Toni Hirvonen

Monitahoinen optimointi ohjelmointikielen kääntäjässä

Tietojenkäsittelytieteen kandidaatintutkielma

21. toukokuuta 2024

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Toni Hirvonen

Yhteystiedot: toni.va.hirvonen@student.jyu.fi

Ohjaaja: Tuomo Rossi

Työn nimi: Monitahoinen optimointi ohjelmointikielen kääntäjässä

Title in English: Polyhedral optimization in a compiler

Työ: Kandidaatintutkielma

Sivumäärä: 23+0

Tiivistelmä: Nykyajan ohjelmointikielten kääntäjät tekevät paljon optimointia, jolla pyritään parantamaan käännetyn ohjelman suorituskykyä. Tähän on monia eri optimointimenetelmiä, joista yksi on monitahoinen optimointi. Se keskittyy sisäkkäisten silmukoiden optimointiin monitahoisen mallin avulla. Tämä kirjallisuuskatsaus tarkastelee monitahoisen mallin käyttöä optimointiprosessissa ja sen vaikutusta käännetyn ohjelman suorituskykyyn.

Avainsanat: monitahoinen optimointi, monitahoinen malli, kääntäjä, sisäkkäiset silmukat

Abstract: Modern compilers perform numerous optimizations, which aim at improving the performance of the compiled program. There are multiple different optimization methods for this, and one of these is polyhedral optimization. It focuses on optimizing nested loops using the polyhedral model. This literature review examines the use of the polyhedral model in the optimization process and its impact on the performance of the compiled program.

Keywords: polyhedral optimization, polytope model, compiler, nested loops

Kuviot

Kuvio 1. Tietoriippuvuuskaavio 6
Kuvio 2. Esimerkki affiinikuvauksesta (Acharya, Bondhugula ja Cohen 2018) 8

Sisällys

1 JOHDANTO 1

2 MONITAHOINEN OPTIMOINTI 3
2.1 Monitahoinen malli 3
2.2 Riippuvuusanalyysi 5

3 SISÄKKÄISTEN SILMUKOIDEN OPTIMOINTI 7
3.1 Transformaatiot 7
3.2 Rinnakkaistaminen 9
3.3 Datan paikallisuuden optimointi 10

4 KOODIN GENEROINTI MONITAHOISESTA MALLISTA 12

5 POTENTIAALI 13
5.1 Vahvuuksia ja heikkouksia 13
5.2 Vertailua AST-pohjaisiin menetelmiin 15

6 YHTEENVETO 16

LÄHTEET 17

1 Johdanto

Korkeatasoisten ohjelmointikielten kääntäjissä tapahtuu hyvin paljon käännösajan optimointia (engl. *Compile-time optimization*), jolla pyritään parantamaan käännetyn ohjelman suorituskykyä. Monitahoinen optimointi (engl. *Polyhedral optimization*) on tehokas käännösajan optimointimenetelmä, jonka käyttömahdollisuuksia tutkitaan edelleen. Se ei kuitenkaan ole kovin yleisessä käytössä eri ohjelmointikielten kääntäjissä ainakaan oletuksena. Esimerkiksi ohjelmointikielen C kääntäjässä GCC on saatavilla Graphite, joka on monitahoista optimointia hyödyntävä kehys (engl. *framework*) (Pop ym. 2006).

Korkeatasoisella ohjelmointikielellä kirjoitettu ohjelma sellaisenaan on harvoin hyvin optimoitu suorituskyvyn, kuten muistinkäytön tai suoritusajan kannalta. Yleensä ohjelmoidessa preferoidaan koodin helppolukuisuutta ja yksinkertaisuutta. Tämä ei aina ole suorituskyvyn kannalta paras käytäntö.

Hyvä suorituskyky on tärkeää etenkin, kun halutaan tehdä suuria määriä laskutoimituksia. Optimointiin etsitään jatkuvasti uusia ja tehokkaampia optimointimenetelmiä ja -algoritmeja. Monitahoinen optimointi on yksi vaihtoehto ja sen potentiaalia on tutkinut esimerkiksi Simburger ym. (2013).

Tässä kandidaatintutkielmassa tarkastellaan sisäkkäisten silmukoiden optimointia monitahoisen mallin avulla. Tähän kuuluu esimerkiksi sisäkkäisten silmukoiden rinnakkaistaminen ja datan paikallisuuden optimointi. Tutkielmassa käydään läpi myös monitahoisen optimoinnin vahvuuksia ja heikkouksia, sekä vertaillaan sitä muihin optimointimenetelmiin.

Monitahoinen optimointi on sisäkkäisten silmukoiden optimointimenetelmä, jossa hyödynnetään monitahoista mallia (Simburger ym. 2013). Monitahoinen malli on esitysmuoto, jossa sisäkkäisten silmukoiden yksittäiset iteraatiot esitetään pisteinä. Tästä pistejoukosta muodostuu monitahokas.

Luvussa 2 käsitellään, miten monitahoinen optimointi yleisesti toimii. Luvussa 3 käydään läpi sisäkkäisten silmukoiden optimointia monitahoisen mallin avulla. Lisäksi käydään läpi keskeisiä käsitteitä, kuten datan paikallisuus, rinnakkaistaminen ja affiinikuvaus. Luku 4 kes-

kittyy koodin generoimiseen monitahoisesta esitysmuodosta. Tämän jälkeen luvussa 5 analysoidaan monitahoisen optimoinnin potentiaalia sisäkkäisten silmukoiden optimointimenetelmänä. Lopuksi luku 6 sisältää yhteenvedon keskeisimmistä asioista, joita tutkielmassa on tullut esille.

2 Monitahoinen optimointi

Monitahoinen optimointi on ohjelmointikielen kääntäjän käyttämä automaattinen optimointi- ja rinnakkaistamismenetelmä (Benabderrahmane ym. 2010). Tämän menetelmän avulla optimoidaan sisäkkäisiä silmukoita käyttäen monitahoista mallia, jota käsitellään luvussa 2.1. Optimointi tehdään yleisimmin käännösaikana (engl. *Compile time*), kun ohjelmaa käännetään korkeatasoisesta ohjelmointikielestä konekielelle.

Monitahoisen optimoinnin toiminta voidaan jakaa kolmeen eri osaan: ohjelman analysointiin, optimointiin ja koodin generointiin (Benabderrahmane ym. 2010). Ensimmäisenä ohjelmakoodi pyritään muuttamaan monitahoiseen esitysmuotoon, jonka avulla suoritetaan riippuvuusanalyysi. Tämän jälkeen optimoidaan, jossa eri optimointialgoritmeja käyttämällä monitahoista mallia muutetaan. Mallia optimoidessa otetaan huomioon edellisessä vaiheessa tehty riippuvuusanalyysi. Lopuksi monitahoisesta mallista generoidaan abstrakti syntaksipuu, minkä jälkeen voidaan jatkaa tavallista kääntämisprosessia.

Monitahoista optimointia voidaan hyödyntää imperatiivisissa kielissä, kuten C ja Fortran (Bastoul 2004). Sisäkkäiset silmukat ovat tyypillisiä rakenteita imperatiivisessa ohjelmoinnissa. Ellmenreich, Lengauer ja Griebl (2000) ovat hyödyntäneet monitahoisesta mallia myös funktionaalisille kielille, kuten Haskell. Tässä sitä on käytetty rinnakkaistamiseen. Imperatiivisten kielten optimointi monitahoisen mallin avulla on paljon yleisempää, joten tässä tutkielmassa keskitytään niihin.

Tässä luvussa kokonaisuudessaan tutustutaan prosessin ensimmäiseen osaan, joka on monitahoisen mallin saaminen ja sen analysointi. Luvussa 2.1 käydään läpi, mikä on monitahoinen malli. Riippuvuusanalyysia tarkastellaan luvussa 2.2.

2.1 Monitahoinen malli

Monitahoinen malli on matemaattinen esitysmuoto, jossa sisäkkäisten silmukoiden jokainen iteraatio voidaan esittää pisteenä. Näistä pisteistä muodostuu monitahokas. Monitahokas on matematiikassa geometrinen objekti, jolla on tasaiset sivut (Jimborean 2012). Monitahok-

kaan pisteet muodostavat äärellisen konveksin joukon pisteitä, joiden välillä on säännöllisiä tietoriippuvuuksia (Lengauer 1993). Joukon konveksisuus tarkoittaa, että kaikkien joukossa olevien pisteiden väliset pisteet kuuluvat myös samaan joukkoon.

Yleisesti ohjelmointikielten kääntäjät tekevät ensin leksikaalianalyysin (engl. *lexical analysis*), jossa ohjelmakoodi muutetaan tokeneiksi. Tämän jälkeen parseri muuttaa yksittäiset tokenit abstraktiksi syntaksipuuksi eli AST:ksi (engl. *Abstract Syntax Tree*). Abstrakti syntaksipuu voidaan sitten muuttaa monitahoiseksi malliksi (Bastoul 2004). Saatuun monitahoiseen malliin voidaan tämän jälkeen tehdä eri transformaatioita, joilla ohjelmaa optimoidaan. Näitä monitahoisen mallin transformaatioita käsitellään luvussa 3.1.

Monitahoinen malli syntyy ohjelmakoodista siten, että ensin etsitään sisäkkäiset silmukat, jotka voidaan esittää monitahoisen mallin avulla (Simburger ym. 2013). Näitä ovat staattiset kontrolliosat eli SCoP:it (engl. *Static Control Part*). SCoP:eja voidaan etsiä kahdella eri tavalla. Niitä voidaan etsiä staattisesti, jolla tarkoitetaan kääntämisen aikana tapahtuvaa etsintää. Niitä on mahdollista etsiä myös dynaamisesti ohjelman ajon aikana, jolloin monitahoinen optimointi on täten ajon aikainen (Simburger ym. 2013). Staattinen on näistä yleisempi tapa, joka on käytössä monitahoisissa optimoijissa, kuten Polly ja Graphite (Grosser, Groesslinger ja Lengauer 2012; Pop ym. 2006).

Tyypillisesti sisäkkäisillä silmukoilla on iteraattorimuuttujat, jotka tieteellisessä laskennassa nimetään yleensä i ja j. Ne usein ovat kokonaisluku (engl. *integer*) tyyppiä, mutta voivat olla esimerkiksi liukulukuja (engl. *floating point number*). Tässä tutkielmassa niiden oletetaan olevan kokonaislukutyyppiä. Yksittäinen sisäkkäisten silmukoiden iteraatio on tilanne, jossa muuttujilla i ja j on jokin arvo. Muuttujien arvot tyypillisesti kasvavat tai pienenevät jokaisella iteraatiolla. Silmukoilla on myös jokin epäyhtälö ehto, joka rajaa muuttujien sallitut arvot.

Kun SCoP ollaan ohjelmakoodista löydetty, täytyy analysoida sen lausekkeiden iteraatioalueita (engl. *iteration domain*). Jokainen molempien silmukoiden sisällä oleva lauseke (engl. *statement*) R muodostaa iteraatiovektorin $\vec{x_R} = (i, j)$ (Pouchet ym. 2008). Jokaisella iteraatiolla suoritettava lauseke R muodostaa oman iteraatiovektorinsa. Kaikista näistä iteraatiovektoreista muodostuu monitahokas D_R (Pouchet ym. 2008). Monitahokkaan rajat saadaan

lausekkeen iteraatioalueen rajoista, jotka määrittävät silmukan rajat (Vasilache, Bastoul ja Cohen 2006). Monitahokas voidaan esimerkiksi määritellä:

$$D_R = \{i, j | 1 \leq i \leq length - 1 \wedge 1 \leq j \leq width - 1\}$$

Yllä oleva määritelmä on joukko iteraatiovektoreita. Silmukoiden sisällä voi olla useampi suoritettava lauseke, jolloin jokainen lauseke muodostaa oman monitahokkaan. Lausekkeen R lisäksi voisi olla myös lauseke L, josta muodostuisi monitahokas D_L.

Optimointi prosessi tulee sitä monimutkaisemmaksi, mitä enemmän lausekkeita silmukat sisältävät. Jokainen monitahokas täytyy ottaa huomioon, kun tarkastellaan lausekkeiden välisiä riippuvuuksia. Nämä riippuvuudet saadaan selville riippuvuusanalyysin avulla.

Myös sisäkkäisten silmukoiden syvyys monimutkaistaa optimointia, koska monitahokkaan ulottuvuuksien määrä kasvaa syvyyden kasvaessa. Esimerkiksi kaksi sisäkkäistä silmukkaa muodostavat kaksiulotteisen monitahokkaan. Ulottuvuuksien kasvaessa tietysti yksittäisten iteraatioiden määrä moninkertaistuu.

Kaikentyyppisiä sisäkkäisiä silmukoita ei voida esittää monitahoisena mallina, joka rajaa monitahoisen optimoinnin käyttökohteita. Näitä ovat epämääräiset silmukkarakenteet, joissa ajon aikana tapahtuu muutoksia. Muutokset voivat olla esimerkiksi silmukoiden rajojen (engl. *loop bounds*) muuttuminen, jota ei voida käännösaikana tietää.

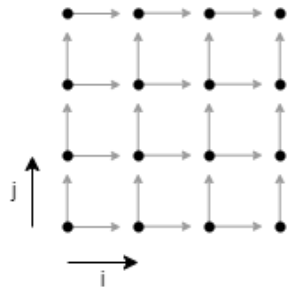
2.2 Riippuvuusanalyysi

Riippuvuusanalyysi (engl. *Dependence analysis*) on tärkeä osa monitahoista optimointia. Sen avulla voidaan varmistaa, että ohjelman logiikka säilyy ennallaan vaikka koodia optimoidaan ja muutetaan. Riippuvuusanalyysissä tarkastellaan sisäkkäisten silmukoiden iteraatioiden välisiä tietoriippuvuuksia (Vasilache ym. 2006).

Yksittäisen lausekkeen suoritus voi olla riippuvainen esimerkiksi toisen lausekkeen tai iteraation laskutoimituksesta. Jos lauseke kirjoittaa dataa johonkin muistipaikkaan ja jokin toinen lauseke lukee tätä, ovat ne tällöin riippuvaisia toisistaan (Benabderrahmane ym. 2010). Vähintään toisen lausekkeista täytyy olla kirjoitusoperaatio, jotta riippuvuus toteutuu. Jos molemmat lausekkeet ovat lukuoperaatioita, eivät ne ole keskenään riippuvaisia. Kaikkien

iteraatioiden välisten tietoriippuvuuksien on säilyttävä, kun monitahoista mallia optimoidaan. Tietoriippuvuuksia voidaan kuvata tietoriippuvuuskaavion (engl. *Data dependence graph*) avulla (Lengauer 1993).

```
for(int i = 1; i < N; i++) {
  for(int j = 1; j < M; j++) {
    S1: A[i][j] = A[i-1][j] + A[i][j-1];
  }
}
```



Kuvio 1: Tietoriippuvuuskaavio

Kuviossa 1 nähdään sitä yllä olevaa koodipätkää vastaava tietoriippuvuuskaavio. Kuviossa mustat pisteet kuvastavat silmukoiden iteraatioita, joillakin arvoilla i ja j. Pisteiden väliset tietoriippuvuudet on kuvattu harmaina nuolina niiden välillä. Nuolien suunta kuvastaa tietoriippuvuuksien kulkua eri iteraatioiden välillä. Jos esimerkiksi pisteestä a lähtee nuoli pisteeseen b, tällöin b on riippuvainen pisteen a laskutoimituksesta.

Ilman riippuvuusanalyysiä ei voida tietää monitahoiseen malliin tehtyjen muutosten laillisuutta. Jos jokin malliin tehty muutos rikkoo jonkin sen tietoriippuvuuksista, ei tämä ole laillinen. Kaikkien tehtyjen muutosten on oltava laillisia, jotta ohjelman logiikka säilyy. Transformaatio on laillinen, jos keskenään riippuvaisten lausekkeiden suoritusjärjestys ja tietoriippuvuudet säilyvät (Jimborean 2012). Monitahoisen mallin transformaatioita käydään läpi luvussa 3.1.

3 Sisäkkäisten silmukoiden optimointi

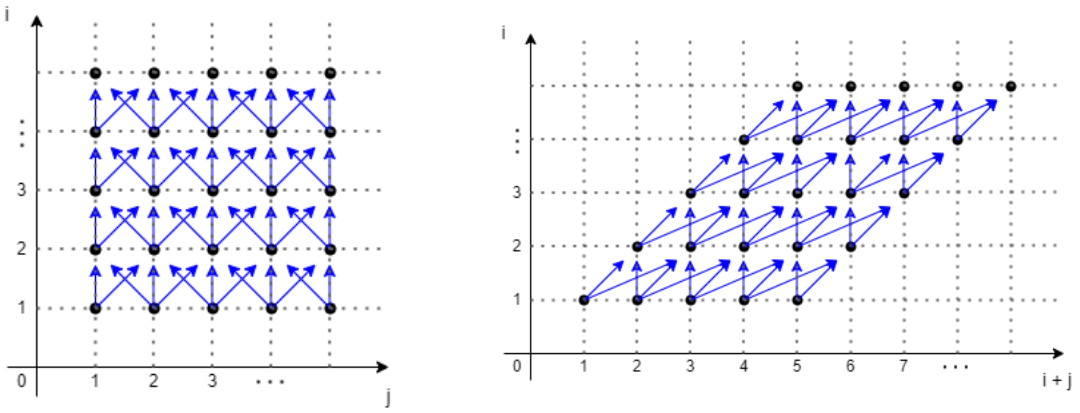
Tässä luvussa käydään läpi monitahoisen optimoinnin toista vaihetta, joka on optimointi. Sisäkkäisten silmukoiden optimointi tapahtuu affiinien kuvausten tai transformaatioiden avulla. Näitä transformaatiota suoritetaan ohjelmakoodin silmukoista muodostuneeseen monitahoiseen malliin.

Yleisimpiä tavoitteita monitahoisessa optimoinnissa ovat rinnakkaistaminen ja datan paikallisuuden tai lokaalisuuden (engl. *Data locality*) optimointi. Rinnakkaistamista käsitellään luvussa 3.2. Luvussa 3.3 käsitellään datan paikallisuuden optimointia.

3.1 Transformaatiot

Monitahoiseen malliin käytettävät transformaatiot ovat affiinikuvauksia (engl. *Affine transformation*). Affiinikuvauksia ovat esimerkiksi peilaaminen, kiertäminen ja skaalaaminen. Affiinisuus takaa, että monitahoisen mallin pisteiden keskinäiset suhteet säilyvät samana. Tämä tarkoittaa, että vierekkäiset pisteet ovat vierekkäin myös kuvauksen jälkeen. Affiinikuvaus T pisteelle x voidaan määritellä $x.T = Ax + b$ (Xinyu ja Ying 2010). Tässä piste x kuvautuu pisteeksi $x.T$ vakio matriisin A ja vakio vektorin b avulla.

Monitahoisessa optimoinnissa yleisiä affiinikuvauksia ovat esimerkiksi silmukoiden yhdistäminen tai jakaminen, uudelleen indeksointi, lausekkeiden uudelleen järjestäminen ja skaalaaminen (Xinyu ja Ying 2010). Kaikki nämä muuttavat silmukoiden lausekkeiden suoritusjärjestystä jollakin tapaa.



(a) Alkuperäinen

(b) Transformoitu

Kuvio 2: Esimerkki affiinikuvauksesta (Acharya, Bondhugula ja Cohen 2018)

Kuviossa 2 on esimerkki yksinkertaisesta monitahoisen mallin affiinikuvauksesta. Mustat pisteet kuvaavat iteraatioita (i, j) ja siniset nuolet niiden välisiä tietoriippuvuuksia. Kuviossa 2b kuvaus olisi $T(i, j) = (i, i + j)$.

Luvussa 2.2 todettiin, että transformaation on oltava laillinen, jotta se voidaan suorittaa. Vasilache ym. (2006) mukaan yksittäinen transformaatio voitaisiin silti suorittaa, vaikka se olisi laiton. Riippuvuusanalyysi mahdollistaa, että transformaatiot voidaan suorittaa sarjassa eikä ainoastaan yksittäin. Tällöin voidaan ensin tehdä halutut transformaatiot, jonka jälkeen niiden laillisuus kokonaisuudessaan voidaan tarkistaa riippuvuusanalyysin avulla.

Tämä on huomattavasti tehokkaampaa verrattuna tavallisiin silmukkaoptimointimenetelmiin, joissa yksittäiset muunnokset tarkastetaan aina ennen, kun ne suoritetaan. Jokaisen muunnoksen laillisuuden tarkastaminen aiheuttaa käännösajan yleisrasitetta (engl. *Compile-time overhead*) (Vasilache ym. 2006). Laillisuuden tarkastaminen transformaatiosarjan suorittamisen jälkeen mahdollistaa paljon monimutkaisempien muutosten tekemisen, jotka voivat olla ohjelman suorituskyvyn kannalta parempia. Samalla vähennetään optimointiin kuluvaa aikaa, koska tehdään vähemmän tarkistuksia.

Monitahoisessa optimoinnissa tätä transformaatiosarjaa kuvataan joukkona affiineja funktioita (Benabderrahmane ym. 2010). Jokaisella lausekkeella on oma affiinifunktionsa kuvaamaan transformaatiota. Funktio voi olla esimerkiksi affiini ajoitusfunktio (engl. *Affine scheduling function*), joka määrittää, milloin lauseke suoritetaan. Näiden avulla saadaan selville,

missä järjestyksessä lausekkeet täytyy suorittaa optimoidussa ohjelmassa.

Jotta affiinikuvauksia voidaan suorittaa, täytyy ne ensin löytää. Niiden löytämiseen voidaan käyttää lineaarista kokonaislukuoptimointia tai lyhennettynä ILP (engl. *Integer Linear Programming*). ILP:n avulla voidaan aina löytää optimaalisin ratkaisu, joka on yksi monitahoisen optimoinnin vahvuuksista (Lengauer 1993).

Hyvien affiinikuvausten löytämiseen on kehitetty monia eri algoritmeja. Yksi suosituimmista monitahoisista optimointikehyksistä on PLuTo (Bondhugula, Ramanujam ja Sadayappan 2008), joka kykenee automaattiseen rinnakkaistamiseen ja lokaalisuuden optimointiin. Myös PLuTo hyödyntää ILP:tä algoritmissaan parhaiden kuvausten löytämiseen.

Silmukkatransformaatiot voidaan löytää myös lineaarisen optimoinnin eli LP avulla (engl. *Linear Programming*) (Simburger ym. 2013). ILP:n käyttö tuottaa haittoja, kun lausekkeiden määrä kasvaa huomattavasti (Acharya, Bondhugula ja Cohen 2018). Tämän seurauksena myös ILP:ssä käytettävien muuttujien ja rajoitteiden määrä kasvaa. Nämä optimointiongelmat ovat ILP:n avulla NP-täydellisiä, joka tarkoittaa, ettei niitä voida ratkaista polynomisessa ajassa, mikä johtaa korkeisiin kääntämisaikoihin. Kokonaislukurajoitteen poistaminen yksinkertaistaa optimointiongelmaa ja siten parantaa monitahoisen mallin skaalautuvuutta isommille optimointiongelmille. LP-ratkaisijat ovat usein myös paljon tehokkaampia kuin ILP-ratkaisijat.

3.2 Rinnakkaistaminen

Rinnakkaistamisessa pyritään selvittämään, voidaanko joitakin sisäkkäisten silmukoiden lausekkeita suorittaa samanaikaisesti hyödyntämällä rinnakkaislaskentaa (engl. *parallel processing*). Tällä saadaan vähennettyä ohjelman suoritusaikaa, koska työmäärää jaetaan useille eri suoritinyksiköille. Voidaan hyödyntää esimerkiksi prosessorin ytimiä tai grafiikkaprosessoria eli GPU:ta (engl. *Graphics Processing Unit*), joka on suunniteltu rinnakkaislaskentaan.

Sekä täydellisesti että epätäydellisesti sisäkkäisiä silmukoita voidaan rinnakkaistaa monitahoisen mallin avulla. Monimutkaisempia muunnoksia vaaditaan epätäydellisesti sisäkkäisten silmukoiden rinnakkaistamisessa, sillä mallin rakenne on monimutkaisempi. Hyvän transfor-

maatiosarjan löytäminen täydellisesti sisäkkäisille silmukoille vaatii vain sopivan kokonaislukumatriisin löytämisen (Kodukula ja Pingali 1996). Tämä on yksinkertaisempaa, koska voidaan hyödyntää lineaarialgebrasta tuttuja matriisilaskutoimituksia.

Silmukat ovat epätäydellisesti sisäkkäiset, jos osa lausekkeista on vain osan mutta ei kaikkien silmukoiden sisällä (Kodukula ja Pingali 1996). Jos kaikki silmukoiden lausekkeet ovat sisimmän silmukan sisällä, ovat silmukat tällöin täydellisesti sisäkkäiset. Täydellisesti sisäkkäisten silmukoiden rinnakkaistaminen tapahtuu kolmessa osassa (Lengauer ja Griebl 1995).

Ensiksi silmukat muutetaan monitahoiseksi malliksi, jossa ulottuvuuksia on sisäkkäisten silmukoiden lukumäärän verran. Tämän jälkeen saadaan toinen monitahokas suorittamalla affiini koordinaattimuunnos alkuperäiseen monitahokkaaseen. Uudessa monitahokkaassa yksi ulottuvuus kuvaa aikaa ja loput avaruutta. Tämä kuvaus on nimeltään aika-avaruuskartoitus (engl. *space-time mapping*).

Saadun monitahokkaan avulla voidaan rakentaa uusi silmukka, jonka sisällä olevat silmukat ovat rinnakkaistettuja. Rinnakkaistetut silmukat saadaan monitahokkaan avaruusulottuvuuksista, joissa jokainen yksittäinen ulottuvuus muodostaa oman silmukan. Silmukoiden sisäinen rakenne saadaan aikaulottuvuuden avulla.

3.3 Datan paikallisuuden optimointi

Optimoidessa datan paikallisuutta halutaan, että ohjelmassa tehdyt muistiviittaukset ovat mahdollisimman paikallisia. Tämä tarkoittaa, että maksimoidaan välimuistin (engl. *Cache*) käyttöä. Välimuistista datan lukeminen on paljon nopeampaa verrattuna esimerkiksi RAM-muistista (engl. *Random-access memory*) lukemiseen. Tämä johtuu siitä, koska RAM-muisti on fyysisesti paljon kauempana prosessoria kuin välimuisti.

Datan paikallisuuden optimointi huomioidaan affiinikuvauksen valinnassa. Ensin etsitään kaikki lailliset kuvaukset, minkä jälkeen niistä valitaan lokaalisuuden kannalta parhain (Xinyu ja Ying 2010). Parhain kuvaus on se, jonka avulla ohjelman muistiviittausten määrä on pienin. Tähän valintaan auttavat kustannusfunktiot (engl. *Cost function*), joiden avulla voidaan laskea esimerkiksi muistinkäytön määrää.

Yksi hyödyllinen menetelmä paikallisuuden optimoinnissa on laatoitus (engl. *Tiling*). Laatoitus vähentää uuden datan hakemista sillä välillä, kun välimuistissa olevaa dataa käytetään (Wolf ja Lam 1991). Ideaalisti käytetään välimuistiin haettua dataa mahdollisimman monta kertaa ennen kuin on tarve hakea uutta dataa.

Laatoituksessa monitahoisen mallin pisteet jaetaan laatoiksi (engl. *Tile*) tai toisin sanoen ryhmitellään pienempiin osiin. Yksittäisten laattojen koko pyritään valitsemaan siten, että saman välimuistin sisällä pystytään säilyttämään laatan sisällä tarvittava data (Wolf ja Lam 1991). Tämän hyöty on, että laatan sisällä ei tarvitse tehdä muistiviittauksia toiseen paikkaan, vaan kaikki tarvittava on samassa välimuistissa. Iteraatiot suoritetaan laatta kerrallaan ja vasta toiseen laattaan siirryttäessä haetaan uutta dataa.

Koodin tasolla, kun monitahoinen malli muutetaan takaisin koodiksi, laatoitus kaksinkertaistaa sisäkkäisten silmukoiden syvyyden. Laatoituksen jälkeen n sisäkkäistä silmukkaa muodostavat silmukan, jossa on $2n$ silmukkaa sisäkkäin (Wolf ja Lam 1991). Sisimmäiset silmukat suorittavat laattojen iteraatioita ja ulommaiset ohjaavat laattojen suoritusjärjestystä.

Tavallisiin kääntäjien optimointi menetelmiin verrattuna laatoitus paransi suorituskykyä Wolf ja Lam (1991) mukaan 2,75 kertaisesti yksittäisellä prosessorilla. Rinnakkaistamisen avulla suorituskyvyn ero kasvoi, mitä enemmän suorittimia oli käytössä. Kahdeksalla suorittimella nopeus oli 64 MFLOPS verrattuna noin 13 MFLOPS (Wolf ja Lam 1991). Optimoitavassa ohjelmassa laskettiin 500 x 500 kokoisten liukulukumatriisien tuloja. Suorituskyky paranee huomattavasti, koska tehdään vähemmän muistiviittauksia.

Laatoitus ei itsenäisesti ole riittävä optimointi menetelmä. Esimerkiksi Xinyu ja Ying (2010) ovat ratkaisussaan ensin suorittaneet affiinikuvaukset, minkä jälkeen on hyödynnetty laatoitusta muutettuun monitahoiseen malliin. Tämä menetelmä teki jopa 30-35 % vähemmän välimuistin ohituksia kuin GCC-kääntäjä, jossa oli käytetty O3-optimointitasoa (Xinyu ja Ying 2010).

Välimuistin ohitus tai ”välimuistihuti” (engl. *Cache miss*) tapahtuu, kun haluttua dataa etsitään välimuistista, mutta sitä ei siellä ole. Tällöin se pitää hakea ulkoisesta muistista välimuistiin, jotta sitä voidaan käyttää. Tätä halutaan minimoida, kun optimoidaan datan paikallisuutta.

4 Koodin generointi monitahoisesta mallista

Monitahoinen malli ei itsessään ole kääntyvä konekielelle. Sen sijaan optimoinnin jälkeen monitahoinen malli täytyy muuntaa takaisin käännettävään muotoon. Monitahoisen optimoinnin viimeisessä vaiheessa malli muutetaan takaisin abstraktiksi syntaksipuuksi.

Monitahoinen malli skannataan läpi, jolla tarkoitetaan sen pisteiden iteroimista ja niiden perusteella muodostuvan AST:n generoimista. Tavoitteena on löytää sisäkkäiset silmukat, jotka käyvät monitahoisen mallin kaikki pisteet läpi vain kerran ja oikeassa järjestyksessä (Bastoul 2004). Tätä varten on kehitetty monia eri algoritmeja.

Yksi tunnetuimmista algoritmeista tähän on Quilleré et al. algoritmi, joka on rekursiivinen (Quilleré, Rajopadhye ja Wilde 2000). Algoritmi ottaa syötteenä listan monitahokkaita, kontekstin C eli parametrien rajoitteet ja ensimmäisen dimension d, jota skannataan. Ulostulona saadaan abstrakti syntaksipuu koodista, joka käy syötteenä annetun monitahokaslistan läpi.

Algoritmista on myös laajennettu versio, jonka on esittänyt Bastoul (2004). Alkuperäisen mukaisesti tämän nimi on laajennettu Quilleré et al. algoritmi. Se vähentää tuotetun koodin kokoa sekä sen generointiaikaa (Bastoul 2004). Erityisen tärkeää on, että koodin suorituskyky ei heikkene, vaikka sen koko pienenee.

Koodin koko on olennaista etenkin sulautetuissa järjestelmissä (engl. *Embedded systems*), joissa muistin määrä on rajoitettu (Bastoul 2004). Parempi muistin käyttö voi niissä parantaa yleistä suorituskykyä ja virrankäyttöä (Xinyu ja Ying 2010). Koodin koon optimointi on aina hyödyllistä, vaikka edut eivät olekaan yhtä merkittäviä, kun resursseja on käytettävissä runsaasti.

Generoimiseen kuluvaa aikaa laajennetussa algoritmissa parannetaan vähentämällä tehtyjen laskutoimituksien määrää. Tähän käytetään avuksi hahmonsovitusta (engl. *Pattern matching*) (Bastoul 2004). Monet ominaisuudet, kuten iteraatioalue, ovat eri laskutoimituksissa samoja, joten näitä tunnistamalla voidaan laskutoimituksen suorittamisen sijaan antaa triviaali ratkaisu. Tämä ratkaisu paransi suorituskykyä Bastoul (2004) mukaan melkein kaksinkertaisesti verrattuna alkuperäiseen Quilleré et al. algoritmiin.

5 Potentiaali

Tässä luvussa käydään läpi monitahoisen optimoinnin potentiaalia sisäkkäisten silmukoiden optimoinnissa. Luvussa 5.1 analysoidaan, mitä etuja monitahoisen mallin käyttö antaa. Tämän jälkeen luvussa 5.2 vertaillaan monitahoista optimointia AST-pohjaisiin silmukoiden optimointimenetelmiin.

5.1 Vahvuuksia ja heikkouksia

Kuten luvussa 3.3 todettiin, hitaita muistiviittauksia saadaan huomattavasti vähennettyä laatoituksen avulla. Kun yhdistettiin paikallisuuden optimointi ja rinnakkaistaminen, saatiin yhä parempia tuloksia. Näiden tuloksien perusteella, monitahoinen optimointi on erinomainen optimointimenetelmä esimerkiksi GPU-laskennassa.

Monitahoinen malli antaa kattavat mahdollisuudet riippuvuusanalyysin suorittamiseen. Sen avulla voidaan suorittaa täsmällinen riippuvuusanalyysi (Pouchet ym. 2008). Tämä mahdollistaa monimutkaisten transformaatiosarjojen suorittamisen yhdellä askeleella.

Simburger ym. (2013) käyttivät tutkimuksessaan mittana suorituksen SCoP kattavuutta (engl. *execution SCoP coverage*) tai *ExecCov*. *ExecCov* mittaa, kuinka suuri osa ohjelman kokonaisesta suoritusajasta käytetään SCoP:ien sisällä. Koska vain SCoP:eja voidaan optimoida, tavoitteena on, että *ExecCov* olisi mahdollisimman suuri.

Monitahoinen optimointi käännösaikana vähensi ohjelman suoritusaikaa 0,79–15,1 % ja *ExecCov* oli keskimäärin 10 % (Simburger ym. 2013). Käännösajan optimointi ei Simburger ym. (2013) mukaan näytä kovin potentiaalisia tuloksia. Simburgerin tutkimuksessa käytettiin vain osaa monitahoisessa optimoinnissa käytettävistä menetelmistä, joten tulokset voivat olla mahdollisesti parempia hyödyntäen toisia algoritmeja tai menetelmiä (Simburger ym. 2013).

ExecCov kasvoi, kun monitahoista optimointia tehtiin ajon aikana. Ajon aikana saadaan ohjelmasta paljon enemmän tietoa, josta monitahoinen optimointi voi hyötyä, kuten muuttuneet silmukoiden rajat. Kuten luvussa 2.1 käsiteltiin, rajoja ei voida kääntämisen aikana ennustaa, jos niillä on mahdollisuus ajon aikana muuttua. Ajon aikaisen optimoinnin avulla *ExecCov*

oli keskimäärin 29 % (Simburger ym. 2013). Koska saadaan lisää tietoa optimoitavasta ohjelmasta, voidaan ajon aikaisen optimoinnin avulla optimoida suurempaa osaa ohjelmakoodista. Tämän myötä ohjelman suoritusaika kasvaa, koska huomattava määrä aikaa käytetään optimointiin ajon aikana. Käännösajan optimoinnissa taas ajon aikana suoritetaan vain ohjelmakoodia, joten suoritusajan kannalta se on parempi vaihtoehto.

Julkaisussaan Bondhugula, Ramanujam ja Sadayappan (2008) tutkivat PLuTon suorituskykyä ja vertailivat sitä esimerkiksi muihin tuotannossa oleviin kääntäjiin. Tunnettuun GCC-kääntäjään verrattuna PLuTon avulla optimoitu ja käännetty ohjelma oli noin 5 kertaa nopeampi (Bondhugula, Ramanujam ja Sadayappan 2008, Kaavio 6). Ohjelmaa suoritettiin yhdellä prosessorin ytimellä ja GCC-kääntäjässä käytettiin O3-optimointitasoa. Tulokset osoittavat käännetyn ohjelman huomattavasti nopeammaksi, mikä eroaa Simburgerin johtopäätöksistä käännösajan optimoinnin kannalta (Simburger ym. 2013).

Suorituskyvyn ero GCC-kääntäjään kasvoi suuremmaksi, kun suoritettiin rinnakkaisesti useammalla eri ytimellä. GCC:llä käännetyn ohjelman nopeus oli neljällä ytimellä noin 0,3 GFLOPS ja PLuTo:lla vastaava oli hieman vajaa 5 GFLOPS (Bondhugula, Ramanujam ja Sadayappan 2008). Rinnakkaislaskennan avulla tulokset vain paranevat, koska voidaan optimoida jokaisen rinnakkaistetun silmukan muistin käyttöä.

Näillä suorituskyvyn eduilla on kuitenkin hintansa. Datan paikallisuuden optimointi monitahoisen mallin avulla vie aikaa, joten kääntämisaika voi kasvaa (Xinyu ja Ying 2010). Kääntämisaika on oleellinen etenkin, kun käännettävän ohjelman koko kasvaa huomattavasti. Tällöin pienetkin nopeus erot voivat olla merkittäviä.

Korkea kääntämisaika ei aina ole ongelma. Esimerkiksi sulautettujen järjestelmien kääntäjien ei tarvitse olla nopeita (Xinyu ja Ying 2010). Muisti ja suoritusteho ovat sulautetuissa järjestelmissä rajallisia, joten näiden optimointi on ainoa tavoite. Optimointiaikaa voidaan eri ratkaisujen avulla vähentää, kuten luvussa 3.1 oli ILP korvattu LP:n avulla.

Toinen rajoite on, että voidaan optimoida vain staattisia kontrolliosia eli SCoP:eja, kuten luvussa 2.1 todettiin. Toisaalta Benabderrahmane ym. (2010) esittävät, että monitahoista optimointia voitaisiin hyödyntää myös epästaattisille kontrolliosille, kuten while-silmukoille. Kontrolliosat ovat epästaattisia, jos esimerkiksi silmukoiden rajoja ei voida esittää affiinina

funktiona. Tämän avulla saataisiin optimoitua suurempaa osaa ohjelmasta.

Ainoa monitahoisen optimoinnin rajoite Benabderrahmane ym. (2010) mukaan on käytettävien algoritmien korkea kompleksisuus, joka rajoittaa skaalautuvuutta. Kompleksisuus kasvaa eksponentiaalisesti sisäkkäisten silmukoiden syvyyden ja lausekkeiden määrän kasvaessa. Monitahoisen optimoinnin skaalautuvuus heikkenee entisestään, jos optimoidaan myös epästaattisia kontrolliosia.

5.2 Vertailua AST-pohjaisiin menetelmiin

Muut sisäkkäisten silmukoiden optimointimenetelmät ovat tyypillisesti AST-pohjaisia, joka tarkoittaa, että optimoinnit tehdään suoraan abstraktiin syntaksipuuhun. Abstraktiin syntaksipuuhun tehdyt muunnokset ovat usein yksinkertaisia verrattuna monitahoisen mallin muunnoksiin. Monitahoisessa mallissa voidaan yhdellä affiinikuvauksella tehdä samat muutokset, jotka AST-mallissa vaatisivat useita silmukkamuunnoksia (Xinyu ja Ying 2010). Tästä syystä monitahoisen mallin avulla voidaan helposti tehdä paljon monimutkaisempia muunnoksia, jotka eivät välttämättä AST-mallissa olisivat mahdollisia.

Koska transformaatiot suoritetaan monitahoisessa optimoinnissa sarjassa, ei yksittäisen muunnoksen laillisuutta tarvitse tarkastaa. Toisin kuin AST-pohjaisissa optimointimenetelmissä, joissa nämä tehdään yksitellen ja samalla tarkistaen. Se aiheuttaa käännösajan yleisrasitetta, kuten luvussa 3.1 todettiin.

Monitahoinen malli antaa matemaattisia lähestymistapoja tietoriippuvuuksien esittämiseen ja affiinikuvausten laillisuuden tarkistamiseen (Shirako, Pouchet ja Sarkar 2014). Tämä mahdollistaa esimerkiksi eri matemaattisten teorioiden ja menetelmien käytön, kuten lineaarinen optimointi, jota käsiteltiin luvussa 3.1. AST-pohjaisten menetelmien etuna on, että niitä voidaan hyödyntää useampiin eri optimoitaviin ohjelmiin. Tämä johtuu siitä, että AST-pohjaisten muunnoksien ei tarvitse olla affiineja (Shirako, Pouchet ja Sarkar 2014). Affiinisuus rajoittaa monitahoisen mallin käyttöä, joten yleiseen optimointiin AST-pohjaiset menetelmät voivat antaa tasaisempia tuloksia. Näitä menetelmiä voidaan Shirakon mukaan myös käyttää yhdessä, joka laajentaa ohjelman osaa, jota voidaan optimoida (Shirako, Pouchet ja Sarkar 2014).

6 Yhteenveto

Kandidaatintutkielmassa käytiin läpi monitahoista optimointia ja sen vahvuuksia sekä heikkouksia sisäkkäisten silmukoiden optimoinnissa. Monitahoinen optimointi voitiin jakaa kolmeen osaan: analysointiin, optimointiin ja koodin generointiin. Tutkielmassa vertailtiin myös monitahoista optimointia toisiin optimointimenetelmiin.

Yksi merkittävimpiä monitahoisen mallin etuja on täsmällinen riippuvuusanalyysi, jonka avulla voidaan selvittää silmukoiden iteraatioiden välisiä tietoriippuvuuksia. Riippuvuusanalyysin avulla voidaan havaita rinnakkaistamisen mahdollisuus ja tarkastaa optimointien laillisuus. Tavoitteena sisäkkäisten silmukoiden optimoinnissa ovat rinnakkaistaminen ja datan paikallisuuden optimointi. Näiden avulla sisäkkäisten silmukoiden suorituskykyä saatiin huomattavasti kasvatettua verrattuna esimerkiksi GCC O3-optimointitason avulla optimoituun ohjelmaan. Suorituskyvyn kasvu saatiin vähentämällä muistiviittausten määrää ja hyödyntämällä rinnakkaislaskentaa.

Ohjelman kääntämiseen kuluva aika kasvaa monitahoista optimointia käyttäessä verrattuna muihin optimointimenetelmiin. Mutta etenkin sulautettujen järjestelmien kääntäjissä monitahoinen optimointi on hyödyllinen, koska niiden muistin määrä ja suoritusteho ovat rajattuja. Se on hyödyllinen myös tieteellisessä- ja numeerisessa laskennassa, joissa on usein paljon silmukkarakenteita. Kaikkia sisäkkäisiä silmukoita ei voida esittää monitahoisena mallina, joka rajaa sen käyttöä.

Lähteet

Acharya, A., U. Bondhugula ja A. Cohen. 2018. “Polyhedral auto-transformation with no integer linear programming”, 529–542. ISBN: 978-1-4503-5698-5. https://doi.org/10.1145/3192366.3192401.

Bastoul, C. 2004. “Code generation in the polyhedral model is easier than you think”. Teoksessa *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* 7–16. ISSN: 1089-795X. https://doi.org/10.1109/PACT.2004.1342537.

Benabderrahmane, Mohamed-Walid, Louis-Noël Pouchet, Albert Cohen ja Cédric Bastoul. 2010. “The Polyhedral Model Is More Widely Applicable Than You Think”. Teoksessa *Compiler Construction,* toimittanut Rajiv Gupta, 283–303. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. ISBN: 978-3-642-11970-5. https://doi.org/10.1007/978-3-642-11970-5_16.

Bondhugula, Uday, J Ramanujam ja P Sadayappan. 2008. “PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System”, https://www.ece.lsu.edu/jxr/Publications-pdf/tr70-07.pdf.

Ellmenreich, Nils, Christian Lengauer ja Martin Griebl. 2000. “Application of the Polytope Model to Functional Programs”. Teoksessa *Languages and Compilers for Parallel Computing,* toimittanut Larry Carter ja Jeanne Ferrante, 219–235. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. ISBN: 978-3-540-44905-8. https://doi.org/10.1007/3-540-44905-1_14.

Grosser, Tobias, Armin Groesslinger ja Christian Lengauer. 2012. “Polly — performing polyhedral optimizations on a low-level intermediate representation”. Publisher: World Scientific Publishing Co. *Parallel Processing Letters* 22 (04). ISSN: 0129-6264. https://doi.org/10.1142/S0129626412500107.

Jimborean, Alexandra. 2012. “Adapting the polytope model for dynamic and speculative parallelization”. Tohtorinväitöskirja, Université de Strasbourg. https://theses.hal.science/tel-00733850.

Kodukula, Induprakas, ja Keshav Pingali. 1996. “Transformations for imperfectly nested loops”. Teoksessa *Proceedings of the 1996 ACM/IEEE conference on Supercomputing,* 12–es. Supercomputing ’96. USA: IEEE Computer Society. ISBN: 978-0-89791-854-1. https://doi.org/10.1145/369028.369051.

Lengauer, C. 1993. “Loop parallelization in the polytope model”. ISBN: 9783540572084, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 715 LNCS:398–416. ISSN: 0302-9743. https://doi.org/10.1007/3-540-57208-2_28.

Lengauer, C., ja M. Griebl. 1995. “On the parallelization of loop nests containing while loops”. Teoksessa *Proceedings the First Aizu International Symposium on Parallel Algorithms/Architecture Synthesis,* 10–18. Fukushima, Japan: IEEE Comput. Soc. Press. ISBN: 978-0-8186-7038-1. https://doi.org/10.1109/AISPAS.1995.401360.

Pop, S., A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber ja N. Vasilache. 2006. “GRAPHITE: Polyhedral analyses and optimizations for GCC”, 179–197. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=fcb36237969922eaeed46613ff7dec14ef888d69#page=185.

Pouchet, Louis-Noël, Cédric Bastoul, Albert Cohen ja John Cavazos. 2008. “Iterative optimization in the polyhedral model: part ii, multidimensional time”. *ACM SIGPLAN Notices* 43 (6): 90–100. ISSN: 0362-1340. https://doi.org/10.1145/1379022.1375594.

Quilleré, Fabien, Sanjay Rajopadhye ja Doran Wilde. 2000. “Generation of Efficient Nested Loops from Polyhedra”. *International Journal of Parallel Programming* 28 (5): 469–498. ISSN: 1573-7640. https://doi.org/10.1023/A:1007554627716.

Shirako, Jun, Louis-Noel Pouchet ja Vivek Sarkar. 2014. “Oil and Water Can Mix: An Integration of Polyhedral and AST-Based Transformations”. Teoksessa *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis,* 287–298. New Orleans, LA, USA: IEEE. ISBN: 978-1-4799-5500-8 978-1-4799-5499-5. https://doi.org/10.1109/SC.2014.29.

Simburger, A., S. Apel, A. Größsslinger ja C. Lengauer. 2013. “The potential of polyhedral optimization: An empirical study”, 508–518. ISBN: 978-1-4799-0215-6. https://doi.org/10.1109/ASE.2013.6693108.

Vasilache, Nicolas, Cedric Bastoul, Albert Cohen ja Sylvain Girbal. 2006. “Violated dependence analysis”. Teoksessa *Proceedings of the 20th Annual International Conference on Supercomputing,* 335–344. ICS ’06. Cairns, Queensland, Australia: Association for Computing Machinery. ISBN: 1595932828. https://doi.org/10.1145/1183401.1183448.

Vasilache, Nicolas, Cédric Bastoul ja Albert Cohen. 2006. “Polyhedral Code Generation in the Real World”. Teoksessa *Compiler Construction,* toimittanut Alan Mycroft ja Andreas Zeller, 185–201. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. ISBN: 978-3-540-33051-6. https://doi.org/10.1007/11688839_16.

Wolf, Michael E., ja Monica S. Lam. 1991. “A data locality optimizing algorithm”. Teoksessa *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation,* 30–44. PLDI ’91. New York, NY, USA: Association for Computing Machinery. ISBN: 978-0-89791-428-4. https://doi.org/10.1145/113445.113449.

Xinyu, Yuan, ja Li Ying. 2010. “Polyhedral Model Based Data Locality Optimization for Embedded Applications”. Teoksessa *2010 IEEE/ACM Int’l Conference on Green Computing and Communications & Int’l Conference on Cyber, Physical and Social Computing,* 926–930. https://doi.org/10.1109/GreenCom-CPSCom.2010.120.