Henri Kumpulainen

Kääntäjäoptimointi ja C++20 [[likely]]- ja [[unlikely]]-attribuutit

```
constexpr double pow(double x, long long n) noexcept {
    if (n > 0) [[likely]]
        return x * pow(x, n - 1);
    else [[unlikely]]
        return 1;
}
constexpr long long fact(long long n) noexcept {
    if (n > 1) [[likely]]
        return n * fact(n - 1);
    else [[unlikely]]
        return 1;
}
constexpr double cos(double x) noexcept {
    constexpr long long precision{16LL};
    double y{};
    for (auto n{0LL}; n < precision; n += 2LL) {
        [[likely]] y += pow(x, n) / (n & 2LL ? -fact(n) : fact(n));
    }
    return y;
}
```

Insinööri
Tieto- ja viestintätekniikka
Kesä / Syksy 2021

Tiivistelmä

Tekijä: Kumpulainen Henri

Työn nimi: Kääntäjäoptimointi ja C++20 [[likely]]- ja [[unlikely]]-attribuutit

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: c++20, optimointi, haarautumisen ennakointi, kääntäjämerkintä

Tässä työssä tutustuttiin C++20-standardin [[likely]]- ja [[unlikely]]-kääntäjäattribuutteihin ja niiden suorituskykyvaikutuksiin sekä niihin liittyviin teoria-aiheisiin, kuten kääntäjäoptimointiin ja C++:aan. C++:aa ohjelmointikielenä käsiteltiin selvittäen lyhyesti sen ominaisuuksia, vahvuuksia ja historiaa. Myös erityisesti C++20-standardin sisältöä ja kielen soveltuvuutta optimointiin tarkasteltiin.

Työssä käsiteltiin optimoinnin perusteita yleisistä käsitteistä ja periaatteista lähtien niin ohjelmakoodin itsensä optimoinnin kuin erityisesti kääntäjäoptimoinnin olennaisten tekniikoiden osalta. Prosessorien haarautumisen ennakoinnin historiaan ja eri tekniikoihin perehdyttiin. Kääntäjäoptimoinnin ja haarautumisen ennakoinnin kohdalla työssä selvitettiin myös, miten [[likely]]- ja [[unlikely]]-attribuutit hyödyntävät näitä ominaisuuksia esimerkiksi ohjelmakoodin järjestystä muuttamalla.

Opinnäytetyössä tutustuttiin attribuutteihin lähtien aiemmista verrokeista ja taustalla olevista ominaisuuksista, kuten builtin_expect. Attribuuttien toimintaan tutustuttiin käytännössä GCC:tä käyttäen todentaen samalla teoriaosuuksissa ilmaistuja liitoksia taustalla oleviin tekniikoihin.

Työssä tehtiin mittauksia attribuuttien suorituskykyvaikutuksista käyttäen yksinkertaista automatisoitua kehystä, jolla muutoksien ja lisäyksien tekeminen on yksinkertaista. Mittauksia tehtiin neljä, joissa verrattiin kolmea eri ohjelmaa: ilman attribuutteja ja attribuutit molemmin eri päin. Kahdessa mittauksista käytössä oli lisäksi ehtolauseiden todennäköisyyden muuttamista eri tasoille, joista jokaisella tehtiin samat mittaukset.

Tuloksista huomattiin, että attribuuteilla on mahdollista saada huomattaviakin vaikutuksia, mutta suurimassa osassa testeistä muutoksia ei ole havaittavissa. Syy tuloksille – ja niiden eroavaisuuksille verrokkituloksiin kuten standardiehdotukseen verrattuna – analysoitiin löytyvän todennäköisesti prosessorien ja kääntäjien kehityksestä ja/tai eroista. Attribuuteista todettiin, että käyttö voi tarjota mahdollisuuksia joissakin tilanteissa, mutta olennaiseksi osaksi C++-optimointia niistä ei liene.

Abstract

Author: Kumpulainen Henri

Title of the Publication: Compiler Optimization and the C++20 [[likely]] and [[unlikely]] Attributes

Degree Title: Bachelor of Engineering, Information and Communication Technology

Keywords: c++20, optimization, branch prediction, compiler flag

This work scrutinizes the C++20-standard [[likely]] and [[unlikely]] compiler attributes and their performance effects, as well as related theory topics, such as compiler optimization and C++. C++ as a programming language, its features, strengths and history was researched. Contents of the C++20-standard and aptitude of C++ for optimization were covered.

Fundamentals of optimization were explored, both regarding optimization of the code itself and especially the essential techniques of compiler optimization. History and different types of processor branch prediction were analyzed. How [[likely]] and [[unlikely]] flags utilize features of compiler optimization and branch prediction was also examined.

The attributes themselves were inspected, starting from earlier similar features like builtin_expect. The functionality of the attributes was examined using GCC, confirming some of the connections suggested earlier when covering the related techniques.

Measurements on the performance effects of the attributes were executed, using simple automatized framework which makes changes and additions to the tests straightforward. Four measurements were carried out, with each one of them comparing three programs: one with no attributes, and one each for the two ways to use the attributes in each case. Additionally, two of the measurements had changes in the branch probability level, with each level going through the same three tests.

The results indicated that it was possible to achieve noticeable effects – both positive and negative – with the attributes. In most cases, however, there were no changes. The reason for these results – and their difference compared to the respective results elsewhere, such as the standard proposal – was deemed to likely stem from advancements and/or changes in processor and compiler architecture. The verdict was that the attributes may offer possibilities in some situations, but it is unlikely that they will be important features in optimization of C++.

Sisällys

1 Johdanto 1

2 C++ 2

2.1 C++ optimoinnissa 3

2.2 C++20 3

3 Optimointi 5

3.1 Ohjelman optimointi 8

3.1.1 Muistinhallinta 9

3.1.2 Tiedostojen ja datan käsittely 10

3.1.3 Iteraatiot ja ehdot 11

3.2 Kääntäjäoptimointi 11

3.2.1 Funktioiden tekeminen avoimiksi (prosessien integrointi) 13

3.2.2 Silmukoiden transformaatiot 13

3.2.3 Toistuvien alilausekkeiden eliminointi 14

3.2.4 Vakioiden taittaminen 15

3.2.5 Kuolleen koodin eliminointi 16

3.2.6 Ohjelmakoodin siirtäminen 16

3.2.7 Lujuuden vähentäminen 17

3.2.8 Kurkistusaukko-optimointi (ikkunaoptimointi) 17

3.2.9 Manuaalinen kääntäjäoptimointi 18

4 Prosessorien haarautumisen ennakointi 19

4.1 Haarautumisen ennakoinnin historia 21

4.2 Haarautumisen ennakoinnin tekniikoita ja tyyppejä 22

5 [[likely]]- ja [[unlikely]]-merkinnät 27

5.1 Vastineet 27

5.2 Toiminta 29

6 Mittaukset 43

6.1 Mittaus 1 46

6.2 Mittaus 2 47

6.3 Mittaus 3 49

6.4 Mittaus 4 51

6.5 Tulosten yhteenveto ja analysointi 52

Lähteet 55

Liitteet

1 Johdanto

Tässä työssä tarkastellaan C++20-standardin [[likely]]- ja [[unlikely]]-kääntäjäattribuutteja ja niiden suorituskykyvaikutuksia. Työssä käydään läpi myös niihin liittyvien teoria-aiheiden kuten kääntäjäoptimoinnin ja C++:n perusteita. Näin työ soveltuu myös optimointia ennestään tekemättömille.

Kappaleessa 2 tutustutaan lyhyesti C++:aan ohjelmointikielenä, sen ominaisuuksiin, vahvuuksiin ja historiaan sekä erityisesti C++20-standardin sisältöön ja kielen soveltuvuuteen optimointia ajatellen.

Kappaleessa 3 käydään läpi optimoinnin perusteita lähtien liikkeelle yleisistä käsitteistä ja periaatteista sekä tarpeesta optimoinnille. Ohjelmakoodin itsensä optimointiin liittyviä huomioita selostetaan ja kääntäjäoptimoinnin olennaiset tekniikat käydään läpi.

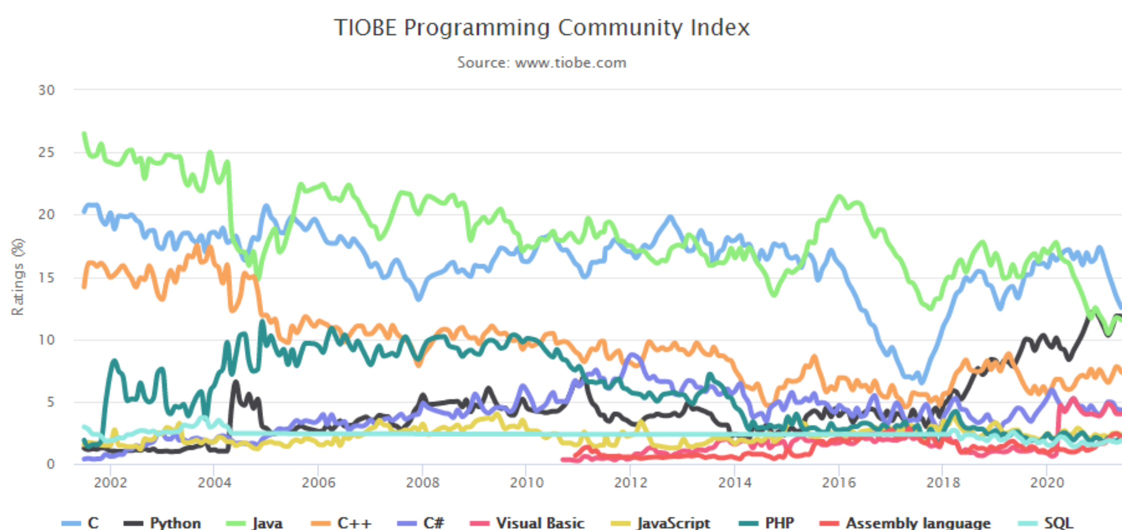
Kappaleessa 4 tutustutaan prosessorien haarautumisen ennakointiin. Erilaisia tekniikoita ja tyyppejä tarkastellaan ja ominaisuuden historiaan otetaan katsaus.

Kappaleessa 5 aiheena ovat C++20 [[likely]]- ja [[unlikely]]-attribuutit itse. Ominaisuuden aiempiin vastineisiin tutustutaan ja attribuuttien toimintaperiaatteita tarkastellaan käyttäen GCC:tä kääntäjäesimerkkinä.

Kappaleessa 6 mitataan attribuuttien suorituskykyvaikutuksista. Mittaukset tehdään automatisoidulla kehyksellä, joka yksinkertaistaa muutosten tai lisäysten tekemisen testeihin, tapausten ja toistojen määriin sekä käytettävään ajoympäristöön tai kääntäjään. Mittauksien tuloksista analysoidaan attribuuttien vaikutuksia ja siten käytettävyyttä.

2 C++

C++ on tanskalaisen Bjarne Stroustrupin alun perin kehittämä yleiskäyttöinen ohjelmointikieli, joka on suunniteltu tarjoamaan sekä joustavuutta että suorituskykyä [1]. Kieli lähti liikkeelle tavoitteesta yhdistää C-kielen tehokkuus ja Simulan tarjoamat mahdollisuudet olio- ja geneeriseen ohjelmointiin [2]. 1990-luvulla standardimallikirjasto mullisti kielen ja on sittemmin ollut yksi kielen monista määrittelevistä piirteistä erityisesti data-abstraktion ja geneerisyyden osalta [3, 4]. Ominaisuuksiensa ansiosta se on laajassa käytössä sijoittuen esimerkiksi kesäkuussa 2021 neljännelle sijalle TIOBE-indeksissä [5].



Kuva 1. Kesäkuun 2021 TIOBE-indeksin kuvaaja [5].

Kielen käyttökohteisiin kuuluu muun muassa käyttöjärjestelmiä, sovellusohjelmistoja, erilaisia servereitä sekä videopelejä ja niihin liittyvää ohjelmistoa. Monet suuret yritykset, kuten Amazon, Google, Intel ja Microsoft käyttävät tai ovat käyttäneet C++:aa toiminnassaan. Vahvuudet kriittistä suorituskykyä vaativissa tai rajallisten resurssien ympäristöissä ovat johtaneet kielen runsaaseen käyttöön myös telekommunikaatiojärjestelmissä (esimerkiksi Yhdysvalloissa AT&T ja Suomessa Nokia). Myös militääriset järjestelmät ja avaruustutkimus, kuten avaruusluotaimet, käyttäjänä esimerkiksi NASA, ovat nähneet C++:aa hyödynnettävän. [6, 7].

2.1 C++ optimoinnissa

C++ on yleensä kokonaan käännetty kieli, eli kääntäjä välittää lähdekoodin tyypillisesti suoraan koneelle luettavaksi koodiksi. Optimoinnin saralla kokonaan käännetyt ohjelmointikielet antavat useimmiten parhaita tuloksia. Tulkitut kielet ovat hitaampia ajamaan – esimerkiksi silmukka joudutaan tulkitsemaan jokaisella kierroksella. Välikielen (usein tavukoodin) kautta käännetyt kielet tarjoavat tulkittuja parempia mahdollisuuksia nopeuden saralla, mutta ohjelman ajonaikaisesti vaatimat viitekehykset käyttävät usein enemmän resursseja kuin itse ohjelma. Näiden suurten kokonaisuuksien kautta saatetaan myös törmätä satunnaisiin ongelmiin, kuten esimerkiksi viiveisiin syötteitä lukiessa. [8.]

C++ tarjoaa käännetyn kielen ominaisuuksien lisäksi myös kattavan valikoiman eri kääntäjiä eri alustoille sekä kokoelman laadukkaita optimoituja kirjastoja. Se sisältää suuren määrän korkean tason abstraktiomahdollisuuksia, mutta myös C-kielen laiteläheisen optimoinnin mahdollisuudet tarjoten näin sekä joustavuutta että tehokkuutta myös optimointia ajatellen. Useat C++-kääntäjät tukevat myös joitakin assembly-kieleen liittyviä ominaisuuksia, kuten linkittämistä assembly-moduuleihin tai kääntämisen assembly-muotoon, jolloin kääntäjän tekemää optimointia voidaan tarkastella luettavammassa muodossa. [8.]

2.2 C++20

C++:n uusin standardi C++20 julkaistiin joulukuussa 2020 [9]. Uusia ominaisuuksia olivat muun muassa konseptit, moduulit, pinottomat alirutiinit, tuki kääntämisen aikaiselle laskennalle, kolmisuuntainen <=>-vertailuoperaattori, kirjastot arvoalueille (range), ajoille ja päivämäärille (date), taulukoista hakemiselle (span) sekä tyyppiturvalliselle printf-vastineelle (format). [8, 10].

Konseptit tarjoavat mahdollisuuden asettaa rajoituksia ja/tai vaatimuksia templaattiargumenteille, esimerkiksi että tietyn operaattorin tulee olla määritelty argumentin tyypille. Moduulit tarjoavat hiukan esimerkiksi JavaScriptin moduuleja muistuttavan modulaarisemman vaihtoehdon perinteiselle #include-rakenteelle, ja mahdollistavat samalla korjauksia useisiin sen aiheuttamiin ongelmiin, kuten sisällytysjärjestyksen aiheuttamiin odottamattomiin muutoksiin, ristiriitaisten määritelmien sisällytyksessä aiheuttamiin (usein kääntäjän ilmoittamatta jääviin) virheisiin tai yksinkertaisesti kääntämisen kestoon. Uudet alirutiiniominaisuudet mahdollistavat asynkronisten tehtävien pysäyttämisen ja jatkamisen. Kääntämisen aikana tapahtuva laskenta on ollut kielessä

jo pidemmän aikaa. Sen modernina pohjana ovat C++11-standardin constexpr-funktiot. C++20 lisää tähän kokonaisuuteen lisää tukea ja useita uusia ominaisuuksia. Kolmisuuntainen operaattori mahdollistaa kolmen vertailun tekemisen yhdellä operaattorilla. Palautettu arvo on negatiivinen, positiivinen tai nolla riippuen vertailun tuloksesta. Range-kirjasto sisältää arvoalueisiin liittyviä ominaisuuksia, date nimensä mukaisesti aikoja ja päivämääriä. Span tarjoaa työkaluja turvalliseen ja tehokkaaseen taulukoista hakemiseen, ja format tyyppiturvallisen printf-vastineen. [4, 10].

Näiden olennaisempien ominaisuuksien lisäksi C++20-standardiin kuuluu myös suuri määrä pienempiä tai vähemmän keskeisiä muutoksia ja lisäyksiä. Näitä ovat esimerkiksi C99-tyyliset initialisaattorit sekä string-literaalin käyttö templaateissa. Samaa linjaa ovat myös [[likely]]- ja [[unlikely]]-merkinnät. [4.]

Huolimatta useista uusista ominaisuuksista, joista monet ovat huomattaviakin vaikutusskaalaltaan, Stroustrup kuvailee retrospektiivissään C++20-standardin kohdalla komitean olevan hiukan tuuliajolla – itse asiassa jo C++17 on dokumentissa otsikoitu ”Lost at Sea”. ”Ensiarviona, jokaista ehdotusta kohti on noin tusina jäsentä, jotka vastustavat voimakkaasti jotakin. WG21:n vaatiessa 80 tai 90 prosentin tukea yksimielisyyden julistamista varten, on ihmeellistä, että C++ on ollut menestyksekäs tähän mennessä.”, Stroustrup kertoo ja jatkaa sitten: ”Omaako WG21 tarkan kuvan siitä, mitä se yrittää tehdä? En usko.” Oireita tästä tilasta löytyy myös ominaisuuksien dokumentoinnissa, sillä esimerkiksi kolmisuuntainen operaattorin <=> kohdalla ominaisuuden hätäinen lisääminen pääsi aiheuttamaan useita ongelmia. [4.]

3 Optimointi

Optimointi tarkoittaa parhaan vastauksen, arvon, määrän tai yleisesti vaihtoehdon etsimistä. Eri osa-alueilla se voi merkitä esimerkiksi käyttäjämäärän maksimointia, kuten sosiaalisen median optimoinnissa [11], tai yksittäisen parhaan vastauksen löytämistä, kuten usein matemaattisessa optimoinnissa. Jo sana itsessään saattaa sisältää erityyppisiä merkityksiä asiayhteydestä riippuen. Matemaattisessa ympäristössä optimoinnilla viitataan tavallisesti perinteiseen merkitykseen globaalin – tai vähintään lokaalin ympäristön – optimin saavuttamisesta, kun taas puhekielessä ja monissa muissa konteksteissa, tavoitteena onkin usein singulaarisen perfektionismin sijaan yleisempi idea parantamisesta [12]. Tälle linjalle kuuluu pääasiallisesti myös optimointi ohjelmoinnissa.

Ohjelmoinnissa optimointi ei kuitenkaan aina ole hyväksi. Liiallinen tai liian aikainen optimointi johtaa usein skaalautumisen, ymmärrettävyyden tai käytettävyyden heikkenemiseen, pahimmillaan myös suorituskyvyn huonontumiseen parantumisen sijaan. Donald Knuth kirjoittaa vuoden 1974 julkaisussaan Structured Programming with go to Statements: "Meidän täytyisi unohtaa pienet parannukset tehokkuudessa, sanotaan noin 97 % ajasta: liian aikainen optimointi on kaiken pahan alku ja juuri. Kuitenkaan meidän ei tule ohittaa mahdollisuuksia tuon kriittisen 3 % alueella." Knuth jatkaa painottamalla, että ensin tulee luoda toimiva ratkaisu, sen jälkeen tutkia sitä – helposti väärässä olevien ennakkoarviointien sijaan – ja siten lopulta saada selville mahdollisia optimointikohteita. [13.] Vastaavan ajatuksen optimoinnista ilman samoja havainnollistavia numeroarvoja Knuth toi esille jo 1968 ensimmäisessä osassa kirjaansa The Art of Computer Programming. [14.]

Toinen vastaavanlainen tunnettu ja usein käytetty lause optimoinnin saralla on Michael A. Jacksonin vuonna 1975 julkaistussa kirjassa Principles of Program Design mainitsemat säännöt optimointia ajatellen:

”Sääntö 1: Älä tee sitä.

Sääntö 2 (vain asiantuntijat): Älä tee sitä vielä – eli, ei ennen kuin sinulla on täydellisen selkeä ja epäoptimoitu ratkaisu.” [15.]

Kuten Knuthin, myös Jacksonin lausunnon taustalla löytyy sama idea suunnitelmallisesta varovaisuudesta optimoinnin suhteen, jotta vältetään mahdolliset negatiiviset vaikutukset. Samalla tulee kuitenkin ilmi selkeästi myös se, että optimointia ei tulisi kokonaan laittaa syrjään, vaan sitä tulisi

mahdollisuuksien mukaan tehdä, oikeassa vaiheessa ja oikealla tavalla. Huomattavaa tarkastelussa on myös, että lausahdukset ovat ajalta, jolloin tietokoneiden komponenttien suorituskyky ja myös ominaisuudet olivat monella tapaa erilaisia tähän päivään verrattuna. Periaatteena nämä ideat ovat kuitenkin silti samalla tavalla käypiä myös nykyään.

Tietokoneiden tehon kasvaessa ajavat ne myös ohjelmistoja nopeammin. Kaikki komponenttien tarjoavat resurssit tekevät oman osansa kokonaisuudessa, mutta traditionaalinen mittari tälle kehitykselle on ollut vuosikymmenien ajan Mooren laki, jonka muun muassa Intelin perustajajäsenenä tunnettu Gordon E. Moore esitteli vuonna 1965 Electronics-lehden artikkelissaan Cramming more components onto integrated circuits. Mooren lain mukaan transistorien määrä kustannustehokkaasti tuotetuissa mikropiireissä kaksinkertaistuu kahden vuoden välein. [16.] Alla kuvassa on havainnollistettu arvoja vuosilta 1970–2020 [17].

Kuva 2. Mooren laki 1970–2020 kuvaajana [17].

Viime vuosina on kuitenkin näyttänyt, että Mooren lain mukainen kehitys olisi hidastumassa, ja sen suuntaisia kommentteja ovat ilmaisseet useat alan ihmiset [18, 19, 20, 21]. Transistorien koon pienentyessä lisääntyvät samalla kvantti-ilmiöiden aiheuttamat ongelmat uuden kehityksessä [22]. Toisaalta Mooren laille esitetään mahdollisuuksia 3D-mikropiirien [23] tai aikansa kontekstia

seuraavan uudelleenmäärittelyn kautta [24]. Todennäköisemmin uudelleenmäärittelyn puoleen kääntänevät keskustelun myös mahdolliset uudenlaiset teknologia, kuten kvanttilaskenta tai nanobioteknologia [20]. Myös nykyisen teknologian pohjalta on vielä päästy uusiin saavutuksiin, kuten maailman ensimmäinen 1 nanometrin kokoinen logiikkaportti [25].

Odotettavaa on kuitenkin joka tapauksessa, että niin prosessorien kuin muiden komponenttien kanssa ei kasvu jatku rajattomasti ilman harppauksia uusissa teknologioissa, ja näin optimointi astunee yhä suurempaan rooliin ohjelmistojen edelleen monimutkaistuessa. Huomattavaa on toisaalta, etteivät raskaat ohjelmistot ole kuitenkaan yksinomaan tämän ajan murhe, vaan ilmiöstä on puhuttu käytännössä halki Mooren eksponentiaalisen kasvun aikakausienkin. Niklaus E. Wirth kirjoitti vuonna 1995 Computer-lehden artikkelissaan A Plea for Lean Software seuraavasti: ”Ohjelmat tulevat hitaammiksi nopeammin kuin tietokonelaitteistot tulevat nopeammiksi.” Lause on sittemmin tunnettu Wirthin lakina, vaikka artikkelissa hän merkitseekin sen Martin Reiserin nimelle. [26.] Samassa yhteydessä hän viittaa myös yleisalaisempaan Parkinsonin lausahdukseen – jonka juuret ovat jo 1950-luvulla [27] – tällä kertaa kontekstiin sopivassa muodossa: ”ohjelmat kasvavat täyttääkseen kaiken saatavilla olevan muistin.” Kyseessä ei siis joka tapauksessa tällöinkään ollut uusi ongelma. Wirth asettaa syyn trendille pitkälti olennaisten ja optionaalisten ominaisuuksien eron häilyvyyteen ohjelmistosuunnittelussa: suuret määrät optionaalisia ominaisuuksia sisällytetään ikään kuin välttämättömyyksinä, ja näin lopullinen tuotos on raskas. Kaiken tämän mahdollistaa komponenttien tehon nousu, joka silti pitää ohjelmistot käytettävällä tasolla suorituskyvyltään. Osittain todistaakseen käytettävän ja joustavan järjestelmän olevan mahdollista uhraamatta tehokkuutta tai keveyttä, Wirth loi yhdessä Jürg Gutknechtin kanssa Oberonin (ohjelmointikieli ja siihen pohjautuva käyttöjärjestelmä), jota hän artikkelissaan käyttää kuvaamaan havaintojaan. [26.]

Vaikka ohjelmistosuunnittelun tulisi toki ottaa aina huomioon myös näitä tekijöitä, ei karsiminen ole kaikissa tilanteissa samalla tapaa vaihtoehto – ohjelmia esimerkiksi tehdään usein asiakkaan tai käyttäjien toiveet mielessä. Näin on olennaista hallita niin yleisesti ohjelmoinnin kuin käytetyn ohjelmointikielen tekniikoita ja periaatteita, joilla ohjelman suorituskykyä voidaan parantaa ilman (esimerkiksi asiakkaan haluamien) ominaisuuksien rajoittamista – jos mahdollista.

3.1 Ohjelman optimointi

Ohjelmaa voidaan yleensä optimoida – parantaa – monella tavoin. Ohjelman käyttämiä resursseja (kuten muistia, prosessorin tai näytönohjaimen kuormitusta, levytilaa tai verkkoyhteyden käyttämistä) voidaan mitata ja yrittää minimoida eri tavoin. Koodin määrää itseäänkin voidaan minimoida, mikä johtaa suoraan myös tiedostojen koon pienentymiseen. Ohjelmakoodin määrän minimointi voi tosin johtaa helposti huomattavaan heikentymiseen luettavuuden osalta, joka jo itsessään voi myös olla yksi optimoinnin kohde. Usein tämä liittyy laajemmin käytettävyyden maksimointiin esimerkiksi jatkokehitystä tai ohjelmistojen pidempiaikaista tukea ajatellen. Yleisin ohjelman optimoinnin muoto – ja samalla todennäköisesti tyypillisin ohjelman suorituskyvyn mittari – lienee kuitenkin ajoaika ja sen minimointi.

Lähdettäessä optimoimaan ohjelmaa on suositeltavaa lähteä liikkeelle monien kappaleessa 3 esiteltyjen periaatteiden kanssa, kuten olemassa oleva ratkaisu tai spesifit, valmiit suunnitelmat, jotta vältettäisiin esitettyjä mahdollisia haittavaikutuksia. Koska optimoinnin kohdistamisella oikeisiin kohtiin (niin sanottuihin ”hot spotteihin”) on radikaali vaikutus lopputuloksiin, olennaista on usein myös tarkastella etukäteen, mitä kohtaa tai toimintoja ohjelmassa optimoidaan, tyypillisesti profiloimalla ohjelman ajoa etukäteen. Tätä vaikutusta voidaan helposti havainnollistaa matemaattisesti: jos 96 % ohjelman ajasta kuluu tehdessä jotakin toimintoa, ei ajoajan puolittuminen jäljelle jäävässä 4 % osiossa – varsin huomattava saavutus itsessään – tuota kuin 2 % leikkauksen kokonaisuudessa. Kun kriittiset kohdat on siis ensin löydetty, voidaan sitten tarkastella, miten niitä voitaisiin mahdollisesti optimoida.

Joskus koodin optimoinnissa päädytään vain kokeilemaan eri vaihtoehtoja mitatessa samalla niiden suorituskykyä. Näin voi olla erityisesti esimerkiksi kolmannen osapuolen resursseja käytettäessä, vaikkapa kirjastojen tai muunlaisten moduulien, mutta myös laajemmin kehitysympäristöjen kanssa (kuten kappaleessa 2.1 mainitaan kieliä tarkasteltaessa). Kun kyseessä on laajempia ominaisuuksia, esimerkiksi grafiikkaa, kolmannen osapuolen kirjastot voivat joskus kuluttaa suuren osan ohjelman ajoajasta ilman, että mahdollisuuksia optimoinnille tarjoajan vaihtamisen ulkopuolelta juuri on. [8.]

Kun kyseessä on kolmannen osapuolen sijaan oma sisältö, kasvavat mahdollisuudet optimoinnille selvästi. Erilaisia enemmän tai vähemmän pieniä vaikuttavia tekijöitä on valtavia määriä, ja ekstensiivinen tarkastelu ylittää tämän dokumentin mittakaavan tässä aiheessa. Kappaleissa 3.1.1 – 3.1.3 on kuitenkin perspektiiviksi lista esimerkkejä yleisistä ja potentiaalisesti huomattavammista tekijöistä, joilla ohjelmakoodia voidaan yrittää optimoida, lyhyine selityksineen.

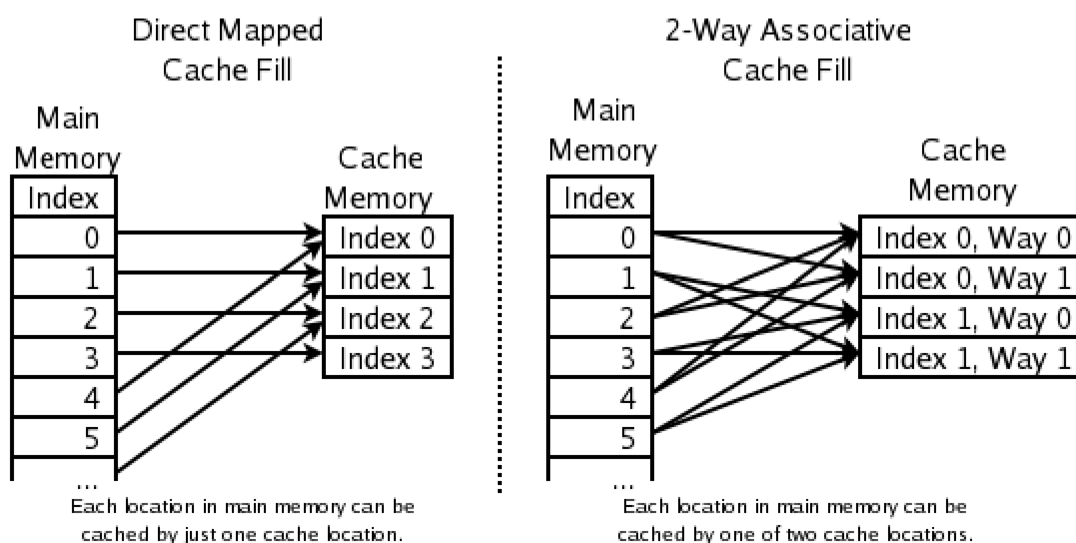
3.1.1 Muistinhallinta

- Allokaatiot

Muistin dynaaminen allokointi tai de-allokointi new-, delete-, malloc- ja free-operaatioilla vie paljon aikaa. Dynaamisen muistin käyttäminen voi olla tarpeellista ja käytännöllistä, ja koska vain tarvittava määrä muistia allokoidaan, joskus myös tehokasta välimuistin hallinnoinnin kannalta. Usein kuitenkin, erityisesti jatkuvista erikokoisista allokoinneista johtuen, ajonaikana varattavissa oleva muisti sirpaloituu ja johtaa sekä heikompaan välimuistin tehokkuuteen että automaattisen muistinhallinnan aiheuttamiin hidastuksiin. C++:ssa vaikkapa string-luokka tai monet muut STL-säiliöluokat tekevät jatkuvaa allokointia koon muuttuessa, mikä johtaa hitaampaan suorituskykyyn kuin esimerkiksi perinteisen taulukon käyttäminen. [8, 28].

- Välimuistit

Välimuistin tarkoituksena on säilyttää dataa prosessorissa niin, että kun sitä (mahdollisesti) tarvitaan uudestaan, on tiedon hakeminen tai kirjoittaminen nopeampaa. Tiedon lukeminen tai kirjoittaminen välimuistista vie tyypillisesti vain muutaman syklin, kun taas kesto voi olla jopa useita satoja, jos data täytyy hakea varsinaisen muistin puolelta. Välimuistissa säilytetään tyypillisesti tietoja, joita oletetaan tarvittavan uudelleen (mahdollisesti monia kertoja). Tällä tavalla voidaan tuottaa haluttu etu nopeudessa. [8, 29].



Kuva 3. Välimuistin toimintaperiaate, suora ja kaksisuuntaisesti liitännäinen [30].

Välimuistin rakenteita on erilaisia. Yllä kuvassa 3 on esitetty yleinen periaate kahdella esimerkillä, joista ensimmäisessä jokainen päämuistin osoite voi olla vain yhdessä välimuistin osoitteessa, ja toisessa vastaavasti kahdessa eri kohteessa. Näiden liitäntöjen lisääminen vähentää ylikirjoittamista ja siten myös välimuistihutien määrää, mutta samalla täytyy lukea useampia kohteita jokaisella kerralla. Tästä johtuen liitäntöjen kaksinkertaistaminen erityisesti pienillä liitännäisyyden määrillä (esimerkiksi kaksisuuntaisesta nelisuuntaiseksi) voi vastata hyödyltään noin välimuistin kaksinkertaistamista, mutta lukujen kasvaessa edut vähenevät. [31.]

Välimuisteja on tyypillisesti useita tasoja (yleensä nykyään 3), joiden koko vaihtelee muutamista kymmenistä tai sadoista kilotavuista useisiin megatavuihin ja nopeus nopeammasta hitaampaan. Kappaleen 6 mittauksissa käytetty AMD Ryzen 7 2700X esimerkiksi sisältää kolme välimuistin tasoa, kooltaan 768 kilo-, 4 mega- ja 16 megatavua [32].

Mitä enemmän ohjelma voi hyödyntää nopeampia välimuistivarastoja, sitä optimoidumpaa toiminta on. Datan sijainti lähellä toisiaan muistissa edesauttaa välimuistin hyödyntämistä, ja näin esimerkiksi lokaalit muuttujat ja/tai yleisesti pinomuistissa oleva data ovat usein (tason 1) välimuistissa. Toisaalta staattiset ja globaalit muuttujat voivat heikentää välimuistin toimivuutta, koska muistisijainteja ei voida käyttää uudelleen koko ohjelman aikana. Yleisenä periaatteena välimuistin kannalta voidaan ajatella, että funktiot, joita käytetään yhdessä, säilytetään yhdessä; ja muuttujat, joita käytetään yhdessä, säilytetään yhdessä. [8.]

3.1.2 Tiedostojen ja datan käsittely

Tiedostojen lukeminen tai kirjoittaminen voi usein viedä enemmän aikaa kuin kyseisen datan käsittely ohjelmassa. Tyypillisesti käsitteleminen on tehokkaampaa suuremmissa palasissa, ja yksi vaihtoehto onkin käyttää yhtä suurempaa puskurimuuttujaa koko datalle kerralla ja näin mahdollistaa lukeminen tai kirjoittaminen yhdellä kertaa. Tiedostojen lisäksi tämä sama periaate voidaan soveltaa myös yleisesti ohjelman sisäisiin puskureihin, esimerkiksi verkkoliikenteeseen. [8.]

Tiedostojen käsittelyä voidaan myös optimoida käyttämällä useampia säikeitä, jos ohjelman on mahdollista työskennellä myös tiedostonhallintaa odotellessa.

Tiedostot tallennetaan yleensä mahdollisuuksien mukaan niille tarkoitettuun välimuistiin ja näin nopeutetaan käsittelyä seuraavilla kerroilla. Ulkoisten tallennustilojen, kuten USB-muistitikkujen, tiedostoja ei voida tallentaa välimuistiin. [8.]

3.1.3 Iteraatiot ja ehdot

Optimoinnin kannalta if-lauseissa olennaisin tekijä on usein, kuinka hyvin ohjelma pystyy ennakoimaan haarautumiset oikein. Tätä voi yrittää edesauttaa esimerkiksi sijoittamalla todennäköisimmät haarat ylimmiksi, tai käyttämällä [[likely]]- ja [[unlikely]]-tyyppisiä merkintöjä kertomaan kääntäjälle oletuksia haarautumisista. Yleensä kuitenkin prosessori ennakoi haarautumiset hyvin. Ennakointi soveltuu suoraan myös silmukoihin, joiden ehdot toimivat sen osalta samalla tavalla.

Silmukoissa yksinkertainen optimointimahdollisuus on myös oikeanlainen break-komennon käyttö, esimerkiksi kun etsitty objekti on löydetty, mutta silmukka jatkaisi vielä kierroksen loppuun. Tyypillisesti tämä yhdistyy break- tai return-komentojen käyttöön tietyissä ehtolauseissa, jolloin silmukasta tai funktiosta voidaan ylimääräisten kierroksien ja/tai rivien välttämiseksi poistua heti, kun se on mahdollista.

3.2 Kääntäjäoptimointi

Kääntäjäoptimointi tarkoittaa kääntäjän kääntämisen yhteydessä tekemiä toimintoja, joilla ohjelmakoodista tehdään optimoidumpaa ajettavaa ohjelmaa luodessa. Kääntäjien tekemä optimointi on useimmiten ajonaikaisiksi jäävien komentojen (toimintojen) minimointia ohjelmassa eri tavoin, oli kyseessä laskentaa, liikkumista tai jotakin muuta. [33.] Eri kääntäjien välillä on jonkin verran eroja ominaisuustarjonnassa, mutta yleisimmät ja olennaisimmat ominaisuudet löytyvät pääasiassa kaikista optimointitukea sisältävistä kääntäjistä. Alla taulukossa 1 on esitetty neljän laajalti käytetyn kääntäjän – Gnu (GCC), Clang, Microsoft (MSVC) ja Intel – tarjoamia tukia joillekin yleisille kääntäjäoptimointiominaisuuksille. [8.]

Taulukko 1. Kääntäjien tarjoamaa tukea joillekin yleisille optimointiominaisuuksille [8].

Optimization method	**Gnu**	**Clang**	**Microsoft**	**Intel**
General optimizations:				
Function inlining	x	x	x	x
Constant folding	x	x	x	x
Constant propagation	x	x	x	x
Constant propagation through loop	x	x	-	-
Pointer elimination	x	x	x	x
Common subexpression elimination	x	x	x	x
Register variables	x	x	x	x
Fused multiply and add	x	x	x	x
Live range analysis	x	x	x	x
Join identical branches	x	x	x	x
Eliminate jumps	x	x	x	x
Tail calls	x	x	x	x
Remove branch that is always false	x	x	x	x
Loop unrolling, array loops	x	x	x	x
Loop unrolling, structures	x	x	x	-
Loop invariant code motion	x	x	x	x
Induction variables for array elements	x	x	x	x
Induction variables for other integer expressions	-	x	-	x
Induction variables for float expressions	-	-	-	x
Multiple accumulators, integer	-	x	x	-
Multiple accumulators, float	-	x	x	-
Devirtualization	x	x	x	x

Monia kääntäjien tekemiä optimointeja on mahdollista suorittaa myös manuaalisesti jo ohjelmointivaiheessa. Kääntäjät tekevät kuitenkin myös paljon, jota voi olla vaikeampaa, epäkäytännöllistä tai mahdotonta manuaalisesti replikoida ohjelmointivaiheessa.

Kääntäjäoptimoinnissa tehtyjen toimenpiteiden alkuperä ja nykyinenkin selkäranka on monella tapaa vuoden 1971 Frances Allenin ja John Cocken julkaisussa A Catalogue of Optimizing Transforms [34]. Dokumentissa käydään läpi useita muitakin tekniikoita, mutta tyypillisesti listalta mainitaan erityisesti seuraavat kappaleissa 3.2.1–3.2.8 esitetyt kahdeksan periaatetta. Suluissa mainitaan alkuperäisessä dokumentissa käytetty nimi, jos se eroaa nykyisestä termistöstä.

Yhtenä Rust-ohjelmointikielen – ja sen kääntäjän – kehittäjistä tunnettu Graydon Hoare on sanonut useiden kääntäjien saavuttavan jopa 80 % optimitilanteen suorituskyvystä pelkästään toteuttamalla nämä toiminnot [35].

3.2.1 Funktioiden tekeminen avoimiksi (prosessien integrointi)

Inlining eli funktion tekeminen avoimeksi tarkoittaa, että kääntäjä siirtää funktion sisällön kokonaisena funktion kutsun paikalle ja poistaa näin tarpeen kutsujen linkittämiselle ja siihen liittyen esimerkiksi argumenttien käsittelylle rekisterissä funktion kutsumisen ja sieltä palaamisen yhteydessä. Avoimien funktioiden hyöty on tyypillisesti sitä suurempi, mitä pienemmistä funktioista on kyse; äärimmäisissä tapauksissa kutsuun yksinään liittyvät proseduurit voivat sisältää enemmän toimintoja ja vaatia enemmän resursseja kuin itse funktio. Toisaalta funktion koon tai käyttökertojen kasvaessa – tai erityisesti molempien tehdessä niin – voi kasvavan koodin määrän aiheuttamat haitat esimerkiksi muistin hyödyntämisen kannalta olla huomattavampia kuin saatavat hyödyt. Välittömien etujen lisäksi funktioiden tekeminen avoimeksi tarjoaa usein myös paljon välillisiä optimointimahdollisuuksia, kun kääntäjä voi muita tekniikoita hyväksikäyttämällä optimoida paikalleen siirrettyä funktion sisältöä myös ympäröivän kontekstin suhteen. [34, 36, 8].

3.2.2 Silmukoiden transformaatiot

Allen ja Cocke esittelivät kolme eri silmukoille tehtävää optimointia: silmukoiden yhdistäminen, erottaminen sekä unrolling eli silmukoiden avaaminen [34]. Yhdistäminen tapahtuu tyypillisesti, jos kahden eri silmukan sisällöt voitaisiin käsitellä yhden sisällä – toiminto, jonka ohjelmoija itsekin yleensä huomannee. Vastaavasti erottaminen voidaan tehdä esimerkiksi, jos silmukan sisällä on ehtolause, jonka tarkastelu voidaan sijoittaa silmukan ulkopuolelle tehtäväksi vain kerran ja päättää silloin, kumpi silmukoista suoritetaan. Koodin määrä kasvaa, mutta toimintojen määrä vähenee.

```
DO I = 1 TO 100 BY 1;
A(I) = A(I) + B(I);
END;
becomes when unrolled by 2:
DO I = 1 TO 100 BY 2;
A(I) = A(I) + B(I);
A(I+1) = A(I+1) + B(I+1);
END;
```

Kuva 4: Esimerkki silmukan vektoroinnista [34].

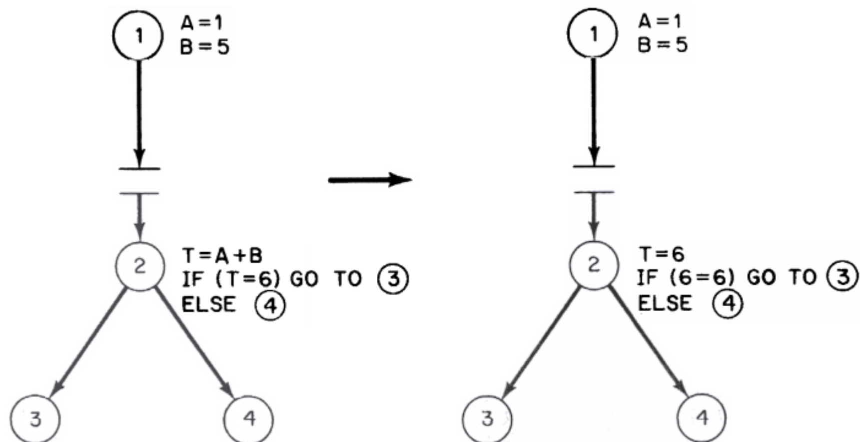
Yllä kuvassa 4 on esitettynä kolmas silmukkatransformaatio eli avaaminen, jota kutsutaan yleensä silmukan (automaattiseksi) vektoroinniksi. Siinä silmukan kierroksia vähennetään ja yhden kierroksen sisällä tehtäviä toimintoja vastaavasti lisätään (”partial unrolling”). Lyhyet silmukat voidaan jopa avata kokonaan (mistä alkuperäinen termi ”unrolling” tulee) ja tehdä kaikki operaatiot normaalissa sekvenssissä luvuilla, jotka silmukka olisi tarjonnut. Vektoroinnin myötä tehtävien toimintojen kokonaismäärä vähenee. Lisäksi kierroksen sisälle jäävissä tehtävissä on yleensä mahdollista hyödyntää yhdenaikaisuutta. [34, 33, 8].

3.2.3 Toistuvien alilausekkeiden eliminointi

Toistuvien alilausekkeiden eliminointi (englanniksi CSE eli common subexpression elimination, myös redundant subexpression elimination eli tarpeettomien alilausekkeiden eliminointi) on kääntäjäoptimointimenetelmä, jossa toistuvat samat tai saman luvun tuottavat lausekkeet lasketaan vain kerran ja korvataan sen jälkeen valmiilla arvolla. Toiminto voidaan tehdä niin muuttujia sisältäville lausekkeille kuin paljaille luvuille, jolloin se usein muistuttaa monella tapaa vakioiden taittamista (kappale 3.2.4). [34, 8].

3.2.4 Vakioiden taittaminen

Vakioiden taittaminen, myös nimellä vakioiden propagaatio, tarkoittaa vakioihin pohjautuvien lausekkeiden laskemista ja korvaamista saadulla tuloksella eli vakiolla. Lauseke voi pohjautua vakioon joko suoraan tai vakioista asetettujen muuttujien kautta. Periaate on esitetty alla kuvassa 5.



Kuva 5. Vakiopropagaation periaate [34].

Propagaatio voidaan tehdä myös funktion kautta käsiteltäville arvoille (kuva 6 alla), jos kääntäjälle on tiedossa funktion sisältö kääntämisen aikana.

```
// Example 8.3a
float parabola (float x) {
   return x * x + 1.0f;}

float a, b;
a = parabola (2.0f);
b = a + 1.0f;
```

The compiler may replace this by

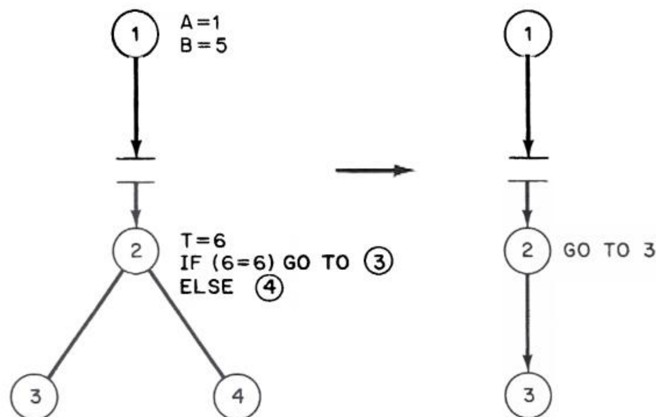
```
// Example 8.3b
a = 5.0f;
b = 6.0f;
```

Kuva 6. Funktion kanssa käytettyjen vakioiden taittaminen [8].

Ulkoisista kirjastoista käytettyihin funktioihin, jotka muuten sopisivat vakioiden taittamiseen, kuten standardin sinifunktio, ei siis voida soveltaa vakioiden propagaatiota. [34, 8].

3.2.5 Kuolleen koodin eliminointi

Jos ohjelma ei kääntäjän arviosta ajettaessa saavuttaisi jotakin osaa koodista, voidaan kyseiseen osaan soveltaa kuolleen koodin eliminointia (alkuperäinen termi DCE eli dead code elimination) eli yksinkertaisesti osa poistetaan. Syitä tähän voi olla useita, mutta yksi tyypillinen syy, jota ohjelmoija itse ei huomaisi, on vakioiden propagaation aiheuttamat muutokset ohjelmakoodiin.



Kuva 7. Kuolleen koodin eliminointi [34].

Suoritettuaan aiemman kuvan 5 mukaisen proseduurin, kääntäjä päätyy tilanteeseen, jossa lausekkeeseen kuuluu kuollutta koodia, jota ei voida saavuttaa (else-haarake). Näin kääntäjä voi jatkaa optimointia kuvan 7 mukaisesti poistamalla kuolleen osan. [34, 8].

3.2.6 Ohjelmakoodin siirtäminen

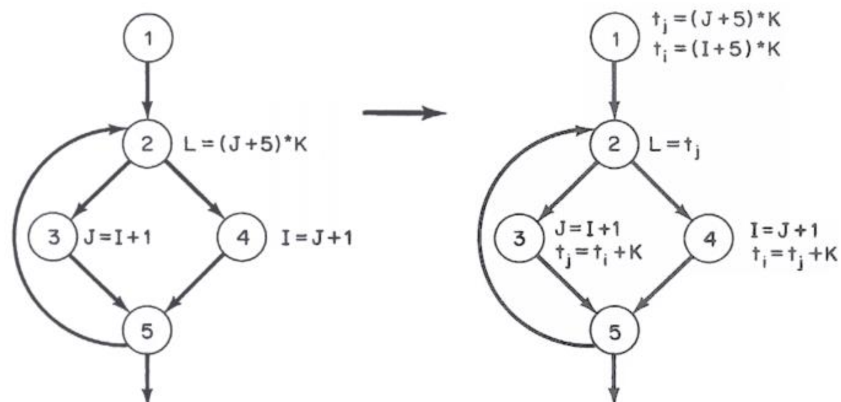
Lauseke voidaan siirtää, jos sen käyttämät tai tuottamat arvot eivät siitä muutu ja voidaan arvioida, ettei siirtämisestä koidu odottamattomia sivuvaikutuksia. Tyypillisiä esimerkkikäyttökohteita koodin siirtämiselle ovat silmukoiden sisällä tapahtuvat lausekkeet, jotka voidaan kontekstin niin salliessa siirtää ulkopuolelle ja vähentää tarvetta laskea niitä useita kertoja. Jos silmukoita on useita ja sama lauseke on niissä kaikissa, eliminoituu tässä samalla ylimääräiset kopiot samasta

toiminnosta. Ohjelmakoodin siirtäminen onkin usein linkittynyt yleisten alilausekkeiden eliminointiin, koska siirtämällä lauseke toiseen paikkaa voidaan joskus eliminoida yksi tai useampia instansseja samasta lausekkeesta toisaalla. [34, 8].

Myös [[likely]]- ja [[unlikely]]-merkintöjen käyttäminen voi johtaa kääntämisessä ohjelmakoodin siirtämiseen, kun kääntäjä todennäköisesti siirtää oletetut prioriteettihaarakkeet ensimmäisiksi, kuten kappaleen 5.2 esimerkeissä.

3.2.7 Lujuuden vähentäminen

Lujuuden vähentäminen tarkoittaa kalliiden operaatioiden korvaamista vastaavilla, mutta kevyemmillä operaatioilla. Tyypillinen tapaus on kertolaskun korvaaminen yhteenlaskulla, esimerkiksi kuvan 8 mukaisesti, jossa kertolasku siirretään tehtäväksi kerran silmukan ulkopuolella ja sisäpuolella tehdään vain yhteenlaskuja. [34, 37, 38].



Kuva 8. Lujuuden vähentäminen [34].

3.2.8 Kurkistusaukko-optimointi (ikkunaoptimointi)

Kurkistusaukkotekniikan idea on, että tarkastelemalla ennalta määriteltyä lukumäärää komentoja, esimerkiksi kahtatoista, voidaan löytää mahdollisuuksia optimoiville transformaatioille. Kääntäjä voi vaikkapa huomata, että jokin muuttuja poimitaan pois välimuistista, mutta laitetaan heti hetken päästä takaisin ja poistaa nämä turhat operaatiot. Tyypillisesti kurkistusaukkovaihe

suoritetaan viimeisenä tai vähintäänkin aivan loppuvaiheessa, kun kääntäjä on soveltanut jo monia muita tekniikoita suollettavaan ohjelmakoodiin. [34, 39].

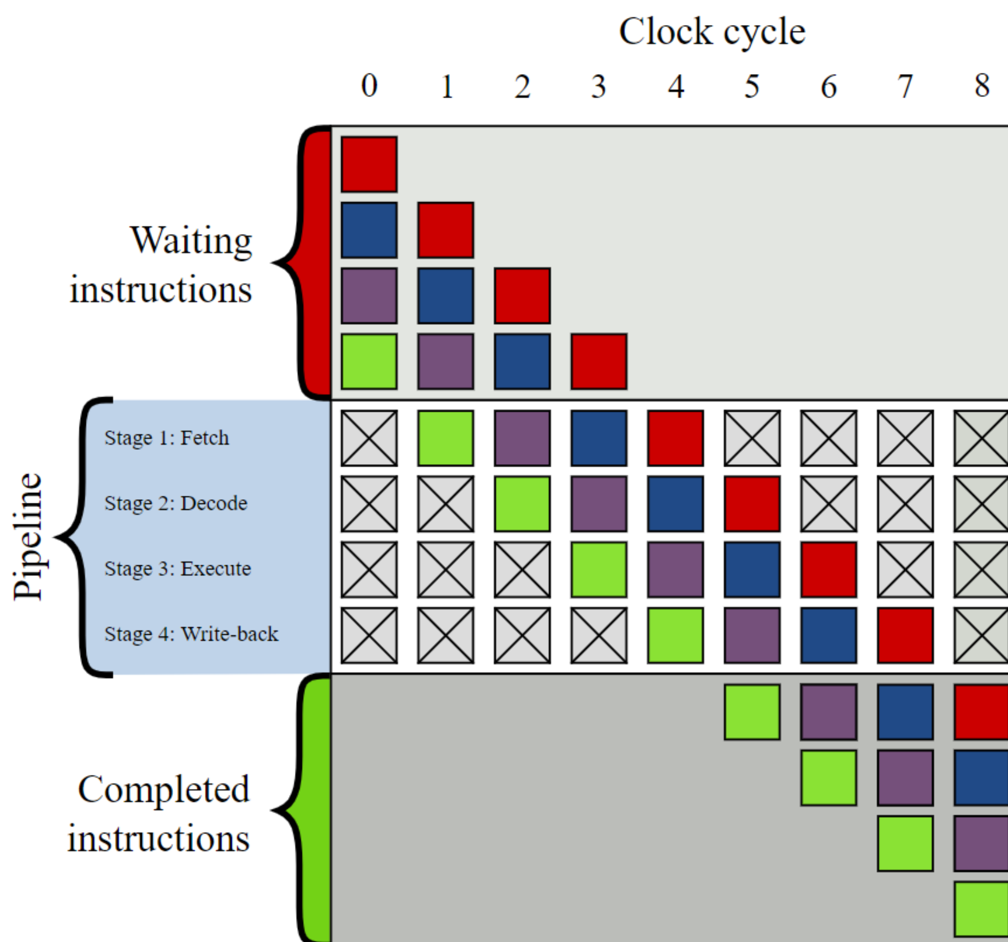
3.2.9 Manuaalinen kääntäjäoptimointi

Automaattisten toimintojen lisäksi kääntäjät voivat tarjota ominaisuuksia, joita kehittäjä voi halutessaan käyttää. Tähän kategoriaan kuuluvat esimerkiksi ohjelmakoodiin sisällytettävät merkinnät, kuten kappaleessa 5 tarkemmin käsitelty haarautumien todennäköisyyttä kertova [[likely]] tai [[noreturn]], joka mahdollistaa kääntäjälle optimoinnin, kun tiedetään, ettei funktio palaa [40]. Samaa linjaa jatkavat kääntämisprosessille annettavat lisämerkinnät, kuten GCC:n -finline-functions-called-once (kääntäjä tekee vain kerran kutsutuista funktioista avoimia), joilla voidaan tyypillisesti poimia joitakin tiettyjä automaattisia toimintoja käyttöön tai määritellä niiden toimintaa. Usein kääntäjät tarjoavat vastaavia määritysmahdollisuuksia myös laajempina kokonaisuuksina, kuten GCC:n optimointitasoa määrittelevät O1-, O2- ja O3-merkinnät. [41.]

4 Prosessorien haarautumisen ennakointi

Haarat ovat ohjelman kohtia, joista ajo tyypillisesti jatkuu normaalista sekvenssistä poikkeavalla tavalla. Haaroja on ehdollisia ja ehdottomia. Ehdolliset haarat toteutuvat perustuen ehtolauseen – esimerkiksi if-haarakkeen tai for- ja while-silmukoiden ehtojen – tulokseen. Ehdottomat haarautumiset, kuten break-, return-, tai goto-komennot, toteutuvat aina ajon saavuttaessa kyseiset komennot. Näiden kahden kategorian tarkastelussa huomattavaa on, että käytännössä ehdottomatkin haarautumiset ovat tyypillisesti käytössä juuri erilaisten ehtojen yhteydessä.

Vaikka haarat ovat usein perustavanlaatuinen osa ohjelmia, voivat ne joskus vaikuttaa huomattavankin negatiivisesti suorituskykyyn ohjelmaa ajettaessa. Yksi tällainen tekijä liittyy moderneissa prosessoreissa käytettävään komentoliukuhihnatekniikkaan, jossa yhdenaikaisuutta hyödynnetään nopeuttamaan prosessia. Eri vaiheet, jotka jokainen komento käy ajossa läpi – esimerkiksi komennon hakeminen, tulkitseminen ja ajaminen – limitetään niin, että ajettaessa yhtä komentoa, voidaan aiempia vaiheita jo tehdä tuleville komennoille. Alla kuvassa 9 periaate on havainnollistettu esimerkkinä nelivaiheinen linjasto [42].



Kuva 9. 4-osainen komentolinjasto [42].

Koska prosessori ei voi tietää etukäteen, mihin haara johtaa, joutuu se arvaamaan valmistellakseen komentoja linjastolla. Väärin ennakoidessa joudutaan valmistellut komennot hylkäämään ja aloittamaan alusta oikeilla. Tästä johtuvat hidastukset kellosykleinä vastaavat yleensä noin prosessorin liukuhihnan pituutta (vaiheiden määrää) [43] – näin mitä edistyneempi ja laajempi linjasto on, sitä suurempaa vahinkoa väärät ennakoinnit aiheuttavat. Tätä ongelmaa pyritään ratkaisemaan haarautumisen ennakoinnin lisäksi myös haarojen välttämisellä tai poistamisella esimerkiksi bittioperaatioiden tai edellyttämismetodien, kuten ehdollisten siirtojen, avulla.

Bittioperaatio tarkoittaa datan käsittelemistä yksittäisten bittien tasolla. Esimerkiksi NOT-operaatio kääntää bitit päinvastoin. Bittipohjaiset operaatiot ovat nopeita (vain yksi komento ja tyypillisesti yksi kellosykli [8]), ja näin niistä voi saada hyötyä monenlaisissakin käyttökohteissa. Kuitenkin niiden soveltaminen kaikkien erilaisten haarojen poistamiseen on vaikeaa tai mahdotonta, ja usein ratkaisutkaan eivät välttämättä vastaa täydellisesti alkuperäistä haaraa, vaan esimerkiksi toteuttavat saman toiminnon jollakin tunnetulla (rajatulla) ryhmällä syötteitä. Näin on myös alla

kuvan 10 haaran eliminoinnin esimerkissä, jonka käyttäjä Mysticial esittelee vastauksessaan haarautumisen ennakointiin liittyvässä kysymyksessä Stack Overflow -sivustolla [44].

Replace:

```
if (data[c] >= 128)
    sum += data[c];
```

with:

```
int t = (data[c] - 128) >> 31;
sum += ~t & data[c];
```

Kuva 10. Esimerkki haaran eliminoinnista bittioperaatioilla [44].

Edellyttävässä logiikassa (predication) ehtolauseen kaikkia haaroja käsitellään komentolinjastolla eteenpäin yhdenaikaisesti, ja niistä oikeaksi osoittautuva validoidaan, kun ehtolause on ajettu ja tulos siitä laskettu [45, 46]. Lopputuloksena ajaminen on näiltä osin hiukan hitaampaa, mutta haaroja ja niistä mahdollisesti aiheutuvia hidastuksia ei käytännössä enää ole. Tämä yleensä ehdolliseksi siirroksi (conditional move) kutsuttu tekniikka on erityisen hyödyllinen, jos käsittelyssä on sattumanvaraisia haaroja, joiden kanssa ennakointimenetelmillä ei saada tyydyttäviä tuloksia. Suurin osa haararakenteista on kuitenkin hyvin ennakoitavissa, ja koska oikein arvatessa haarat ovat aina suorituskyvykkäämpi vaihtoehto [47], on ennakoiminen olennaisin metodi haaroista aiheutuvien mahdollisten haittojen minimoinnissa.

4.1 Haarautumisen ennakoinnin historia

Yhtenä ensimmäisistä haarautumisen ennakointia suorittavista järjestelmistä voitaneen pitää IBM:n 1950-luvulla suunnittelemaa supertietokonetta IBM 7030, joka tunnetaan nimellä Stretch. 7030 kykeni tulkitsemaan komentoja ajon kanssa yhdenaikaisesti ja ajoi ennalta kaikki ehdottomat haarat sekä haarat, jotka riippuivat järjestelmän indeksirekisteristä. Muut haarat ennakoitiin epätosiksi (kaksi ensimmäistä mallia) tai ennakoitiin käyttämällä indikaattoribittiä todennäköiselle tulokselle (myöhemmät mallit). [48, 49, 50].

Vuonna 1977 Tom McWilliams ja Curt Widdoes esittelivät kaksibittisen haarautumisen ennakoinnin Lawrence Livermoren kansalliselle laboratoriolle rakennetussa supertietokoneessa S-1. Ennakointijärjestelmä vaihtoi tilaansa vain, kun tulos arvataan väärin kaksi kertaa putkeen. [51.]

Samankaltainen järjestelmä oli rakennettu myös 1982 julkaistuun COBOL-pohjaiseen Burroughs B4900 -tietokoneeseen, jossa kahden bitin sijaan oli neljä heksadesimaalimuotoista koodia, joiden välillä laitteisto päätti. B4900 omasi haarautumisen ennakoinnin lisäksi myös yhdenaikaistettua komentojen käsittelyä. Järjestelmälle myönnettiin patentti vuonna 1984. [52.]

Vastaavaa kehityksen linjaa edusti myös DEC:n vuonna 1989 esitelty VAX 9000. Haarautumisen ennakoinnissa käytössä oli 1024 haaraa kattava välimuisti, jonka avulla ennakoimista tehtiin dynaamisesti käyttäen yhtä indikaattoribittiä. Jos haaraa ei löytynyt välimuistista, käytössä oli staattinen ennakointi. VAX 9000 kykeni jopa kuuden komennon yhtäaikaiseen käsittelyyn liukuhihnalla, joten väärinennakoimisesta palaaminen aiheutti siten myös vastaavasti huomattavia hidastuksia. [53.]

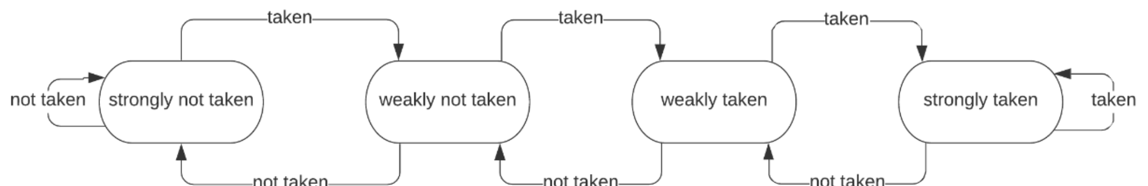
Haarautumisen ennakointi ei kuitenkaan pitkään aikaan ollut kovin keskeinen ominaisuus useissa järjestelmissä, koska suosiossa olleiden mikro-ohjelmoitujen prosessorien monimutkaisten komentojen lukeminen vei yleensä useita kellosyklejä itsessään eikä haarojen ennakoiminen väärin välttämättä siten tuottanut huomattavia menetyksiä. Vasta kehityksen siirtyessä enemmän superskalaaristen prosessorien ja komentolinjastojen puoleen alkoi haarautumisen ennakoinnin kehitys saada suurempaa roolia yleisemmilläkin markkinoilla. [54, 55, 56].

4.2 Haarautumisen ennakoinnin tekniikoita ja tyyppejä

Yksinkertaisin haarautumisen ennakoinnin muoto on staattinen ennakointi, jossa ainoastaan komento itse vaikuttaa tehtävään päätökseen, eikä ajonaikaista tiedonkeruuta ja sen vaikutuksia ennakointiin ole. Staattinen haarautumisen ennakointi on usein yksisuuntainen logiikka, jossa kaikki haarat joko ennakoidaan otettaviksi tai ei. Variaatio tähän on niin sanottu ”backwards taken/forwards not-taken” -logiikka (BTFNT), jossa haarat, joiden osoite on aiemmin, ennakoidaan otettaviksi, ja muut ohitettaviksi. Aiemmin määritellyt osoitteet ovat yleensä silmukoita, jotka tavallisesti otetaan useaan kertaan. Staattista ennakointia voidaan yrittää parantaa profiloimalla ohjelmaa ajon tai kääntämisen aikana, jolloin algoritmeilla saadaan kuvaa ohjelman mahdollisesta käyttäytymisestä ja yleisestä rakenteesta. Näin parannetaan tulevia staattisia ennakointeja. Toinen parannusvaihtoehto on yleisten ohjelmointiperiaatteiden soveltaminen. Tällaisia voivat

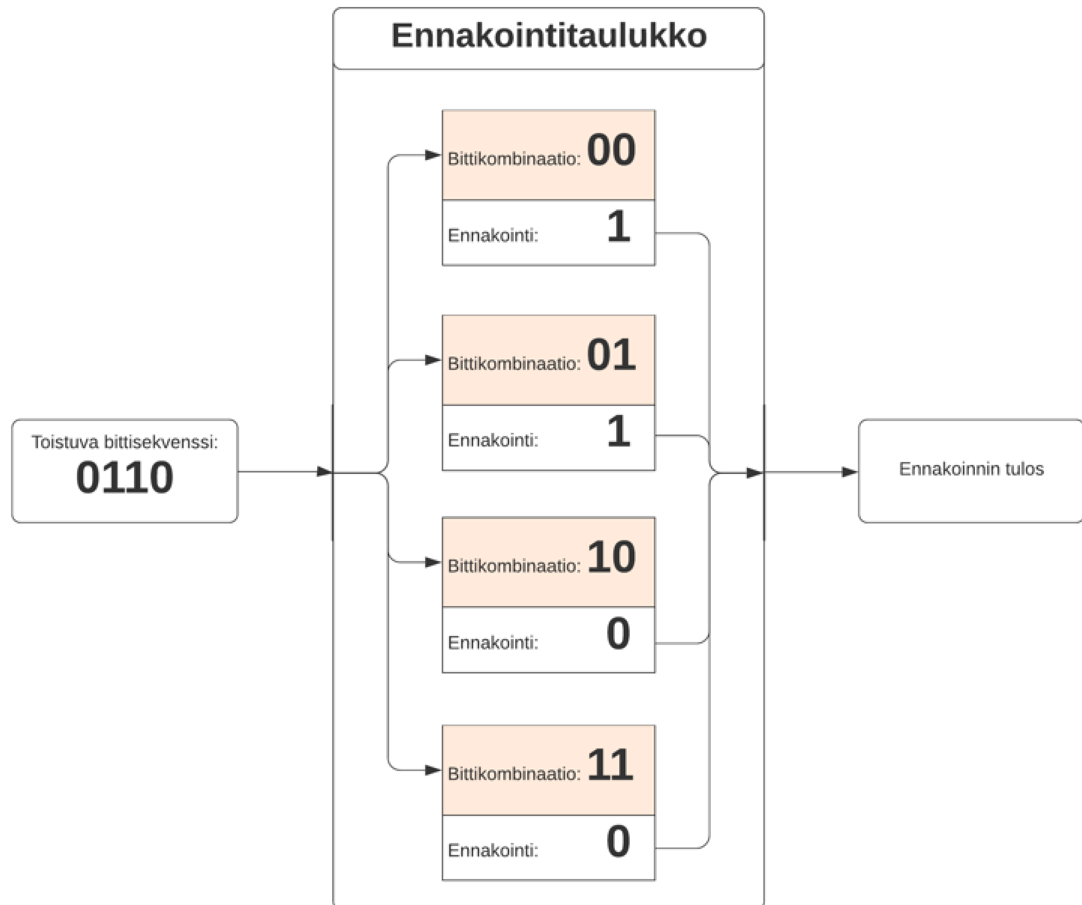
olla esimerkiksi tyypillisten virhetarkastusten, kuten vertaamisen NULL-arvoon, ennakoiminen ohitettaviksi. [57.] Kääntäjä sijoittaa profiloidut staattiset ennakoinnit ja tällaiset Ball-Larus -heuristiikan tulokset ohjelmakoodiin yleensä haaravihjeinä. Vastaavalla tavalla joidenkin prosessorien kanssa vihjeiden käyttäminen on mahdollista myös käyttäjälle, esimerkiksi [[likely]]-tyyppisten attribuuttien kautta. Staattinen ennakointi voi parhaimmillaan saavuttaa jopa 80 % tarkkuuden [58], mutta jää silti huomattavasti jälkeen parhaista dynaamisista metodeista. Moderneissa prosessoreissa käytössä onkin lähinnä dynaamisia järjestelmiä, ja staattinen ennakointi on korkeintaan varalta olemassa [56].

Dynaamisessa haarojen ennakoinnissa ajonaikaiset tapahtumat vaikuttavat tehtäviin päätöksiin. Yksinkertaisimmillaan käytössä voi olla yksi indikaattoribitti, joka kertoo, otettiinko kyseinen haara viimeksi. Saavuttaessa samaan haaraan uudestaan, ohjelma ennakoi bitin mukaisesti. Näin esimerkiksi silmukoiden kohdalla ennustus on aina väärin kaksi kertaa, kun sisään mentäessä ennakoidaan ohittaminen ja ulos mennessä silmukkaan jääminen [55]. Edistyneempi versio tästä indikaattorista on kaksibittinen, jossa kahden tilan sijaan on neljä tilaa, jolloin ennakointi vaihtuu vain kahden peräkkäisen väärän tuloksen jälkeen, korjaten näin esimerkiksi perustoiminnallisuuden silmukoiden kanssa. Periaate on esitetty alla kuvassa 11. [51, 56].



Kuva 11. Kaksibittinen ennakointilogiikka

Sekä yksi- että kaksibittiset indikaattoribitit ovat yksitasoisia dynaamisia ennakointijärjestelmiä. Kaksitasoisissa metodeissa käytössä on kaksiulotteinen taulukko, jonka kaikki jäsenet ovat kaksibittisiä indikaattoreita. Jokaisella eri bittikombinaatiolla on taulukossa oma indikaattorinsa, jota muokataan kuvan 11 mukaisesti ajon aikana. Esimerkiksi jos taulukon koko on 2 suuntaansa, eri bittikombinaatiot ovat 00, 01, 10 ja 11. Näin tämän järjestelmän tila lyhyelle toistuvalle esimerkkisekvenssille 0110, jossa haara ohitetaan kerran, otetaan kaksi kertaa ja ohitetaan kerran, olisi todennäköisesti alla esitetyn kuvan 12 mukainen. [59.]



Kuva 12. Kaksitasoisen ennakointijärjestelmän toimintaperiaate

Sekvenssin toistuessa bittikombinaatiota 00 seuraa aina 1, 01 seuraa aina 1, 10 seuraa aina 0, ja 11 seuraa aina 0. Näin ennakointi saavuttaisi tämän esimerkin kanssa 100 % tarkkuuden. Käytännössä tämä ”kaksitasoinen adaptiivinen oppiva haarautumisen ennakoiminen” [59] voi n-kokoisella historiataulukolla ennakoida oikein sekvenssejä 2^n periodipituuteen asti, kunhan kaikki periodin n-pituiset alisekvenssit ovat uniikkeja [56]. Esimerkiksi 4-bittistä taulukkoa käyttäessä ei silti voida täydellisesti ennakoida toistuvaa sekvenssiä 1100111, koska neljän bitin kombinaatiota 1111 voi seurata joko 0 tai 1. Lisäksi muistikooltaan pienemmät taulukot saattavat joutua käyttämään samoja tietoja useilla haaroille, mikä heikentää tuloksia [60]. Näistä pienistä vajavaisuuksista huolimatta tämä tekniikka mahdollisti huomattavia parannuksia ja saavutti siten suuren suosion.

Yksi moniin moderneihin prosessoreihin – esimerkiksi AMD:n ensimmäisiin Zen-sukupolviin [61] – sisältyvä variantti on suurempaa globaalia taulukkoa käyttävä järjestelmä, jossa näin vältetään

vaadittavien resurssien radikaali kasvu tarkkuuden kustannuksella. Tarkkuutta parannetaan käyttämällä lisäksi haarakohtaisia indikaattoreita, yleensä esimerkiksi kuvan 11 mukaisia kaksibittisiä indikaattoreita, ja muodostamalla lopullinen ennustus yhdistelmänä näitä kahta. Koska tuloksessa taulukon suositus käsitellään yleensä muodossa ”samaa tai eri mieltä” lokaalin indikaattorin kanssa, kulkee tämä tekniikka nimellä ”samanmielinen ennakointi” (alkuperäinen termi ”agree predictor”). [62, 56].

Globaalin järjestelmän heikkoa yksittäisten haarojen erottelemista voidaan myös parantaa yhdistämällä globaali historia haarakomennon osoitteen kanssa, ja indeksoimalla ennakointitaulukko tämän mukaan. Näin tuloksista saadaan tarkempia yksittäisiä haaroja kohtaan. Tästä tekniikasta on pääasiassa kaksi variaatiota, ”gselect” [63], jossa bitit yhdistetään AND-komennon kautta, ja ”gshare” [64], jossa käytössä on XOR. Kuten alla taulukon 2 esimerkistä voidaan nähdä, XOR antaa tuloksena paremman erottelun indeksointiin tuottaen neljä uniikkia indeksiä kolmen sijaan ja siten lopulta korkeamman ennakoinnin tarkkuuden.

Taulukko 2. gselect- ja gshare-tekniikat

Haaran osoite	Globaali historia	gselect (AND)	gshare (XOR)
0000 0000	0000 0001	0000 0001	0000 0001
0000 0000	0000 0000	0000 0000	0000 0000
1111 1111	0000 0000	1111 0000	1111 1111
1111 1111	1000 0000	1111 0000	0111 1111

Viimeisintä kehitystä haarautumisen ennakoinnissa on ollut tekoälypohjaisten ratkaisujen hyödyntäminen, erityisesti kaksitasoisten metodien yhteydessä. Idea tuotiin esille ensimmäistä kertaa 1999, lähtökohtana kaksitasoisten järjestelmien toisen tason korvaaminen neuroverkolla, joka ottaa vastaavat parametrit sisäänsä, ja tunnistaisi sekvenssit kuuluviksi joko ryhmään 1 tai 0 (eli haara otetaan tai ohitetaan) [65]. Seuraavina vuosina käyttöön otettiin tekoälytoiminnassa paljon hyödynnetyt perseptronit ja laskenta suoritettiin vektorien pistetulona, tekijöinä painovektori ja haarojen historiaa sisältävä syötevektori. Syötteen indeksissä 0 käytetään riippumatonta vakiotermiä 1, jolloin vakiotermin painotus painovektorin indeksissä 0 oppii juuri kyseisen haaran vinouman, riippumatta muusta historiasta. [66.] Tämä perseptroniin pohjautuva haarojen

ennakointi saavuttaa parhaimmillaan paremman tarkkuuden kuin mitkään muut vaihtoehdot [67]. Kun muut metodit pääasiassa vaativat eksponentiaalisesti lisää resursseja vaadittavan historian tai haarojen lisääntyessä, neuroverkkopohjainen ratkaisu voi hyödyntää pitkiä historioita lineaarisella resurssien lisäyksellä. Potentiaalisena ongelmana tekniikassa on kuitenkin toistaiseksi sen monimutkaisuus ja mahdollinen hitaus käytännössä. [68.]

Myös erilaisten ennakointijärjestelmien yhdistämistä on tutkittu. Vaihtoehtojen suorituskykyä voidaan tallentaa ja poimia tulos siltä järjestelmältä, joka on suoriutunut paremmin kyseisen haaran kanssa. Erityisen hyödyllistä voi olla yhdistää yksinkertainen vaihtoehto, kuten kaksibittinen indikaattori, ja globaali järjestelmä, kuten gshare. Näin globaalia haarainformaatiota voidaan käyttää sen käydessä ilmi hyödylliseksi, mutta suoraviivaisiin tilanteisiin riittää kevyt ja tehokas tekniikka. [64.]

5 [[likely]]- ja [[unlikely]]-merkinnät

Tässä kappaleessa merkintöihin itseensä tutustutaan tarkemmin. Kuten myös myöhemmin kappaleessa 6 mittauksiin liittyen, laajalti käytettyä ja vapaasti tutustuttavissa olevaa GCC:tä käytetään tässä kääntäjäesimerkkinä, kun tarkastellaan, miten nämä attribuutit on toteutettu tai voitaisiin toteuttaa. GCC:n repositorioon pohjaavat lähteet on luettavuuden, saatavuuden ja käytettävyyden helpottamiseksi haettu GCC:n GitHub-alustalle peilatusta repositoriosta. Alkuperäiseen voi tutustua seuraavan osoitteen kautta: https://gcc.gnu.org/git/?p=gcc.git;a=summary.

Attribuutit [[likely]] ja [[unlikely]] ovat kääntäjämerkintöjä, jotka vihjaavat kääntäjälle – ja sitä kautta prosessorille – ehtolauseiden, kuten haarojen tai silmukoiden, todennäköistä lopputulosta. Merkinnät lisättiin C++-kieleen uusimmassa C++20-standardissa vuonna 2020 [10]. Ne pohjautuvat vuonna 2016 tehtyyn selvitykseen [69]. Taustalla on myös C++11-standardiin lisätty tuki standardisoiduille attribuuteille [70]. Vuoden 2008 ehdotuksessa eräänä attribuuttivaihtoehtona mainitaankin ”probably”, joka idealtaan vastaa nykyisiä [[likely]]- ja [[unlikelyl]]-merkintöjä [71].

Yleisimmistä kääntäjistä GCC on tukenut näitä attribuutteja versiosta 9 asti [72], Clang versiosta 12 [73] ja MSVC versiosta 19.26 (Visual Studio 2019 16.6).

5.1 Vastineet

Erilaisia haarautumien vihjemerkintöjä on suuriakin määriä, vähintäänkin sisäisesti kääntäjien ja prosessorien keskusteluun esimerkiksi käännetyissä assembly-tiedostoissa. Käyttäjille suunnatuista ominaisuuksista tunnetuimpiin kuuluu makropohjainen __builtin_expect-funktio, joka saa parametrikseen lausekkeen - esimerkiksi funktion tai muuttujan – sekä oletuksen sen arvosta, 0 tai 1. Alla olevassa esimerkissä kääntäjälle kerrotaan siis, että muuttujan x oletetaan olevan 0, eli funktiota foo ei kutsuttaisi.

```
if (__builtin_expect (x, 0))
  foo ();
```

Kuva 13. Esimerkki __builtin_expect -funktion käyttämisestä [74].

Sekä GCC että Clang tukevat __builtin_expect -ominaisuutta [74, 75], mutta se ei ole standardisoitu ominaisuus, mikä lisää tarvetta varovaisuudelle sen käytössä projekteissa. Merkintää voi

käyttää myös manuaalisen todennäköisyyden kanssa, käyttämällä funktiota __builtin_expect_with_probability, jolloin ylimääräinen parametri kertoo oletetun prosenttitodennäköisyyden. Alla esimerkki switch-case -rakenteen kanssa [75], jossa tuloksen 5 oletetaan tapahtuvan 70 prosentin todennäköisyydellä. Clang ottaa todennäköisyyden liukulukuna välillä 0.0–1.0, kun taas GCC vastaavasti kokonaislukuna 0–100. Oletusarvona todennäköisyydelle GCC käyttää 90 prosenttia [74].

```
switch (__builtin_expect_with_probability(x, 5, 0.7)) {
default: break;  // Take this case with probability 10%
case 0:  break;  // Take this case with probability 10%
case 3:  break;  // Take this case with probability 10%
case 5:  break;  // This case is likely to be taken with probability 70%
}
```

Kuva 14. Esimerkki __builtin_expect_with_probability -funktion käyttämisestä [75].

Koska syntaksi __builtin_expect -funktion kanssa on monimutkainen ja työläs, käytetään sille pääasiassa helppokäyttöisemmiksi määriteltyjä vaihtoehtofunktioita likely ja unlikely. Valtavat koodikokoelmat kuten GCC ja Clang ehtivät sisältämään suuren määrän hiukan erityyppisiä määriteltyjä, mutta perusidea on tyypillisesti alla olevien kuvien 15 ja 16 mukainen. [76, 77].

```
40  #define likely(X)      __builtin_expect((X) != 0, 1)
41  #define unlikely(X)    __builtin_expect((X), 0)
```

Kuva 15. GCC unlikely ja likely -makrot [76].

```
14  #define likely(x) __builtin_expect(!!(x), 1)
15  #define unlikely(x) __builtin_expect(x, 0)
```

Kuva 16. Clang unlikely ja likely -makrot [77].

Näiden ominaisuuksien suurimpia käyttäjiä (suluissa mainittu komentojen määrä, likely/unlikely) ovat muun muassa Linux-käyttöjärjestelmä (3000+/14000+), Mozilla (200+/7000+) ja Chromium (satoja molempia), kertoo Trychta dokumentissaan, kun hän esittää pohjaa [[unlikely]]- ja [[likely]]-attribuuttien käyttötarpeille. [69.]

5.2 Toiminta

Haaravihjeinä [[unlikely]] ja [[likely]] kertovat kääntäjälle, mikä on haaran oletettu tulos, mahdollistaen oletuksena näin kääntäjälle paremman ennakoinnin ja optimoinnin [69, 78]. Tässä kappaleessa tarkastellaan tarkemmin, mitä tämän abstraktin määritelmän taustalla näyttää tapahtuvan käytännössä ja käydään läpi lähdekoodia sekä kääntäjien näiden ominaisuuksien kanssa tuottamia tiedostoja. Esimerkkinä käytetään GCC:n versiota 10.3.

Attribuuttien käsittely lähtee jäsentimestä (parser.c-tiedosto), jossa ohjelmakoodia luetaan läpi ja myös attribuutit poimitaan. Alla on kuvaa osasta cp_parser_statement -funktiota, jossa eri lausekkeita tarkastellaan, ja relevanttien tyyppien – ehtolauseiden – kohdalla käydään läpi mahdolliset vihjeattribuutit funktiolla process_stmt_hotness_attribute (manuaalisesti korostettu kuvassa). [79.]

```
if (token->type == CPP_KEYWORD)
  {
    enum rid keyword = token->keyword;

    switch (keyword)
      {
      case RID_CASE:
      case RID_DEFAULT:
        /* Looks like a labeled-statement with a case label.
           Parse the label, and then use tail recursion to parse
           the statement.  */
        cp_parser_label_for_labeled_statement (parser, std_attrs);
        in_compound = false;
        in_omp_attribute_pragma = parser->lexer->in_omp_attribute_pragma;
        goto restart;

      case RID_IF:
      case RID_SWITCH:
        std_attrs = process_stmt_hotness_attribute (std_attrs, attrs_loc);
        statement = cp_parser_selection_statement (parser, if_p, chain);
        break;

      case RID_WHILE:
      case RID_DO:
      case RID_FOR:
        std_attrs = process_stmt_hotness_attribute (std_attrs, attrs_loc);
        statement = cp_parser_iteration_statement (parser, if_p, false, 0);
        break;
```

Kuva 17. Ohjelmakoodin tarkastelua jäsentimessä [79].

Kuvan 18 funktiossa process_stmt_hotness_attribute katsotaan, onko lausekkeen attribuuteissa haaravihjeisiin viittaavia avainsanoja ”likely”, ”unlikely”, ”hot” tai ”cold”, ja sen mukaan luodaan PREDICT_EXPR [80].

```
2767  /* If [[likely]] or [[unlikely]] appear on this statement, turn it into a
2768     PREDICT_EXPR.  */
2769
2770  tree
2771  process_stmt_hotness_attribute (tree std_attrs, location_t attrs_loc)
2772  {
2773    if (std_attrs == error_mark_node)
2774      return std_attrs;
2775    if (tree attr = lookup_hotness_attribute (std_attrs))
2776      {
2777        tree name = get_attribute_name (attr);
2778        bool hot = (is_attribute_p ("hot", name)
2779                    || is_attribute_p ("likely", name));
2780        tree pred = build_predict_expr (hot ? PRED_HOT_LABEL : PRED_COLD_LABEL,
2781                                        hot ? TAKEN : NOT_TAKEN);
2782        SET_EXPR_LOCATION (pred, attrs_loc);
2783        add_stmt (pred);
2784        if (tree other = lookup_hotness_attribute (TREE_CHAIN (attr)))
2785          warning (OPT_Wattributes, "ignoring attribute %qE after earlier %qE",
2786                   get_attribute_name (other), name);
2787        std_attrs = remove_hotness_attribute (std_attrs);
2788      }
2789    return std_attrs;
2790  }
```

Kuva 18. Attribuuttien käsittelemistä koodissa [80].

Hot ja cold ovat GCC:n omia ehtolauseiden todennäköisyyksiä indikoivia attribuutteja. Kylmäksi merkatut funktiot optimoidaan koon ja kuumaksi merkatut nopeuden mukaan. Myös komentojen uudelleenryhmittelyä tehdään kuumuuden ja kylmyyden perusteella, jotta rakennetta saadaan välimuistin käytön kannalta optimoitua. [81]. Koska näiden merkintöjen toiminta on hyvin samantapaista, tekee GCC linkitystä C++-standardin attribuuttien ja näiden sisäisten merkintöjen välillä, kuten kuvasta 19 nähdään [82].

```
4850  /* The C++20 [[likely]] and [[unlikely]] attributes on labels map to the GNU
4851     hot/cold attributes.  */
4852
4853  static tree
4854  handle_likeliness_attribute (tree *node, tree name, tree args,
4855                               int flags, bool *no_add_attrs)
4856  {
4857    *no_add_attrs = true;
4858    if (TREE_CODE (*node) == LABEL_DECL
4859        || TREE_CODE (*node) == FUNCTION_DECL)
4860      {
4861        if (args)
4862          warning (OPT_Wattributes, "%qE attribute takes no arguments", name);
4863        tree bname = (is_attribute_p ("likely", name)
4864                      ? get_identifier ("hot") : get_identifier ("cold"));
4865        if (TREE_CODE (*node) == FUNCTION_DECL)
4866          warning (OPT_Wattributes, "ISO C++ %qE attribute does not apply to "
4867                   "functions; treating as %<[[gnu::%E]]%>", name, bname);
4868        tree battr = build_tree_list (bname, NULL_TREE);
4869        decl_attributes (node, battr, flags);
4870        return NULL_TREE;
4871      }
4872    else
4873      return error_mark_node;
4874  }
```

Kuva 19. C++-attribuuttien korvaaminen sisäisillä hot- ja cold-merkinnöillä [82].

Luotu PREDICT_EXPR luo määritelmän mukaan myös vihjeen haarojen ennakoimiselle, ja alustavan profiloinnin jälkeen merkitsee myös haaraan johtavan polun ennakoinnin muine ehtolauseineen haarautumisten ennakoijalle. Koodissa annettu määritelmä on esitetty kuvassa 20 alla [83]. Kuvassa 21 on esitetty esimerkiksi tällaista solmujen (komentojen) todennäköisyyksien merkitsemistä attribuuttien pohjalta [84].

```
1444  /* PREDICT_EXPR.  Specify hint for branch prediction.  The
1445     PREDICT_EXPR_PREDICTOR specify predictor and PREDICT_EXPR_OUTCOME the
1446     outcome (0 for not taken and 1 for taken).  Once the profile is guessed
1447     all conditional branches leading to execution paths executing the
1448     PREDICT_EXPR will get predicted by the specified predictor.  */
1449  DEFTREECODE (PREDICT_EXPR, "predict_expr", tcc_expression, 1)
```

Kuva 20. PREDICT_EXPR määritelmä [83].

```
int flags = flags_from_decl_or_type (current_function_decl);
if (lookup_attribute ("cold", DECL_ATTRIBUTES (current_function_decl))
    != NULL)
  node->frequency = NODE_FREQUENCY_UNLIKELY_EXECUTED;
else if (lookup_attribute ("hot", DECL_ATTRIBUTES (current_function_decl))
         != NULL)
  node->frequency = NODE_FREQUENCY_HOT;
else if (flags & ECF_NORETURN)
  node->frequency = NODE_FREQUENCY_EXECUTED_ONCE;
else if (MAIN_NAME_P (DECL_NAME (current_function_decl)))
  node->frequency = NODE_FREQUENCY_EXECUTED_ONCE;
else if (DECL_STATIC_CONSTRUCTOR (current_function_decl)
         || DECL_STATIC_DESTRUCTOR (current_function_decl))
  node->frequency = NODE_FREQUENCY_EXECUTED_ONCE;
return;
```

Kuva 21. Todennäköisyyksien määrittelyä attribuuttien pohjalta [84].

Kääntäjän (tässä tapauksessa GCC:n) käytännössä tekemiä toimenpiteitä voi katsoa tarkastelemalla sen tuottamaa objektitiedostoa esimerkiksi seuraavasti:

1. Käännä ohjelma komennolla ”g++ -c -O3 -std=c++20 main.cpp”
   - ”-c” kertoo kääntäjälle, että ohjelmakoodi käännetään objektiksi, mutta ei linkitetä ja tehdä ajettavaksi exe-tiedostoksi
   - ”-O3” kertoo kääntäjälle käytetyn optimoinnin tason, tässä tilanteessa korkein eli 3 [41].
   - ”-std” kertoo kääntäjälle käytetyn kielistandardin, tässä tilanteessa C++20
   - ”main.cpp” on käytetty lähdetiedosto, jonka nimi tässä tilanteessa on ”main”
2. Lue ja tulosta objektitiedoston sisältö komennolla ”objdump -dr main.o”
   - ”objdump” on objektien sisältöjen tarkasteluun tarkoitettu työkalu
   - ”-dr” on ”-d” ja ”-r” komennot yhdistettynä, eli objektin purkaminen luettavaksi, ja relokaatiotaulukon käyttäminen operaatiossa
   - ”main.o” on lähdeobjekti, joka avataan

Alla kuvassa 22 on testiohjelma, jonka avulla kääntäjän toimintaa tutkitaan. Muuttujan initialisoimiseen käytetään rand-funktiota, ettei kääntäjä optimoisi koko rakennetta vakiona pois kappaleessa 3.2 esitettyjen periaatteiden mukaisesti. Tulostuksissa käytetään kahta eri funktiota, että erottaminen toisistaan tulostiedostossa on suoraviivaisempaa. Tämän merkinnättömän pohjan objektin antama sisältö on esitetty kuvassa 23. Huomataan, että printf- ja puts-funktiot ovat, kuten alkuperäisessä tiedostossa oli määritelty.

```
 1  #include "stdio.h"
 2  #include "stdlib.h"
 3  int main()
 4  {
 5      int i = rand() % 2;
 6      if (i)
 7      {
 8          printf("true");
 9      }
10      else
11      {
12          puts("false");
13      }
14
15      return 0;
16  }
```

Kuva 22. Testiohjelman pohja

```
0000000000000000 <main>:
   0:   48 83 ec 28             sub    $0x28,%rsp
   4:   e8 00 00 00 00          call   9 <main+0x9>
                        5: IMAGE_REL_AMD64_REL32        __main
   9:   e8 00 00 00 00          call   e <main+0xe>
                        a: IMAGE_REL_AMD64_REL32        rand
   e:   a8 01                   test   $0x1,%al
  10:   74 13                   je     25 <main+0x25>
  12:   48 8d 0d 00 00 00 00    lea    0x0(%rip),%rcx        # 19 <main+0x19>
                        15: IMAGE_REL_AMD64_REL32       .rdata
  19:   e8 00 00 00 00          call   1e <main+0x1e>
                        1a: IMAGE_REL_AMD64_REL32       _Z6printfPKcz
  1e:   31 c0                   xor    %eax,%eax
  20:   48 83 c4 28             add    $0x28,%rsp
  24:   c3                      ret
  25:   48 8d 0d 05 00 00 00    lea    0x5(%rip),%rcx        # 31 <main+0x31>
                        28: IMAGE_REL_AMD64_REL32       .rdata
  2c:   e8 00 00 00 00          call   31 <main+0x31>
                        2d: IMAGE_REL_AMD64_REL32       puts
  31:   eb eb                   jmp    1e <main+0x1e>
  33:   90                      nop
  34:   90                      nop
  35:   90                      nop
  36:   90                      nop
  37:   90                      nop
  38:   90                      nop
  39:   90                      nop
  3a:   90                      nop
  3b:   90                      nop
  3c:   90                      nop
  3d:   90                      nop
  3e:   90                      nop
  3f:   90                      nop
```

Kuva 23. Testiohjelmapohjan objektin sisältö

Jos ensimmäiseen haaran osaan lisätään [[unlikely]]-merkintä (kuva 24), on tulos kuvan 25 mukainen. Huomataan, että printf-komento on nyt siirretty puts-komennon jälkeen. Vastaavasti, jos else-haaraan laitetaan kuvan 26 mukaisesti [[likely]]-merkintä, tai kuvien 24 ja 26 merkinnät laitetaan molemmat (kuva 27), on tuloksena sama kuvan 25 mukainen tiedosto. Likely-haarakkeet saavat siis prioriteetin sekä normaaleihin että unlikely-haarakkeisiin nähden ja normaalit haarakkeet prioriteetin unlikely-haarakkeisiin nähden.

```
1   #include "stdio.h"
2   #include "stdlib.h"
3   int main()
4   {
5       int i = rand() % 2;
6       if (i) [[unlikely]]
7       {
8           printf("true");
9       }
10      else
11      {
12          puts("false");
13      }
14
15      return 0;
16  }
```

Kuva 24. Testiohjelma [[unlikely]]-merkinnällä

```
0000000000000000 <main>:
   0:   48 83 ec 28             sub    $0x28,%rsp
   4:   e8 00 00 00 00          call   9 <main+0x9>
                        5: IMAGE_REL_AMD64_REL32        __main
   9:   e8 00 00 00 00          call   e <main+0xe>
                        a: IMAGE_REL_AMD64_REL32        rand
   e:   a8 01                   test   $0x1,%al
  10:   75 13                   jne    25 <main+0x25>
  12:   48 8d 0d 05 00 00 00    lea    0x5(%rip),%rcx        # 1e <main+0x1e>
                        15: IMAGE_REL_AMD64_REL32       .rdata
  19:   e8 00 00 00 00          call   1e <main+0x1e>
                        1a: IMAGE_REL_AMD64_REL32       puts
  1e:   31 c0                   xor    %eax,%eax
  20:   48 83 c4 28             add    $0x28,%rsp
  24:   c3                      ret
  25:   48 8d 0d 00 00 00 00    lea    0x0(%rip),%rcx        # 2c <main+0x2c>
                        28: IMAGE_REL_AMD64_REL32       .rdata
  2c:   e8 00 00 00 00          call   31 <main+0x31>
                        2d: IMAGE_REL_AMD64_REL32       _Z6printfPKcz
  31:   eb eb                   jmp    1e <main+0x1e>
  33:   90                      nop
  34:   90                      nop
  35:   90                      nop
  36:   90                      nop
  37:   90                      nop
  38:   90                      nop
  39:   90                      nop
  3a:   90                      nop
  3b:   90                      nop
  3c:   90                      nop
  3d:   90                      nop
  3e:   90                      nop
  3f:   90                      nop
```

Kuva 25. Testiohjelman objektin sisältö [[unlikely]]-merkinnällä

```
1  #include "stdio.h"
2  #include "stdlib.h"
3  int main()
4  {
5      int i = rand() % 2;
6      if (i)
7      {
8          printf("true");
9      }
10     else [[likely]]
11     {
12         puts("false");
13     }
14
15     return 0;
16 }
```

Kuva 26. Testiohjelma [[likely]]-merkinnällä

```
1  #include "stdio.h"
2  #include "stdlib.h"
3  int main()
4  {
5      int i = rand() % 2;
6      if (i) [[unlikely]]
7      {
8          printf("true");
9      }
10     else [[likely]]
11     {
12         puts("false");
13     }
14
15     return 0;
16 }
```

Kuva 27. Testiohjelma molemmilla merkinnöillä

Kuvassa 28 on esitetty tapaus, jossa yhdessä ehtolauseessa on käytetty enemmän kuin yhtä samaa merkintää. Tämä aiheuttaa varoituksen kääntäjältä (kuva 29), mutta prosessi menee kuitenkin läpi. Tulos nähdään kuvassa 30.

```cpp
1  #include "stdio.h"
2  #include "stdlib.h"
3  #include "iostream"
4  #include "time.h"
5  int main()
6  {
7      int i = rand() % 3;
8      if (i == 0) [[unlikely]]
9      {
10         printf("0");
11     }
12     else if (i == 1) [[likely]]
13     {
14         puts("1");
15     }
16     else [[likely]]
17     {
18         int t = time(NULL);
19     }
20
21     return 0;
22 }
```

Kuva 28. Testiohjelma kahdella [[likely]]-merkinnällä

```
$ gcc -c -O3 -std=c++20 1.cpp
1.cpp: In function 'int main()':
1.cpp:12:22: warning: both branches of 'if' statement marked as 'likely' [-Wattributes]
   12 |     else if (i == 1) [[likely]]
      |                      ^~~~~~~~~~
......
   16 |     else [[likely]]
      |          ~~~~~~~~~~
```

Kuva 29. Varoitus useiden samojen attribuuttien käyttämisestä

```
0000000000000000 <main>:
   0:   48 83 ec 28             sub    $0x28,%rsp
   4:   e8 00 00 00 00          call   9 <main+0x9>
                        5: IMAGE_REL_AMD64_REL32        __main
   9:   e8 00 00 00 00          call   e <main+0xe>
                        a: IMAGE_REL_AMD64_REL32        rand
   e:   89 c2                   mov    %eax,%edx
  10:   48 98                   cltq
  12:   48 69 c0 56 55 55 55    imul   $0x55555556,%rax,%rax
  19:   89 d1                   mov    %edx,%ecx
  1b:   c1 f9 1f                sar    $0x1f,%ecx
  1e:   48 c1 e8 20             shr    $0x20,%rax
  22:   29 c8                   sub    %ecx,%eax
  24:   8d 0c 40                lea    (%rax,%rax,2),%ecx
  27:   89 d0                   mov    %edx,%eax
  29:   29 c8                   sub    %ecx,%eax
  2b:   74 22                   je     4f <main+0x4f>
  2d:   83 f8 01                cmp    $0x1,%eax
  30:   74 0f                   je     41 <main+0x41>
  32:   31 c9                   xor    %ecx,%ecx
  34:   ff 15 00 00 00 00       call   *0x0(%rip)        # 3a <main+0x3a>
                        36: IMAGE_REL_AMD64_REL32       __imp__time64
  3a:   31 c0                   xor    %eax,%eax
  3c:   48 83 c4 28             add    $0x28,%rsp
  40:   c3                      ret
  41:   48 8d 0d 02 00 00 00    lea    0x2(%rip),%rcx        # 4a <main+0x4a>
                        44: IMAGE_REL_AMD64_REL32       .rdata
  48:   e8 00 00 00 00          call   4d <main+0x4d>
                        49: IMAGE_REL_AMD64_REL32       puts
  4d:   eb eb                   jmp    3a <main+0x3a>
  4f:   48 8d 0d 00 00 00 00    lea    0x0(%rip),%rcx        # 56 <main+0x56>
                        52: IMAGE_REL_AMD64_REL32       .rdata
  56:   e8 00 00 00 00          call   5b <main+0x5b>
                        57: IMAGE_REL_AMD64_REL32       _Z6printfPKcz
  5b:   eb dd                   jmp    3a <main+0x3a>
  5d:   0f 1f 00                nopl   (%rax)
```

Kuva 30. Testiohjelman objektin sisältö kahdella [[likely]]-merkinnällä

Huomataan, että ainakin tässä testitapauksessa, huolimatta varoituksesta, merkinnät toimivat, ja todennäköiseksi asetetut haarat ovat jälleen sijoitettu ensimmäisiksi. Ne eivät kuitenkaan ole alkuperäisessä järjestyksessä keskenään, vaan myöhempi [[likely]]-merkitty haarake on ensin. Todennäköinen syy tällaiselle käytökselle on, että ehtojen järjestystä muokataan sitä mukaa, kun kääntäjä törmää attribuutteihin. Näin viimeisimpänä löytynyt todennäköiseksi merkitty haara laitetaan viimeisimpänä ensimmäiseksi eli järjestys kääntyy. Vastaavasti siis [[unlikely]]-merkintöjen kanssa keskinäinen järjestys pysyy, kuten kuvien 31 ja 32 esimerkistä voidaan nähdä.

```
1   #include "stdio.h"
2   #include "stdlib.h"
3   #include "iostream"
4   #include "time.h"
5   int main()
6   {
7       int i = rand() % 3;
8       if (i == 0) [[unlikely]]
9       {
10          printf("0");
11      }
12      else if (i == 1) [[unlikely]]
13      {
14          puts("1");
15      }
16      else [[likely]]
17      {
18          int t = time(NULL);
19      }
20
21      return 0;
22  }
```

Kuva 31. Testiohjelma kahdella [[unlikely]]-merkinnällä

```
0000000000000000 <main>:
   0:   48 83 ec 28             sub    $0x28,%rsp
   4:   e8 00 00 00 00          call   9 <main+0x9>
                        5: IMAGE_REL_AMD64_REL32        __main
   9:   e8 00 00 00 00          call   e <main+0xe>
                        a: IMAGE_REL_AMD64_REL32        rand
   e:   89 c2                   mov    %eax,%edx
  10:   48 98                   cltq
  12:   48 69 c0 56 55 55 55    imul   $0x55555556,%rax,%rax
  19:   89 d1                   mov    %edx,%ecx
  1b:   c1 f9 1f                sar    $0x1f,%ecx
  1e:   48 c1 e8 20             shr    $0x20,%rax
  22:   29 c8                   sub    %ecx,%eax
  24:   8d 0c 40                lea    (%rax,%rax,2),%ecx
  27:   89 d0                   mov    %edx,%eax
  29:   29 c8                   sub    %ecx,%eax
  2b:   74 14                   je     41 <main+0x41>
  2d:   83 f8 01                cmp    $0x1,%eax
  30:   74 1d                   je     4f <main+0x4f>
  32:   31 c9                   xor    %ecx,%ecx
  34:   ff 15 00 00 00 00       call   *0x0(%rip)        # 3a <main+0x3a>
                        36: IMAGE_REL_AMD64_REL32       __imp__time64
  3a:   31 c0                   xor    %eax,%eax
  3c:   48 83 c4 28             add    $0x28,%rsp
  40:   c3                      ret
  41:   48 8d 0d 00 00 00 00    lea    0x0(%rip),%rcx        # 48 <main+0x48>
                        44: IMAGE_REL_AMD64_REL32       .rdata
  48:   e8 00 00 00 00          call   4d <main+0x4d>
                        49: IMAGE_REL_AMD64_REL32       _Z6printfPKcz
  4d:   eb eb                   jmp    3a <main+0x3a>
  4f:   48 8d 0d 02 00 00 00    lea    0x2(%rip),%rcx        # 58 <main+0x58>
                        52: IMAGE_REL_AMD64_REL32       .rdata
  56:   e8 00 00 00 00          call   5b <main+0x5b>
                        57: IMAGE_REL_AMD64_REL32       puts
  5b:   eb dd                   jmp    3a <main+0x3a>
  5d:   0f 1f 00                nopl   (%rax)
```

Kuva 32. Testiohjelman objektin sisältö kahdella [[unlikely]]-merkinnällä

Käytettäessä useampia merkintöjä näin ei kääntäjä näyttänyt saaneen optimoitua samalla tavalla, vaikka näiden merkintöjen osuus pitkälti samoin toimikin edelleen. Toisin kuin aiemmissa esimerkeissä, kuvista 30 ja 32 nähdään, että ohjelmiin ei ole lisätty tyhjiä bittejä nop-komennoilla sen paremmaksi linjaamiseksi muistin lukemisen kannalta. [41.]

Sen sijaan switch-rakenteen kanssa ei valitusta tule käytettäessä samaa merkintää useampien kohtien kanssa. Alla olevassa kuvan 33 esimerkissä muuttuja on satunnaisesti välillä 0–9. Todennäköiseksi on merkattu kaikki kolmella jaolliset sekä 0, epätodennäköiseksi on merkattu jäljellä olevat kahdella jaolliset ja loput vaihtoehdot ovat ilman merkintää. Jotta tapahtuva ryhmittely attribuuttien osalta nähtäisiin paremmin, haarojen tulostuksiin lisätään kyseisen haaran numero, ettei kääntäjä automaattisesti optimoisi muuten identtisiä komentoja kolmeen vaihtoehtoon useamman sijaan. Saatu tulos on esitetty kuvassa 34.

```
#include "stdio.h"
#include "stdlib.h"
#include "iostream"
#include "time.h"
int main()
{
    int i = rand() % 10;
    int t;
    switch(i)
    {
        case 0: [[likely]]
            printf("0: likely");
            break;
        case 1:
            t = time(NULL);
            break;
        case 2: [[unlikely]]
            puts("2: unlikely");
            break;
        case 3: [[likely]]
            printf("3: likely");
            break;
        case 4: [[unlikely]]
            puts("4: unlikely");
            break;
        case 5:
            t = time(NULL);
            break;
        case 6: [[likely]]
            printf("6: likely");
            break;
        case 7:
            t = time(NULL);
            break;
        case 8: [[unlikely]]
            puts("8: unlikely");
            break;
        case 9: [[likely]]
            printf("9: likely");
            break;
        default: [[unlikely]]
            puts("default: unlikely");
            break;

    }
    return 0;
}
```

Kuva 33. Testiohjelma switch-casen kanssa

```
0000000000000000 <main>:
   0:   48 83 ec 28             sub    $0x28,%rsp
   4:   e8 00 00 00 00          call   9 <main+0x9>
                        5: IMAGE_REL_AMD64_REL32        __main
   9:   e8 00 00 00 00          call   e <main+0xe>
                        a: IMAGE_REL_AMD64_REL32        rand
   e:   48 63 d0                movslq %eax,%rdx
  11:   89 c1                   mov    %eax,%ecx
  13:   48 69 d2 67 66 66 66    imul   $0x66666667,%rdx,%rdx
  1a:   c1 f9 1f                sar    $0x1f,%ecx
  1d:   48 c1 fa 22             sar    $0x22,%rdx
  21:   29 ca                   sub    %ecx,%edx
  23:   8d 14 92                lea    (%rdx,%rdx,4),%edx
  26:   01 d2                   add    %edx,%edx
  28:   29 d0                   sub    %edx,%eax
  2a:   83 f8 09                cmp    $0x9,%eax
  2d:   0f 87 81 00 00 00       ja     b4 <main+0xb4>
  33:   48 8d 15 60 00 00 00    lea    0x60(%rip),%rdx        # 9a <main+0x9a>
                        36: IMAGE_REL_AMD64_REL32       .rdata
  3a:   48 63 04 82             movslq (%rdx,%rax,4),%rax
  3e:   48 01 d0                add    %rdx,%rax
  41:   ff e0                   jmp    *%rax
  43:   31 c9                   xor    %ecx,%ecx
  45:   ff 15 00 00 00 00       call   *0x0(%rip)        # 4b <main+0x4b>
                        47: IMAGE_REL_AMD64_REL32       __imp__time64
  4b:   31 c0                   xor    %eax,%eax
  4d:   48 83 c4 28             add    $0x28,%rsp
  51:   c3                      ret
  52:   48 8d 0d 00 00 00 00    lea    0x0(%rip),%rcx        # 59 <main+0x59>
                        55: IMAGE_REL_AMD64_REL32       .rdata
  59:   e8 00 00 00 00          call   5e <main+0x5e>
                        5a: IMAGE_REL_AMD64_REL32       _Z6printfPKcz
  5e:   eb eb                   jmp    4b <main+0x4b>
  60:   48 8d 0d 16 00 00 00    lea    0x16(%rip),%rcx        # 7d <main+0x7d>
                        63: IMAGE_REL_AMD64_REL32       .rdata
  67:   e8 00 00 00 00          call   6c <main+0x6c>
                        68: IMAGE_REL_AMD64_REL32       _Z6printfPKcz
  6c:   eb dd                   jmp    4b <main+0x4b>
  6e:   48 8d 0d 2c 00 00 00    lea    0x2c(%rip),%rcx        # a1 <main+0xa1>
                        71: IMAGE_REL_AMD64_REL32       .rdata
  75:   e8 00 00 00 00          call   7a <main+0x7a>
                        76: IMAGE_REL_AMD64_REL32       _Z6printfPKcz
  7a:   eb cf                   jmp    4b <main+0x4b>
  7c:   48 8d 0d 42 00 00 00    lea    0x42(%rip),%rcx        # c5 <main+0xc5>
                        7f: IMAGE_REL_AMD64_REL32       .rdata
  83:   e8 00 00 00 00          call   88 <main+0x88>
                        84: IMAGE_REL_AMD64_REL32       _Z6printfPKcz
  88:   eb c1                   jmp    4b <main+0x4b>
  8a:   48 8d 0d 36 00 00 00    lea    0x36(%rip),%rcx        # c7 <main+0xc7>
                        8d: IMAGE_REL_AMD64_REL32       .rdata
  91:   e8 00 00 00 00          call   96 <main+0x96>
                        92: IMAGE_REL_AMD64_REL32       puts
  96:   eb b3                   jmp    4b <main+0x4b>
  98:   48 8d 0d 20 00 00 00    lea    0x20(%rip),%rcx        # bf <main+0xbf>
                        9b: IMAGE_REL_AMD64_REL32       .rdata
  9f:   e8 00 00 00 00          call   a4 <main+0xa4>
                        a0: IMAGE_REL_AMD64_REL32       puts
  a4:   eb a5                   jmp    4b <main+0x4b>
  a6:   48 8d 0d 0a 00 00 00    lea    0xa(%rip),%rcx        # b7 <main+0xb7>
                        a9: IMAGE_REL_AMD64_REL32       .rdata
  ad:   e8 00 00 00 00          call   b2 <main+0xb2>
                        ae: IMAGE_REL_AMD64_REL32       puts
  b2:   eb 97                   jmp    4b <main+0x4b>
  b4:   48 8d 0d 4c 00 00 00    lea    0x4c(%rip),%rcx        # 107 <_GLOBAL__sub_I_main+0x37>
                        b7: IMAGE_REL_AMD64_REL32       .rdata
  bb:   e8 00 00 00 00          call   c0 <main+0xc0>
                        bc: IMAGE_REL_AMD64_REL32       puts
  c0:   eb 89                   jmp    4b <main+0x4b>
  c2:   66 66 2e 0f 1f 84 00    data16 cs nopw 0x0(%rax,%rax,1)
  c9:   00 00 00 00
  cd:   0f 1f 00                nopl   (%rax)
```

Kuva 34. Testiohjelman objektin sisältö switch-casen kanssa

Vaihtoehdot, joissa ei ollut merkintöjä, ovat optimoituna yhteen ryhmään. Näin tapahtuisi myös ilman attribuutteja käännettävässä versiossa, koska kääntäjä saanee käytettyä tehokkaampia taulukkoratkaisuja näille laskupohjaisille haaroille. Käytettäessä tulostuksia laskujen sijaan kaikille haaroille, on tuloksena 10 vaihtoehtoa järjesteltynä todennäköisyysryhmien mukaan.

Koska nämä tavalliset haarat ovat tuloksessa ensimmäisenä huolimatta muista annetuista attribuuteista, voidaan olettaa, että kääntäjän luoma taulukkoratkaisu on aina tärkeysjärjestyksessä ensimmäinen. Koska se käytännössä ottaa sisäänsä useita eri switch-haarakkeeseen saapuvia arvoja, on sen teoreettinen todennäköisyys lähtökohtaisesti verrokkejaan korkeammalla. Esimerkki tällaisesta kääntäjän taulukkokomennosta luettavammassa muodossa on kuvassa 35.

```
jmp [QWORD PTR .L4[0+rax*8]]
```

Kuva 35. Switch-case taulukoituna jmp-komentona

C++20-attribuuttien linkittämistä GCC:n hot- ja cold-attribuutteihin voidaan todentaa kokeilemalla C++20-attribuutteja funktioiden yhteydessä. Standardin mukaan [[likely]] ja [[unlikely]] eivät ole tarkoitettu funktioiden käyttöön [9], mutta GCC:n hot ja cold sen sijaan ovat alun perin juuri funktioattribuutteja [81]. Kuvan 36 esimerkki aiheuttaa kuvassa 37 esitetyn varoituksen. Kyseinen ilmoitus tulee aiemmin kuvassa 19 esitetystä funktiosta, jossa C++-attribuutteja korvataan GCC:n hot- ja cold-merkinnöillä, varoituksesta riippumatta. Operaatiot tapahtuvat taustalla siten tavalliseen tapaan, ja lopputuloksena on kuvassa 38 esitetty sisältö.

```
1   #include "stdio.h"
2   #include "stdlib.h"
3
4   void likely_f(); [[likely]]
5   void likely_f(){ printf("likely");}
6
7   void unlikely_f(); [[unlikely]]
8   void unlikely_f(){ puts("unlikely");}
9
10  int main()
11  {
12      int i = rand() % 2;
13      if (i)
14      {
15          unlikely_f();
16      }
17      else
18      {
19          likely_f();
20      }
21  }
```

Kuva 36. Attribuuttien käyttäminen funktioiden kanssa

```
$ gcc -c -O3 -std=c++20 3.cpp
3.cpp:4:15: warning: ISO C++ 'likely' attribute does not apply to functions; treating as '[[gnu::hot]]' [-Wattributes]
    4 | void likely_f(){ printf("likely");}
      |               ^
3.cpp:6:17: warning: ISO C++ 'unlikely' attribute does not apply to functions; treating as '[[gnu::cold]]' [-Wattributes]
    6 | void unlikely_f(){ puts("unlikely");}
      |                 ^
```

Kuva 37. Varoitus C++-attribuuttien käyttämisestä funktioiden kanssa

```
0000000000000000 <main>:
   0:   48 83 ec 28             sub    $0x28,%rsp
   4:   e8 00 00 00 00          call   9 <main+0x9>
                        5: IMAGE_REL_AMD64_REL32        __main
   9:   e8 00 00 00 00          call   e <main+0xe>
                        a: IMAGE_REL_AMD64_REL32        rand
   e:   a8 01                   test   $0x1,%al
  10:   75 13                   jne    25 <main+0x25>
  12:   48 8d 0d 00 00 00 00    lea    0x0(%rip),%rcx        # 19 <main+0x19>
                        15: IMAGE_REL_AMD64_REL32       .rdata
  19:   e8 00 00 00 00          call   1e <main+0x1e>
                        1a: IMAGE_REL_AMD64_REL32       _Z6printfPKcz
  1e:   31 c0                   xor    %eax,%eax
  20:   48 83 c4 28             add    $0x28,%rsp
  24:   c3                      ret
  25:   e8 00 00 00 00          call   2a <main+0x2a>
                        26: IMAGE_REL_AMD64_REL32       .text.unlikely
  2a:   eb f2                   jmp    1e <main+0x1e>
  2c:   90                      nop
  2d:   90                      nop
  2e:   90                      nop
  2f:   90                      nop
```

Kuva 38. Tulos attribuuttien käyttämisestä funktioiden kanssa

Koska todennäköisiin funktioihin johtavat tiet ja ehdot optimoidaan, on kuumaksi merkitty funktio siirretty kylmän edelle. Lisäksi kuuman funktion sisältö on avattu kappaleessa 3.2 esitetyn periaatteen mukaisesti.

Vaikka [[likely]]- ja [[unlikely]]-attribuutteja voi näin ilmeisesti käyttää myös standardin ulkopuolisilla tavoilla, kuten funktioiden kanssa tai useita samoja merkintöjä samassa ehtolauseessa käyttäen, ei tätä voitane käytännössä juuri hyödyntää. Tämä johtuu tiukasta riippuvuudesta pohjaavaan kääntäjän implementaatioon, joka kehitysympäristön vaihtuessa tai pelkästään ajan kuluessa voi muuttua standardipohjaisen tuen pysyessä silti vakaana.

6 Mittaukset

Attribuuttien suorituskykyvaikutuksia mitattiin neljällä eri kokeella, joista kaikista löytyy verrokkituloksia. Yksi koe löytyy cppreference-sivuston sivusta [[unlikely]]- ja [[likely]]-attribuuteille [85] ja kolme standardiehdotuksesta [69].

Testeihin tehtiin joitakin muutoksia esimerkiksi ajan mittaamiseen ja tulosten tallentamiseen liittyen. Lisäksi standardiehdotuksen testit muokattiin käyttämään C++-standardin attribuutteja ehdotuksessa käytetyn builtin_expect -ominaisuuden sijaan. Jokainen mittaus on jaettu kolmeen versioon: ilman attribuutteja ja kaksi testiä attribuutit eri päin. Alkuperäiset testit löytyvät lähteiden kautta. Ilman attribuutteja olevat versiot näissä mittauksissa käytetyistä ohjelmakoodeista löytyvät liitteistä.

Jokainen testi ajettiin 5000 kertaa jaettuna viiteen erilliseen tuhannen ryhmään. Näin voitiin vähentää mahdollista muiden järjestelmän taustaprosessien vaikutusta laskennalle tuloksissa. Jokaisella kerralla ohjelma luotiin kääntämisestä lähtien uudelleen, jotta profiloidun optimoinnin vaikutusta voitaisiin vähentää. Eri todennäköisyysarvoja käyttävissä testeissä 2 ja 3 jokainen eri todennäköisyys ajettiin samat 5000 kertaa. Tuloksissa käytettiin yksinkertaista keskiarvoa kaikista saaduista arvoista.

Ajaminen tapahtui MSYS2 MingW64 -ympäristössä Windows 10 -käyttöjärjestelmällä, mikä mahdollistaa saman järjestelmän käyttämisen myös UNIX-pohjaisissa ympäristöissä. Kääntäjänä käytössä oli GCC 10.3. Prosessi automatisoitiin käyttämällä kuvien 39 ja 40 mukaisia skriptitiedostoja, joilla määrää on helppo kontrolloida ja uusia testejä lisätä.

```
1  #!/bin/bash
2  tests=( */ )
3  for test in "${tests[@]}"
4  do
5      echo $test
6      ./build.sh $test
7  done
```

Kuva 39. Mittausjärjestelmän ulompi skripti measure.sh

```bash
#!/bin/bash
cd $1
cases=( $( ls . ) )
for case in "${cases[@]}"
do
    echo $case
    cd $case
    for ((i=0; i<1000; i++))
    do
        if [[ $1 == "4/" ]]
        then
            g++ -O3 -std=c++20 "${case}.cpp" "picohttpparser.c" -o "${case}.exe"
        else
            g++ -O3 -std=c++20 "${case}.cpp" -o "${case}.exe"
        fi
        ./"${case}.exe"
    done
    cd ..
done
```

Kuva 40. Mittausjärjestelmän sisempi skripti build.sh

Testissä 4 lähdetiedostoja on enemmän kuin yksi ja komento täytyi siksi asettaa erilaiseksi. Mikäli tällaisia eroavaisuuksia sisältäviä testejä luo lisää, voi olla edullisempaa siirtyä käyttämään esimerkiksi taulukkoa ja eval-komentoa tauluhakuratkaisun luomiseksi, jotta ekstensiiviset ehtotarkastelut voidaan välttää silmukassa.

Ajan mittaamisessa käytettiin std::chrono -pohjaista laskuria (kuva 41). Tulokset tallennettiin csv-tiedostoon (kuva 42), jonka kautta jatkokäsittely esimerkiksi Excelissä on suoraviivaista.

```cpp
//chrono-based timer setup
#include <chrono>
namespace Timer
{
    std::chrono::time_point<std::chrono::high_resolution_clock> t;
    inline void Start()
    {
        t = std::chrono::high_resolution_clock::now();
    }
    inline std::chrono::duration<double> Read()
    {
        return std::chrono::high_resolution_clock::now() - t;
    }
    inline std::chrono::duration<double> Stop()
    {
        const auto res =  std::chrono::high_resolution_clock::now() - t;
        t = std::chrono::time_point<std::chrono::high_resolution_clock>();
        return res;
    }
}
```

Kuva 41. std::chrono -pohjainen ajanottojärjestelmä.

```cpp
#include <iostream>
#include <fstream>
namespace csv
{
    void dump_seconds(float data);
    void dump_ticks(long long data);
    void wipe();

    void dump_seconds(float data)
    {
        std::ofstream f;
        f.open("data.csv", std::ofstream::app);
        if (f)
        {
            f << data << std::endl;
        }
        else
        {
            std::cout << "Problem occurred opening file" << std::endl;
        }
        f.close();
    }

    void dump_ticks(long long data)
    {
        std::ofstream f;
        f.open("data.csv", std::ofstream::app);
        if (f)
        {
            f << data << std::endl;
        }
        else
        {
            std::cout << "Problem occurred opening file" << std::endl;
        }
        f.close();
    }

    void wipe()
    {
        std::ofstream f;
        f.open("data.csv", std::ofstream::trunc);
        f.close();
    }
}
```

Kuva 42. Datan tallentaminen csv-tiedostoon.

6.1 Mittaus 1

Ensimmäinen mittaus on cppreference-sivustolta löytyvä esimerkkiohjelma, jossa attribuutteja käytetään kosinia laskevan ohjelman yhteydessä [85]. Merkintöjä käytetään kohdassa, jossa tarkastellaan annettua parametria rekursiivisissa potenssilaskuissa ja kertomissa ja toimitaan sen mukaan. Alla kuvassa on esitettynä kyseiset kohdat, kun attribuutteja käytetään todennäköisyyden kannalta oikein. Koko ohjelmakoodi ilman attribuutteja löytyy liitteestä 1.

```
constexpr double pow(double x, long long n) noexcept {
    if (n > 0) [[likely]]
        return x * pow(x, n - 1);
    else [[unlikely]]
        return 1;
}
constexpr long long fact(long long n) noexcept {
    if (n > 1) [[likely]]
        return n * fact(n - 1);
    else [[unlikely]]
        return 1;
}
constexpr double cos(double x) noexcept {
    constexpr long long precision{16LL};
    double y{};
    for (auto n{0LL}; n < precision; n += 2LL) {
        [[likely]] y += pow(x, n) / (n & 2LL ? -fact(n) : fact(n));
    }
    return y;
}
```

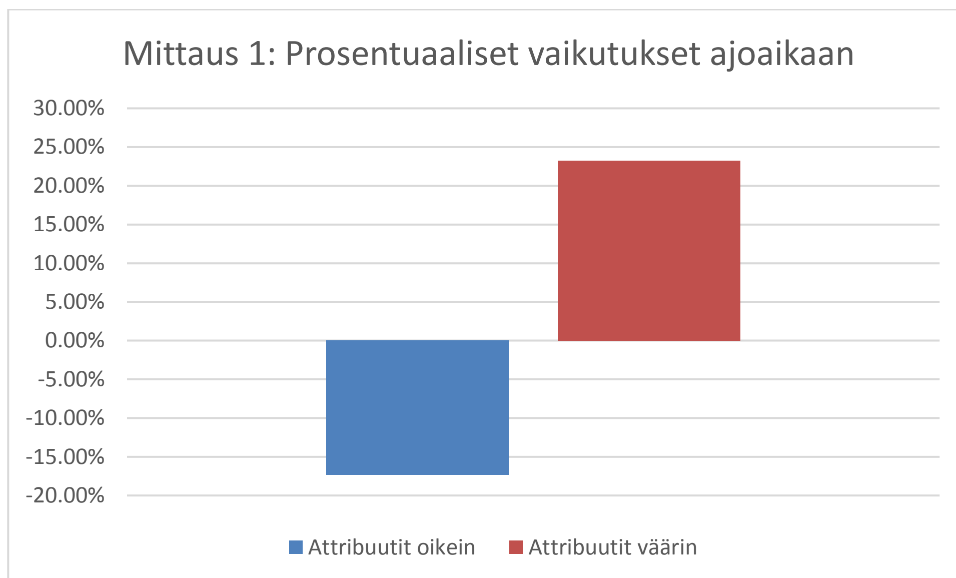
Kuva 43. Attribuuttien käyttäminen mittauksessa 1.

Taulukko 3. Mittauksen 1 ajoaikatulokset

Aika ilman attribuutteja (s)	Aika attribuutit oikein päin (s) n > 0 ja n > 1 merkitty [[likely]]	Aika attribuutit väärin päin (s) n > 0 ja n > 1 merkitty [[unlikely]]
0,375456	0,3103105	0.4627425

Tuloksista nähdään, että attribuuttien käyttämisellä oli selkeä vaikutus ohjelman ajoon. Oikeanlainen käyttö johti keskimäärin jopa yli 17 prosentin optimointiin, kun taas väärin käyttäminen

nosti ajoaikaa yli 23 prosenttia. Prosentuaaliset muutokset ajoajassa on havainnoituna alla kaaviossa.



Kuva 44. Mittauksen 1 prosentuaaliset vaikutukset.

6.2 Mittaus 2

Toisessa mittauksessa ohjelma lukitsee lukuja alueelle [0, 65535]. Merkintöjä käytetään, kun tarkastellaan, onko luku välillä valmiiksi, vai tarvitseeko sitä muokata. Kyseinen funktio merkintöjen ennakoidessa tarvetta muokata lukua on esitettynä kuvassa 41. Luvut luodaan Bernoulli-distribuution avulla, käyttäen eri todennäköisyyksiä. Koko ohjelma ilman attribuutteja löytyy liitteestä 2.

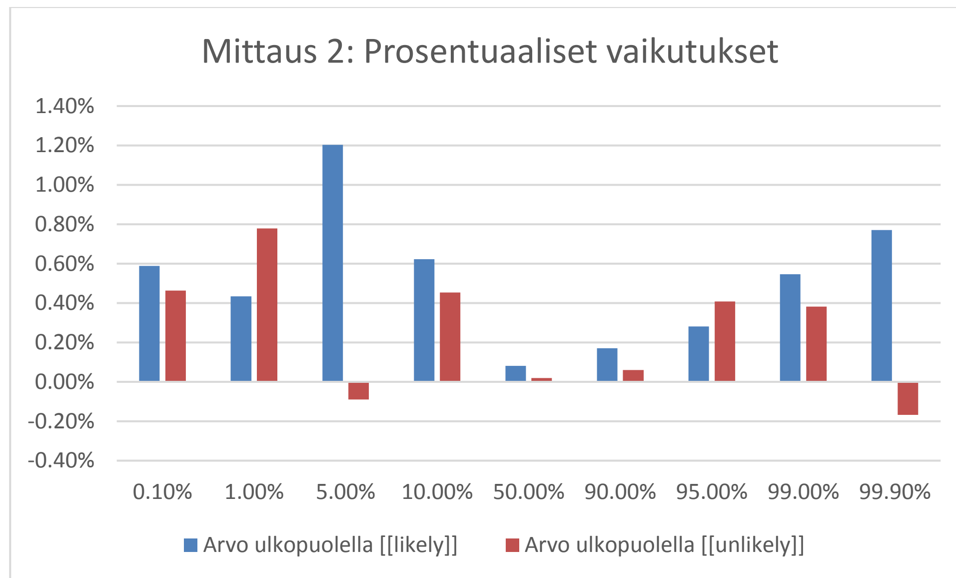
```cpp
std::uint16_t clamp(int i) {
    if (i < 0) [[likely]]
    {
        return 0;
    }
    else if (i > 0xFFFF) [[likely]]
    {
        return 0xFFFFu;
    }
    else [[unlikely]]
    {
        return i;
    }
}
```

Kuva 45. Attribuuttien käyttäminen mittauksessa 2.

Taulukko 4. Mittauksen 2 ajoaikatulokset

% arvoja alueen [0,65535] ulkopuolella	Aika ilman attribuutteja (s)	Aika, kun luku alueen ulkopuolella merkitty [[likely]]	Aika, kun luku alueen ulkopuolella merkitty [[unlikely]]
0,1	0,003756118	0,003778243	0,003773518
1	0,003891617	0,003908491	0,003921911
5	0,004525331	0,004579787	0,004521308
10	0,005340753	0,00537399	0,005365
50	0,011856321	0,011865973	0,011858657
90	0,006130725	0,006141216	0,006134395
95	0,005350427	0,005365429	0,005372284
99	0,004778444	0,004804559	0,004796721
99,9	0,004661057	0,004696984	0,00465326

Tuloksista nähdään, ettei attribuuttien käytöllä ollut käytännössä vaikutusta. Muutokset ovat minimaalisia, eivätkä ne näytä seuraavaan mitään kaavaakaan, vaan vaikuttavat enemmän luonnolliselta heittelyltä. Prosentuaaliset tulokset on havainnollistettu alla kaaviossa.



Kuva 46. Mittauksen 2 prosentuaaliset vaikutukset.

6.3 Mittaus 3

Kolmannen mittauksen ohjelmassa luodaan taulukko lukuja 1 ja 0 käyttämällä eri todennäköisyyksiä määrittämään kummankin luvun arvioitua määrää. Sen jälkeen tarkastellaan, onko luku 0 vai ei, käyttäen attribuutteja tässä ehtolauseessa kuten alla olevassa kuvassa. Koko ohjelma ilman attribuutteja löytyy liitteestä 3.

```
std::uint32_t result = 0;
for (int i = 0; i != 10000; ++i) {
    for (auto val : data) {
        if (val != 0) [[likely]]
        {
            result = i;
        }
    }
}
```

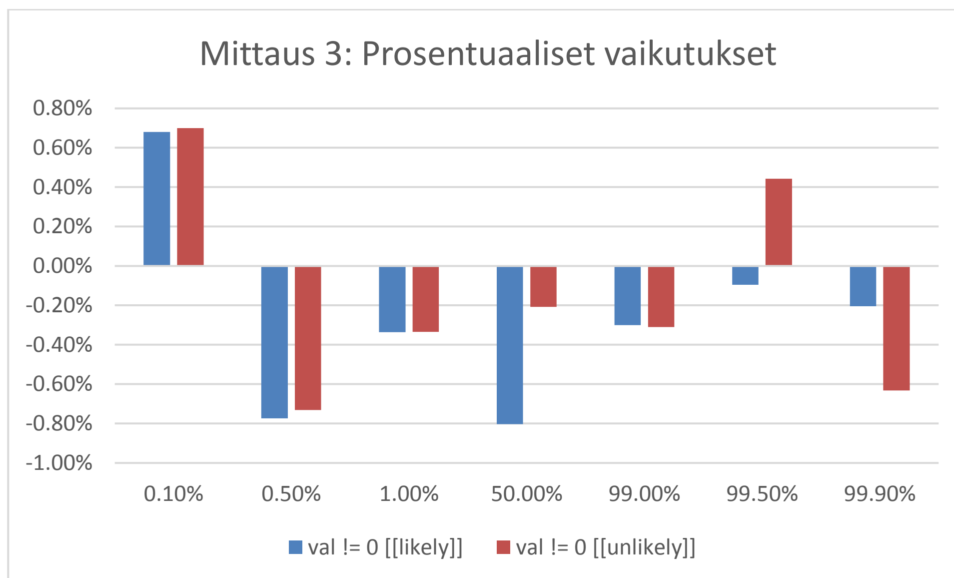
Kuva 47. Attribuuttien käyttäminen mittauksessa 3.

Standardiehdotus mainitsee tarkastelleensa tällä kokeella myös attribuuttien suorituskyvyn vertautumista cmov-komentoon [69], mutta näissä mittauksissa huomattiin, että kääntäjä ei käyttänyt cmov-komentoja lainkaan, riippumatta ehdotuksessa mainituista kääntäjäkomennoista, vaan luodut objektit sisälsivät ehdottomia siirtymisiä. Tämä ero johtunee kääntäjien kehityksestä tällä aikavälillä, sillä näissä mittauksissa käytetty kääntäjän versio on huomattavasti uudempi kuin standardiehdotuksessa.

Taulukko 5. Mittauksen 3 ajoaikatulokset.

% lukuja asetettu arvoon 1	Aika ilman attribuutteja (s)	Aika, kun luku = 1 merkitty [[likely]]	Aika, kun luku = 1 merkitty [[unlikely]]
0,1	0,001347153	0,001356317	0,001356585
0,5	0,001362134	0,001351607	0,00135218
1	0,001360911	0,001356335	0,001356367
50	0,001374511	0,00136348	0,001371669
99	0,001360369	0,001356276	0,00135616
99,5	0,001357722	0,001356423	0,001363736
99,9	0,00135732	0,001354553	0,001348749

Tuloksista nähdään, että attribuuttien käytöllä ei ollut merkittäviä vaikutuksia ohjelman ajoon. Kuten mittauksessa 2, erot ovat marginaalisia eivätkä näytä noudattavan säännöllisiä kuvioita. Toisin kuin mittauksessa 2, ajoajat eivät muutu juurikaan huolimatta käytettävän todennäköisyyden muutoksista. Tämä johtuu mainituista siirtymäkomennoista, joita käännetty ohjelma käyttää ehtojen sijaan. Koska ehtojen tarkastelua ei ajettaessa ole, eivät ehtojen todennäköisyyttä manipuloivat muutokset luonnollisesti vaikuta ajoaikaan. Prosentuaaliset tulokset on havainnollistettu alla kaaviossa.



Kuva 48. Mittauksen 3 prosentuaaliset vaikutukset

6.4 Mittaus 4

Neljäntenä mittauksena käytössä oli käytännönläheisempi esimerkki standardiehdotuksesta, jossa attribuutteja hyödynnetään picohttp-jäsentimen suorituskykytestin yhteydessä [69, 86]. Bench.c -tiedoston testiä muokattiin niin, että varsinainen prosessi tehdään, jos rand-funktio antaa muuta kuin nollan – eli hyvin suurella todennäköisyydellä. Else-haarake simuloi näin siis oikeassa vastaavanlaisessa järjestelmässä olevaa virhehaaraketta, johon tyypillisesti harvoin saavutaan. Tällä rakenteella kymmenestä miljoonasta kierroksesta keskimäärin noin 300 päätyi else-haaraan. Alla kuvassa on esitettynä kyseiset kohdat, kun attribuutteja käytetään todennäköisyyden kannalta oikein. Main-tiedoston ohjelmakoodi ilman attribuutteja löytyy liitteestä 4.

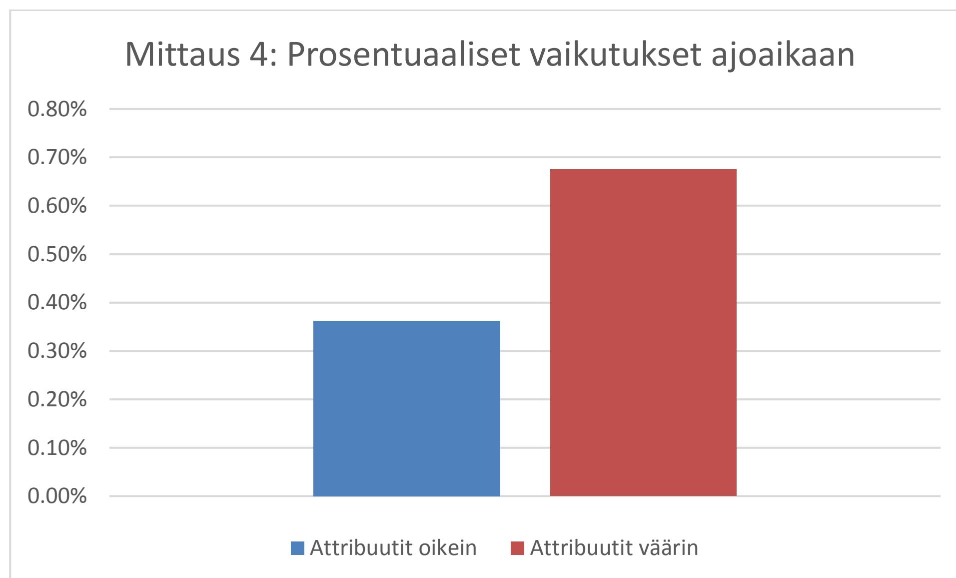
```
for (i = 0; i < 10000000; i++) {
    if (rand()) [[likely]]
    {
        num_headers = sizeof(headers) / sizeof(headers[0]);
        ret = phr_parse_request(REQ, sizeof(REQ) - 1, &method, &method_len,
                                &path, &path_len, &minor_version, headers, &num_headers, 0);
        assert(ret == sizeof(REQ) - 1);
    }
    else [[unlikely]]
    {
        zeros++;
    }
}
```

Kuva 49. Attribuuttien käyttäminen mittauksessa 4.

Taulukko 6. Mittauksen 4 ajoaikatulokset.

Aika ilman attribuutteja (s)	Aika attribuutit oikein päin (s) rand != 0 merkitty [[likely]]	Aika attribuutit väärin päin (s) rand != 0 merkitty [[unlikely]]
2,97687121	2,98766198	2,99697491

Tuloksista nähdään, ettei attribuuttien käytöllä ollut käytännössä vaikutusta. Muutokset eivät ole pelkästään minimaalisia, vaan riippumatta kummin päin attribuutit olivat, keskimääräinen aika meni yli vastaavan attribuutittoman version. Prosentuaaliset tulokset on havainnollistettu alla kaaviossa.



Kuva 50. Mittauksen 4 prosentuaaliset vaikutukset.

6.5 Tulosten yhteenveto ja analysointi

Tuloksista huomataan, että kolmessa neljästä mittauksesta attribuuttien käyttämisellä ei ollut huomattavia vaikutuksia. Erot ovat marginaalisia ja satunnaisia, joten on syytä olettaa, ettei yksinkertaista jatkuvaa variaatioita suurempia muutoksia ohjelman suorituskyvyssä mittauksissa 2–4 tapahtunut, toisin kuin standardiehdotuksessa samojen mittauksien kanssa. Syy tälle erolle löytynee prosessorien ja kääntäjien kehityksestä. Ero kääntäjässä tuli ilmi esimerkiksi mittauksessa

3, jossa luotu tiedosto käytti siirtymäkomentoja myös attribuuttien kanssa. Näin tuloksena oli samankaltaisia vakioajoaikoja kuin standardiehdotuksen cmov-versiossa kyseisestä mittauksesta. Attribuuteilla ei siten myöskään ollut vaikutusta.

Eroja voi aiheuttaa myös eri ominaisuuksien käyttäminen standardiehdotuksen käyttäessä builtin_expect -funktiota C++20-standardin attribuuttien sijaan. Kovin olennainen tekijä tämä ei oletuksena kuitenkaan liene, sillä molemmat ominaisuudet ovat monella tapaa samaan tarkoitukseen, ja voivat olla myös linkitettyjä samoihin sisäisiin toteutuksiin kääntäjissä.

Prosessorien kehitys voidaan heti nähdä suurista eroista tulosten tasossa: standardiehdotuksen mittauksissa samoihin ohjelmiin menee useita sekunteja näissä mittauksissa saatujen minimaalisten aikojen sijaan. Toisaalta mittauksessa 4 ajat ovat suunnilleen samaa suuruusluokkaa molemmissa. Huomattavampaa ovat kuitenkin muutokset kuten mittauksessa 2 havaitut. Standardiehdotuksessa aika kasvaa jatkuvasti, kun todennäköisyyttä luvun alueen ulkopuolella olemiselle lisätään, mikä johtaa lopulta nelin- tai kuusinkertaiseen aikaan riippuen attribuuttien käytöstä. Näissä mittauksissa sen sijaan attribuuttien käytöllä ei ollut merkitystä ja aika maksimoitui noin kolminkertaiseksi satunnaisimman kierroksen kohdalla, eli 50 % todennäköisyydellä. Vastaavasti todennäköisyyden kasvaessa tai vähentyessä ajoaika pienentyi. Voidaan siis huomata, että prosessorin haarojen käsittely ottaa huomattavasti paremmin huomioon ehdon tuloksen todennäköisyyden muutokset. Syy attribuuttien vaikutuksen puuttumiselle tässä ja muissa mittauksissa johtunee yksinkertaisesti siitä, että useat modernit prosessorit ohittavat ehtolausevihjeet kokonaan [56], jolloin ainoa jäljelle jäävä vaikutus on mahdollinen optimointi, jota kääntäjä saa aikaiseksi, esimerkiksi kappaleissa 3.2.1–3.2.8 esitettyjen periaatteiden mukaisesti.

Näin voidaan olettaa, että mittauksissa 2–4 kääntäjä joko ei saanut huomattavaa optimointia tehtyä tai luotu ratkaisu ei enää riippunut ehtolauseista, kuten mittauksen 3 kohdalla havainnointiin. Vastaavasti mittauksen 1 tulokset viittaavat selkeästi onnistuneeseen optimointiin. Huomattavin ero mittausten 1 ja 2–4 ohjelmakoodien välillä tätä ajatellen on, että mittauksessa 1 attribuutteja käyttävät kohteet eivät riipu satunnaisuudesta tai todennäköisyydestä, vaan kyseessä on for-silmukan iteraation sisäiset tapahtumat, jotka suoritetaan aina samalla tavalla. Siten voidaan päätellä, että kääntäjälle – ainakin käytössä olleen GCC:n toteutukselle – oli mahdollista optimoida tätä rakennetta paremmin kuin muissa mittauksissa käytettyjä yksinkertaisia satunnaisarvoihin perustuvia ehtolauseita. Toisaalta vaikutuksen voi saada aikaan myös jokin muu erottava tekijä ohjelmakoodien välillä, ja näiden testien perusteella on mahdotonta vetää lopullisia johtopäätöksiä.

Mittauksista kokonaisuutena voidaan nähdä, että joissakin tapauksissa attribuuttien käyttämisellä voi olla huomattavia vaikutuksia, niin parempaan kuin huonompaan. Suurimmassa osassa tapauksista minkäänlaisia vaikutuksia ei kuitenkaan ole havaittavissa. Lisää optimointimahdollisuuksia etsivä voi kokeilla C++20-standardin attribuutteja, mutta laajalle käyttäjäkunnalle paremmat optimointiominaisuudet ja -periaatteet löytynevät toisaalta.

Lähteet

1 Stroustrup, Bjarne. (1994). The Design and Evolution of C++. Reading, MA, USA: Addison-Wesley.

2 Stroustrup, Bjarne. (1991). A History of C++: 1979-1991. Saatavilla osoitteesta: https://www.stroustrup.com/hopl2.pdf. Haettu 14.6.2021.

3 Stroustrup, Bjarne. (2007). Evolving a language in and for the real world: C++ 1991-2006. Saatavilla osoitteesta: https://www.stroustrup.com/hopl-almost-final.pdf. Haettu 16.6.2021.

4 Stroustrup, Bjarne. (2020). Thriving in a Crowded and Changing World: C++ 2006–2020. Saatavilla osoitteesta: https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf. Haettu 16.6.2021.

5 TIOBE Software BV. (2021). TIOBE Index for June 2021. Saatavilla osoitteesta: https://www.tiobe.com/tiobe-index/. Haettu 15.6.2021.

6 Stroustrup, Bjarne. (Modified May 13, 2021). C++ Applications. Saatavilla osoitteesta: https://www.stroustrup.com/applications.html. Haettu 15.6.2021.

7 Stroustrup, Bjarne. (2013). The C++ Programming Language Fourth Edition. Upper Saddle River, NJ, USA: Addison-Wesley.

8 Fog, Agner. (2004-2021). Optimizing software in C++. Saatavilla osoitteesta: https://www.agner.org/optimize/optimizing_cpp.pdf. Haettu 4.6.2021

9 International Organization for Standardization. (2020). ISO/IEC 14882:2020 – Programming Languages – C++. Saatavilla osoitteesta: https://www.iso.org/standard/79358.html. Haettu 16.6.2021.

10 International Organization for Standardization. (2020). ISO/IEC JTC1 SC22 WG21 N 4860. Saatavilla osoitteesta: https://isocpp.org/files/papers/N4860.pdf. Haettu 5.5.2021.

11 Bhargava, Rohit. (2006). 5 Rules of Social Media Optimization (SMO). Saatavilla osoitteesta: https://www.rohitbhargava.com/2006/08/5_rules_of_soci.html. Haettu 18.6.2021.

12 Floudas Christodoulos A, Pardalos Panos M. (2009). Encyclopedia of Optimization Second Edition. New York, NY, USA: Springer Science+Business Media LLC.

13 Knuth, Donald. (1974). Structured Programming with go to Statements. Saatavilla osoitteesta: https://pic.plover.com/knuth-GOTO.pdf. Haettu 25.6.2021.

14 Knuth, Donald. (1968). The Art of Computer Programming, Volume 1: Fundamental Algorithms. Reading, MA, USA: Addison-Wesley.

15 Jackson, Michael A. (1975). Principles of Program Design. London, Great Britain: Academic Press.

16 Moore, Gordon E. (1965). Cramming more components onto integrated circuits. Electronics, 38(8). Saatavilla osoitteesta: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf. Haettu 29.6.2021.

17 Ritchie Hannah, Roser Max. (2020) Moore's Law: The number of transistors on microchips doubles every two years. Saatavilla osoitteesta: https://ourworldindata.org/uploads/2020/11/Transistor-Count-over-time.png. Haettu 29.6.2021.

18 Merritt, Rick. (2013). Moore’s Law Dead by 2022, Expert Says. Saatavilla osoitteesta: https://www.eetimes.com/moores-law-dead-by-2022-expert-says/#. Haettu 29.6.2021.

19 Chojecki, Przemek. (2019). Moore’s law is dead. Saatavilla osoitteesta: https://towardsdatascience.com/moores-law-is-dead-678119754571. Haettu 29.6.2021.

20 Shein, Esther. (2020). Moore's Law turns 55: Is it still relevant? Saatavilla osoitteesta: https://www.techrepublic.com/article/moores-law-turns-55-is-it-still-relevant/. Haettu 29.6.2021.

21 Tibken, Shara. (2019). CES 2019: Moore's Law is dead, says Nvidia's CEO. Saatavilla osoitteesta: https://www.cnet.com/news/moores-law-is-dead-nvidias-ceo-jensen-huang-says-at-ces-2019/. Haettu 29.6.2021.

22 Sperling, Ed. (2018). Quantum Effects At 7/5nm And Beyond. Saatavilla osoitteesta: https://semiengineering.com/quantum-effects-at-7-5nm/. Haettu 29.6.2021.

23 DeBenedictis Erik P, Badaroglu Mustafa, Chen An, Conte Thomas M, Gargini Paolo. (2017). Sustaining Moore's Law with 3D Chips. Saatavilla osoitteesta: http://debenedictis.org/erik/ComputerColumn/SAND2017-9177_j_3D_Chips.pdf. Haettu 29.6.2021.

24 DeBenedictis Erik P. (2017). It's Time to Redefine Moore's Law Again. Saatavilla osoitteesta: https://www.osti.gov/pages/servlets/purl/1377816. Haettu 29.6.2021.

25 Yang, Sarah. (2016). Smallest. Transistor. Ever. Saatavilla osoitteesta: https://newscenter.lbl.gov/2016/10/06/smallest-transistor-1-nm-gate/. Haettu 29.6.2021.

26 Wirth, Niklaus. (1995). A Plea for Lean Software. Computer, 28(2), 64–68. Saatavilla osoitteesta: https://people.inf.ethz.ch/wirth/Articles/LeanSoftware.pdf. Haettu 29.6.2021.

27 Parkinson, C. Northcote. (1955). Parkinson's Law. The Economist, 19 November 1955. Saatavilla osoitteesta: https://www.economist.com/news/1955/11/19/parkinsons-law. Haettu 29.6.2021.

28 Calder Brad G, Grunwald Dirk C, Zorn Benjamin G. (1994). Quantifying Behavioral Differences Between C and C++ Programs. Saatavilla osoitteesta: https://scholar.colorado.edu/concern/reports/z316q2624. Haettu 5.7.2021.

29 Smith, Alan Jay. (1982). Cache Memories. ACM Computing Surveys, 14(3), 473–530. Saatavilla osoitteesta: https://dl.acm.org/doi/10.1145/356887.356892. Haettu 5.7.2021.

30 Hellisp. (2006). Cache, associative-fill-both. Saatavilla osoitteesta: https://upload.wikimedia.org/wikipedia/commons/9/93/Cache%2Cassociative-fill-both.png. Haettu 21.9.2021.

31 Drepper, Ulrich. (2007). Memory part 2: CPU caches. Saatavilla osoitteesta: https://lwn.net/Articles/252125/. Haettu 21.9.2021.

32 Advanced Micro Devices Inc. AMD Ryzen™ 7 2700X Processor. Saatavilla osoitteesta: https://www.amd.com/en/products/cpu/amd-ryzen-7-2700x. Haettu 5.7.2021.

33 Aho Alfred V, Lam Monica S, Sethi Ravi, Ullman Jeffrey D. (2014). Compilers: Principles, Techniques, and Tools Second Edition. Harlow, Great Britain: Pearson Education.

34 Allen Frances E, Cocke John. (1971). A Catalogue of Optimizing Transforms. Saatavilla osoitteesta: https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf. Haettu 6.7.2021.

35 Cassel, David. (2019). Rust Creator Graydon Hoare Recounts the History of Compilers. Saatavilla osoitteesta: https://thenewstack.io/rust-creator-graydon-hoare-recounts-the-history-of-compilers/. Haettu 6.7.2021.

36 Peyton Jones Simon, Marlow Simon. (1999). Secrets of the Glasgow Haskell Compiler inliner. Journal of Functional Programming, 12, 393-434. Saatavilla osoitteesta: https://www.microsoft.com/en-us/research/wp-content/uploads/2002/07/inline.pdf. Haettu 7.7.2021.

37 Kennedy, Ken. (1973). Reduction in Strength Using Hashed Temporaries. Saatavilla osoitteesta: http://www.softwarepreservation.org/projects/SETL/setl/newsletter/setl_102_1973-03-12.pdf. Haettu 9.7.2021.

38 Cooper Keith, Simpson Taylor, Vick Christopher. (1995). Operator Strength Reduction. Saatavilla osoitteesta: http://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR95635-S.pdf. Haettu 9.7.2021.

39 McKeeman, William M. (1965). Peephole optimization. Communications of the ACM, 8(7), 443-444. Saatavilla osoitteesta: https://dl.acm.org/doi/10.1145/364995.365000. Haettu 12.7.2021.

40 Maurer Jens, Wong Michael. (2008). Towards support for attributes in C++. Saatavilla osoitteesta: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf. Haettu 18.7.2021.

41 Free Software Foundation Inc. 3.11 Options That Control Optimization. Saatavilla osoitteesta: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Haettu 12.7.2021.

42 Cburnett, Inductiveload, Nyq. (2006-2015). Pipeline, 4 stage. Saatavilla osoitteesta: https://upload.wikimedia.org/wikipedia/commons/c/cb/Pipeline%2C_4_stage.svg. Haettu 18.7.2021.

43 Eyerman Stijn, Smith James E, Eeckhout Lieven. (2006). Characterizing the Branch Misprediction Penalty. Saatavilla osoitteesta: https://users.elis.ugent.be/~leeckhou/papers/ispass06-eyerman.pdf. Haettu 19.7.2021.

44 Mysticial. (2012). Vastaus kysymykseen "Why is processing a sorted array faster than processing an unsorted array?" sivustolla StackOverflow. Saatavilla osoitteesta: https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array. Haettu 19.7.2021.

45 Vinyard, Rick. (2000). Predication. Saatavilla osoitteesta: https://www.cs.nmsu.edu/~rvinyard/itanium/predication.htm. Haettu 21.7.2021.

46 Park Joseph C H, Schlansker Mike. (1991). On Predicated Execution. Saatavilla osoitteesta: https://www.hpl.hp.com/techreports/91/HPL-91-58.pdf. Haettu 21.7.2021.

47 Torvalds, Linus. (2007). Re: kernel + gcc 4.1 = several problems. Saatavilla osoitteesta: https://yarchive.net/comp/linux/cmov.html. Haettu 21.7.2021.

48 Smotherman, Mark. (2010). IBM Stretch (7030) -- Aggressive Uniprocessor Parallelism. Saatavilla osoitteesta: https://people.cs.clemson.edu/~mark/stretch.html. Haettu 2.8.2021.

49 Bahnsen R J, Dirac J F. (1957). Sigma Computer Memo #2. Saatavilla osoitteesta: http://archive.computerhistory.org/resources/text/IBM/Stretch/pdfs/06-19/102632583.pdf. Haettu 2.8.2021.

50 Chen, Tien Chi. (1961). Design of Floating Point Arithmetic. Saatavilla osoitteesta: http://archive.computerhistory.org/resources/text/IBM/Stretch/pdfs/09-05/102634376.pdf. Haettu 2.8.2021.

51 Smotherman, Mark. (2013). S-1 Supercomputer (1975-1988). Saatavilla osoitteesta: https://people.cs.clemson.edu/~mark/s1.html. Haettu 3.8.2021.

52 Potash, Hanan. (1984). Branch predicting computer. US 4435756. Saatavilla osoitteesta: https://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO2&Sect2=HITOFF&p=1&u=%2Fnetahtml%2FPTO%2Fsearch-bool.html&r=1&f=G&l=50&co1=AND&d=PTXT&s1=4435756.PN.&OS=PN/4435756&RS=PN/4435756. Haettu 6.8.2021.

53 Fite, David B Jr, Fossum Tryggve, Manley Dwight. (1990). Design Strategy for the VAX 9000 System. Julkaisussa: VAX 9000 SERIES, Digital Technical Journal of Digital Equipment Corporation, 2(4). Bedford, MA, USA: Digital's Educational Services Media Communications Group.

54 Boswoth, Edward. The Evolution of Microprogramming. Saatavilla osoitteesta: http://www.edwardbosworth.com/My5155_Slides/Chapter09/Microprogramming-History.htm. Haettu 3.8.2021.

55 Warford, J. Stanley. (2010). Computer Systems 4th Edition. Sudbury, MA, USA: Jones and Bartlett Publishers.

56 Fog, Agner. (1996-2021). The microarchitecture of Intel, AMD, and VIA CPUs. Saatavilla osoitteesta: https://www.agner.org/optimize/microarchitecture.pdf. Haettu 3.6.2021.

57 Shen John Paul, Lipasti Mikko H. (2005). Modern Processor Design: Fundamentals of Superscalar Processors. Long Grove, IL, USA: Waveland Press Inc.

58 Calder Brad, Grunwald Dirk, Jones Michael, Lindsay Donald, Martin James, Mozer Michael, Zorn Benjamin. (1997). Evidence-based Static Branch Prediction using Machine Learning. Saatavilla osoitteesta: https://cseweb.ucsd.edu/~calder/papers/TOPLAS-97-ESP.pdf. Haettu 8.8.2021.

59 Yeh Tse-Yu, Patt Yale N. (1991). Two-Level Adaptive Training Branch Prediction. MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture, 51-61. Saatavilla osoitteesta: https://dl.acm.org/doi/10.1145/123465.123475. Haettu 8.8.2021.

60 Cheng, Chih Cheng. The Schemes and Performances of Dynamic Branch predictors. Saatavilla osoitteesta: http://bwrcs.eecs.berkeley.edu/Classes/CS252/Projects/Reports/terry_chen.pdf. Haettu 4.6.2021.

61 Advanced Micro Devices Inc. (2017). Software Optimization Guide for AMD Family 17h Processors. Saatavilla osoitteesta: https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf. Haettu 3.6.2021.

62 Sprangle Eric, Chappell Robert S, Alsup Mitch, Patt Yale N. (1997). The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. ACM SIGARCH Computer Architecture News, 25(2), 284-291. Saatavilla osoitteesta: https://dl.acm.org/doi/10.1145/384286.264210. Haettu 10.8.2021.

63 Pan Shien-Tai, So Kimming, Rahmeh Joseph T. (1992). Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. ACM SIGPLAN Notices, 27(9), 76-84. Saatavilla osoitteesta: https://dl.acm.org/doi/10.1145/143371.143490. Haettu 10.8.2021.

64 McFarling, Scott. (1993). Combining Branch Predictors. Saatavilla osoitteesta: https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf. Haettu 8.8.2021.

65 Vintan, Lucian N. (1999). Towards a High Performance Neural Branch Predictor. Saatavilla osoitteesta: https://web.archive.org/web/20190713224752/http://webspace.ulbsibiu.ro/lucian.vintan/html/USA.pdf. Haettu 10.8.2021.

66 Jiménez Daniel A, Lin Calvin. (2001). Dynamic Branch Prediction with Perceptrons. Saatavilla osoitteesta: https://www.cs.utexas.edu/~lin/papers/hpca01.pdf. Haettu 4.6.2021.

67 Jiménez Daniel A. (2003). Fast Path-Based Neural Branch Prediction. MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, 243-253. Saatavilla osoitteesta: https://www.microarch.org/micro36/html/pdf/jimenez-FastPath.pdf. Haettu 10.8.2021.

68 Parihar, Raj. (2009). Branch Prediction Techniques and Optimizations. Saatavilla osoitteesta: http://www2.ece.rochester.edu/~parihar/pres/Paper_BrPrediction.pdf. Haettu 10.8.2021.

69 Trychta, Clay. (2016). Attributes for Likely and Unlikely Branches. Saatavilla osoitteesta: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0479r0.html. Haettu 16.6.2021.

70 International Organization for Standardization. (2012). Working Draft, Standard for Programming Language C++. Saatavilla osoitteesta: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf. Haettu 11.8.2021.

71 Maurer Jens, Wong Michael. (2008). Towards support for attributes in C++ (Revision 6). Saatavilla osoitteesta: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf. Haettu 18.7.2021.

72 Free Software Foundation Inc. (2019). GCC 9 Release Series – Changes, New Features, and Fixes. Saatavilla osoitteesta: https://gcc.gnu.org/gcc-9/changes.html. Haettu 12.8.2021.

73 LLVM Project. (2021). Clang 12.0.0 Release Notes. Saatavilla osoitteesta: https://releases.llvm.org/12.0.0/tools/clang/docs/ReleaseNotes.html. Haettu 12.8.2021.

74 Free Software Foundation Inc. 6.59 Other Built-in Functions Provided by GCC. Saatavilla osoitteesta: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. Haettu 12.8.2021.

75 LLVM Project. LLVM Branch Weight Metadata. Saatavilla osoitteesta: https://llvm.org/docs/BranchWeightMetadata.html. Haettu 12.8.2021.

76 Free Software Foundation Inc. gcc/libitm/common.h. Saatavilla osoitteesta: https://github.com/gcc-mirror/gcc/blob/16e2427f50c208dfe07d07f18009969502c25dc8/libitm/common.h. Haettu 12.8.2021.

77 LLVM Project. llvm-project/libc/src/__support/common.h. Saatavilla osoitteesta: https://github.com/llvm/llvm-project/blob/d480f968ad8b56d3ee4a6b6df5532d485b0ad01e/libc/src/__support/common.h. Haettu 12.8.2021.

78 Trychta, Clay. (2018). Proposed wording for likely and unlikely attributes (Revision 5). Saatavilla osoitteesta: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0479r5.html. Haettu 12.6.2021.

79 Free Software Foundation Inc. gcc/cp/parser.c. Saatavilla osoitteesta: https://raw.githubusercontent.com/gcc-mirror/gcc/master/gcc/cp/parser.c. Haettu 11.8.2021.

80 Free Software Foundation Inc. gcc/cp/cp-gimplify.c. Saatavilla osoitteesta: https://github.com/gcc-mirror/gcc/blob/16e2427f50c208dfe07d07f18009969502c25dc8/gcc/cp/cp-gimplify.c. Haettu 11.8.2021.

81 Free Software Foundation Inc. 6.33.1 Common Function Attributes. Saatavilla osoitteesta: https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html. Haettu 11.8.2021.

82 Free Software Foundation Inc. gcc/cp/tree.c. Saatavilla osoitteesta: https://github.com/gcc-mirror/gcc/blob/master/gcc/cp/tree.c. Haettu 11.8.2021.

83 Free Software Foundation Inc. gcc/tree.def. Saatavilla osoitteesta: https://github.com/gcc-mirror/gcc/blob/master/gcc/tree.def. Haettu 13.8.2021.

84 Free Software Foundation Inc. gcc/predict.c. Saatavilla osoitteesta: https://github.com/gcc-mirror/gcc/blob/master/gcc/predict.c. Haettu 12.8.2021.

85 C++ attribute: likely, unlikely (since C++20). Saatavilla osoitteesta: https://en.cppreference.com/w/cpp/language/attributes/likely. Haettu 7.6.2021.

86 The H2O Project. (2009-2014). picohttpparser. Saatavilla osoitteesta: https://github.com/h2o/picohttpparser. Haettu 12.8.2021.

Liitteet

```
#include <chrono>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <random>

#include "../../csv.h"
#include "../../timer.h"

namespace no_attributes {
    constexpr double pow(double x, long long n) noexcept {
        if (n > 0)
            return x * pow(x, n - 1);
        else
            return 1;
    }
    constexpr long long fact(long long n) noexcept {
        if (n > 1)
            return n * fact(n - 1);
        else
            return 1;
    }
    constexpr double cos(double x) noexcept {
        constexpr long long precision{16LL};
        double y{};
        for (auto n{0LL}; n < precision; n += 2LL) {
            y += pow(x, n) / (n & 2LL ? -fact(n) : fact(n));
        }
        return y;
    }
}  // namespace no_attributes

double gen_random() noexcept {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    static std::uniform_real_distribution<double> dis(-1.0, 1.0);
    return dis(gen);
}

volatile double sink{}; // ensures a side effect

int main() {
    auto benchmark = [](auto fun, auto rem) {
        Timer::Start();
        for (auto size{1ULL}; size != 10'000'000ULL; ++size) {
            sink = fun(gen_random());
        }
        csv::dump_seconds(Timer::Read().count());
        std::cout << "Time: " << Timer::Read().count()
                  << " sec " << rem << std::endl;
    };
    benchmark(no_attributes::cos, "(without attributes)");
}
```

```
#include <array>
#include <cstdint>
#include <random>
#include <iostream>

#include "../../csv.h"
#include "../../timer.h"

std::uint16_t clamp(int i) {
    if (i < 0)
    {
        return 0;
    }
    else if (i > 0xFFFF)
    {
        return 0xFFFFu;
    }
    else
    {
        return i;
    }
}

int main() {

    Timer::Start();

    std::mt19937 gen(42);
    std::bernoulli_distribution d(.999), up_down(.5);

    std::array<int, 1'000'000> data;
    for (std::size_t i = 0; i != data.size(); ++i) {
        if (d(gen)) {
            data[i] = up_down(gen) ? -1 : 0xFFFFF;
        } else {
            data[i] = 1;
        }
    }

    std::uint32_t result = 0;
    for (int i = 0; i != 10000; ++i) {
        for (auto val : data) {
            result += clamp(val);
        }
    }
    csv::dump_seconds(Timer::Read().count());
    std::cout << "Time: " << Timer::Read().count() << std::endl;
    return 0;
}
```

```
#include <array>
#include <cstdint>
#include <random>

#include "../../csv.h"
#include "../../timer.h"

int main() {

    Timer::Start();
    std::mt19937 gen(42);
    std::bernoulli_distribution d(.999);
    std::array<std::uint32_t, 1'000'000> data;
    for (std::size_t i = 0; i != data.size(); ++i) {
        data[i] = d(gen) ? 1 : 0;
    }

    std::uint32_t result = 0;
    for (int i = 0; i != 10000; ++i) {
        for (auto val : data) {
            if (val != 0)
            {
                result = i;
            }
        }
    }
    csv::dump_seconds(Timer::Read().count());
    std::cout << "Time: " << Timer::Read().count() << std::endl;
    return 0;
}
```

```
/*
 * Copyright (c) 2009-2014 Kazuho Oku, Tokuhiro Matsuno, Daisuke Murase,
 *                         Shigeo Mitsunari
 *
 * The software is licensed under either the MIT License (below) or the Perl
 * license.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to
 * deal in the Software without restriction, including without limitation the
 * rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
 * sell copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
 * IN THE SOFTWARE.
 */

#include <assert.h>
#include <stdio.h>
#include "picohttpparser.h"
#include <iostream>
#include <stdlib.h>

#include "../../csv.h"
#include "../../timer.h"

#define REQ                                                                    \
    "GET /wp-content/uploads/2010/03/hello-kitty-darth-vader-
pink.jpg HTTP/1.1\r\n"                                                         \
    "Host: www.kittyhell.com\r\n"                                              \
    "User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; ja-JP-
mac; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3 "                                \
    "Pathtraq/0.9\r\n"                                                         \
    "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"  \
    "Accept-Language: ja,en-
us;q=0.7,en;q=0.3\r\n"                                                         \
    "Accept-
Encoding: gzip,deflate\r\n"                                                    \
    "Accept-Charset: Shift_JIS,utf-
8;q=0.7,*;q=0.7\r\n"                                                           \
    "Keep-
Alive: 115\r\n"                                                                \
```

```
    "Connection: keep-
alive\r\n"
                          \
    "Cookie: wp_ozh_wsa_visits=2; wp_ozh_wsa_visit_lasttime=xxxxxxxxxx; "
                                                    \
    "__utma=xxxxxxxxx.xxxxxxxxxx.xxxxxxxxxx.xxxxxxxxxx.xxxxxxxxxx.x; "
                                                    \
    "__utmz=xxxxxxxxx.xxxxxxxxxx.x.x.utmccn=(referral)|utmcsr=reader.livedoor.com|utmc
t=/reader/|utmcmd=referral\r\n"                     \
    "\r\n"

int main(void)
{
    const char *method;
    size_t method_len;
    const char *path;
    size_t path_len;
    int minor_version;
    struct phr_header headers[32];
    size_t num_headers;
    int i, ret;

    int zeros = 0;
    srand(time(NULL));

    Timer::Start();

    for (i = 0; i < 10000000; i++) {
        if (rand())
        {
            num_headers = sizeof(headers) / sizeof(headers[0]);
            ret = phr_parse_request(REQ, sizeof(REQ) - 1, &method, &method_len, &path,
&path_len, &minor_version, headers, &num_headers,
                                    0);
            assert(ret == sizeof(REQ) - 1);
        }
        else
        {
            zeros++;
        }
    }
    std::cout << "Zeros: " << zeros << std::endl;
    csv::dump_seconds(Timer::Read().count());
    std::cout << "Time: " << Timer::Read().count() << " sec " <<  std::endl;
    return 0;
}
```