TAMPEREEN TEKNILLINEN YLIOPISTO

LAURI HAAVISTO

CLANG++- JA G++ -KÄÄNTÄJIEN VAIHEIDEN JA NIIDEN TUOTOSTEN VERTAILU

Kandidaatintyö

Tarkastaja: Mikko Nurminen
27. huhtikuuta 2017

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
HAAVISTO, LAURI: Clang++- ja G++ -kääntäjien vaiheiden ja niiden tuotosten vertailu
Kandidaatintyö, 21 sivua, 14 liitesivua
Huhtikuu 2017
Pääaine: Ohjelmistotuotanto
Tarkastaja: Mikko Nurminen
Avainsanat: gcc, clang, vertailu, kääntäminen, optimointi

Työssä vertaillaan kahden C++-kääntäjän vaiheita ja niiden tuotoksia. Käytetyt kääntäjät ovat Clang++ ja G++. Tutkimusta varten on luotu kaksi ohjelmaa, joihin on sisällytetty erilaisia ominaisuuksia ja rakenteita. Vertailu toteutetaan kääntämällä näitä kahta ohjelmaa vaiheittain ja tarkastelemalla vaiheista saatuja tuotoksia. Kääntäminen suoritetaan molemmilla kääntäjillä sekä optimointien kanssa, että ilman. Tämän lisäksi vertaillaan koko kääntöprosessiin kuluvaa aikaa sekä käännettyjen ohjelmien suoritusaikoja ja tiedostokokoja.

Tutkimuksessa huomataan, että kääntäjät ovat hyvin vastaavia keskenään ja kääntäminen onnistuu niillä samalla tavalla. Eroja kuitenkin havaitaan kääntäjien lopputuotoksista, sekä varsinkin assemble-vaiheen assembly-kielisistä tuotoksista. Myös optimoinneilla havaitaan olevan suuri vaikutus tuotetun koodin kannalta. Kääntäjistä ei kuitenkaan selkeästi erotu, että toinen olisi jollain tietyllä osa-alueella parempi.

SISÄLLYS

1 JOHDANTO

Tutkimuskysymyksenä tässä kandidaatintyössä on, kuinka kahden C++-kääntäjän käännösprosessien vaiheet ja niiden tuotokset eroavat. Molemmissa kääntäjissä perusvaiheina ovat esikääntö, assemble, kääntäminen ja linkitys, jotka suoritetaan vastaavassa järjestyksessä. Työssä tutkitaan kääntäjien luomien suoritettavien tiedostojen välisiä eroavaisuuksia. Tämän lisäksi tarkastellaan optimointien vaikutuksia tuotetun koodin kannalta sekä niiden eroavaisuuksia kääntäjien kesken. Vertailussa kiinnitetään huomiota ohjelmien rakenteeseen, nopeuteen sekä kokoon.

Vertailu on tärkeää, sillä on hyvä tietää, kuinka suuria eroja kääntäjien välillä löytyy. Joissain projekteissa suorituskyky voi olla oleellisen tärkeää, jolloin myös eri kääntäjää käyttämällä voidaan saavuttaa parempia tuloksia. Kääntäjät voivat myös suoriutua paremmin erilaisten koodien kääntämisessä. Näitä eroja pyritään tuomaan esille kahden vertailtavan ohjelman avulla, joihin on sisällytetty erilaisia rakenteita.

Vertailussa käytetyt kääntäjät ovat Free Software Foundationin GCC ja LLVM:n Clang. GCC on noin 20 vuotta vanhempi kääntäjäympäristö kuin Clang, ja tämä onkin yksi motiivi vertailun tekemiselle, sillä on kiintoisaa nähdä, tuleeko ohjelman ikä jotenkin esille vertailua suorittaessa. Clang pyrkii kääntäjänä olemaan modernimpi ja yksinkertaisempi vaihtoehto yleisemmin käytetylle GCC-kääntäjälle.

Työ alkaa teoreettisen taustan läpi käymisellä, jossa aluksi vertaillaan eri ohjelmointikielten tyyppejä ja kerrotaan taustatietoja C++:sta yleisesti. Tämän jälkeen esitellään käytetyt kääntäjät, sekä tarkastellaan tarkemmin kääntöprosessia. Tutkimusmenetelmissä ja -aineistoissa kerrotaan, millaisella alustalla kääntäminen tapahtuu sekä miten kääntäminen suoritetaan eri kääntäjillä ja optimoinneilla. Seuraavassa kappaleessa esitellään käännettävien ohjelmien lähdekoodit ja kerrotaan tarkemmin niiden rakenteesta. Tämän jälkeen on tehty varsinainen vertailu, jossa tarkastellaan saatuja tuloksia eri kääntäjillä. Työn päättää johtopäätökset, joihin on tiivistetty oleellisimmat vertailussa esille tulleet erot.

2 C++-KÄÄNTÄJÄT

Ohjelmointikieliä on olemassa paljon erityyppisiä ja niitä pystyykin jaottelemaan monin eri tavoin. Matalan tason kielet ovat laitteistoläheisempiä, eivätkä ne tarjoa kovinkaan paljon abstraktiota konekieleen nähden, jota voisi kutsua alimmaksi tasoksi. Konekieli on suoraan prosessorin ymmärtämässä muodossa ja koostuu vain biteistä eli ykkösistä ja nollista, jotka muodostavat käskyjä. Seuraava taso konekielestä on assembly-kieli, jossa käskyt ovat ihmisen helpommin ymmärrettävässä tekstimuodossa. Molemmille yhteistä on se, että jokainen käsky suorittaa jonkin operaation prosessorilla ja käytettävissä olevat käskyt määräytyvät käytetyn prosessorin käskykannan mukaan. Konekieli ja assembly ovat siten laitteistoriippuvaisia, eli jollekin prosessorille kirjoitettu ohjelma ei toimi toisella prosessorilla, jos sillä on eri käskykanta. Korkeamman tason kielet tarjoavat enemmän abstraktiota matalampaan tasoon nähden, jolloin ei tarvitse tietää tarkemmin käytetystä laitteistosta ja yksi koodirivi usein vastaakin montaa eri käskyä prosessorilla. [1, s. 3-5]

Toinen luokittelutapa perustuu ohjelman suoritustapaan eli siihen, onko kieli käännettävä vai tulkattava. Käännettävät kielet joudutaan kääntämään ennen suorittamista prosessorin ymmärtämään konekieleen käyttäen ohjelmointikielen kääntäjää. Tulkattavat kielet puolestaan käyttävät tulkkia, joka kääntää ja suorittaa koodia ohjelman ajon aikana. Molemmissa suoritustavoissa on omat hyötynsä ja haittansa, mutta yksi oleellinen käännettävien kielten etu on kirjoitettujen ohjelmien nopeus. C++ on käännettävä kieli, joten tähän perustuu myöskin sen tehokkuus.

C++ on olio-orientoitunut kieli, joka on kehitetty matalamman tason C-kielestä. Kieleen on lisätty muunmuassa olio-ohjelmointiin ja geneerisyyteen liittyviä ominaisuuksia, kuten luokat, mallit ja tuki poikkeuksille. C++:n avulla pystyy luomaan hyvin tehokkaita ohjelmia, joten sitä usein käytetäänkin esimerkiksi palvelinohjelmistojen ja pelien tekemiseen, joissa suorituskyvyllä on iso merkitys. Suurin osa tyypillisistä tietokoneella käytetyistä ohjelmista onkin kirjoitettu sitä käyttäen [7, s. 2]. C++ on ISO:n (International Organization for Standardization) standardoima ja tämänhetkinen uusin versio standardista on C++14, jota tullaan myöskin käyttämään myöhemmin työssä (ks. luku 4.1).

Ensimmäinen kääntäjä C++:lle oli vuonna 1983 luotu Cfront, jonka kirjoitti Bjarne Stroustrup. Toinen merkittävä kääntäjä oli Borlandin Turbo C:n pohjalta luotu Turbo C++, jonka ensimmäinen versio on vuodelta 1990. Se oli julkaisunsa jälkeen yleisesti käytössä monen vuoden ajan ja sen pohjalta on myöskin luotu ammattikäyttöön tarkoitettu versio, Borland C++. Tuki näihin vanhoihin kääntäjiin on jo nykyisin

lopetettu, mutta Borland myi oikeudet kääntäjiinsä Embarcadero Technologiesille, joka on luonut vielä toiminnassa olevan C++ Builderin niiden pohjalta. [10]

Tällä hetkellä yleisimmin käytössä olevia kääntäjiä ovat Microsoftin Visual C++ -kääntäjä (MSVC), GNU Projectin GCC sekä LLVM Projectin Clang, joista kahta jälkimmäistä tullaan tarkastelemaan tarkemmin. Kaikki kolme tukevat uusinta C++14-standardia.

2.1 GCC

GCC, eli GNU Compiler Collection, on yksi Free Software Foundationin projekteista. Sille löytyy tuki monelta eri alustalta ja se onkin hyvin yleisesti käytetty kääntäjä. Sen kehitys on ollut jatkuvaa ja vapaaehtoisten toimin tapahtuvaa jo vuodesta 1987 lähtien, jolloin GCC julkaistiin ensimmäisen kerran. [2, s. 259]

Ensimmäisen version GCC:stä loi Richard Stallman. Tällöin kääntäjä oli luotu tukemaan vain 1980-luvun lopun prosessoreita, mutta tämän jälkeen se siirrettiin muille CISC-prosessoreille, kuten Intelin 80386:lle. Tässä vaiheessa optimoinnit olivat rajoitettuja, sillä kääntäjä kävi lähdekoodista lauseita vain yksi kerrallaan läpi, mutta tämä rajoitus poistui nopeasti ja kääntäjälle lisättiin tuki yhdistää käskyjä yhdeksi tehokkaammaksi käskyksi. Tämän jälkeen optimointeja on tehostettu keskittymällä kerrallaan yhteen metodiin, kääntöyksikköön tai jopa koko ohjelmaan. Optimointien avulla koodista saadaan nopeammin suoriutuvaa tai kompaktimpaa. [2, s. 260]

GCC-kääntäjän ensisijaisesti tuetut kielet ovat Fortran, Java, Ada, Objective C sekä tietenkin C ja C++. Näiden lisäksi sille löytyy tuki yli kolmestakymmenestä tietokonearkkitehtuurista. GNU Projectin kehitysympäristö tarjoaa kokonaisuudessaan assemblerin, linkkerin, objektitiedostotyökalut, debuggerin sekä C-kirjaston. GCC on avoimen lähdekoodin ohjelma ja se on kirjoitettu C-kielellä. [2, s. 259]

2.2 LLVM ja Clang

LLVM on projekti, johon on koottu modulaarisia ja uudelleenkäytettäviä kääntäjien sekä työkalujen teknologioita. Se on verrattavissa GCC:hen siten, että molemmat tarjoavat lukuisia kääntäjiä, jotka tukevat eri ohjelmointikieliä. LLVM:n kehitys alkoi alun perin Illinoisin yliopistossa kehitysprojektina, jossa päämääränä oli tukea sattumanvaraisia ohjelmointikieliä sekä modernia, dynaamista, käännösprosessia. [9]

Yksi tärkeimmistä LLVM:n alaprojekteista on Clang, joka tukee kieliä C, C++ sekä Objective C. Clang on selvästi uudempi kuin GCC, sillä sen ensimmäinen versio on julkaistu vuonna 2007. Sen tavoitteet ovat nopeat kääntöajat, alhainen muistin käyttö sekä yksinkertainen lähdekoodi, jota on helppo laajentaa tarvittaessa. Näiden ominaisuuksien lisäksi se pyrkii olemaan mahdollisimman informatiivinen virheistä sekä varoituksista käännösaikana, jotta viat saadaan korjattua mahdollisimman helposti. [8]

3 KÄÄNTÄMISEN VAIHEET

Kääntäminen on monivaiheinen prosessi, jonka suorittaa jollekin ohjelmointikielelle erityisesti suunniteltu kääntäjä. Prosessin seurauksena projektin lähdekooditiedostoista luodaan suoritettava ohjelma. C++:n tapauksessa kääntöprosessi sisältää neljä eri vaihetta, jotka ovat esikääntäminen, assemble, kääntäminen ja linkitys [11, s. 12].

Eri kääntäjillä voi olla erilaisia käytäntöjä siitä, miten eri vaiheet suoritetaan. Lopputuloksissa voi siis olla pieniä eroja, mutta yleensä ne eivät ole käyttäjän havaittavissa, sillä ohjelmat kuitenkin toimivat eroista huolimatta samalla tavalla. Jotta koodi olisi suorituskykyistä, kääntäjä pyrkii optimoimaan sitä parhaimman mukaan käytetylle kohdealustalle. Kääntäjästä ja valituista asetuksista riippuen optimointi voidaan suorittaa eri tavoin, tai se voidaan myöskin jättää kokonaan pois.

3.1 Esikääntäminen

Esikääntö on kääntöprosessin ensimmäinen vaihe ja sen suorittaa tekstin prosessointiin erikoistunut esikääntäjä. Lähdekooditiedoston alussa on usein `#include`-avainsanoja. Niiden tarkoitus on sisällyttää tiedostoon ulkopuolisia otsikkotiedostoja, joissa on määritelty joitain hyödyllisiä koodissa käytettyjä funktioita. Yksi esikääntäjän tehtävä on käydä nämä otsikkotiedostot läpi ja sisällyttää ne kooditiedostoon siten, että siinä on kaikki tarpeellinen tieto kääntämistä varten. [11, s. 12-13]

Toinen esikääntäjän tehtävä on `#define`-lausekkeiden määrittämien arvojen muunto vakioiksi koodissa siten, että jos koodin alussa on vaikka määritelty `#define VAKIO 2`, niin kaikki koodissa olevat avainsanat `VAKIO` vaihdetaan luvuksi 2. Ohjeilla `#if`, `#elif` ja `#endif` voidaan myöskin neuvoa kääntäjää siten, että joitain osia koodista ei oteta käännökseen mukaan jos jokin ehto täyttyy. Niitä käytetään yleisimmin otsikkotiedostoissa, kun halutaan varmistua siitä, että niiden sisältämät funktioesittelyt käydään vain kertaalleen läpi. (engl. include guard). Koodi saattaa myös sisältää joitain makroja, jotka nekin muunnetaan koodimuotoon. [11, s. 13]

Esikääntö suoritetaan jokaiselle projektin lähdekooditiedostolle, eli käännösyksikölle, joita on aina vähintään yksi kappale. Lopputuloksena saadaan yhtenäisiä, pelkkää koodia sisältäviä tiedostoja, jotka voidaan prosessoida seuraavassa vaiheessa. [11, s. 11, 12]

3.2 Assemble

Assemble-vaiheessa käydään läpi edellisen vaiheen tuotokset ja muunnetaan ne assembly-kieliseen muotoon. Ennen varsinaista muuntoa koodista karsitaan pois sen sisältämät kommentit ja muut ylimääräiset osat, jonka jälkeen sille suoritetaan analyysi. Analyysissa tarkastetaan, onko koodi syntaksin mukaista. Jos virheitä havaitaan, käyttäjälle annetaan ilmoitus ja kääntäminen jätetään kesken. Tässä vaiheessa kääntäjä voi myöskin antaa varoituksia, jotka eivät kuitenkaan estä kääntämisen viemistä loppuun. [11, s. 14]

Analyysin jälkeisessä muunnossa jokaisesta koodirivistä muodostuu käskyjä, jotka suorittavat sitä vastaavan operaation. Jotta koodi olisi suoritettavaa prosessorilla, tulee ottaa huomioon sen käskykanta. Tässä vaiheessa suoritetaan myös koodin optimoinnit, jotka pyrkivät yksinkertaistamaan tuotettua assembly-koodia ja siten tekemään siitä nopeammin suoritutuvaa. Kääntäjä voi esimerkiksi huomata, ettei joitain osia koodista suoriteta lainkaan, jolloin ne voidaan jättää kääntämättä kokonaan. Käytettyihin optimointeihin voi vaikuttaa kääntäjiin luotujen vipujen avulla, jotka annetaan ohjelmaa suorittaessa. [11, s. 15-18]

3.3 Kääntäminen

Kääntämisessä luodaan jokaisesta assembly-muotoisesta tiedostosta objektitiedosto. Jokaisella assembly-kielen käskyllä on jokin määrätty binääriarvo, joka suorittaa sitä vastaavan konekäskyn prosessorilla. Kääntäjä käy läpi kaikki käskyt ja muuntaa ne konekieleksi objektitiedoston sisälle. Käännöksen jälkeen tiedostot eivät ole enää ihmisen luettavassa tekstimuodossa. [11, s. 19]

Objektitiedostot ovat siis binäärimuotoisia tiedostoja, jotka ovat luotu yhden käännösyksikön pohjalta. Käskyjen lisäksi ne sisältävät neuvoja kääntäjälle siitä, miten niistä tulisi luoda suoritettava ohjelma. Objektitiedostot voivat myöskin olla erilaisessa muodossa eri käyttöjärjestelmien välillä. [11, s. 26]

Objektitiedostot nopeuttavat projektin kääntämistä pieniä muutoksia tehdessä. Kääntöprosessissa voidaan jättää sellaiset kooditiedostot kääntämättä uudelleen, joihin ei ole tehty mitään muutoksia, jolloin ne tarvitsee vain linkittää uudelleen.

3.4 Linkitys

Linkitys on kääntöprosessin viimeinen vaihe, jossa edellisessä vaiheessa saadut objektitiedostot yhdistetään yhdeksi tiedostoksi. Jos projektissa on käytetty joitain ulkoisia ohjelmointikirjastoja, niin myöskin niiden objektitiedostot otetaan linkitykseen mukaan. Linkityksessä objektitiedostojen sisältämät osiot siirretään omille paikoilleen ohjelmassa ja symbolit muunnetaan muistiosoitteiksi. Koodin pitää pystyä viittaamaan ohjelman eri osiin tarpeen mukaan, joten niiden välille pitää luoda tarvittavat yhteydet.

Vaihe yksinkertaistuu huomattavasti, jos projekti koostuu vain yksittäisestä objektitiedostosta. Lopputuloksena saadaan suoritettava ohjelma. [11, s. 29-30]

4 TUTKIMUSMENETELMÄT JA -AINEISTO

Kääntäjiä tullaan käyttämään suoraan komentoriviltä ja niiden vertailussa tarkastellaan kahta erityisesti tutkimusta varten luotua ohjelmaa. Käytetyt ohjelmat ovat kirjoitettu C++-kielellä, joten kääntäjistä käytetään niiden C++:aa tukevia versioita: GCC:stä (versio 6.3.1) G++:aa ja Clangista (versio 3.9.1) Clang++:aa. Testijärjestelmänä toimii kannettava tietokone, jossa on Intelin i3-prosessori, joka tukee 64-bittistä x86-käskykantaa. Tietokoneen käyttöjärjestelmä on Arch Linux.

Ohjelmien käännökset tullaan suorittamaan molemmilla kääntäjillä sekä optimointien kanssa, että ilman. Käännösvaiheet suoritetaan yksitellen vaihe kerrallaan ja niiden kesken tehdään vertailuja. Esikäännön sekä assemblen tuloksia vertaillaan suoraviivaisesti, sillä niiden tuotokset ovat tekstimuotoisia.

Binäärimuotoisten tiedostojen analysointiin olisi mahdollista käyttää objdump-nimistä työkalua, jolla voidaan takaisinmallintaa objektitiedostot ja valmiit ohjelmat takaisin luettavissa olevaan muotoon [4, s. 260]. Objektitiedostoja ei kuitenkaan suoraan vertailla keskenään, sillä ne ovat käytännössä samoja kuin assembly-kieliset versiot, mutta vain käännettynä konekieleen. Valmiita, suoritettavia ohjelmia vertaillaan niiden suoritusaikojen ja tiedostokokojen perusteella. Lisäksi koko kääntöprosessiin kuluvaa aikaa vertaillaan kääntäjien kesken ja siihen käytetään time-työkalua.

4.1 Kääntäminen

Tyypillisesti ohjelmaa käännettäessä sen lähdekoodista, kääntäjä tekee suoraan kaikki käännösvaiheet ja luo suoritettavan ohjelman. Yleisesti kääntäjiin pystyy kuitenkin käännösvaiheessa syöttämään niin kutsuttuja vipuja, joilla voi määrätä sen tekemään vain osan vaiheista. [11, s. 32-33]

Sekä G++:lla että Clang++:lla käännösvaiheiden suorittamiseen pystyy vaikuttamaan seuraavilla vivuilla:

- -E suorittaa vain esikäännön
- -S suorittaa edellisen lisäksi assembly-muunnon
- -c suorittaa edellisten lisäksi kääntämisen

Jos edellisiä vipuja ei määritä, niin kääntäjä suorittaa edellisten vaiheiden lisäksi myös linkityksen. On hyvä huomata, että G++:n vivut toimivat myöskin Clang++:ssa. Tämä johtuu siitä, että Clang pyrkii olemaan yhteensopiva GCC:n kanssa [8]. Edellä olevien vipujen lisäksi Clang++ sisältää myöskin omat, helpommin muistettavat, vivut

edellisille toimille: --preprocess, --assemble sekä --compile. Esiteltyjä vipuja käytetään hyödyksi tutkimuksessa käytettävien ohjelmien kääntämiseen vaiheittain. [5, 12]

Lisäksi, koska ohjelmat käännetään käyttäen uusinta standardia, tulee syöttää yhtenä vipuna -std=c++14, joka asettaa käytetyksi standardiksi C++14:n. Tämä toimii molemmissa käytetyissä kääntäjissä. Viimeisenä parametrina komentoa kutsuttaessa tulee määrittää tiedostot, jotka kääntäjälle syötetään. Jos siis haluttaisiin tehdä G++:lla kaikki vaiheet kääntämiseen asti ja lähdekooditiedosto olisi main.cpp, tarvittava komento olisi g++ -c -std=c++14 main.cpp. Lopputuloksena saataisiin yksittäinen binäärimuotoinen objektitiedosto. Jos haluttaisiin luoda vastaava tiedosto Clangia käyttäen tulisi vain vaihtaa kutsuttavaksi ohjelmaksi g++:n tilalle clang++. [5]

4.2 Koodin optimointi

Käytetyissä kääntäjissä on useita asetuksia erilaisille optimoinneille, mutta oletuksena ne eivät kuitenkaan ole päällä. Tämä johtuu siitä, että ilman optimointeja käännösajat ovat lyhyempiä ja tuotettuja ohjelmia pystyy analysoimaan helpommin. Usein optimoinnit otetaankin käyttöön vasta käyttäjille päätyvää, valmista ohjelmaa käännettäessä. Koodin nopeuden lisäksi voidaan myös keskittyä optimoimaan sen kokoa, tosin se ei kovinkaan usein ole oleellista. Nopeammassa koodissa on yleensä myös vähemmän käskyjä, jolloin sen kokokin on pienempi. [6, s. 9]

Sekä G++:ssa että Clangissa asetetaan halutut optimoinnit vipujen avulla. Jokaisessa korkeammassa optimointitasossa on lisättynä aina joitain uusia optimointeja edellisten lisäksi. Optimointeja voi myös ottaa käyttöön yksitellen, jos halutaan määrätä tarkemmin, mitä kääntäjä pyrkii optimoimaan. [6, s. 10]

Käytettävissä olevat tasot on esitetty taulukossa 1 ja ne toimivat samoin periaattein molemmissa kääntäjissä [12, 5, s. 93-95].

Taulukko 1. *Kääntäjien optimointitasot selityksineen*

-O0 (oletus)	Ei optimointeja. Nopein kääntöprosessi.
-O1	Ottaa vain helpoiten tehdyt optimoinnit käyttöön. Nopeuttaa koodia ja pienentää sen kokoa.
-O2	Ottaa suurimman osan optimoinneista käyttöön. Nopeuttaa ja pienentää koodia entisestään.
-O3	Ottaa kaikki nopeuden optimoinnit käyttöön. Saattaa kasvattaa koodin kokoa.
-Ofast	Edellisten lisäksi ottaa käyttöön myös mahdollisesti standardeja rikkovat optimoinnit.

-Os	Pyrkii mahdollisimman pieneen koodin kokoon, nopeuttamatta sen suoritusta.

Näistä optimoinneista työssä käytetään tasoa -O2 ja sitä verrataan optimoimattomaan tasoon -O0.

5 TUTKIMUKSESSA KÄYTETYT OHJELMAT

Tutkimuksessa käytetään kahta eri ohjelmaa, jotka sisältävät erityylisiä rakenteita ja metodeja erilaisten tutkimustulosten saamiseksi. Koodin erottelu kahteen ohjelmaan myöskin helpottaa myöhempää optimoinnin vaikutusten analyysiä, kun kaikki koodi ei ole samassa tiedostossa. Molemmista ohjelmista löytyy silmukkarakenteita. Tämä johtuu siitä, että suurin osa ohjelmien suoritusajasta vietetään silmukoissa, joten kääntäjistä löytyy usein paljon optimointeja niihin liittyen.

Ohjelmissa on pyritty välttämään kirjastojen käyttöä analysoinnin helpottamiseksi. Ainoat käytetyt kirjastot ovat aika-analyysiä helpottavan *clock()*-rutiinin sisällyttävä ctime sekä cstdio, joka sisältää suoritusajan tulostamiseen käytetyn *printf()*-funktion. Rutiinia *clock() k*äytetään siten, että ohjelman suorituksen alussa alkuaika kirjataan muistiin ja lopussa lasketaan aikaero, jonka perusteella lasketaan suoritukseen kulunut aika.

Käytettyjen ohjelmien lähdekoodit selityksineen on esitetty seuraavissa alakappaleissa. On hyvä huomata, ettei ohjelmilla ole varsinaista käyttötarkoitusta tutkimuksen ulkopuolella.

5.1 Ohjelma A, neliön laskenta

Ensimmäinen ohjelma koostuu kolmesta eri tiedostosta: *main.cpp*, *nelio.h* sekä *nelio.cpp*. Useamman tiedoston käyttö tekee kääntöprosessista hieman erilaisen ja se mahdollisesti paljastaa lisäksi eroavaisuuksia kääntäjien kesken. Ohjelman perusrakenteena toimii yksinkertainen for-silmukka, joka kutsuu toistuvasti funktiota *nelio()*. Ennen kuin ohjelma sulkeutuu, se tulostaa suoritukseen kuluneen ajan käyttäjälle.

main.cpp

```
1       #include <time.h>
2       #include <stdio.h>
3       #include "nelio.h"
4
5       // Valitaan iterointien määrä
6       #define ISO_MAARA
7
8       #ifdef ISO_MAARA
9           #define MAARA 1000
10      #else
11          #define MAARA 10
12      #endif
13
14      int main() {
15          clock_t t{clock()}; // Suorituksen alkuaika
16
17          const int iterointeja{MAARA};
18          int vakio{0};
19
20          for (int i{0}; i < iterointeja; ++i) {
21              // Pysyy samana koko suorituksen ajan
22              vakio = MAARA / 10;
23
24              if (i > MAARA) {
25                  // Osa, jota ei suoriteta koskaan ohjelmassa
26                  nelio(i - vakio);
27              }
28              else {
29                  nelio(i + vakio);
30              }
31          }
32
33          t = clock() - t; // Suoritusaika
34          printf("Suoritusaika: %fs\n", (float)t/CLOCKS_PER_SEC);
35      }
```

Koodilistaus 1: *Ohjelma A:n main.cpp-lähdekooditiedosto*

Tiedoston alussa on `#define`-lauseke, jolla voidaan valita iterointikertojen määrä. Iteraatioiden määrä on 1000 jos *ISO_MAARA* on määritelty, muuten sen arvo on 10. Silmukan sisältä löytyy kaksi kohtaa, joista löytyy helposti optimoitavia rakenteita: vakiota *vakio* lasketaan joka iterointikerralla, vaikka se pysyy samana koko suorituksen ajan ja if-lausekkeessa on kohta, joka ei koskaan päädy suoritukseen.

nelio.h

```
1       #ifndef NELIO_H
2       #define NELIO_H
3
4       // Korottaa annetun numeron neliöön ja palauttaa sen
5       int nelio(int numero);
6
7       #endif
```

Koodilistaus 2: *Ohjelma A:n nelio.h-otsikkotiedosto*

Ohjelman ainoassa otsikkotiedostostossa on funktioesittely metodille *nelio()*.

nelio.cpp

```
1       #include "nelio.h"
2
3       int nelio(int numero) {
4           return numero * numero;
5       }
```

Koodilistaus 3: *Ohjelma A:n nelio.cpp-lähdekooditiedosto*

Metodi *nelio()* koostuu vain yhdestä koodirivistä, joka korottaa parametrina annetun luvun neliöön ja palauttaa sen.

5.2 Ohjelma B, yksinkertainen lajittelualgoritmi

Toinen ohjelma sisältää ensimmäisestä poiketen vain yhden lähdekooditiedoston, joka on *main.cpp*. Ohjelmaan on kovakoodattu taulukko, joka sisältää luvut väliltä 0..99 satunnaisessa järjestyksessä. Tämän lisäksi ohjelma sisältää funktion, joka järjestää nämä luvut pienimmästä suurimpaan. Käytetty järjestelyalgoritmi on lisäyslajittelu (engl. insertion sort), joka on yksinkertainen, mutta suhteellisen tehoton. Myös ensimmäisen ohjelman tavoin tämäkin ohjelma tulostaa ajoaikansa suorituksen päätteeksi.

main.cpp

```
1       #include <time.h>
2       #include <stdio.h>
3
4       // Yksinkertainen, mutta hidas järjestelyalgoritmi
5       void jarjesta_taulukko(int taulukko[], int koko) {
6           if (koko < 2) {
7               // Jos alkioita on alle kaksi, taulukko on
8               // järjestyksessä
9               return;
10          }
11
12          int j, tmp;
13
14          // Käydään kaikki paitsi ensimmäinen taulukon alkio läpi
15          for (int i{1}; i < koko; ++i) {
16              j = i;
17              // Siirretään alkio oikealle paikalleen
18              while (taulukko[j] < taulukko[j - 1] && j > 0) {
19                  // Vaihdetaan alkioiden j ja j-1 paikkaa
20                  tmp = taulukko[j];
21                  taulukko[j] = taulukko[j - 1];
22                  taulukko[j - 1] = tmp;
23
24                  --j;
25              }
26          }
27      }
28
29      int main() {
```

```
30          clock_t t{clock()}; // Suorituksen lakuaika
31          const int taulukon_koko{100};
32
33          // Numerot väliltä 1..99 sattumanvaraisessa järjestyksessä
34          int taulukko[] = {
35              91, 24, 8,  64, 95, 5,  15, 10, 98, 53, 77, 69, 17, 83,
36              7,  96, 45, 93, 32, 22, 12, 51, 81, 74, 9,  37, 47, 25,
37              54, 86, 19, 11, 42, 2,  89, 50, 68, 27, 94, 57, 3,  56,
38              62, 87, 43, 33, 84, 41, 66, 67, 4,  52, 16, 40, 61, 1,
39              44, 65, 88, 34, 49, 20, 48, 14, 73, 71, 60, 39, 6,  92,
40              99, 59, 90, 29, 58, 72, 80, 0,  70, 18, 97, 30, 79, 63,
41              13, 46, 31, 55, 21, 78, 75, 23, 35, 82, 38, 28, 76, 36,
42              85, 26
43          };
44
45          jarjesta_taulukko(taulukko, taulukon_koko);
46
47          t = clock() - t; // Suoritusaika
48          printf("Suoritusaika: %fs\n", (float)t/CLOCKS_PER_SEC);
49      }
```

Koodilistaus 4: *Ohjelma B:n main.cpp-lähdekooditiedosto*

Järjestämiseen käytetty funktio löytyy koodista nimellä *jarjesta_taulukko()* ja siihen syötetään taulukon lisäksi sen koko. Funktiossa käytetty algoritmi toimii siten, että taulukon jokainen, paitsi ensimmäinen, alkio käydään läpi ja sitä siirretään vasemmalle, kunnes sen oikealla puolella oleva alkio on sitä suurempi. Siirtäminen tapahtuu siten, että vierekkäiset alkiot vaihtavat keskenään paikkaa, ja siinä käytetään hyväksi apumuuttujaa *tmp*.

6 TULOKSET

Kuten jo aiemmin tutkimusmenetelmien esittelyssäkin (ks. luku 5) on todettu, molemmille kääntäjille pystyy käännösvaiheessa syöttämään samoja vipuja. Ohjelmien kääntäminen kokonaan sekä vaiheittain onnistui siis samalla lailla molemmilla kääntäjillä. Kääntäjissä on kuitenkin olemassa myös niille ominaisia vipuja, jos haluttaisiin tehdä joitain erikoislaatuisempia toimintoja.

Ohjelmien suoritusajat olivat hyvin lyhyitä, mikrosekuntien luokkaa. Tämä ei ollut kovinkaan yllättävää, sillä ohjelmat olivat yksinkertaisia, eikä silmukoissa tehty mitään raskaita toimenpiteitä. Toistuvasti ohjelmia suoritettaessa havaittiin, että niiden suoritusajoissa oli vaihtelua. Tämä oli oletettavissa, sillä käytetyssä käyttöjärjestelmässä on suorituskykyä parantavia välimuisteja, jolloin samojen käskyjen suorittamiseen voi kulua eri ajokerralla eri aika. Suoritusaikojen vaihtelua kompensoitiin ajamalla ohjelmia moneen kertaan ja laskemalla saaduista ajoista keskiarvoja.

6.1 Käännösvaiheet

Esikääntäjien tuotokset vastasivat toisiaan koodillisesti, mikä olikin odotettua, sillä saman toiminnallisuuden saavuttamiseksi koodiin ei tulisi koskea juuri ollenkaan. Ainoita eroja esikäännössä olivat kääntäjäkohtaiset kommentit ja rivitys, millä ei ole lopputuloksen kannalta merkitystä.

Assemble-vaiheessa, eli missä koodi muunnetaan assembly-kieleen ja optimoidaan, esiintyi suurimmat eroavaisuudet. Tuotokset vastasivat toisiaan pääosin, mutta niistä oli havaittavissa erilaisia ratkaisuja: joitain asioita tehtiin eri tavoin ja eri järjestyksessä. Optimoinnit vaikuttivat koodin rakenteeseen paljon ja oli huomattavissa, kuinka molemmilla kääntäjillä oli omia käytäntöjään tuottaa koodia.

Compile-vaihe sujui molemmilla kääntäjillä samalla tavalla. Sen tuotoksia ei tutkittu tarkemmin, sillä vaihe käytännössä vain muuntaa assembly-kielisen muodon konekieliseksi, eikä lopputuloksessa ole eroavaisuutta. Myös linkitys onnistui samalla tavalla kääntäjästä riippumatta. Ohjelman A kääntämisessäkään eroja ei näistä vaiheista löytynyt, vaikka siinä lähdekooditiedostoja olikin useampi. Eroavaisuuksien vähäinen määrä johtuu luultavasti siitä, että Clang++ pyrkii olemaan yhteensopiva G++:n kanssa.

6.2 Ohjelma A

Kun ohjelman lähdekooditiedostot syötettiin G++:n esikääntäjään, saatiin koodi, joka on esitetty koodilistauksessa 5. Koodista on jätetty pois kirjastojen sisällyttämä osa tiedoston alusta, sillä sitä oli yli 500 riviä eikä se ole oleellista tutkimuksen kannalta.

```
1       int nelio(int numero);
2       int main() {
3           clock_t t{clock()};
4           const int iterointeja{1000};
5           int vakio{0};
6           for (int i{0}; i < iterointeja; ++i) {
7               vakio = 1000 / 10;
8               if (i > 1000) {
9                   nelio(i - vakio);
10              }
11              else {
12                  nelio(i + vakio);
13              }
14          }
15          t = clock() - t;
16          printf("Suoritusaika:%fs\n", (float)t/((__clock_t)1000000));
17      }
18      int nelio(int numero);
19      int nelio(int numero) {
20          return numero * numero;
21      }
```

Koodilistaus 5: *Ohjelma A esikäännettynä G++:lla*

Saatua koodia vertailtaessa ohjelman lähdekoodiin (ks. koodilistaukset 1-3) huomataan, että *main.cpp*:n alussa ollut `#include`-avainsana on vaihtunut metodin *nelio()* esittelyyn, koska otsikkotiedosto on sisällytetty koodiin. Alussa olleet avainsanat `#define` ja `#ifdef` ovat poistuneet ja niiden määrittelemän makron *MAARA* paikalle on vaihtunut koodiin luku 1000. Myöskin makro *CLOCKS_PER_SEC*, joka on määritelty ctime-kirjastossa, on vaihtunut *__clock_t*:ksi (rivi 16). Clang++:n esikääntäjän tuottama koodi vastasi täysin G++:n tuotosta.

Assemble-muunnosta löytyivät varsinaiset erot kääntäjien välillä. Kun tutkitaan tiedoston *main.cpp* (ks. koodilistaus 1) assembly-muotoisia versioita (liitteet A, B, C ja D) huomataan, että kääntäjät ovat asettaneet joitain käskyjä eri järjestykseen ja joissain niistä on käytetty eri rekistereitä. Joitain toiminnallisuuksia on myöskin toteutettu eri tavoin, kuten inkrementointi G++:n versiossa toteutuu käskyllä *add ebx* ja Clang++:ssa käskyllä *inc ebx*. Molempien kääntäjien luomista optimoimattomista versioista (liitteet A ja C) on havaittavissa osat koodista, joita ei koskaan suoriteta: esimerkiksi metodi *nelio()* esiintyy kahdesti. Optimoiduista versioista (liitteet C ja D) ne on kuitenkin järkevästi jätetty kokonaan pois, ja molempien kääntäjien tuotoksissa koko for-silmukka toteutuu viidellä käskyllä (rivit 11-15). Optimoimattomissa versioissa on lähes sama määrä käskyjä, mutta Clang++:n optimoitu versio sisältää hieman enemmän käskyjä verrattaen G++:n versioon.

Pienemmän lähdekooditiedoston *nelio.cpp* (ks. koodilistaus 3) assembly-versioista ei kuitenkaan kääntäjien välillä ollut eroja. Tämä johtuu luultavimmin siitä, että tiedostossa esitetty metodi *nelio() s*isältää koodia vain yhden rivin, eikä siitä muodostu käskyjä kovinkaan paljoa.

```
1       push   rbp
2       mov    rbp, rsp
3       mov    dword ptr [rbp - 4], edi
4       mov    edi, dword ptr [rbp - 4]
5       imul   edi, dword ptr [rbp - 4]
6       mov    eax, edi
7       pop    rbp
8       ret
```

Koodilistaus 6: *nelio.cpp:n assembly-muoto Clang++:lla ja G++:lla*

Metodi *nelio()* näyttää seuraavalta optimoimattomana versiona. Konekäskyjä on kahdeksan kappaletta.

```
1       imul   edi, edi
2       mov    eax, edi
3       ret
```

Koodilistaus 7: *nelio.cpp:n optimoitu assembly-muoto Clang++:lla ja G++:lla*

Optimoidussa versiossa metodi toteutetaan kolmella käskyllä, joka on viisi vähemmän optimoimattomaan versioon verrattuna. Molemmista versioista löytyy käsky *imul*, jossa suoritetaan varsinainen kertolasku, sekä *ret*, jossa palataan takaisin metodista pääohjelmaan. Koska metodia kutsutaan koodissa tuhat kertaa, vähempi konekäskyjen määrä nopeuttaa ohjelman suoritusta.

Ohjelmien suoritusajoista ei ollut havaittavissa eroja kääntäjien välillä. Molemmilla kääntäjillä optimoimattoman version suoritusaika oli noin 15µs ja optimoidun version suoritusaika 8µs. Suoritusaika oli siis melkein puolet lyhyempi optimoidussa versiossa. Yllättävää oli, että optimoidut versiot ohjelmista olivat saman kokoisia kuin optimoimattomat: G++:lla lopputuloksena saadut suoritettavat ohjelmat molemmat olivat 8488 tavua ja Clang++:lla 8416 tavua, eli ero on marginaalinen. Kääntöprosessiin kului kokonaisuudessaan G++:lla aikaa ennen optimointeja 130ms ja 140ms optimointien jälkeen, kun taas Clang++:lla kulunut aika oli aluksi 135ms ja optimointien kanssa 145ms.

6.3 Ohjelma B

G++:lla esikäännetty versio ohjelmasta B on esitetty koodilistauksessa 8. Kuten myös ohjelmassa A, siitäkin on jätetty kirjastot pois.

```
1       void jarjesta_taulukko(int taulukko[], int koko) {
2           if (koko < 2) {
3               return;
4           }
5           int j, tmp;
6           for (int i{1}; i < koko; ++i) {
7               j = i;
8               while (taulukko[j] < taulukko[j - 1] && j > 0) {
9                   tmp = taulukko[j];
10                  taulukko[j] = taulukko[j - 1];
11                  taulukko[j - 1] = tmp;
12                  --j;
13              }
14          }
15      }
16      int main() {
17          clock_t t{clock()};
18          const int taulukon_koko{100};
19          int taulukko[] = {
20              91, 24, 8, 64, 95, 5, 15, 10, 98, 53, 77, 69, 17, 83,
21              7, 96, 45, 93, 32, 22, 12, 51, 81, 74, 9, 37, 47, 25,
22              54, 86, 19, 11, 42, 2, 89, 50, 68, 27, 94, 57, 3, 56,
23              62, 87, 43, 33, 84, 41, 66, 67, 4, 52, 16, 40, 61, 1,
24              44, 65, 88, 34, 49, 20, 48, 14, 73, 71, 60, 39, 6, 92,
25              99, 59, 90, 29, 58, 72, 80, 0, 70, 18, 97, 30, 79, 63,
26              13, 46, 31, 55, 21, 78, 75, 23, 35, 82, 38, 28, 76, 36,
27              85, 26
28          };
29          jarjesta_taulukko(taulukko, taulukon_koko);
30          t = clock() - t;
31          printf("Suoritusaika:%fs\n", (float)t/((__clock_t)1000000));
32      }
```

Koodilistaus 8: *Ohjelma B esikäännettynä G++:lla*

Koska lähdekooditiedostossa (ks. koodilistaus 4) ei ole juurikaan käytetty makroja tai muitakaan avainsanoja, saatu koodi vastaa lähes täysin alkuperäistä *main.cpp*:n koodia. Ainoa oleellinen ero lähdekoodiin on, että kommentit ovat poistuneet koodista. Koodillisesti G++:n ja Clang++:n esikäännöksistä ei ollut eroja.

Sekä optimoimattomista että optimoiduista assembly-versioista (liitteet E, F, G, H) löytyy kääntäjien kesken paljon eroja. Käännöksiä tutkittaessa huomataan myös, että pienestä määrästä koodia voi muodostua suuri määrä käskyjä - metodissa *jarjesta_taulukko()* varsinaista koodia on noin kymmenen riviä, mutta esimerkiksi G++:n optimoimattomassa käännöksessä (liite E) käskyjä on noin 70 kappaletta. Verrattuna ohjelman A tuotoksiin kääntäjät tuottivat toisiinsa nähden hyvin erilaista koodia, joten ilmeisesti taulukoiden käsittely on toteutettu kääntäjissä eri tavoin. Optimoinnit vähensivät käskyjen määrää huomattavasti, sillä vaikka lähdekoodissa suorittamattomia koodirivejä ei ollut ollenkaan, molemmissa optimoiduissa versioissa

(liitteet F ja H) metodin *jarjesta_taulukko()* käskymäärä puolittui. Clang++ onnistui kuitenkin optimoimaan koodia tehokkaammin ja ero on huomattavissa tuloksista.

Ohjelmien suoritusajoissa oli huomattavissa pientä eroa kääntäjien kesken, toisin kuin ohjelman A tapauksessa. Niiden vertailu oli myöskin helpompaa, sillä suoritusajat olivat pidempiä. G++:lla suoritusaika oli aluksi noin 50µs, ja Clang++:lla 55µs. Optimoiduissa versioissa suoritusaika oli G++:lla noin 20µs ja Clang++:lla 18µs. Kuten myös ohjelmassa A, optimoimattomat ja optimoidut versiot olivat samankokoisia: G++:lla 8464 tavua ja Clang++:lla 8448 tavua. Koko kääntöprosessiin aikaa kului G++:lla 110ms ilman optimointeja ja 130ms optimointien kanssa. Clang+:lla kääntämiseen kului hieman enemmän aikaa: aluksi 115ms ilman optimointeja ja niiden jälkeen 145ms.

7 JOHTOPÄÄTÖKSET

Kääntäjät olivat keskenään yllättävän samankaltaisia varsinkin käytettävyydeltään, eikä suuria eroja löytynyt. Molemmilla kääntöprosessi onnistui samalla tavalla, samoja vipuja käyttämällä. Eniten eroja löytyi assemble-vaiheen tuotoksista ja optimoinneista, mutta myös suoritus- ja kääntöajoissa oli havaittavissa pieniä eroavaisuuksia. Myöskin tiedostoko'oissa oli hieman eroja, joskin ne olivat vain marginaalisia. Optimointia ei kuitenkaan kohdistettu tiedostokokoon, joten sen tehokkuudesta ei voida sen tarkemmin sanoa.

Ohjelman A suoritusajoissa ei kääntäjien kesken ollut eroja, mutta ohjelmassa B niitä esiintyi jonkin verran. Ilman optimointeja G++:n kääntämä versio oli nopeampi, mutta optimointien kanssa Clang++:n versio oli hieman nopeampi. Molemmilla kääntäjillä kääntöprosessiin kului aikaa reilu 100ms ja optimoinnit pidensivät aikaa 10-30ms. G++ kuitenkin osoittautui hieman nopeammaksi kaikissa kääntötilanteissa.

Vaikka optimointien käyttö ei nostanut käännösaikaa paljoa, tulee huomata, että käännetyt ohjelmat olivat hyvin pieniä. Isompia ohjelmia käännettäessä optimointiin voi kulua paljon enemmän aikaa. Koska ohjelmien kehityksen aikana suorituskyvyllä ei ole suurta merkitystä, optimointeja ei silloin kannata ottaa mukaan. Optimoinnit kuitenkin lyhensivät huomattavasti ohjelman suoritusaikaa molemmilla kääntäjillä, joten lopulliseen versioon ne kannattaa sisällyttää.

Tutkimuksessa ei paljastunut, että toinen kääntäjä olisi suoriutunut selvästi paremmin jollakin osa-alueella. Käännetyissä ohjelmissa oli kuitenkin havaittavissa eroavaisuuksia ja on hyvä huomata, että suurempia ohjelmia käännettäessä erot voivat olla merkittävämpiä. Jos siis suorituskyky on kriittistä, on hyvä kokeilla kääntää ohjelmaa eri kääntäjillä ja vertailla lopputuotoksia.

LÄHTEET

[1] S.P. Dandamudi, Guide to Assembly Language Programming in Linux, Springer Verlag, 1st. Ed, 2005, 543 p.

[2] D. Edelsohn, W. Gellerich, M. Hagog, D. Naishlos, M. Namolaru, E. Pasch, H. Penner, U. Weigand, A. Zaks, Contributions to the GNU compiler collection, IBM Systems Journal, Vol. 44, Iss. 2, 2005, pp. 259-278.

[3] W. von Hagen, Definitive Guide: The Definitive Guide to GCC, Apress, 2nd. Ed, 2006, 550 p.

[4] S. Koranne, Handbook of Open Source Tools, Springer US, 2011, 484 p.

[5] Richard M. Stallman and the GCC Developer Community, Using the GNU Compiler Collection, GNU Press, 2016, 839 p. Saatavissa: https://gcc.gnu.org/onlinedocs/

[6] P.A. Ballal, H. Sarojadevi, H.P. S, Compiler Optimization: A Genetic Algorithm Approach, International Journal of Computer Applications, Vol. 112, Iss. 10, 2015, pp. 9-13.

[7] S.R. Davis, C++ For Dummies, Wiley, 7th. Ed, 2014, 461 p.

[8] Clang – Features and Goals. Saatavissa: https://clang.llvm.org/features.html

[9] The LLVM Compiler Infrastructure Project. Saatavissa: http://llvm.org

[10] Remembering... Turbo C++, Micro Mart, Iss. 1427, 2016, pp. 34.

[11] M. Stevanovic, Advanced C and C++ Compiling, Apress, 2014, 314 p.

[12] Clang command line argument reference. Saatavissa: https://clang.llvm.org/docs/ClangCommandLineReference.html

LIITE A: OHJELMA A, G++, ASSEMBLE-MUUNTO, EI OPTIMOINTEJA

```
1       .LC1:
2       .string      "Suoritusaika: %fs\n"
3       main:
4       .LFB0:
5       push   rbp
6       mov    rbp, rsp
7       sub    rsp, 32
8       call   clock
9       mov    QWORD PTR [rbp-16], rax    # t, tmp95
10      mov    DWORD PTR [rbp-20], 1000   # iterointeja,
11      mov    DWORD PTR [rbp-24], 0      # vakio,
12      mov    DWORD PTR [rbp-4], 0       # i,
13      .L5:
14      cmp    DWORD PTR [rbp-4], 999     # i,
15      jg     .L2
16      mov    DWORD PTR [rbp-24], 100    # vakio,
17      cmp    DWORD PTR [rbp-4], 1000    # i,
18      jle    .L3
19      mov    eax, DWORD PTR [rbp-4]     # tmp96, i
20      sub    eax, DWORD PTR [rbp-24]    # _11, vakio
21      mov    edi, eax      #, _11
22      call   _Z5nelioi
23      jmp    .L4
24      .L3:
25      mov    edx, DWORD PTR [rbp-4]     # tmp97, i
26      mov    eax, DWORD PTR [rbp-24]    # tmp98, vakio
27      add    eax, edx      # _13, tmp97
28      mov    edi, eax      #, _13
29      call   _Z5nelioi
30      .L4:
31      add    DWORD PTR [rbp-4], 1       # i,
32      jmp    .L5
33      .L2:
34      call   clock
35      sub    rax, QWORD PTR [rbp-16]    # tmp100, t
36      mov    QWORD PTR [rbp-16], rax    # t, tmp100
37      pxor   xmm0, xmm0    # _19
38      cvtsi2ssq     xmm0, QWORD PTR [rbp-16]   # _19, t
39      movss  xmm1, DWORD PTR .LC0[rip]  # tmp101,
40      divss  xmm0, xmm1    # _20, tmp101
41      cvtss2sd      xmm0, xmm0    # _21, _20
42      mov    edi, OFFSET FLAT:.LC1
43      mov    eax, 1
44      call   printf
45      mov    eax, 0 # _23,
46      leave
47      ret
48      .LC0:
49      .long  1232348160
```

LIITE B: OHJELMA A, G++, ASSEMBLE-MUUNTO, OPTIMOITU

```
1       .LC1:
2       .string      "Suoritusaika: %fs\n"
3       main:
4       push   rbp
5       push   rbx
6       mov    ebx, 100      # ivtmp.9,
7       sub    rsp, 8
8       call   clock
9       mov    rbp, rax      # t,
10      .L2:
11      mov    edi, ebx      #, ivtmp.9
12      add    ebx, 1 # ivtmp.9,
13      call   _Z5nelioi
14      cmp    ebx, 1100     # ivtmp.9,
15      jne    .L2
16      call   clock
17      pxor   xmm0, xmm0    # tmp100
18      sub    rax, rbp      # t, t
19      mov    edi, OFFSET FLAT:.LC1
20      cvtsi2ssq     xmm0, rax     # tmp100, t
21      mov    eax, 1
22      divss  xmm0, DWORD PTR .LC0[rip]  # tmp101,
23      cvtss2sd      xmm0, xmm0    # tmp103, tmp101
24      call   printf
25      add    rsp, 8
26      xor    eax, eax
27      pop    rbx
28      pop    rbp
29      ret
30      .LC0:
31      .long  1232348160
```

LIITE C: OHJELMA A, CLANG++, ASSEMBLE-MUUNTO, EI OPTIMOINTEJA

```
1       .LCPI0_0:
2       .long  1232348160               # float 1.0E+6
3       main:                                   # @main
4       push   rbp
5       mov    rbp, rsp
6       sub    rsp, 48
7       mov    dword ptr [rbp - 4], 0
8       call   clock
9       mov    qword ptr [rbp - 16], rax
10      mov    dword ptr [rbp - 20], 1000
11      mov    dword ptr [rbp - 24], 0
12      mov    dword ptr [rbp - 28], 0
13      .LBB0_1:
14      cmp    dword ptr [rbp - 28], 1000
15      jge    .LBB0_7
16      mov    dword ptr [rbp - 24], 100
17      cmp    dword ptr [rbp - 28], 1000
18      jle    .LBB0_4
19      mov    eax, dword ptr [rbp - 28]
20      sub    eax, dword ptr [rbp - 24]
21      mov    edi, eax
22      call   _Z5nelioi
23      mov    dword ptr [rbp - 32], eax # 4-byte Spill
24      jmp    .LBB0_5
25      .LBB0_4:
26      mov    eax, dword ptr [rbp - 28]
27      add    eax, dword ptr [rbp - 24]
28      mov    edi, eax
29      call   _Z5nelioi
30      mov    dword ptr [rbp - 36], eax # 4-byte Spill
31      .LBB0_5:
32      jmp    .LBB0_6
33      .LBB0_6:
34      mov    eax, dword ptr [rbp - 28]
35      add    eax, 1
36      mov    dword ptr [rbp - 28], eax
37      jmp    .LBB0_1
38      .LBB0_7:
39      call   clock
40      movabs rdi, .L.str
41      movss  xmm0, dword ptr [rip + .LCPI0_0]
42      sub    rax, qword ptr [rbp - 16]
43      mov    qword ptr [rbp - 16], rax
44      cvtsi2ss     xmm1, qword ptr [rbp - 16]
45      divss  xmm1, xmm0
46      cvtss2sd     xmm0, xmm1
47      mov    al, 1
48      call   printf
49      mov    ecx, dword ptr [rbp - 4]
50      mov    dword ptr [rbp - 40], eax # 4-byte Spill
```

```
51      mov    eax, ecx
52      add    rsp, 48
53      pop    rbp
54      ret
55      .L.str:
56      .asciz "Suoritusaika: %fs\n"
```

LIITE D: OHJELMA A, CLANG++, ASSEMBLE-MUUNTO, OPTIMOITU

```
1       .LCPI0_0:
2       .long  1232348160               # float 1.0E+6
3       main:                                   # @main
4       push   rbp
5       push   rbx
6       push   rax
7       mov    ebp, 100
8       call   clock
9       mov    rbx, rax
10      .LBB0_1:
11      mov    edi, ebp
12      call   _Z5nelioi
13      inc    ebp
14      cmp    ebp, 1100
15      jne    .LBB0_1
16      call   clock
17      sub    rax, rbx
18      cvtsi2ss      xmm0, rax
19      divss  xmm0, dword ptr [rip + .LCPI0_0]
20      cvtss2sd      xmm0, xmm0
21      mov    edi, .L.str
22      mov    al, 1
23      call   printf
24      xor    eax, eax
25      add    rsp, 8
26      pop    rbx
27      pop    rbp
28      ret
29      .L.str:
30      .asciz "Suoritusaika: %fs\n"
```

LIITE E: OHJELMA B, G++, ASSEMBLE-MUUNTO, EI OPTIMOINTEJA

```
1       _Z17jarjesta_taulukkoPii:
2       .LFB0:
3       push   rbp
4       mov    rbp, rsp
5       mov    QWORD PTR [rbp-24], rdi    # taulukko, taulukko
6       mov    DWORD PTR [rbp-28], esi    # koko, koko
7       cmp    DWORD PTR [rbp-28], 1      # koko,
8       jle    .L7
9       mov    DWORD PTR [rbp-8], 1       # i,
10      .L6:
11      mov    eax, DWORD PTR [rbp-8]     # tmp111, i
12      cmp    eax, DWORD PTR [rbp-28]    # tmp111, koko
13      jge    .L1
14      mov    eax, DWORD PTR [rbp-8]     # tmp112, i
15      mov    DWORD PTR [rbp-4], eax     # j, tmp112
16      .L5:
17      mov    eax, DWORD PTR [rbp-4]     # tmp113, j
18      cdqe
19      lea    rdx, [0+rax*4]       # _11,
20      mov    rax, QWORD PTR [rbp-24]    # tmp114, taulukko
21      add    rax, rdx     # _13, _11
22      mov    edx, DWORD PTR [rax]       # _14, *_13
23      mov    eax, DWORD PTR [rbp-4]     # tmp115, j
24      cdqe
25      sal    rax, 2 # _16,
26      lea    rcx, [rax-4]  # _17,
27      mov    rax, QWORD PTR [rbp-24]    # tmp116, taulukko
28      add    rax, rcx     # _18, _17
29      mov    eax, DWORD PTR [rax]       # _19, *_18
30      cmp    edx, eax     # _14, _19
31      jge    .L4
32      cmp    DWORD PTR [rbp-4], 0       # j,
33      jle    .L4
34      mov    eax, DWORD PTR [rbp-4]     # tmp117, j
35      cdqe
36      lea    rdx, [0+rax*4]       # _21,
37      mov    rax, QWORD PTR [rbp-24]    # tmp118, taulukko
38      add    rax, rdx     # _22, _21
39      mov    eax, DWORD PTR [rax]       # tmp119, *_22
40      mov    DWORD PTR [rbp-12], eax    # tmp, tmp119
41      mov    eax, DWORD PTR [rbp-4]     # tmp120, j
42      cdqe
43      lea    rdx, [0+rax*4]       # _25,
44      mov    rax, QWORD PTR [rbp-24]    # tmp121, taulukko
45      add    rdx, rax     # _26, tmp121
46      mov    eax, DWORD PTR [rbp-4]     # tmp122, j
47      cdqe
48      sal    rax, 2 # _28,
49      lea    rcx, [rax-4]  # _29,
50      mov    rax, QWORD PTR [rbp-24]    # tmp123, taulukko
51      add    rax, rcx     # _30, _29
52      mov    eax, DWORD PTR [rax]       # _31, *_30
53      mov    DWORD PTR [rdx], eax       # *_26, _31
```

```
54      mov    eax, DWORD PTR [rbp-4]     # tmp124, j
55      cdqe
56      sal    rax, 2 # _34,
57      lea    rdx, [rax-4]  # _35,
58      mov    rax, QWORD PTR [rbp-24]    # tmp125, taulukko
59      add    rdx, rax      # _36, tmp125
60      mov    eax, DWORD PTR [rbp-12]    # tmp126, tmp
61      mov    DWORD PTR [rdx], eax       # *_36, tmp126
62      sub    DWORD PTR [rbp-4], 1       # j,
63      jmp    .L5
64      .L4:
65      add    DWORD PTR [rbp-8], 1       # i,
66      jmp    .L6
67      .L7:
68      nop
69      .L1:
70      pop    rbp
71      ret
72      .LC2:
73      .string       "Suoritusaika: %fs\n"
74      .LC0:
75      .long  91
76      .long  24
77      .long  8
78      <... taulukon arvoja ...>
79      .long  36
80      .long  85
81      .long  26
82      main:
83      .LFB1:
84      push   rbp
85      mov    rbp, rsp
86      sub    rsp, 416
87      call   clock
88      mov    QWORD PTR [rbp-8], rax     # t, tmp93
89      mov    DWORD PTR [rbp-12], 100    # taulukon_koko,
90      lea    rax, [rbp-416]      # tmp94,
91      mov    esi, OFFSET FLAT:.LC0      # tmp95,
92      mov    edx, 50       # tmp96,
93      mov    rdi, rax      # tmp94, tmp94
94      mov    rcx, rdx      # tmp96, tmp96
95      rep movsq
96      lea    rax, [rbp-416]      # tmp97,
97      mov    esi, 100
98      mov    rdi, rax      #, tmp97
99      call   _Z17jarjesta_taulukkoPii
100     call   clock
101     sub    rax, QWORD PTR [rbp-8]     # tmp99, t
102     mov    QWORD PTR [rbp-8], rax     # t, tmp99
103     pxor   xmm0, xmm0    # _10
104     cvtsi2ssq     xmm0, QWORD PTR [rbp-8]    # _10, t
105     movss  xmm1, DWORD PTR .LC1[rip]  # tmp100,
106     divss  xmm0, xmm1    # _11, tmp100
107     cvtss2sd      xmm0, xmm0    # _12, _11
108     mov    edi, OFFSET FLAT:.LC2
109     mov    eax, 1
110     call   printf
111     mov    eax, 0 # _15,
112     leave
113     ret
114     .LC1:
115     .long  1232348160
```

LIITE F: OHJELMA B, G++, ASSEMBLE-MUUNTO, OPTIMOITU

```
1       _Z17jarjesta_taulukkoPii:
2       .LFB12:
3       cmp    esi, 1 # koko,
4       lea    rdx, [rdi+4]  # ivtmp.24,
5       mov    r10d, 1       # i,
6       jle    .L1
7       .L8:
8       mov    ecx, DWORD PTR [rdx]
9       mov    edi, DWORD PTR [rdx-4]
10      xor    eax, eax      # ivtmp.7
11      lea    r9, [rdx-4]   # _46,
12      mov    r8d, r10d     # j, i
13      cmp    edi, ecx      # _16, _12
14      jg     .L9
15      jmp    .L6
16      .L11:
17      test   r8d, r8d      # j
18      je     .L6
19      .L9:
20      mov    DWORD PTR [rdx+rax], edi
21      mov    DWORD PTR [r9+rax], ecx
22      sub    r8d, 1 # j,
23      mov    ecx, DWORD PTR [rdx-4+rax]
24      mov    edi, DWORD PTR [rdx-8+rax]
25      sub    rax, 4 # ivtmp.7,
26      cmp    ecx, edi      # _12, _16
27      jl     .L11
28      .L6:
29      add    r10d, 1       # i,
30      add    rdx, 4 # ivtmp.24,
31      cmp    esi, r10d     # koko, i
32      jne    .L8
33      rep ret
34      .L1:
35      rep ret
36      .LC2:
37      .string       "Suoritusaika: %fs\n"
38      .LC0:
39      .long  91
40      .long  24
41      .long  8
42      <... taulukon arvoja ...>
43      .long  36
44      .long  85
45      .long  26
46      main:
47      .LFB13:
48      push   rbx
49      sub    rsp, 400
50      call   clock  #
```

```
51      mov    rdi, rsp      # tmp118,
52      mov    esi, OFFSET FLAT:.LC0      # tmp119,
53      mov    ecx, 50       # tmp120,
54      rep movsq
55      lea    rdx, [rsp+4]  # ivtmp.50,
56      mov    rbx, rax      # t,
57      mov    r9d, 1 # i,
58      .L15:
59      mov    ecx, DWORD PTR [rdx]
60      mov    esi, DWORD PTR [rdx-4]
61      xor    eax, eax      # ivtmp.32
62      lea    r8, [rdx-4]   # _51,
63      mov    edi, r9d      # j, i
64      cmp    esi, ecx      # _30, _19
65      jg     .L18
66      jmp    .L16
67      .L21:
68      cmp    ecx, esi      # _19, _30
69      jge    .L16
70      .L18:
71      mov    DWORD PTR [rdx+rax], esi
72      mov    DWORD PTR [r8+rax], ecx
73      mov    ecx, DWORD PTR [rdx-4+rax]
74      mov    esi, DWORD PTR [rdx-8+rax]
75      sub    rax, 4 # ivtmp.32,
76      sub    edi, 1 # j,
77      jne    .L21
78      .L16:
79      add    r9d, 1 # i,
80      add    rdx, 4 # ivtmp.50,
81      cmp    r9d, 100      # i,
82      jne    .L15
83      call   clock
84      pxor   xmm0, xmm0    # tmp127
85      sub    rax, rbx      # t, t
86      mov    edi, OFFSET FLAT:.LC2
87      cvtsi2ssq     xmm0, rax     # tmp127, t
88      mov    eax, 1
89      divss  xmm0, DWORD PTR .LC1[rip]  # tmp128,
90      cvtss2sd      xmm0, xmm0    # tmp130, tmp128
91      call   printf
92      add    rsp, 400
93      xor    eax, eax
94      pop    rbx
95      ret
96      .LC1:
97      .long  1232348160
```

LIITE G: OHJELMA B, CLANG++, ASSEMBLE-MUUNTO, EI OPTIMOINTEJA

```
1       _Z17jarjesta_taulukkoPii:
2       push   rbp
3       mov    rbp, rsp
4       mov    qword ptr [rbp - 8], rdi
5       mov    dword ptr [rbp - 12], esi
6       cmp    dword ptr [rbp - 12], 2
7       jge    .LBB0_2
8       jmp    .LBB0_11
9       .LBB0_2:
10      mov    dword ptr [rbp - 24], 1
11      .LBB0_3:
12      mov    eax, dword ptr [rbp - 24]
13      cmp    eax, dword ptr [rbp - 12]
14      jge    .LBB0_11
15      mov    eax, dword ptr [rbp - 24]
16      mov    dword ptr [rbp - 16], eax
17      .LBB0_5:
18      xor    eax, eax
19      mov    cl, al
20      movsxd rdx, dword ptr [rbp - 16]
21      mov    rsi, qword ptr [rbp - 8]
22      mov    eax, dword ptr [rsi + 4*rdx]
23      mov    edi, dword ptr [rbp - 16]
24      sub    edi, 1
25      movsxd rdx, edi
26      mov    rsi, qword ptr [rbp - 8]
27      cmp    eax, dword ptr [rsi + 4*rdx]
28      mov    byte ptr [rbp - 25], cl # 1-byte Spill
29      jge    .LBB0_7
30      cmp    dword ptr [rbp - 16], 0
31      setg   al
32      mov    byte ptr [rbp - 25], al # 1-byte Spill
33      .LBB0_7:
34      mov    al, byte ptr [rbp - 25] # 1-byte Reload
35      test   al, 1
36      jne    .LBB0_8
37      jmp    .LBB0_9
38      .LBB0_8:
39      movsxd rax, dword ptr [rbp - 16]
40      mov    rcx, qword ptr [rbp - 8]
41      mov    edx, dword ptr [rcx + 4*rax]
42      mov    dword ptr [rbp - 20], edx
43      mov    edx, dword ptr [rbp - 16]
44      sub    edx, 1
45      movsxd rax, edx
46      mov    rcx, qword ptr [rbp - 8]
47      mov    edx, dword ptr [rcx + 4*rax]
48      movsxd rax, dword ptr [rbp - 16]
49      mov    rcx, qword ptr [rbp - 8]
50      mov    dword ptr [rcx + 4*rax], edx
```

```
51      mov    edx, dword ptr [rbp - 20]
52      mov    esi, dword ptr [rbp - 16]
53      sub    esi, 1
54      movsxd rax, esi
55      mov    rcx, qword ptr [rbp - 8]
56      mov    dword ptr [rcx + 4*rax], edx
57      mov    edx, dword ptr [rbp - 16]
58      add    edx, -1
59      mov    dword ptr [rbp - 16], edx
60      jmp    .LBB0_5
61      .LBB0_9:
62      jmp    .LBB0_10
63      .LBB0_10:
64      mov    eax, dword ptr [rbp - 24]
65      add    eax, 1
66      mov    dword ptr [rbp - 24], eax
67      jmp    .LBB0_3
68      .LBB0_11:
69      pop    rbp
70      ret
71      .LCPI1_0:
72      .long  1232348160              # float 1.0E+6
73      main:                                  # @main
74      push   rbp
75      mov    rbp, rsp
76      sub    rsp, 448
77      call   clock
78      mov    esi, 100
79      lea    rcx, [rbp - 416]
80      movabs rdx, .L_ZZ4mainE8taulukko
81      mov    edi, 400
82      mov    r8d, edi
83      mov    qword ptr [rbp - 8], rax
84      mov    dword ptr [rbp - 12], 100
85      mov    rax, rcx
86      mov    rdi, rax
87      mov    dword ptr [rbp - 420], esi # 4-byte Spill
88      mov    rsi, rdx
89      mov    rdx, r8
90      mov    qword ptr [rbp - 432], rcx # 8-byte Spill
91      call   memcpy
92      mov    rdi, qword ptr [rbp - 432] # 8-byte Reload
93      mov    esi, dword ptr [rbp - 420] # 4-byte Reload
94      call   _Z17jarjesta_taulukkoPii
95      call   clock
96      movabs rdi, .L.str
97      movss  xmm0, dword ptr [rip + .LCPI1_0]
98      sub    rax, qword ptr [rbp - 8]
99      mov    qword ptr [rbp - 8], rax
100     cvtsi2ss     xmm1, qword ptr [rbp - 8]
101     divss  xmm1, xmm0
102     cvtss2sd     xmm0, xmm1
103     mov    al, 1
104     call   printf
105     xor    esi, esi
106     mov    dword ptr [rbp - 436], eax # 4-byte Spill
107     mov    eax, esi
108     add    rsp, 448
109     pop    rbp
110     ret
```

```
111     .L_ZZ4mainE8taulukko:
112     .long  91                       # 0x5b
113     .long  24                       # 0x18
114     .long  8                        # 0x8
115     <... taulukon arvoja ...>
116     .long  36                       # 0x24
117     .long  85                       # 0x55
118     .long  26                       # 0x1a
119     .L.str:
120     .asciz "Suoritusaika: %fs\n"
```

LIITE H: OHJELMA B, CLANG++, ASSEMBLE-MUUNTO, OPTIMOITU

```
1       _Z17jarjesta_taulukkoPii:
2       cmp    esi, 2
3       jl     .LBB0_7
4       dec    esi
5       xor    r8d, r8d
6       mov    r9d, 1
7       .LBB0_2:
8       mov    edx, dword ptr [rdi + 4*r9]
9       mov    ecx, dword ptr [rdi + 4*r9 - 4]
10      cmp    edx, ecx
11      jge    .LBB0_6
12      mov    rax, r8
13      .LBB0_4:
14      mov    dword ptr [rdi + 4*rax + 4], ecx
15      mov    dword ptr [rdi + 4*rax], edx
16      test   rax, rax
17      jle    .LBB0_6
18      mov    ecx, dword ptr [rdi + 4*rax - 4]
19      dec    rax
20      cmp    edx, ecx
21      jl     .LBB0_4
22      .LBB0_6:
23      inc    r9
24      inc    r8
25      cmp    r8d, esi
26      jne    .LBB0_2
27      .LBB0_7:
28      ret
29      .LCPI1_0:
30      .long  1232348160              # float 1.0E+6
31      main:                                   # @main
32      push   rbx
33      sub    rsp, 400
34      call   clock
35      mov    rbx, rax
36      lea    rdi, [rsp]
37      mov    esi, .L_ZZ4mainE8taulukko
38      mov    edx, 400
39      call   memcpy
40      xor    eax, eax
41      mov    ecx, 1
42      .LBB1_1:
43      mov    edx, dword ptr [rsp + 4*rcx]
44      mov    edi, dword ptr [rsp + 4*rcx - 4]
45      cmp    edx, edi
46      jge    .LBB1_5
47      mov    rsi, rax
48      .LBB1_3:
49      mov    dword ptr [rsp + 4*rsi + 4], edi
50      mov    dword ptr [rsp + 4*rsi], edx
51      test   rsi, rsi
52      jle    .LBB1_5
53      mov    edi, dword ptr [rsp + 4*rsi - 4]
```

```
54      dec    rsi
55      cmp    edx, edi
56      jl     .LBB1_3
57      .LBB1_5:
58      inc    rcx
59      inc    rax
60      cmp    rax, 99
61      jne    .LBB1_1
62      call   clock
63      sub    rax, rbx
64      cvtsi2ss      xmm0, rax
65      divss  xmm0, dword ptr [rip + .LCPI1_0]
66      cvtss2sd      xmm0, xmm0
67      mov    edi, .L.str
68      mov    al, 1
69      call   printf
70      xor    eax, eax
71      add    rsp, 400
72      pop    rbx
73      ret
74      .L_ZZ4mainE8taulukko:
75      .long  91                      # 0x5b
76      .long  24                      # 0x18
77      .long  8                       # 0x8
78      <... taulukon arvoja ...>
79      .long  36                      # 0x24
80      .long  85                      # 0x55
81      .long  26                      # 0x1a
82      .L.str:
83      .asciz "Suoritusaika: %fs\n"
```