hyväksymispäivä arvosana

arvostelija

Mahdottomien polkujen etsintä ja poisto suoritusaika-analyysissä

Allan Holsti

Helsinki dd.mm.yyyy

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET

Tiedekunta/Osasto	Laitos – Institution
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä – Författare

Allan Holsti

Työn nimi – Arbetets titel

Mahdottomien polkujen etsintä ja poisto suoritusaika-analyysissä

Oppiaine – Läroämne

Tietojenkäsittelytiede

Työn laji – Arbetets art	Aika – Datum	Sivumäärä – Sidoantal
Pro gradu -tutkielma	28.9.2004	

Tiivistelmä – Referat

Avainsanat – Nyckelord

Aika-analyysi, tosiaikaisuus, mahdottomat polut

Säilytyspaikka – Förvaringställe

Kumpulan tiedekirjasto, sarjanumero C-2004-

Muita tietoja – Övriga uppgifter

.

Sisällysluettelo

1. Johdanto

Tämä dokumentti on työn alla oleva pro gradu -tutkielma. Tutkielman tavoitteena on tehdä katsaus suoritusaika-analyysin ja selvittää mahdollisia tapoja käsitellä *mahdottomien polkujen* aiheuttamaa epätarkkuutta analyysissä. Mikäli sopiva menetelmä etsiä ja poistaa mahdottomia polkuja analyysissä löytyy, olisi tämän menetelmän toteutuksen Bound-T [BT] suoritusaika-analyysityökaluun osa tutkielmaa. Bound-T:n toimittaja Tidorum Oy tukee ja osallistuu toteutukseen.

2. Suoritusaika-analyysin haasteet

Suoritusaika-analyysissä on tavoitteena löytää aikarajat ohjelman suorituksen kestolle. Yleensä keskitytään ylärajan eli pahimman suoritusajan (PSA:n) etsimiseen, vaikka alaraja voi myös olla kiinnostava. Esimerkiksi kontrollijärjestelmissä, joissa halutaan reagoida tapahtumiin tietyssä aikaikkunassa voi turhan aikainen suoritus olla yhtä haitallista kuin liian myöhäinen. On tosin huomattavasti helpompaa lisätä viive liian aikaiseen suoritukseen, kuin vähentää suorituksen aikavaativuutta.

Kova tosiaikajärjestelmä (*eng. hard real-time system*) on järjestelmä, jolle ei ole hyväksyttävää, että suoritus kestää sille asetettua aikarajaa pidempään. Tällaisia järjestelmiä tehtäessä on tarpeen osoittaa, että ohjelman suoritus poikkeuksetta päättyy viimeistään aikarajaan mennessä. Suoritusaikarajaa jota ei koskaan ylitetä kutsutaan *turvalliseksi rajaksi* ja järjestelmää, jolle tällaiset rajat ovat tärkeitä joskus myös *turvallisuuskriittiseksi järjestelmäksi*. Esimerkiksi henkilöauton ohjausjärjestelmässä ei olisi hyväksyttävää, että jarrujen painamisen ja auton jarruttamisen välinen aika olisi epämääräinen.

Rajan etsiminen ohjelman suorituksen vaatimalle ajalle on kuitenkin vaikea ongelma. Jos se voitaisiin yleisessä tapauksessa ratkaista olisi myös pysähtymisongelma ratkaistu, sillä rajan löytäminen takaisi pysähtymisen löydettyyn rajaan mennessä. Koska pysähtymisongelma tiedetään ratkeamattomaksi, voidaan täten todeta myös pahimman suoritusajan etsiminen ratkeamattomaksi ongelmaksi ainakin yleisessä tapauksessa. Käytännössä tämä tarkoittaa, että todellisen pahimman suoritusajan (TPSA:n) sijasta etsitään turvallista ylärajaa pahimmalle suoritusajalle.

Jotta PSA-ongelma saataisiin käsiteltävään muotoon joudutaan tekemään joitakin rajauksia. Ongelman vaikeus perustuu ohjelman suorituspolkujen suureen määrään. Kovia tosiaikajärjestelmiä tehtäessä käytetään ohjelmointityylejä, joilla tätä määrää vähennetään. Esimerkiksi rekursion ja ite-

raation käyttö joko kielletään kokonaan tai sallitaan rajoitetusti siten, että rekursiosyvyys ja iteraatioiden lukumäärä on *staattisesti määritelty*, eli tiedossa ennen suoritusta. Ilman näitä rajoituksia ei olisi mahdollista selvittää ohjelman PSA:ta, sillä iteraatio tai rekursio voisi jatkua loputtomiin.

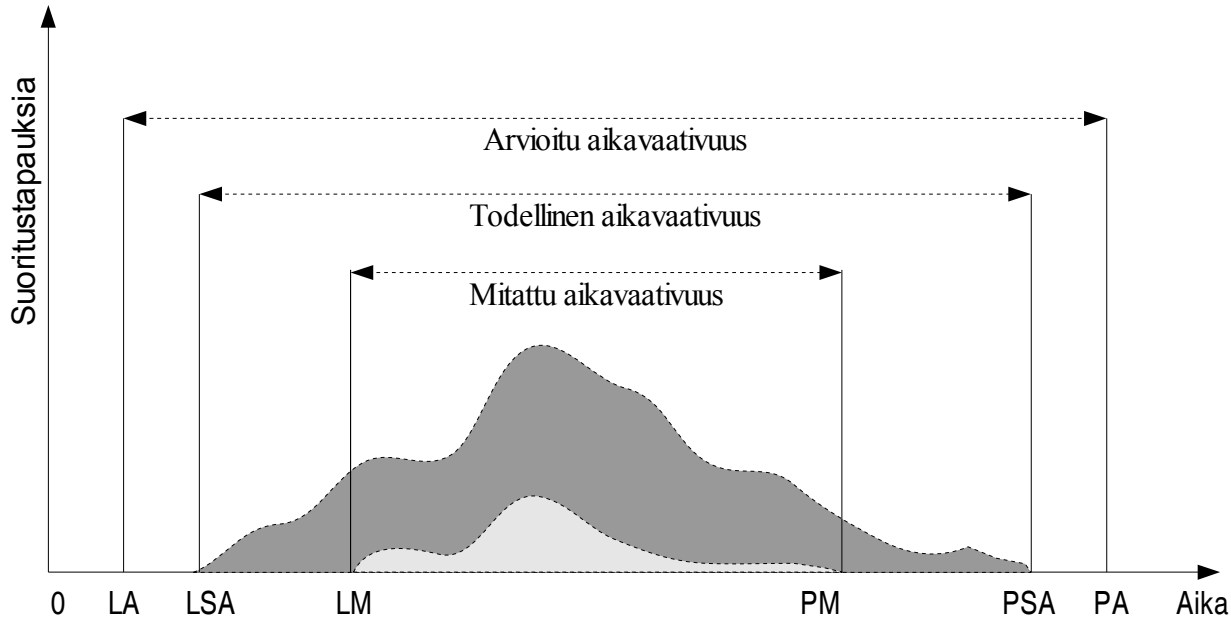
Suoritusajan selvittämistä voidaan lähestyä kahdella tavalla: dynaamisesti mittaamalla tai analysoimalla ohjelmaa staattisesti. Dynaaminen mittaus on yksinkertainen ja suoraviivainen menetelmä. Suoritetaan ohjelma ja mitataan kulunut aika. Menetelmä sisältää kuitenkin monia ongelmia. Jotta mitattu arvo olisi ohjelman suorituksen yläraja täytyisi ohjelma suorittaa pahimman tapauksen syötteellä, jonka selvittäminen on kuitenkin mahdollista vain erittäin harvoin. Syöteavaruuden suuren koon vuoksi suoritusta ei voida myöskään mitata kaikilla syötteillä. Esimerkiksi funktiolla, joka saa syötteenään kolme 16 bittistä lukua on syöteavaruus jossa on noin $3*10^{14}$ alkiota. Jos suoritus ja mittaus yhdellä syötteellä kestäisi millisekunnin, tarvittaisiin yli 9000 vuotta koko syöteavaruuden läpikäymiseen.

Kun käytetään dynaamista mittausta ohjelma suoritetaan syöteavaruuden osajoukolla, joka pyritään valitsemaan edustavasti. Näin voidaan selvittää ohjelman aikavaativuudelle rajoja, jotka ovat melko hyviä kun osajoukko on riittävän suuri ja kattava. Saadut rajat eivät kuitenkaan ole turvallisia koska ei ole takeita, että pahin syöte kuuluisi valittuun osajoukkoon. Dynaaminen mittaus ei täten ole riittävä menetelmä kovien tosiaikajärjestelmien suoritusaika-analyysiin, vaikka sitä voidaan käyttää ja usein käytetään pehmeiden tosiaikajärjestelmien tapauksessa [Wilh+07].

Vaihtoehto dynaamiselle mittaukselle on staattinen analyysi, jonka vahvuutena on juuri, että analyysissä selvitetään turvalliset rajat ohjelman suorituksen aikavaativuudelle. Staattinen analyysi on täten soveltuva kovien tosiaikajärjestelmien aikavaativuuden selvittämiseen.

Tyypillisesti analyysi tehdään ohjelman kontrollivuokaavioon. Kaavion tutkimiseen on eri menetelmiä, joita yhdistää se, että kaikissa yliarvioidaan aikavaativuutta turvallisuuden takaamiseksi. Staattisen analyysin heikkous on, että yliarvioinnin seurauksena löydetyt aikaraja-arviot voivat olla liian kaukana todellisista rajoista ollakseen hyödyllisiä. Staattisen analyysimenetelmän hyvyyden mitta on kuinka *tiukkoja*, eli kuinka lähellä todellisia rajoja sen löytämät raja-arviot ovat.

Kuva 1 esitettää dynaamisen mittauksen, staattisen analyysin ja todellisuuden väliset suhteet kuvitteelliselle ohjelmalle. Tumma alue on suoritusaikojen todellinen jakauma kaikilla syötteillä, joka ulottuu lyhimmästä suoritusajasta (LSA) pahimpaan (PSA). Vaalea alue on dynaamisella mittauksella löydettävä aikavaativuuden aliarvioiva jakauma, joka ulottuu lyhimmästä mitatusta suoritusajasta (LM) pahimpaan mittaukseen (PM). Uloimmat rajat LA ja PA ovat staattisella analyysillä löydettävät turvalliset raja-arviot.



Kuva 1: Tumma alue esittää kuvitteellisen ohjelman kaikkien suoritusaikojen jakaumaa. Vaalea alue on syöteavaruuden osajoukolla mitattu jakauma. Kuvassa lyhin arvio LA, lyhin suoritusaika LSA, lyhin mittaus LM, pahin mittaus PM, pahin suoritusaika PSA ja pahin arvio PA. Muokattu lähteestä [Wilh+07].

2.1. Staattinen analyysi

Suoritusaika-analyysi ja analyysin tekevä työkalu voidaan jakaa kahteen osaan. Ensimmäisessä osassa (*analysoijan etupää, frontend*) tutkitaan syötteenä saatua ohjelmaa ja rakennetaan siitä malli, esimerkiksi vuokaavio. Toisessa osassa (*takapää, backend*) mallista selvitetään aikavaativuudeltaan pisin suorituspolku (*kriittinen polku*), josta lasketaan turvallinen arvio ohjelman PSA:lle. Etuosassa siis mallinnetaan ohjelman rakenne ja takaosassa suoritetaan mallin analyysi suoritusaikarajojen löytämiseksi. Analyysi jaetaan joskus myös *korkean tason* ja *matalan tason* analyysiin (*high level*, *low level*). Korkealla tasolla tarkoitetaan laitteistoriippumatonta koodin analyysiä ja matalan tason analyysillä laitteistoriippuvaista selvitystä koodin suorituksesta tietyllä laitteistolla.

2.1.1. Analyysin syötteet

Suoritusaika-analyysi tehdään aina käännetyn ohjelman konekieli- eli objektikoodiin. Vain tällöin on saatavilla kaikki tarvittavat tiedot, esimerkiksi kääntäjän tekemistä optimoinneista, joita ilman analyysi ei olisi mahdollista. Konekoodin lisäksi analyysin syötteenä tarvitaan usein myös lisätietoja analyysityökalun käyttäjältä. Tyypillinen esimerkki tällaisista tiedoista ovat iteraatio- ja rekursiorajat. Analyysityökalut voivat usein selvittää automaattisesti joitakin iteraatio- ja rekursiorajoja, mutta loput on saatava syötteenä. Tarvittavat lisätiedot annetaan joko lisämerkintöinä, *anno-*

taatioina, koodissa tai erillisessä tiedostossa. Yleisesti pidetään suotavana, että annotaatioiden tarve on mahdollisimman vähäinen, eli työkalun automaatio on mahdollisimman korkea [Gust00]. Syynä on osittain lisätietojen selvittämisen käyttäjälle aiheuttama työ, mutta tärkeämpänä ovat mahdolliset virheet tiedoissa, jotka voivat tehdä löydetystä PSA:sta epäturvallisen.

Iteraatiorajojen käsin löytämiseen liittyvä turvallisuusriski ilmenee seuraavasta esimerkistä. Olkoon seuraava *for*-silmukka C-kielisessä ohjelmassa:

```
for (i=14; i>=0; i--) {...}
```

Olisi helppoa kuvitella silmukan päättyvän 15 iteraation jälkeen. Näin ei kuitenkaan välttämättä ole, sillä muuttujan *i* tyyppiä ei ole määritelty silmukan yhteydessä. Jos *i* on esimerkiksi etumerkitön kokonaisluku jatkuu iteraatio loputtomiin, sillä etumerkitön luku on aina suurempi tai yhtä kuin nolla. Tällainen ohjelmointivirhe paljastuisi, jos analyysityökalu suorittaisi iteraatiorajan etsimisen, mutta voisi helposti jäädä lisätiedot antavalta käyttäjältä huomaamatta. Analyysi voisi tällöin antaa turvalliseksi luullun ylärajan vaikka todellisuudessa ohjelma juuttuu ikuiseen silmukkaan.

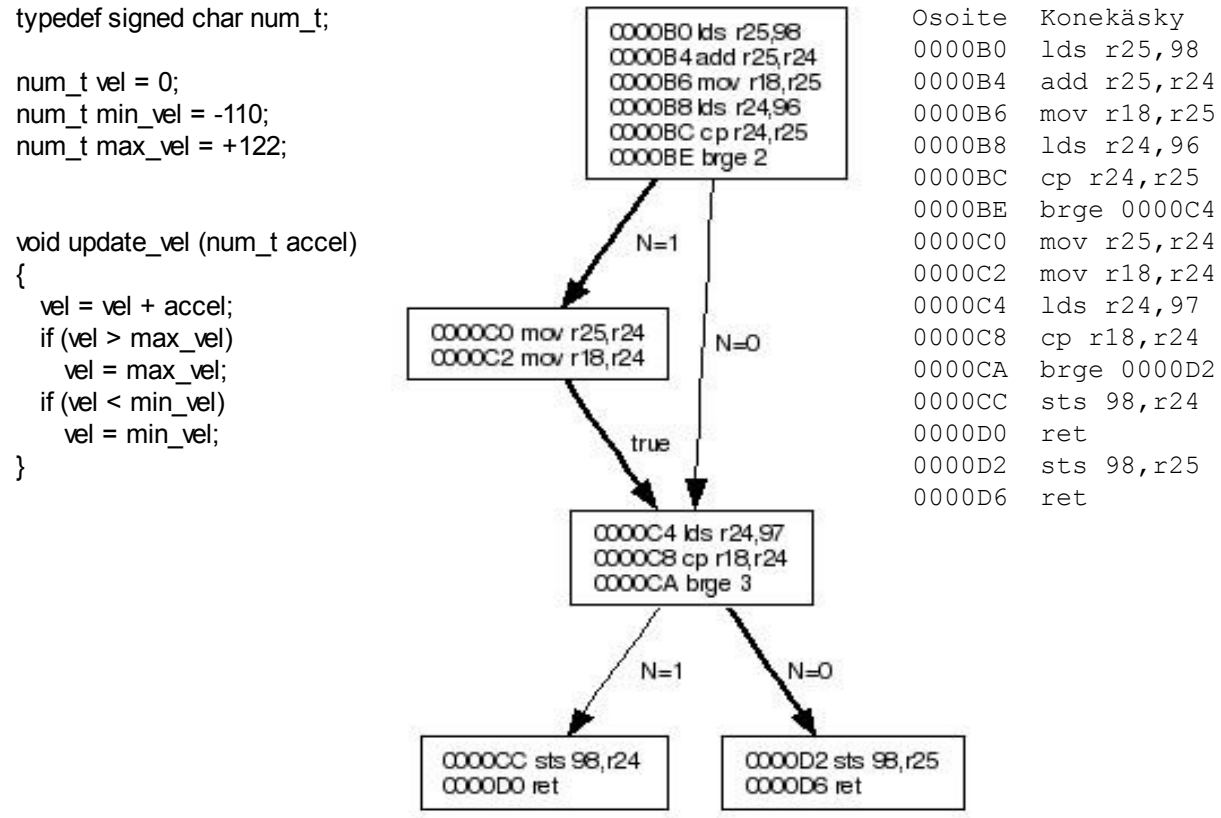
2.1.2. Kontrollivuokaavion laatiminen

Analyysin ensimmäinen askel on kontrollivuokaavion tai muun vastaavan mallin laatiminen syöteohjelman konekoodista. Aika-analyysityökalun on tunnettava ohjelman kohdesuorittimen käskykanta, mikä tekee analyysistä suoritinkohtaisen.

Vuokaavion rakentamista voidaan lähestyä *osittavasti* (*top-down*) tai *kokoavasti* (*bottom-up*). Osittava oli yleinen lähestymistapa aika-analyysin alkuaikoina, mutta on sittemmin todettu epäkelvolliseksi [Theil00, Hol07]. Syynä tähän on, että menetelmässä virheellisesti oletetaan aliohjelmien käskyjen koostuvan kaikesta konekoodista aliohjelman ja sitä seuraavan aliohjelman alkukohtien välillä. Kääntäjät eivät kuitenkaan tuota oletuksen mukaista koodia, esimekiksi konekoodin keskellä voi olla vakiodataa, joten menetelmästä on luovuttu.

Kokoavassa menetelmässä ei tehdä oletuksia ohjelman rakenteesta vaan tutkitaan konekielisiä käskyjä yksi kerrallaan. Aluksi muodostetaan vuokaavio, jossa kutakin käskyä vastaa solmu. Solmujen väliset särmät kuvaavat siirtymiä eli hyppyjä tai haaraumia koodissa. Käskyt ryhmitetään yhteen uusiksi peruslohko-solmuiksi siten, että uudet solmut vastaavat yhtenäisiä koodilohkoja, joiden keskeltä ei ole siirtymiä ulos tai ulkoa siirtymiä niiden keskelle. Täten solmun suoritus sisältää aina kaikki sen käskyt. Kuvassa 2 on esimerkki C-funktiosta, vastaavasta konekoodista ja konekoodista kokoavasti tuotetusta vuokaaviosta.

Aika-analyysillä halutaan usein selvittää miten eri ohjelman osat tai aliohjelmat vaikuttavat koko ohjelman PSA:han. Analyysimenetelmät tarvitsevat myös usein tietoa aliohjelmista voidakseen tehdä muuttujien arvoanalysointia, joka on osa suoritusajan arviointia. Analyysityökalut voivat myös yhdistellä eri menetelmiä PSA:n arviointiin. Menetelmää joka olisi liian raskas koko ohjelman analysointiin, esimerkiksi kaikkien suorituspolkujen läpikäyntiä, voidaan soveltaa aliohjelmiin ja tulokset käyttää koko ohjelman analyysissä tarkkuuden parantamiseksi.



Kuva 2: Esimerkki lähdekoodista, vuokaaviosta ja konekoodista. Koodi päivittää muuttujaa vel lisäämällä siihen muuttujan accel ja valvomalla ylä- ja alarajoja. Konekoodi on ATMEL-AVR suorittimelle ja vuokaavio on Bound-T työkalun tuottama.

Vuokaaviosta on näin ollen eroteltava aliohjelmat. Tehtävän vaikeus riippuu siitä, kuinka helposti aliohjelman alku ja loppu voidaan tunnistaa konekoodista. Helpossa tapauksessa aliohjelman kutsu tapahtuu aina tietyllä konekäskyllä, jolla ei ole muuta käyttöä, ja aliohjelman koodi päättyy samantapaiseen yksiselitteiseen paluukäskyyn. Vaikea tilanne muodostuu, jos aliohjelmakutsut ja palautukset koostuvat monista käskyistä, joita myös käytetään muissa tarkoituksissa. Esimerkiksi *switch-case*-lause voi usein muistuttaa aliohjelman kutsua, jolloin analyysi voi joutua hakoteille. Käytännössä tämä tarkoittaa, että analyysityökalun on tunnettava konekoodin tuottaneen kääntäjän käyttämät konventiot. Analyysityökaluja voidaan täten käyttää turvallisesti vain niiden tuntemien kääntäjien tuottamaan konekoodiin.

2.1.3. Laitteistomallin tarve

Jotta kontrollivuokaaviosta olisi mahdollista selvittää suoritusaikarajoja on tiedettävä kunkin solmun ja särmän aikavaativuus. Tämä tarkoittaa kunkin konekielisen käskyn aikavaativuuden selvittämistä. Ongelma on, että käskyn vaatima suoritusaika ei usein ole helposti saatavilla. Käskyn tarvitsema aika riippuu ensisijaisesti laitteistosta, jolla se suoritetaan. Voidakseen määritellä aikavaativuuksia käskyille analyysityökalulle ei enään riitä tuntemus kohdesuorittimen käskykannasta vaan tarvitsee myös mallin suorittimen toiminnasta. Tämän tarve rajoittaa kunkin analyysityökalun käytettävyyttä.

Suorittimet ovat usein monimutkaisia ja vaikeita mallintaa tarkasti. *Liukuhihnat* (*pipeline*) ja *välimuistin* (*cache*) käyttö ovat kaksi ominaisuutta, jotka kehittyneissä suorittimissa ovat erityisen hankalia. Liukuhihnasuoritin jakautuu eri osiin ja kunkin käskyn suoritus etenee vaiheittain osien välillä. Suorittimen osat voivat samanaikaisesti käsitellä eri käskyjä. Jos nyt aikavaativuutta arvioitaisiin laskemalla yhteen yksittäisten käskyjen suoritusaikoja (instruction latency) saataisiin suuri yliarvio ohjelman PSA:lle, sillä käskyjen suoritus ei ole peräkkäistä vaan päällekkäistä. Jos esimerkiksi käskyjen suoritusaika olisi viisi kellojaksoa voitaisiin kuitenkin suorittaa yksi käsky jokaisella jaksolla suorittimessa, jossa on viisivaiheinen liukuhihna.

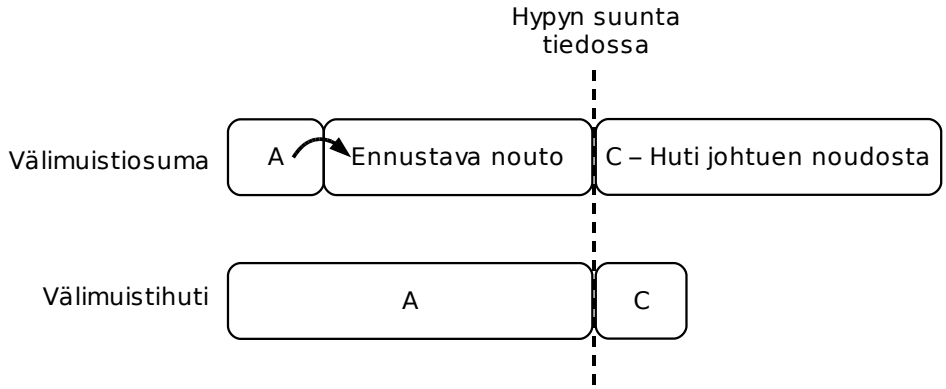
Mallintamisen hankaluutta lisää, että käskyn suoritusaika ei ole vakio. Suoritukseen voi tulla tauko, koska tarvittu suorittimen osa ei ole vapaa. Käskyjen suoritusaika riippuu täten suorittimen tilasta, eli muista samalla suoritettavista käskyistä. Samanaikainen suoritus rikkoo myös vuokaavion ajatusta, sillä esimerkiksi hyppyehtoa tutkittaessa voidaan jo suorittaa hypyn jälkeisiä käskyjä. Näiden käskyjen valinta voi osoittautua vääräksi, jos ne valittiin ennustaen hyppyehdon tulosta, eli käskyt haettiin suoritukseen ennen varmuutta suorituksen suunnasta haaraumakohdassa. Tällöin käskyjen suoritus on peruttava ja korvattava oikeilla käskyillä. Suoritin voi myös toimia siten, että hypyn jälkeisiä käskyjä suoritetaan aina vakio määrä hyppyehdosta riippumatta (*viivästetty hyppy*, *delayed jump*), jonka ilmaiseminen vuokaaviossa on ongelmallista.

Välimuistin käyttö nopeuttaa suorittimen toimintaa keskivertotapauksessa. Jos suorituksen tarvitsema tieto löytyy välimuistista säästyy aikaa, joka olisi muuten kulunut tiedon hakemiseen keskusmuistista. Koska käskyn suoritus nyt riippuu välimuistin sisällöstä, on tarpeen mallintaa myös tämän tilaa ja käyttäytymistä suuren yliarvioinnin välttämiseksi sitä käyttävillä suorittimilla.

Ennustavassa suorittimessa välimuisti voi aiheuttaa *ajoitus-säännöttömyyksiä* (*timing anomalies*), jotka ovat intuition vastaisia vaikutuksia yhden käskyn suoritusajan ja koko ohjelman suoritusajan välillä [Rein+06, Wilh+07]. Yhden käskyn tarvitseman tiedon löytyminen välimuistista voi nopeuttaa käskyn suoritusta mutta samalla hidastaa koko ohjelman suoritusta. Syynä tähän on, että

tiedon löytyminen välimuistista jättää muistiväylän vapaaksi keskusmuistiin, jolloin suoritin voi ennustavasti hakea seuraavan käskyn tarvitsemaa tietoa välimuistiin. Jos ennustus osoittautuu vääräksi on mahdollista, että ennustus poisti todellisen seuraavan käskyn tarvitseman tiedon välimuistista. Tilanne on tätäkin hankalampi, sillä poistettu tieto on voinut olla tarpeen ei vain seuraavalle käskylle, mutta mille tahansa käskylle lopun ohjelman suorituksessa.

Kuva 3 esittää yksinkertaisen tapauksen, jossa vaikutus on seuraavaan käskyyn. Käskyn *A* löytyminen välimuistista aiheuttaa ennustavan noudon, joka syrjäyttää välimuistista raskaamman käskyn *C* tarvitseman tiedon. Näin käskyjen yhteenlaskettu suoritus kestää kauemmin kuin jos käskyjen välimuistiosuma ja muistihuti olisivat toisinpäin (alempi tapaus).



Kuva 3: Ennustavan noudon aiheuttama ajoitus-säännöttömyys[Rein+06].

Laitteiston tila-avaruus ja tilasiirtymät ovat mallintamisen kannalta samalla tavalla hankalia kuin ohjelman suorituspolkujen eksponentiaaliset kombinaatiot. Mallintaminen on täten tehtävä epätarkasti (mutta turvallisesti) korkealla abstraktiotasolla, mikä heikentää aika-arvioiden tiukkuutta.

Usein joudutaan toteamaan suorittimen toiminta liian vaikeaksi mallintaa tarkasti ja käyttää joko turvallisia pahimman tapauksen arvoja käskyjen suoritukselle tai tehdä turvallinen analyysi vain yksinkertaisille suorittimille. Puutteet kehittyneiden suorittimien mallintamisessa voivat näin pakottaa käyttämään vähemmän tehokkaita mutta ennustettavampia deterministisiä suorittimia turvallisuuskriittisissä järjestelmissä [DHPS03].

2.1.4. Arvo- ja kontrollivuoanalyysi

Arvo- ja *kontrollivuoanalyysi* ovat kaksi staattisen analyysin vaihetta, joilla pyritään parantamaan sen tarkkuutta. Arvoanalyysissä tutkitaan ohjelman muuttujia ja mikäli mahdollista löytää niille rajoja ohjelman eri pisteissä. Pyrkimyksenä on siis tiukentaa suoritusaika-arviota rajaamalla

analyysin ulkopuolelle muuttujien sellaisia arvoja, joiden tiedetään olevan mahdottomia ohjelman suorituksen eri kohdissa. Esimerkiksi jos kyseessä on iteraation laskuri voidaan arvoanalyysillä mahdollisesti löytää iteraatiolle yläraja.

Aika-analyysistä saadaan hienojakoisempi siten, että aliohjelman aikavaativuus selvitetään jokaiselle kutsulle erikseen käyttäen arvoanalyysin kutsukohdalle antamia arvovälejä parametreille. Arvoanalyysi on myös avuksi jos halutaan tutkia välimuistin käyttäytymistä, koska tällöin on tarpeen tuntea muistihakujen osoitteiden arvot [Wilh+07].

Tyypillinen tapa tehdä arvoanalyysiä on seurata muuttujien *määrittely*- ja *käyttökohtia* (*def-use pairs*) koodissa. Vuokaaviosta etsitään takautuvasti ne määrittelykohdat, jotka voivat asettaa muuttujalle arvon ennen jokaista käyttökohtaa. Usein myös seurataan yksinkertaisia muutoksia arvoissa, kuten vakiolla kertomista tai lisäystä.

Kontrollivuoanalyysillä pyritään vastaavasti rajaamaan analyysiin mukaan otettavia suorituspolkuja. Analyysityökalun etupää tuottama vuokaavio on jo itsessään yliarvio ohjelman mahdolliselle suoritukselle, sillä se sisältää paljon sellaisia polkuja, joita suoritus ei käytännössä voi seurata. Näiden *mahdottomien polkujen (infeasible path)* rajaaminen analyysin ulkopuolelle tiukentaa aikavaativuuden arvioita mikäli nämä polut muuten olisivat osa pisintä polkua.

Arvoanalyysin tuloksia käytetään kontrollivuoanalyysissä. Esimerkiksi jos muuttujan mahdolliset arvot hypyn kohdalla määräävät hypyn suunnan voidaan polku, joka sisältää toteutumattoman suunnan todeta mahdottomaksi.

Analyysit tukevat toisiaan sillä kontrollivuoanalyysin tuloksia voidaan myös käyttää tarkentamaan muuttujien arvovälejä arvoanalyysissä. Sulkemalla pois suorituspolkuja voidaan vähentää muuttujan arvoon vaikuttavia määrittelykohtia ohjelman eri pisteissä.

2.2. Käytetyt menetelmät PSA:n arvioimiseen

Kun kontrollivuokaavio on saatu rakennettua ja merkittyä analyysin tarvitsemilla tiedoilla on analyysin seuraavana vaiheena laskea arvio pahimmalle suoritusajalle. Käytetyt menetelmät vuokaavion staattiseen analysointiin jakaantuvat seuraaviin luokkiin [Wilh+07]:

- *rakenneperustaiset* menetelmät
- *polkupohjaiset* menetelmät
- *implisiittiseen polkujen luettelointiin* perustuvat menetelmät (*Implicit Path Enumeration Technique*, *IPET*).

Seuraavassa esitellään nämä menetelmät lyhyesti.

2.2.1. Rakenneperustainen menetelmä

Menetelmässä käytetään ohjelman syntaksipuuta, joka on johdettava vuokaaviosta. Suntaksipuu jäsentää vuokaavion osiin, jotka tehdään peräkkäin (käsky; käsky;...), vaihtoehtoisiin osiin (if(ehto) then käsky else käsky) ja toistettaviin osiin (while(ehto) do käsky). Puuhun sovelletaan *muunnos-sääntöjä* (*timing schema*) toistuvasti siten, että lopulta puukaavio koostuu yhdestä ainoasta solmusta, jonka aikavaativuus on samalla koko ohjelman PSA:n arvio. Peräkkäiset solmut liitetään yhteen ja suoritusajaksi asetetaan niiden summa. Vaihtoehdoista valitaan suurempi suoritusaika ja lisätään siihen ehdon suoritusaika. Tämä aika asetetaan uudelle solmulle, joka korvaa vaihtoehto-alipuun. Toiston tapauksessa muodostetaan uusi solmu, joka kattaa toiston ehtosolmusta loppuun. Suoritusaika uudelle solmulle saadaan kertomalla korvattujen solmujen suoritusaika toiston rajaa yhtä pienemällä luvulla ja lisäämällä tähän ehtosolmun aika vielä kerran viimeisen tarkistuksen vuoksi.

Ongelma on, että syntaksipuuta ei voida tehdä kaikista vuokaavioista. Menetelmässä on lisäksi vaikea huomioida lisätietoja ohjelman suorituspoluista [Wilh+07].

2.2.2. Polkupohjainen menetelmä

Polkupohjaisessa menetelmässä etsitään vuokaaviosta aikavaativuudeltaan pisin polku. Menetelmä tutkii kaikki polut ja on täten mahdollinen vain rajoitetusti sovellettuna. Sitä on joko käytettävä pienten ohjelmapalasien analysointiin toisen menetelmän osana tai ainoastaan sellaisen koodin analysointiin, jossa suorituspolkujen määrä on voimakkaasti rajoitettu. Suorituspolkujen määrä voidaan ääritapauksessa rajata tasan yhteen (*single-path programming*) [Pusc02], mutta yleisessä tapauksessa polkupohjaisen menetelmän käyttö ei ole suotavaa.

2.2.3. IPET-menetelmä

Implisiittinen polkujen luettelointi eroaa muista menetelmistä siinä, ettei se löydä yksittäistä pisintä polkua. PSA-ongelmasta muodostetaan maksimointiongelma, joka ratkaistaan *lineaarisella kokonaislukuohjelmoinnilla* (*Integer Linear Programming*, *ILP*). ILP on yleinen menetelmä ratkaista ongelma, joka koostuu maksimoitavasta lineaarisesta funktiosta sekä joukosta *lineaarisia rajoitteita* funktion muuttujille. Sen soveltamista suoritusaika-analyysissä ehdotettiin ensimmäisen kerran 1995 [LiMa95] ja se on käytössä useissa työkaluissa (SWEET, aiT, Bound-T).

Maksimointiongelma muodostetaan asettamalla jokaiselle särmälle ja solmulle vuokaaviossa muuttuja, joka ilmaisee kuinka monta kertaa kyseinen särmä tai solmu suoritetaan. ILP-ongelman funktio saadaan kertomalla muuttujat vastaavan vuokaavioelementin aikavaativuudella ja laskemalla näiden tulojen summa. Tämä lauseke kuvaa ohjelman suorituksen aikavaativuutta ja sen maksimointi antaa PSA-arvion.

Lineaariset rajoitteet antavat mahdollisuuden tarkentaa analyysiä. Vuokaaviosta saadaan suoraan joitakin rajoitteita, esimerkiksi että solmusta ulos johtavia särmiä suoritetaan yhteensä saman verran kuin solmuun sisään tulevia. Näiden vuokaaviosta saatavien *rakenteellisten* rajoitteiden lisäksi iteraatio- ja rekursiorajat voidaan myös ilmaista rajoitteiden muodossa. Arvo- ja kontrollivuoanalyysistä voidaan edelleen johtaa lisää rajoiteita tiukkuuden parantamiseksi.

Kun rajoitteet ja maksimoitava funktio on muodostettu voidaan soveltaa ILP-ratkojaa tuloksen saamiseksi. Ratkojia on saatavilla sekä parempia kaupallisia että vähemmän tehokkaita avoimia ja ilmaisia versioita. ILP-ongelmalla on teoriassa eksponentiaalinen aikavaativuus, mutta käytännössä ratkojat ovat riittävän hyviä menetelmän soveltamiseen suoritusaika-analyysissä.

2.3. Mahdottomat polut

Suoritusaika-analyysin tarkentamista voidaan lähestyä kahdella tapaa, parantamalla joko suorittimien tai ohjelman mallintamista. Suoritinmallien parantaminen vähentäisi kunkin käskyn vaatiman ajan epätarkkuutta. Parannukset tällä saralla voisivat mahdollistaa tehokkaampien suorittimien käytön kovissa tosiaikajärjestelmissä. Parannukset olisivat kuitenkin suoritinkohtaisia ja tarpeen tehdä jokaiselle suorittimelle erikseen. Tästä syystä tämä tutkielma keskittyy ohjelman mallintamiseen, jonka kehittämisen hyödyt ovat laajemmin käytettävissä.

Ohjelman malli koostuu vuokaaviosta sekä arvo- ja kontrollivuoanalyysistä. Kuten aikaisemmin todettiin kontrollivuokaavio yliarvioi ohjelman mahdollista suoritusta, sillä se sisältää suorituspolkuja, joita ohjelman todellinen suoritus ei voi seurata. Tällaiset mahdottomat polut vaikuttavat suoritusaika-analyysiin lisäämällä tarkasteltavien polkujen määrää ja mikäli ne sijaitsevat kriittisellä polulla kasvattavat saatavaa PSA:n arviota.

Mahdottomat suorituspolut on todettu merkittäväksi haasteeksi suoritusaika-analyysissä, mutta myös ohjelmistoanalyysissä yleisemmin. Esimerkiksi automaattisessa testitapausten generoinnissa halutaan välttää turhat yritykset generoida syötteitä mahdottomille poluille [JBWZC94].

Jos ohjelma määritellään tilasiirtymäkaaviona, josta lähdekoodi automaattisesti tuotetaan niin vuokaavioon voi syntyä suuri määrä mahdottomia polkuja, koska alkuperäisen määritelmän tilojen väliset tarkat suhteet on kadotettu. Tilat ovat nyt muuttujien arvoja, mitkä eivät vastaa vuokaavion solmuja, mutta vaikuttavat oleellisesti suorituspolkuihin. Tämä tilanne on kohdattu esimerkiksi laivateollisuudessa [Assa07].

2.3.1. Mahdottomien polkujen luokittelu

Edellä esitetty mahdottoman polun määritelmä on hyvin yleinen. Esimerkiksi iteraatiorajan ylittävä iteraation suoritus sisältyisi määritelmään, vaikka sitä ei yleensä mielletä osaksi mahdottomien polkujen aiheuttamaa ongelmaa, koska iteraatiorajat käsitellään erikseen suoritusaika-analyysissä. Vaikka mahdottomat polut ovat olleet tutkimuksen kohteena jonkin aikaa, ei kirjallisuudessa esiinny yleistä luokittelua tai jakoa niille. Luokittelua lähinnä oleva julkaisu on ehkä yritys luetella halutut ominaisuudet annotaatiokielille [Kirn07].

Jotta luokittelu olisi käyttökelpoinen tulisi sen erotella mahdottomia polkuja etsivien menetelmien löytämät polut sekä perustua ohjelman rakenteisiin. Nämä tavoitteet samastuvat koska menetelmät tyypillisesti jakavat etsinnän ohjelman rakenteiden mukaan. Seuraava luokittelu perustuu näin ollen polun kattamiin ohjelman rakenteisiin:

- **Yksinkertainen polku**; aliohjelman sisäinen polku, joka on mahdoton koska sen varrella on kaksi ehtoa tai ehto ja ehtomuuttujan määrittely, jotka ovat ristiriitaisia.
- **Kutsupolku**; kuten edellinen mutta ensimmäinen ehto tai ehtomuuttujan määrittely sijaitsee aliohjelman ulkopuolella, sitä kutsuneessa ohjelman osassa tai sen kutsumassa aliohjelmassa.
- **Silmukkapolku**; silmukan mahdollisen iteraatiorajan ylittävä polku.
- **Kolmiosilmukkapolku**; silmukan iteraatiorajojen sisällä pysyvä polku, joka on mahdoton koska silmukan sisällä oleva ehtolauseke voi saada tietyn tuloksen vain osalla iteraatioista.

2.3.2. Mahdottomien polkujen esittäminen

Jotta polkuja voitaisiin käsitellä tarvitaan niille esitystapa. IPET-menetelmän lineaariset rajoitteet ovat yksi vaihtoehto tähän. Lineaarisella rajoitteella voidaan helposti ilmaista joitakin polkuja. Kolmiosilmukkapolku on helposti ilmaistavissa asettamalla ehtolausetta seuraaville vuokaavioelementtien IPET-muuttujille polkua vastaavat rajoitukset.

Yksinkertainen polku voi sen sijan olla hankala esittää. Olkoon ohjelmassa kaksi peräkkäistä ehtolausetta, jossa ensimmäisen ehdon *tosi*-arvo johtaa ristiriitaan jälkimmäisen kanssa. Mikäli tämä kuvaa tilannetta kokonaisuudessaan voidaan asettaa lineaarinen rajoite siten, että ehtolauseiden tosi-haaraumia kuljetaan yhteensä korkeintaan kerran. Ongelma on, että jos nyt tämä polku on silmukan sisällä ei voida muodostaa rajoitetta, joka esittäisi sitä tarkasti. Syy tähän on, että IPET ei huomioi polkuja sellaisinaan, vaan ainostaan elementtien suorituskertoja. Voidaan muodostaa rajoite sille, että ehtojen tosi-haaraumia suoritetaan yhteensä korkeintaan silmukan toistojen verran, mutta tieto, että samalla silmukan suorituskerralla voidaan suorittaa vain toinen ei ole esitettävissä.

3. Työkalu SWEET

- Ruotsalaisen suoritusaika-analyysityökalun SWEET:in esittely

3.1. Abstract Execution

- SWEET:in käyttämä erikoinen abstrakti suoritus menetelmä
- Edut/haitat

3.2. Mahdottomien polkujen etsintä

- Miten SWEET etsii mahdottomia polkuja
- Mitä jää löytämättä

4. Työkalu aiT

- Kaupallisen suoritusaika-analyysityökalun aiT:n esittely
- Vahvuutena hankalien dynaamisten suorittimien mallintaminen
- Miten huomioi mahdottomat polut

5. Esitettyjä algoritmeja

- Katsaus kirjallisuudessa esitettyihin tapoihin ja menetelmiin käsitellä mahdottomia polkuja

5.1. Healy ja Whalley

5.2. Boík, Gupta ja Soffa

5.3. Altenbernd

5.4. Suhendra, Mitra, Roychoudhury ja Chen

5.5. Jasper, Brennan, Williamson, Currier ja Zimmerman

6. Vertailua

- Mitkä ovat hyvän algoritmin ominaisuudet
- Löydettyjen polkujen määrä, tarvittavat käyttäjän antamat lisätiedot, tulosten käytettävyys analyysissä
- Esitettyjen algoritmien löytämät polut
- Miten ne eroavat toisistaan
- Algoritmien aika- ja tilavaativuus

7. Bound-T

7.1. Bound-T:n analyysin menetelmät

- `IPET ja ILP`
- `Koodin lisäksi tarvittavat käyttäjän merkinnät`
- `Miten Bound-T:n nykyversio käsittelee mahdottomia polkuja`

8. Algoritmien soveltuvuus

- Kuinka esitetyt algoritmit voitaisiin liittää osaksi Bound-T:n analyysiä
- Valita paras tai muutama hyvä ja/tai helposti liitettävä algoritmi implementoitavaksi

9. Toteutuksen esittely

- Muutokset Bound-T työkaluun

10. Tulosten arviointi

- Benchmark-ohjelmilla kokeilu ja tulosten vertailu

11. Yhteenveto

Lähteet

[AbsInt] AbsInt Execution Time Analyzer. http://www.absint.com/ait/.

[Alte96] Peter Altenbernd, On the False Path Problem in Hard Real-Time Programs. *Proceedings of EURWRTS* 1996.

[Assa07] Elie Manouel Assaf, WCET Analysis of Automatically Generated Code for CC-Systems AB. Master of Science Thesis, Department of Computer Science and Engineering Mälardalen University, Västerås, August 15, 2007.

[BGS97] Rasitslav Bodík, Rajiv Gupta ja Mary Lou Soffa, Refining Data Flow Information Using Infeasible Paths. *Proceedings of the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Switzerland, September 1997.

[BT] Bound-T Execution Time Analyzer. http://www.bound-t.com/.

[DHPS03] Martin Delvai, Wolfgang Huber, Peter Puschner ja Andreas Steininger, Processor Support for Temporal Predictablility - The SPEAR Design Example. *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, 2003.

[GEL05] Jan Gustafsson, Andreas Ermedahl ja Björn Lisper, Towards a Flow Analysis for Embedded System C Programs. *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems* 2005.

[GEL06a] Jan Gustafsson, Andreas Ermedahl ja Björn Lisper, Algorithms for Infeasible Path Calculation. *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis* (WCET'06), Dresden, Germany, July 4, 2006. http://moss.csc.ncsu.edu/~mueller/wcet06/wcet06.pdf tai http://drops.dagstuhl.de/portals/WCET06/.

[GEL06b] Jan Gustafsson, Andreas Ermedahl ja Björn Lisper, Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution. *Proceedings of the 27th IEEE International Real-Time Systems Symposium* 2006.

[Gust00] Jan Gustafsson, Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. PhD Thesis, Department of Computer Engineering, Mälardalen University. DoCS 00/115, ISSN 0283-0574, tai vaihtoehtoisesti MRTC 00/10, ISSN 1404-3041.

[HeWh02] Christopher A. Healy ja David Whalley, Automatic Detection and Exploitation of Branch Constraints for Timing Analysis. *IEEE Transactions on Software Engineering*, August 2002, sivut 763-781.

[Hol07] Niklas Holsti, Analysing Switch-Case Tables by Partial Evaluation. *Proceedings of the 7th International Workshop on Worst-Case Execution-Time Analysis*, Pisa, Italy, July 3, 2007.

[JBWZC94] Robert Jasper, Mike Brennan, Keith Williamson, Bill Currier ja David Zimmerman, Test Data Generation and Feasible Path Analysis. *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[Kirn07] Raimund Kirner et al., WCET Analysis: The Annotation Language Challenge. *Proceedings of the 7th International Workshop on Worst-Case Execution-Time Analysis*, Pisa, Italy July 3, 2007.

[LiMa95] Yau-Tsun Steven Li ja Sharad Malik, Performance Analysis of Embedded Software Using Implicit Path Enumeration. *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers & Tools for Real-Time Systems*.

[Pusc02] Peter Puschner, Is Worst-Case Execution-Time Analysis a Non-Problem? - Towards New Software and Hardware Architectures. *Proceedings of the 2002 Workshop on WCET Analysis*.

[Rein+06] Jan Reineke et al., A Definition and Classification of Timing Anomalies. *Proceedings of the 6th International Workshop on Worst-Case Execution-Time Analysis*, 2006.

[SMRC06] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury ja Ting Chen, Efficient Detection and Explotation of Infeasible Paths for Software Timing Analysis. *Proceedings of the 43rd Annual Conference of Design Automation*, San Francisco, July 2006.

[StMa07] Ingmar Stein ja Florian Martin, Analysis of path exclusion at machine code level. *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis*, Pisa, Italy, July 3, 2007.
http://www.irit.fr/wcet2007/WCET07_proceedings.pdf.

[Theil00] Henrik Theiling, Extracting Safe and Precise Control Flow from Binaries. *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, 2000.

[Wilh+07] Reinhard Wilhelm et al., The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. Hyväksytty julkaistavaksi lehdessä *ACM Transactions on Embedded Computing Systems*. Preprint saatavilla osoitteesta http://www.mrtc.mdh.se/index.php?choice=publications&year=any&id=1257.