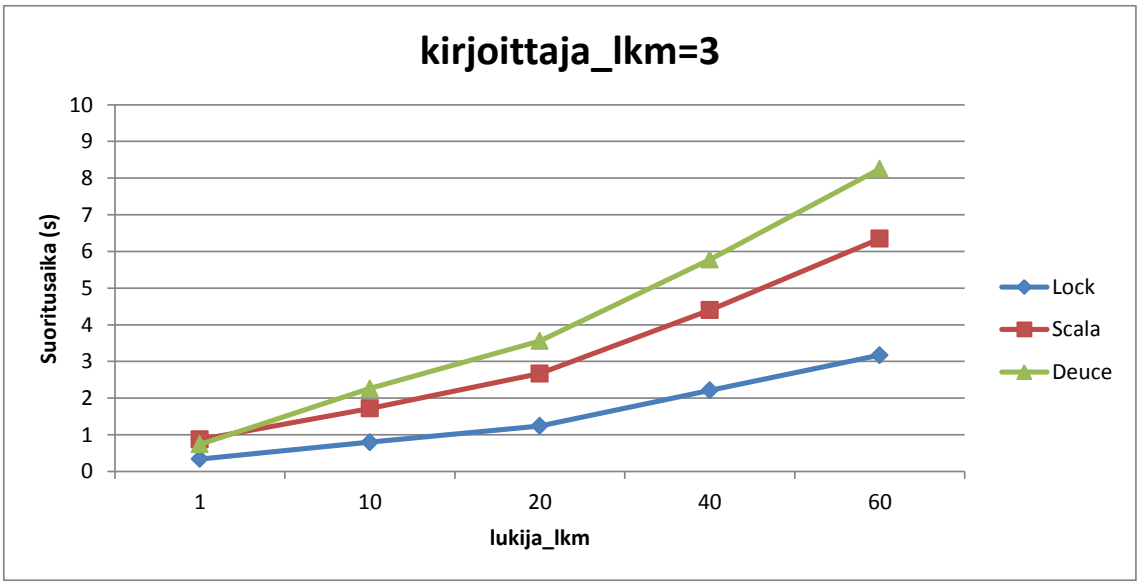
Kuva 8. Kymmenen jaettua resurssia, lukija- ja kirjoittajasäikeitä on saman verran.



Kuva 9. Kymmenen jaettua resurssia, lukijasäikeitä on huomattavasti kirjoittajasäikeitä enemmän.



Kuva 10. Kymmenen jaettua resurssia, kirjoittajasäikeitä on huomattavasti lukijasäikeitä enemmän.



Kuva 11. Kymmenen jaettua resurssia, kirjoittajasäikeitä on kolme ja lukijasäikeiden lukumäärää kasvatetaan.

6.4 Lukija-kirjoittaja-toteutusten mittaustulosten analysointi

Kun jaettuja resursseja oli yksi, lukija-kirjoittaja-toteutusten suorituskyky vaihteli riippuen lukija- ja kirjoittajasäikeiden lukumäärien suhteesta. Kun lukija- ja kirjoittajasäikeitä oli yhtä paljon, lukkopohjainen toteutus oli suorituskyvyltään paras. Scala STM-toteutuksen suorituskyky oli hieman huonompi. Lisäksi sen suorituskyky huononi hieman säikeiden lukumäärän lisääntyessä verrattuna lukkopohjaiseen toteutukseen. Deuce STM-toteutuksen suorituskyky oli huonoin suurella erolla muihin toteutuksiin verrattaessa (Kuva 4.) Kun kirjoittajasäikeitä oli huomattavasti enemmän kuin lukijasäikeitä, suorituskykyjen erot olivat hyvin samankaltaiset (kuva 6). Kun lukijasäikeitä oli huomattavasti kirjoittajasäikeitä enemmän, toteutusten suorituskyvyt kääntyivät ympäri. Deuce STM-toteutuksen suorituskyky oli tällöin paras, toiseksi paras

oli ScalaSTM-toteutus ja huonoiten suoriutui lukkopohjainen toteutus (kuva 5). Kun kirjoittajasäikeiden lukumäärä kasvatettiin kolmeen vastasi Deuce STM-toteutuksen suorituskyky lukkopohjaista toteutusta kuitenkin sillä erotuksella, että yksittäisten suoritusaikojen välillä oli vaihtelua (kuva 7). Kun kirjoittajasäikeiden lukumäärää kasvatettiin tästä eteenpäin, heikkeni Deuce STM-toteutuksen suorituskyky huomattavasti. Mittausten perusteella vaikuttaisi siis, että Deuce on käyttökelpoinen vain niissä tilanteissa, joissa rinnakkaisesti suoritettavien kirjoittajasäikeiden lukumäärä on pienempi kuin käytettävien laskentaydinten lukumäärä.

Kun jaettuja resursseja oli kymmenen, lukija-kirjoittaja-toteutusten suorituskyvyissä esiintyi tiettyjä eroavaisuuksia yhden jaetun resurssin mittauksiin nähden. Kun lukija- ja kirjoittajasäikeitä oli yhtä paljon, muistutti eri toteutusten suorituskyky yhden resurssin mittauksia sillä erotuksella, että suoritusajat olivat muutamia sekunteja pienempiä (kuvat 4 ja 8). Kun lukijasäikeitä oli huomattavasti enemmän kuin kirjoittajasäikeitä, lukkopohjaisen toteutuksen suorituskyky parantui huomattavasti verrattuna muihin toteutuksiin (kuvat 5 ja 9). Erot toteutusten välillä eivät kuitenkaan olleet suuret. Kun kirjoittajasäikeitä oli huomattavasti enemmän kuin lukijasäikeitä, muistuttivat eri toteutusten suorituskyvyt jälleen yhden resurssin mittauksia sillä erotuksella, että suoritusajat olivat pienempiä (kuvat 6 ja 10). Kirjoittajasäikeiden lukumäärällä kolme Deuce STM-totetutuksen suorituskyky vastasi suurin piirtein ScalaSTM-toteutuksen suorituskykyä. Lukkopohjaisen toteutuksen suorituskyky oli kuitenkin parempi verrattuna muihin toteutuksiin (kuvat 7 ja 11).

7. Pohdinta ja yhteenveto

Tässä tutkielmassa rakennettiin kolme eri toteutusta kumpaankin valittuun synkronointiongelmaan. Koska tässä tutkielmassa käytettiin vain kahta synkronointiongelmaa, eli tuottaja-kuluttaja-ongelmaa sekä lukija-kirjoittaja-ongelmaa, rajoittaa tämä jonkin verran tutkielmassa saatujen tulosten yleistettävyyttä. Tuottaja-kuluttaja-ongelmasta saadut mittaustulokset auttavat selventämään miten käytetyt ohjelmalliset transaktiomuistit suoriutuvat kun niitä käytetään korvaamaan perinteisiä poissulkevia lukkoja. Lukija-kirjoittaja-ongelmasta saadut mittaustulokset auttavat vastaavasti selventämään miten eri ohjelmalliset transaktiomuistit suoriutuvat kun niitä käytetään korvaamaan perinteisiä lukija-kirjoittajatyyppisiä lukkomekanismeja. Vaikka tehtyjen toteutusten toiminnan oikeellisuutta testattiin ennen mittausten tekemistä, ei tämä tarkoita, että ne olisivat virheettömiä. Tämä puolestaan vaikuttaa jonkin verran saatujen tulosten luotettavuuteen.

Käytettyjen ohjelmallisten transaktiomuistien, eli Deuce STM:n ja ScalaSTM:n, välillä oli huomattavia eroja. Deuce STM-toteutuksen suorituskyky oli huomattavasti ScalaSTM-toteutusta huonompi kummassakin synkronointiongelmassa kun kirjoitusoperaatioita oli paljon. Toisaalta tilanteissa joissa lukuoperaatioita oli huomattavasti kirjoitusoperaatioita enemmän Deuce STM-toteutuksen suorituskyky vastasi suurin piirtein muita toteutuksia tai oli tietyissä tilanteissa jopa parempi. Lisäksi kummankin toteutuksen vuonohjaustavat erosivat huomattavasti toisistaan. ScalaSTM-toteutuksen vuonohjausta käytettäessä transaktio voi jäädä odottamaan tiettyjen ehtojen täyttymistä, jolloin transaktiota ei tarvitse tietyissä tilanteissa suorittaa jatkuvasti uudestaan. Deuce STM-toteutuksen vuonohjaus on toteutettu käyttämällä poikkeuksia eikä transaktiota voi jättää odottamaan. ScalaSTM soveltuukin tämän takia paremmin tuottaja-kuluttaja-ongelman ratkaisemiseen, koska puskurin ollessa tyhjä tai täynnä voi transaktio jäädä odottamaan, että puskurin tila muuttuu. Koska Deuce STM-toteutus käyttää laskennallisesti raskaita poikkeuksia vuonohjaukseen voi tämä olla yksi selittävä tekijä miksi sen suorituskyky on tuottaja-kuluttaja-toteutuksissa niin heikko.

Helppokäyttöisyyden näkökulmasta Deuce STM-toteutus on huomattavasti ScalaSTM-toteutusta yksinkertaisempi. Deuce STM-toteutuksissa transaktion rakenne on hyvin yksinkertainen. Deuce STM:ää käytettäessä riittää, että transaktiona suoritettavat metodit merkataan huomautuksella `@Atomic`. Deuce STM:ää voi siis varsin vähällä vaivalla käyttää korvaamaan valmiiden sovelluksien lukkiutumismekanismeja. Deuce STM-toteutusten helppokäyttöisyys ei kuitenkaan riitä oikeuttamaan niiden huonoa suorituskykyä ratkaistaessa tähän tutkielmaan valittuja synkronointiongelmia, kun kirjoitusoperaatioita oli huomattavasti lukuoperaatioita enemmän. ScalaSTM-toteutuksissa jaetut resurssit, eli ohjelman muuttujat, on korvattava transaktioviitteillä. Lisäksi ScalaSTM-toteutusten käyttämä transaktioiden rakenne on huomattavasti Deuce STM-toteutusten transaktioiden rakennetta monimutkaisempi. ScalaSTM:ää käytettäessä valmiiden sovelluksien lukitusmekanismien korvaaminen on siis huomattavasti työläämpää kuin käytettäessä Deuce STM:ää.

Ohjelmalliset transaktiomuistit lupaavat tavallisesti tuoda kaksi selvää etua perinteiseen lukkiutumiseen hallintaan nähden. Ensimmäinen etu on lukkiutumattomuus, eli ohjelmallisen transaktiomuistin toimiessa oikein lukkiutumia ei pääse tapahtumaan. Toinen etu on helppokäyttöisyys, ohjelmoija voi kirjoittaa koodia välittämättä rinnakkaisuudesta, eli kirjoittaa koodia samalla tavalla kuin kirjoitettaessa tavallista

peräkkäin suoritettavaa koodia. Ohjelmallisten transaktiomuistien tapauksessa nämä edut saavutetaan kuitenkin suorituskyvyn kustannuksella. Kysymys kuuluukin, ovatko nämä edut riittävät ratkaistaessa tässä tutkielmassa käytettyjä synkronointiongelmia Java-ohjelmointikielellä? Koska tässä tutkielmassa käytetyt synkronointiongelmat ovat yleisiä, löytyy niiden ratkaisemista varten valmiit hyvin dokumentoidut luokat, joilla voidaan varsin vaivattomasti tehdä toimivat lukkopohjaiset ratkaisut kyseisiin synkronointiongelmiin. Näitä luokkia käytettäessä tässä tutkielmassa käytettyjen synkronointiongelmien ratkaiseminen sekä rinnakkaisuuden hallitseminen on huomattavasti helpompaa. Lisäksi niiden suorituskyky on tämän tutkielman tulosten perusteella yleensä ottaen parempi verrattuna ohjelmallisiin transaktiomuistitoteutuksiin. Java-ohjelmointikielen tapauksessa on siis tässä tutkielmassa käytetyt synkronointiongelmat parempi ratkaista käyttäen hyväksi valmiiksi tehtyjä lukkopohjaisia ratkaisuja. Jos synkronointiongelman ratkaisemiseen ei kuitenkaan löydy valmiiksi tehtyjä luokkia ja sen tehokas ratkaiseminen edellyttää hyvin hienojakoisen lukkiutumismekanismin rakentamista, hyvin optimoidusta ohjelmallisesta transaktiomuistista tulee varteenotettava vaihtoehto. Lisäksi jos synkronointiongelmaan kuuluu huomattavasti enemmän lukuoperaatioita suhteessa kirjoitusoperaatioihin, voi ohjelmallinen transaktiomuistitoteutus tarjota mahdollisesti jopa suorituskyvyn lisäystä verrattuna perinteisiin lukkopohjaisiin ratkaisuihin. Java-pohjaiset ohjelmalliset transaktiomuistit eivät siis vielä tällä hetkellä ole varteenotettava vaihtoehto lukkopohjaisille ratkaisuille kuin tietyissä erityisissä tilanteissa. Tulevaisuudessa tilanne voi kuitenkin muuttua prosessorien laskentaytimien lukumäärän kasvaessa, jolloin ohjelmallisten transaktiomuistien suorittamat transaktioiden uudelleen yritykset jakautuvat yhä suuremmalle määrälle laskentaytimiä. Lisäksi laitteistotuen lisääntyminen transaktiomuisteille auttaa parantamaan niiden suorituskykyä entisestään.

Lähteet

Adl-Tabatabai, A. R. Kozyrakis, C. Saha, B. (2006). Unlocking concurrency. *Queue*, *4*(10), 24-33.

Afek, Y. Korland, G. Zilberstein, A. (2011). Lowering STM overhead with static analysis. *Languages and Compilers for Parallel Computing* , 31-45.

AtomJava-sivusto. (2015). Lainattu 24.2.2015, saatavilla: http://wasp.cs.washington.edu/wasp_atomjava.html

Cascaval, C. Blundell, C. Michael, M. Cain, H. W. Wu, P. Chiras, S. Chatterjee, S. (2008). Software transactional memory: Why is it only a research toy? *Queue*, *6*(5), 40.

Deuce STM-sivusto. (2015). Lainattu 25.2.2015, saatavilla: https://sites.google.com/site/deucestm/

Downey, A. B. (2005). *The Little Book of Semaphores* (Version 2.1.5). Green Tea Press.

Emerick, C. Carper, B. Grand, C. (2012). *Clojure programming*. O'Reilly Media, Inc.

Ennals, R. (2006). Software transactional memory should not be obstruction-free. *Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report*, 54.

Fraser, K. (2004). *Practical lock-freedom.* Doctoral dissertation, University of Cambridge.

GCC 4.7 Release. (2012). Lainattu 25.2.2015, saatavilla: https://gcc.gnu.org/gcc-4.7/

Geva, R. (2012). Intel® C++ STM Compiler, Prototype Edition. Lainattu 24.2.2015, saatavilla: https://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition

GHC compiler version 6.4. (2005). Lainattu 25.2.2015, saatavilla: https://www.haskell.org/ghc/download_ghc_64

Harris, T. Larus, J. Rajwar, R. (2010). *Transactional memory. Synthesis Lectures on Computer Architecture*. Morgan and Claypool Publishers.

Herlihy, M. Moss, J. (1993). Transactional memory: Architectural support for lock-free data structures. *Conference Proceedings - Annual Symposium on Computer Architecture*, 289-300.

Jones, S. (2007). *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media, Inc.

JVSTM-sivusto. (2015). Lainattu 24.2.2015, saatavilla: http://inesc-id-esw.github.io/jvstm/

Järvinen, P. (1999). *On research methods*. Opinpajan kirja.

Knight, T. (1986). An architecture for mostly functional languages. *In Proceedings of the 1986 ACM conference on LISP and functional programming*, 105-112. ACM.

Korland, G. Shavit, N. Felber, P. (2010). Noninvasive concurrency with Java STM. *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*.

Kugler, F. (2013). Concurrent Programming: APIs and Challenges, Issue 2. Lainattu 10.2.2015, saatavilla: http://www.objc.io/issue-2/concurrency-apis-and-pitfalls.html

Lomet, D. (1977). Process structuring, synchronization, and recovery using atomic actions. *In ACM Conference on Language Design for Reliable Software*, 128–137.

Marshall, C. (2005). Software Transactional Memory. *Computer Science*, *203*.

Multiverse-sivusto. (2015). Lainattu 25.2.2015, saatavilla: http://multiverse.codehaus.org/overview.html

Oracle-sivusto. (2015a). Lainattu 25.2.2015, saatavilla: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.htm

Oracle-sivusto. (2015b). Lainattu 25.2.2015, saatavilla: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.WriteLock.html

Oracle-sivusto. (2015c). Lainattu 25.2.2015, saatavilla: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.ReadLock.html

Oracle-sivusto. (2015d). Lainattu 25.2.2015, saatavilla: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.htm

Oracle-sivusto. (2015e). Lainattu 25.4.2015, saatavilla: https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html

ScalaSTM-sivusto. (2015a). Lainattu 25.2.2015, saatavilla: http://nbronson.github.io/scala-stm/index.html

ScalaSTM-sivusto. (2015b). Lainattu 25.2.2015, saatavilla: http://nbronson.github.io/scala-stm/api/0.7/index.html#scala.concurrent.stm.japi.STM$

Scherer III, W. Scott, M. (2004). Contention management in dynamic software transactional memory. *PODC Workshop on Concurrency and Synchronization in Java programs*, 70-79.

Shavit, N. Touitou, D. (1997). Software transactional memory. *Distributed Computing*, *10*(2), 99-116.

Silberschatz, A. Galvin, P. (1998). *Operating System Concepts.* Addison-Wesley publishing Company.

Spear, M. Dalessandro, L. Marathe, V. Scott, M. (2009). A comprehensive strategy for contention management in software transactional memory. *ACM Sigplan Notices*, 44(4), 141-150.

Spear, M. Marathe, V. Scherer III, W. Scott, M. (2006). Conflict detection and validation strategies for software transactional memory. *Distributed Computing*, 179-193.

Stallings, W. (1998). *Operating Systems: Internals and Design Principles.* Prentice Hall.

Stone, J. Stone, H. Heidelberger, P. Turek, J. (1993). Multiple reservations and the Oklahoma update. *IEEE Concurrency*, *1*(4), 58-71.

TMJava-sivusto. (2015). Lainattu 25.2.2015, saatavilla: http://www.tmware.org/tmjava

VisualVM-sivusto. (2015). Lainattu 2.3.2015, saatavilla: http://visualvm.java.net/

Waliullah, M. Stenstrom, P. (2007). Starvation-free transactional memory-system protocols. *Euro-Par 2007 Parallel Processing*, 280-291.

Weimerskirch, M. (2008). Software Transactional Memory. Lainattu 25.2.2015, saatavilla: http://michel.weimerskirch.net/wp-content/uploads/2008/02/software_transactional_memory.pdf

Wiki Blue Gene. (2015). Lainattu 25.2.2015, saatavilla: http://en.wikipedia.org/wiki/Blue_Gene

WikiClojure-sivusto. (2015). Lainattu 25.2.2015, saatavilla: http://en.wikipedia.org/wiki/Clojure

Wiki Rock Processor. (2015). Lainattu 25.2.2015, saatavilla: http://en.wikipedia.org/wiki/Rock_(processor)

WikiSTM-sivusto. (2015). Lainattu 25.2.2015, saatavilla: http://en.wikipedia.org/wiki/Software_transactional_memory

Wiki TSX. (2015). Lainattu 25.2.2015, saatavilla: http://en.wikipedia.org/wiki/Transactional_Synchronization_Extensions

Zyulkyarov, F. Gajinov, V. Unsal, O. S. Cristal, A. Ayguadé, E. Harris, T. Valero, M. (2009). Atomic quake: using transactional memory in an interactive multiplayer game server. *ACM Sigplan Notices*, 44(4), 25-34.

Liite A. Tuottaja-kuluttaja mittaustulokset.

Tuottaja-kuluttaja	tuottaja_lkm	kuluttaja_lkm	Lock (10)	Scala	Deuce	Lock (100)
	1	1	1,08	5,72	5,27	1,05
	2	2	1,93	6,17	5,29	1,29
	4	4	3,63	6,27	12,20	1,36
	20	20	4,50	6,65	90,00	1,68
	40	40	4,45	7,43		2,24
	60	60	4,53	7,80		2,59
	80	80	4,57	7,86		2,52
	100	100	4,51	7,91		2,54
	tuottaja_lkm	kuluttaja_lkm	Lock (10)	Scala	Deuce	Lock (100)
	1	1	1,10	5,74	5,20	1,07
	1	2	1,79	6,09	5,32	1,82
	1	4	3,40	7,20	7,27	3,23
	1	20	5,83	7,19	47,00	5,78
	1	40	5,92	7,39		5,78
	1	60	5,84	7,96		5,81
	1	80	5,87	7,87		5,90
	1	100	5,94	8,45		5,82
	tuottaja_lkm	kuluttaja_lkm	Lock (10)	Scala	Deuce	Lock (100)
	1	1	1,09	5,95	5,19	1,06
	2	1	1,42	6,13	5,26	0,95
	4	1	2,38	6,45	7,08	1,07
	20	1	5,79	6,59	48,00	3,95
	40	1	5,94	6,80		5,36
	60	1	5,96	7,11		5,74
	80	1	6,00	7,19		6,03
	100	1	6,04	7,61		5,98

obj_lkm	puskuri_kok
2000000	10

Liite B. Lukija-kirjoittaja mittaustulokset (yksi resurssi).

lukija-kirjoittaja	kirjoittaja_lkm	lukija_lkm	Lock	Scala	Deuce
yksi jaettu resurssi	1	1	0,29	0,51	0,44
	2	2	0,66	0,81	0,88
	4	4	1,27	1,79	4,11
	20	20	6,59	9,02	131
	40	40	13,2	18,98	
	60	60	20,6	31,7	
	kirjoittaja_lkm	lukija_lkm	Lock	Scala	Deuce
	1	1	0,29	0,52	0,43
	1	2	0,35	0,57	0,49
	1	4	0,56	0,72	0,55
	1	20	2,77	1,95	0,92
	1	40	5,61	3,6	1,82
	1	60	8,6	5,21	2,25
	kirjoittaja_lkm	lukija_lkm	Lock	Scala	Deuce
	1	1	0,28	0,49	0,44
	2	1	0,58	0,71	0,83
	4	1	0,96	1,67	2,03
	20	1	3,92	8,41	76
	40	1	7,95	17,57	
	60	1	12,18	26,9	
	kirjoittaja_lkm	lukija_lkm	Lock	Scala	Deuce
	3	1	0,78	1,17	1,22
	3	20	3,43	2,57	3,88
	3	40	6,1	4,23	6,43
	3	60	9,1	5,8	7,51

kirjoitusten_lkm	lukujen_lkm
1000000	1000000

Liite C. Lukija-kirjoittaja mittaustulokset (kymmenen resurssia).

lukija-kirjoittaja	kirjoittaja_lkm	lukija_lkm	Lock	Scala	Deuce
kymmenen jaettua resurssia	1	1	0,18	0,56	0,48
	2	2	0,34	0,79	0,63
	4	4	0,63	1,34	2,21
	10	10	1,33	3,25	33
	20	20	2,57	6,48	120
	40	40	5,05	13,7	
	60	60	7,62	21,9	
	kirjoittaja_lkm	lukija_lkm	Lock	Scala	Deuce
	1	1	0,18	0,56	0,48
	1	2	0,24	0,61	0,49
	1	4	0,37	0,76	0,55
	1	10	0,64	1,3	0,91
	1	20	1,14	2,2	1,57
	1	40	2,12	4,02	2,45
	1	60	3,09	5,98	4,23
	kirjoittaja_lkm	lukija_lkm	Lock	Scala	Deuce
	1	1	0,18	0,55	0,48
	2	1	0,27	0,76	0,62
	4	1	0,41	1,07	1,34
	10	1	0,87	2,38	24
	20	1	1,71	4,71	63
	40	1	3,42	9,61	
	60	1	5,12	15,15	
	kirjoittaja_lkm	lukija_lkm	Lock	Scala	Deuce
	3	1	0,34	0,87	0,74
	3	10	0,8	1,72	2,26
	3	20	1,24	2,67	3,56
	3	40	2,21	4,4	5,78
	3	60	3,17	6,35	8,25

kirjoitusten_lkm	lukujen_lkm
1000000	1000000