Teknillinen korkeakoulu Tietotekniikan osasto
Tietojenkäsittelyopin laboratorio
Espoo 2004 TKO-C92/04

Seminaariraportti sulautetuista reaaliaikaohjelmista ja niiden suorituskykyanalyysistä

Vesa Hirvisalo (toim.)

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Tietojenkäsittelyopinlaboratorio
Tietotekniikan osasto
Teknillinen korkeakoulu
PL 5400
02015 TKK
Espoo

Avainsanat: sulautetut ohjelmistot, reaaliaikajärjestelmät

ISBN: 951-22-7455-8
ISSN: 1239-6907

Esipuhe

Keväällä 2003 järjestettiin Teknillisessä korkeakoulussa seminaari, jonka aiheena oli ohjelmistojen suorituskyky. Seminaarissa perehdyttiin erityisesti sulautettujen ohjelmistojen suorituskykyanalyysiin. Sulautetuissa laitteissa ohjelmisto sulautuu yhteen laitteiston kanssa. Tämän vuoksi seminaari keskittyi käsittelemään ohjelmistojen suorituskykyä laitteistoläheisellä tavalla.

Viime vuosien aikana prosessoreiden laskentanopeuden kehitys on ollut ripeää. Osaksi laskentanopeuden kehitys on perustunut perustekniikan muutoksiin. Ratkaisevaa on kuitenkin ollut prosessoriarkkitehtuurien kehitys, ja sitä myötä aktiivisten komponenttien määrän nopea kasvu prosessoreissa. Prosessoriarkkitehtuurien muutos on myös mahdollistanut yhä korkeampien kellotaajuuksien käytön.

Uusia arkkitehtuuriratkaisuja on otettu laajalti käyttöön sovelluksissa, joissa laskentanopeus on keskeisin suorituskykytekijä. Sen sijaan reaaliaikajärjestelmissä kehitys on ollut selvästi konservatiivisempaa. Niissä suurta laskentanopeutta tärkeämpää on laskentanopeuden takaaminen. Näin sopivien analyysimenetelmien puuttuessa erilaiset välimuistit tai käskyliukuhihnat eivät ole reaaliaikajärjestelmissä vastaavalla tavalla yleistyneet.

Uudet arkkitehtuuriratkaisut tekevät laskentanopeuden ohella myös monet muut suorituskyvyn mitat vaikeaksi ennustaa. Tällainen mitta on erityisesti tehonkulutus, joka monille nykypäivän akkukäyttöisille laitteille on jopa laskentanopeutta tärkeämpi suorituskyvyn mitta.

Staattisen ohjelma-analyysin kehitys 1990-luvulla on tehnyt mahdolliseksi antaa ehdottomia suorituskykyrajoja myös prosessoreille, jotka käyttävät välimuisteja tai liukuhihnoja. Niinpä tällaisten prosessoreiden yleistyminen reaaliaikajärjestelmissä vaikuttaa todennäköiseltä. Joillekin prosessoreille työkaluja on jo kaupallisesti saatavilla, ja niitä on sovellettu käytännön kehitystyössä esimerkiksi lentokone- ja autoteollisuudessa.

Seminaarin tavoitteena oli luoda suppea katsaus alan nykytilaan, jota voitaisin eri yhteyksissä hyödyntää. Seminaarissa ei niinkään pyritty selvittämään tutkimuksen tämänhetkistä eturintamaa, vaan pikemmin yritettiin etsiä laajamittaista peruskatsausta aihepiiriin.

Espoossa, 3. joulukuuta 2004
Seminaarin vetäjä

Vesa Hirvisalo

Sisältö

Lyhyt katsaus reaaliaikajärjestelmiin

Antti Kantee
Sampo Vuorinen

1 Johdanto

Reaaliaikajärjestelmät ovat tietojärjestelmiä, joilta vaaditaan vasteen laskennallisen oikeellisuuden lisäksi sen täsmällistä oikea-aikaisuutta. Tyypillisiä reaaliaikaisuutta vaativia sovelluskohteita ovat mm. teollisuuden ohjausprosessit, kulkuneuvojen automaattinen ohjaus sekä tietoliikenne- ja multimediajärjestelmät. Kaikissa näissä sovelluksissa tietotekninen osa ohjaa tai valvoo jotain ulkoista prosessia, jonka tila muuttuu ajan funktiona. Jotta prosessi pysyisi halutussa tilassa tai ylipäänsä hallittavissa, ohjaustoimenpiteiden on tapahduttava sille luonteenomaisten aikarajojen puitteissa.

Reaaliaikaisuuden ja laskentanopeuden käsitteet sekoitetaan usein toisiinsa. Kyseessä on kuitenkin kaksi eri tavoitetta, joihin pyritään osittain keskenään ristiriitaisin keinoin. Nopea laskenta pyrkii tarjoamaan mahdollisimman lyhyen keskimääräisen suoritusajan. Tähän päästään erilaisten suoritinytimen käyttöastetta parantavien laitteistoratkaisujen kuten välimuistien ja käskyliukuhihnojen avulla. Reaaliaikajärjestelmissä kullakin prosessilla on valmistumisajan suhteen yksilölliset vaatimukset, joiden on toteuduttava täsmällisesti. Tämä edellyttää sekä laitteistolta että ohjelmistolta ennustettavaa käyttäytymistä. Keskimääräistä suorituskykyä parantaville menetelmille on tyypillistä, että ne heikentävät ennustettavuutta ja huonontavat pahimman tapauksen suorituskykyä. Esimerkiksi nykyaikaisten työasemaprosessorien monitasoiset välimuistiratkaisut aiheuttavat sen, ettei muistihaun kestoa tunneta etukäteen, ja jos tieto joudutaan hakemaan päämuistista asti, haku kestää kauemmin kuin kokonaan ilman välimuisteja. Tästä syystä reaaliaikajärjestelmissä on perinteisesti vältetty välimuistien käyttöä.

2 Peruskäsitteitä

Prosessin käsite on keskeinen sekä reaaliaikaisissa että perinteisissä aikajakoisissa käyttöjärjestelmissä. Prosessi on yksittäinen laskentatapahtuma, jota keskusyksikkö suorittaa ajalla, jonka käyttöjärjestelmä on antanut prosessin käyttöön. Yleiskäyttöisissä tietokoneissa kukin sovellusohjelma muodostaa ajettaessa oman prosessinsa. Reaaliaikajärjestelmien sulautetusta luonteesta johtuen ohjelmisto on yleensä ulkoisesti yksi kokonaisuus, ja prosesseilla tarkoitetaan yksittäisiä tehtäviä (task), joilla on yksilölliset ajastusvaatimukset ja joista koko järjestelmän toiminnallisuus koostuu.

Skedulointi eli prosessoriajan jakaminen prosessien kesken on tarpeen, kun järjestelmässä on samanaikaisesti useampia aktiivisia prosesseja kuin fyysisiä suoritinyksiköitä. Skedu-

lointialgoritmi on joukko sääntöjä, joiden avulla käyttöjärjestelmä valitsee kulloinkin suoritettavan prosessin. Skedulointia käsitellään tarkemmin luvussa 3.

Prioriteetti on prosessiin liittyvä luku, joka kuvaa prosessin tärkeyttä. Perusperiaate on, että ajossa on aina korkeimman prioriteetin omaava prosessi. Alemman prioriteetin prosessit pääsevät ajoon vain ylemmän prosessin valmistuessa tai siirtyessä jostain muusta syystä (esim. oheislaitteen vastetta odottaessaan) pois ajovuorosta.

Prosessia kutsutaan aktiiviseksi, mikäli se on tarkasteluhetkellä ajossa tai ajovalmiina. Ajovalmis prosessi odottaa ajovuoroa käyttöjärjestelmän ylläpitämässä jonossa. Prosessia, joka ei ole aktiivinen, kutsutaan odottavaksi prosessiksi. Aktiivinen prosessi voi joutua siirtymään odotustilaan esimerkiksi, mikäli se tarvitsee käyttöönsä jotakin resurssia (oheislaitetta tms.) jota vain yksi prosessi voi käyttää kerrallaan ja joka on juuri jonkin toisen prosessin käytössä. Odottava prosessi ei voi muuttua aktiiviseksi (tulla siirretyksi valmiusjonoon) ennen kuin varattu resurssi vapautuu.

Reaaliaikajärjestelmät jaetaan karkeasti koviin (hard) ja pehmeisiin (soft) sen mukaan, kuinka tärkeää aikarajojen täyttäminen on. Koviksi katsotaan sellaiset järjestelmät, joissa aikarajoista lipsuminen voi aiheuttaa vakavaa vahinkoa, kuten ympäristökatastrofin tai lentokoneen maahansyöksyn. Pehmeissä järjestelmissä aikarajan pettäminen huonontaa suorituskykyä, mutta ei aiheuta suoranaista vahinkoa (esim. parin kuvaruudun poisjääminen videokuvasta). Jako on sinänsä tietenkin varsin keinotekoinen ja sen käytännön hyöty rajoittuu lähinnä tehtävien tärkeysjärjestyksen määräämiseen tietyn sovelluksen sisällä.

3 Skedulointi

Reaaliaikaskeduloinnin tarkoitus on määrätä prosesseille sellainen alkamisaika ja suoritusjärjestys, että ne valmistuvat kaikki annettuna määräaikana (deadline). Määräaika käsitetään usein takarajaksi (ts. laskenta saa valmistua milloin tahansa ennen määräaikaa), mutta sovelluksesta riippuen se voi olla myös tarkka (on-time) eli siitä ei saa poiketa kumpaankaan suuntaan. Kovissa reaaliaikasovelluksissa erotetaan usein toisistaan "hard task"ja "firm task", joista edellisessä myöhästyminen on kohtalokasta ja jälkimmäisessä myöhästynyt tulos on hyödytön. Pehmeissä sovelluksissa tuloksesta saatava hyöty vähenee asteittain (ks. ed).

Prosessien joukkoa, jolle on mahdollista löytää jollakin algoritmilla aikataulu, jota noudattamalla kaikki prosessit valmistuvat määräaikana, kutsutaan skeduloituvaksi (schedulable).

Jotta prosesseille voidaan asettaa määräaikoja, niiden suoritukseen kuluva aika on luonnollisesti tunnettava. Yksi reaaliaikajärjestelmien olennainen ero yleiskäyttöisiin tietokoneisiin nähden on, että käyttöjärjestelmä tuntee kaiken niissä ajettavan koodin suoritusajat etukäteen. Suoritusajan arviointimenetelmiä ei käsitellä tässä lähemmin; formaalien ohjelma-analyysimenetelmien kehittyessä alalle on tullut erilaisia työkaluja, mutta valtaosa käytännön työstä on yhä ad-hoc-pohjaista.

Pre-emptiivisessä skeduloinnissa käyttöjärjestelmä voi keskeyttää ajossa olevan prosessin ennen sen valmistumista ja ryhtyä ajamaan toista, tärkeämpää prosessia. Keskeytetty

prosessi siirretään valmiusjonoon. Ei-pre-emptiivisessä skeduloinnissa prosessi suoritetaan keskeytymättä loppuun asti, ja seuraavia prosesseja koskevat skedulointipäätökset tehdään prosessin päättäessä toimintansa. Pre-emptiivisyys on hyödyllinen ominaisuus järjestelmissä, joissa uusia prosesseja luodaan dynaamisesti, koska sen avulla aikatauluista saadaan tehokkaampia ja järjestelmän vastetta kriittisiin tapahtumiin voidaan nopeuttaa.

Skedulointi voi tapahtua off-line- tai on-line-periaatteella. Edellisessä kaikki suoritettavat prosessit tunnetaan etukäteen ja aikataulu lasketaan valmiiksi koko prosessijoukolle järjestelmän kehitysvaiheessa, ja käyttöjärjestelmän tehtäväksi jää suorittaa prosesseja aikataulun mukaan. On-line-periaatteessa skedulointipäätökset tehdään ajonaikaisesti aina prosesseja käynnistettäessä ja niiden päättyessä.

Käyttöjärjestelmän tasolla skeduloinnin pääasialliset toteutusmenetelmät ovat kellopohjainen skedulointi, painotettu Round Robin sekä prioriteettipohjainen skedulointi. Kellopohjainen (time-driven) skedulointi on menetelmistä yksinkertaisin ja käytännössä synonyymi off-line-skeduloinnille. Siinä prosesseja käynnistetään määrättyinä aikoina ennalta laaditun, staattisen aikataulun mukaan, ja ne käyvät keskeytyksettä loppuun saakka. Menetelmä ei ole kovin joustava, mutta sen aiheuttama ajonaikainen kuormitus on pieni ja se soveltuu hyvin tiukkoja aikarajoja vaativiin sovelluksiin.

Painotettu Round Robin on perinteisistä aikajakoisista käyttöjärjestelmistä omaksuttu pre-emptiivinen menetelmä, jossa kullekin aktiiviselle prosessille annetaan vuorollaan aikaannos (time quantum), jonka suuruus riippuu prosessin painoarvosta (weight). Koska kaikki prosessit saavat suoritusaikaa, menetelmä sopii luontevasti esimerkiksi tietoliikennekäyttöön, jossa yhteyksille on taattava tietty palvelun laatu (quality of service, QoS). Usein toistuvat ajovuoron vaihdot aiheuttavat kuitenkin ylimääräistä kuormitusta eikä menetelmä sovellu järjestelmiin, joissa prosessien keskinäisellä suoritusjärjestyksellä on rajoituksia (precedence constraints).

Prioriteettipohjaisessa skeduloinnissa prosessori on aina korkeimman prioriteetin omaavan prosessin käytössä. Mikäli järjestelmään saapuu uusi prosessi, jonka prioriteetti on senhetkisen ajossa olevan prosessin prioriteettia korkeampi, tämä keskeytetään ja uusi prosessi saa ajovuoron. Prioriteettipohjainen skedulointi on dynaamisissa ympäristöissä joustavin menetelmä ja useimmat on-line-skedulointialgoritmit ovat prioriteettipohjaisia.

Kovissa reaaliaikajärjestelmissä prosessijoukon skeduloituvuus on pystyttävä todistamaan etukäteen, ennen uuden prosessin luomista. Mikäli prosessijoukko on staattinen, off-line-skeduloinnin käyttö on toimiva ratkaisu. Dynaamisissa järjestelmissä skeduloituvuustesti on suoritettava jokaiselle uudelle prosessille erikseen. Testillä tarkistetaan, säilyykö järjestelmässä olevien prosessien joukko skeduloituvana, kun uusi prosessi lisätään siihen. Ellei niin käy, prosessi hylätään. Koska skeduloituvuuden takaaminen edellyttää pahimman tapauksen tarkastelua, prosesseja saatetaan hylätä tarpeettomasti. Tämä aiheuttaa resurssien vajaakäyttöä ja keskimääräisen suorituskyvyn laskua (vrt. nopeat järjestelmät). Toisaalta näin vältetään etukäteen ylikuormitustilanteet, erityisesti ns. dominoefekti, jossa uuden korkean prioriteetin prosessin lisääminen aiheuttaa kaikkien aikaisempien prosessien myöhästymisen.

Pehmeissä järjestelmissä, joissa prosessien ajoittainen viivästyminen ei ole kohtalokasta, on tehokkaampaa käyttää best effort -tyyppistä skedulointia, jossa prosessit ajastetaan mah-

dollisuuksien mukaan täyttämään aikarajat, mutta niiden toteutumisesta ei anneta mitään takeita. Ylikuormitustilanteen sattuessa jo käynnistettyjä prosesseja voidaan hylätä. Oikein mitoitetussa järjestelmässä tällaisia tilanteita esiintyy melko harvoin, ja keskimääräinen suorituskyky on resurssien tehokkaamman käytön ansiosta huomattavasti kovaa reaaliaikaskedulointia parempi.

4 Reaaliaikakäyttöjärjestelmistä

Reaaliaikakäyttöjärjestelmät (RTOS) tarjoavat sovelluksille pitkälti samanlaisia palveluita (prosessien ja muistin hallinta, I/O, synkronointipalvelut, jne.) kuin tavallisetkin käyttöjärjestelmät. Monet kaupallisesti saatavilla olevat RTOS:t pohjautuvatkin suoraan perinteisiin aikajakoisiin käyttöjärjestelmiin. Tämä helpottaa sovelluskehitystä, mutta haittana on, että pohjalla olevista reaaliaikakäyttöön soveltumattomista rakenteista on yhteensopivuussyistä vaikea kokonaan luopua.

Yksi reaaliaikakäytössä suurinta huomiota vaativista ongelmista on ulkoisten keskeytysten käsittely. Tavallisissa käyttöjärjestelmissä keskeytyksiä käsittelevät laiteajurit saavat sovellusprosesseja korkeamman prioriteetin, jolloin nämä joutuvat odottamaan keskeytyksen palvelemisen ajan. Koska keskeytysten määrää ja niiden vaatimaa aikaa ei yleensä pystytä ennalta rajaamaan, menettely on sopimaton reaaliaikajärjestelmiin, joissa sovellusprosesseilla on tiukat aikarajat. Joissakin erityiskäyttöön tarkoitetuissa RTOS:eissa ongelma on ratkaistu estämällä ulkoiset keskeytykset järjestelmäkelloa lukuunottamatta kokonaan ja käyttämällä oheislaitteita pollausperiaatteella. Laajempaan käyttöön soveltuvampi ratkaisu on sallia keskeytykset, mutta käyttää niitä vain aktivoimaan oheislaitetta käsittelevä ajuriprosessi, joka skeduloidaan samoin kuin sovellusprosessit.

Työasematyyppistä sivuttavaa virtuaalimuistia ei voida käyttää RTOS:eissa ennustamattomien ja pahimmillaan erittäin aikaavievien sivuhakujen vuoksi. Käyttöjärjestelmät tarjoavat sovelluskehittäjälle mahdollisuuden dynaamiseen muistinvaraukseen, mutta kiinteä muistin allokointi on turvallisin ratkaisu.

Käyttöjärjestelmän systeemikutsujen on itsensä toimittava täsmällisten aikarajojen puitteissa. Niiden tulee olla keskeytettäviä (pre-emptable kernel), jotta ne eivät aiheuta kovia reaaliaikavaatimuksia omaavien prosessien myöhästymistä.

Resurssien jako ja synkronointi vaativat erityisjärjestelyitä. Tavalliset käyttöjärjestelmät tarjoavat semaforeja, joilla voidaan toteuttaa kriittisiä sektioita. Semaforeille on ominaista ns. priority inversion -ilmiö, jossa kriittisen sektion varannut alemman prioriteetin prosessi estää korkeamman prioriteetin omaavaa prosessia pääsemästä ajettavaksi. Reaaliaikajärjestelmässä ilmiö olisi tuhoisa, joten semaforit on korvattava RTOS:eissa erityisillä protokollilla (Priority Inheritance, Priority Ceiling, Stack Resource Policy), jotka rajoittavat odotusaikaa muuttamalla dynaamisesti kriittistä sektiota tavoittelevien prosessien prioriteetteja.

5 Esimerkkijärjestelmä: Solaris

Solaris on Sunin UNIX-tyylinen käyttöjärjestelmä. Tänä päivänä se perustuu AT&T:n SystemV Release 4:aan, mutta omaa vielä jonkin verran perinteitään vanhalle BSD-pohjaiselle SunOS4-järjestemälle. Solaris toimii Sunin SPARC- ja UltraSPARC-arkkitehtuureissa, ja lisäksi se on saatavilla Intelin x86-raudalle. x86-tuki on kuitenkin viime aikoina ollut hieman ailahtelevaista, ja usein laahaa SPARC-tuen jäljessä. Koska Solaris on täysimittainen UNIX-käyttöjärjestelmä, on sitä helpompi kuvitella työpöytä- tai serverikäyttöjärjestemänä kuin sulautetussa järjestelmässä, joissa reaaliaikakäyttöjärjestelmiä tavataan useammin. Solariksessa on kuitenkin joitain piirteitä, jotka tekevät siitä reaaliaikakäyttöön sopivan. Tosin ainakaan vielä ei kannata alkaa odottaa mikroaaltouunia, joka toimii Solariksella.

Solariksen alkuaikoina (Solaris 2.0) Solarista pidettiin ankeana ja kankeana järjestelmänä, mutta se on kehittynyt huomattavasti noista ajoista, ja nykyään järjestelmä on sisältä kohtuullisen kaunis. Nykyään onkin tavanomaista, että Open Source -maailmassa järjestelmiä toteutetaan Solariksen esimerkkien mukaisesti.

5.1 Solariksen reaaliaikaominaisuudet

Solaris tarjoaa lukuisia klassisesta UNIX-järjestelmästä[1] puuttuneita ominaisuuksia, jotka edesauttavat järjestelmän mahdollisten reaaliaikavaatimuksien toteutumista.

5.1.1 Lukitus

Kuten aiemmin tekstissä mainittiin, priority inversion voi aiheuttaa pahoja ongelmia reaaliaikakäyttöjärjestelmässä. Priority inversion on ratkaistu Solariksessa liittämällä lukitusrakenteisiin *turnstile*. Turnstile on tietorakenne, jonka avulla on mahdollista saada tietää kuka pitää hallussaan lukkoa, sekä mahdollisesti pakottaa tämä säie ajoon, jotta se voi vapauttaa lukon. Tämä tapahtuu niin, että kun korkeamman prioriteetin säie havaitsee haluavansa lukkoa, joka on matalamman prioriteetin säikeen halussa, se lainaa (engl. *will*) oman prioriteettinsa lukkoa hallitsevalle säikeelle, ja tämä pääsee ajoon niin pitkäksi aikaa, että ehtii vapauttaa lukon. Tämän jälkeen prioriteetit palautetaan ennalleen, ja alkuperäinen korkean prioriteetin säie pääsee ajoon ja saa lukittua haluamansa tietorakenteen. Kyseinen toimintamalli tunnetaan yleisesti nimellä *priority inheritance*.

Lisäksi, koska Solariksen lukkojen hienojakoisuus on hiottu korkeaksi, on todennäköistä, että lukko alkunperinkin on vapaana.

5.1.2 Skedulointiluokat

Solaris tarjoaa reaaliaikaohjelmille oman skedulointiluokan. Käytännössä tämä tarkoittaa sitä, että reaaliaikaluokassa olevat ohjelmat saavat CPU:n käyttöönsä korkeammalla prioriteetilla kuin käyttöjärjestelmän omat system threadit, puhumattamaan muista "normaaleista"ohjelmista. Skedulointi reaaliaikaluokan sisällä hoituu taulukkopohjaisella skeduloinnil-

[1] "Klassinen"tässä yhteydessä tarkoittaa luokkaa Bell Labs UNIX tai Berkeley CSRG:n BSD.

la, jossa järjestelmän pystyttäjällä on suhteellisen vapaat kädet päättää siitä, miten tiettyä säiettä skeduloidaan. Hän pystyy määrittelemään mm. prioriteetilla X pyörivien säikeiden aika-annoksen koon, sekä mihin säikeen prioriteettia lasketaan, vai lasketaanko ollenkaan, jos se käyttää aika-annoksensa loppuun.

5.1.3 Keskeytysten käsittely

Solariksessa keskeytyksillä on aina korkeampi prioriteetti kuin millään sovelluksella, mukaanlukien reaaliaikaluokassa olevat sovellukset. Tämä ei kuitenkaan välttämättä ole ongelma kahdestakaan syystä. Ensinnäkin, keskeytyskäsittelijät ovat yleensä "tarkkaa"koodia, joka yrittää suoriutua tehtävästään mahdollisimman nopeasti. Toisekseen, moniprosessorijärjestelmässä on mahdollista ohjata kaikki keskeytykset yhdelle prosessorille, ja säästää toinen kokonaan vapaaksi esim. reaaliaikasovellusten suorittamiseen.

5.2 Johtopäätökset

Solaris tarjoaa hienoja, houkuttelevia ja teknisesti kauniita ratkaisuja eri ongelmiin reaaliaikajärjestelmissä. On kuitenkin havaittavissa selvästi, että Solariksen ratkaisut vaativat usein suuria määriä laitteistoresursseja, kuten useamman prosessorin ja (kymmeniä, jopa satoja) megatavuja keskusmuistia. On selvää, ettei tällaisia resursseja ole käytettävissä tyypillisissä RTOS:n sovelluskohteissa, joissa raudalle on hyvin rajoitetut resurssit. Siksi Solaris tuskin ihan lähipäivinä tulee huolehtimaan konjakkilasien pesusta astianpesukoneessa tai Takahuhdin mummon sydämentahdistimesta.

6 Esimerkkijärjestelmä: Chorus

Chorus-käyttöjärjestelmän juuret löytyvät ranskalaisesta akateemisesta maailmasta. Se oli INRIAn tutkimusprojekti vuodesta 1979 vuoteen 1986, jonka jälkeen vastuun kehityksestä ja tuesta otti Chorus Systems -niminen yritys. Jokin aika sitten Sun Microsystems osti kyseisen yrityksen, ja sen jälkeen ilmoitti lopettavansa Choruksen tukemisen ja kehittämisen[2].

Surmastaan huolimatta Chorus ehti saavuttaa huomattavan jalansijan, ja on vahvasti käytössä erityisesti mobiilimaailmassa. Koska Sun hiljattain pisti Choruksen lähdekoodin vapaaseen levitykseen, on luultavaa, että Chorus saavuttaa kiinnostusta myös harrastelijahakkereiden keskuudessa tulevaisuudessa.

Solariksen tavoin Chorus on suuremman mittakaavan käyttöjärjestelmä, josta löytyy ominaisuuksia lähes joka lähtöön. Järjestelmästä löytyy esimerkiksi TCP/IP-pino, ja tuki Java-sovelluksille on saatavissa. Sovellusrajapintojakin löytyy aina POSIX-rajapinnasta lähtien.

Päätellen Chorus 5.1:n mukana tulevista lähdekoodihakemistoista, tuki on olemassa ainakin seuraaville prosessoriarkkitehtuureille:

- MIPS

[2]Me emme millään tavalla vihjaa, että Sun halusi täten poistaa kilpailijan Solarikselta.

- PowerPC
  - MPC8xx
  - PPC60x/750
- UltraSPARC
- Intel x86

On kuitenkin todennäköistä, että Chorus on portattu monille muillekin platformeille, mutta tuki ei syystä tai toisesta ole päätynyt mukaan päähaaraan.

6.1 Choruksen perusteet

Chorus on hajautettu käyttöjärjestelmä, joka rakentuu seuraavien käsitteiden ja perusterminologian varaan:

Site. Site on yksi Chorus-järjestelmän node, joka sitoo itseensä ko. paikassa fyysisesti olevan raudan ja siihen liittyvät resurssit.

Nucleus. Kuten nimikin hieman vihjaa, nucleus on Choruksen ydin. Jokaisella sitellä on nucleus. Nucleuksen tehtävät muistuttavat hyvin paljon minkä tahansa muun käyttöjärjestelmän ytimen tehtäviä: laitteiston abstrahointi, muistinhallinta, skedulointi ja IPC. Ainakin teoriassa Choruksen nucleuksen pitäisi olla hyvin portattava, sillä hyvin suuri osa siitä aivan alinta tasoa lukuunottamatta kuuluisi olla MI koodia.

Subsystem. Nucleuksen päällä ajetaan nk. subsysteemeitä. Nämä ovat kirjastopaketteja, jotka tarjoavat sovelluksille tarvittavat rajapinnat, jotta sovellusten ajaminen on mielekästä. Yksi nucleus voi tukea useaa subsystemiä samanaikaisesti. Hauska esimerkki subsystemistä on *Chorus/MiX*, joka tarjoaa sovelluksille UNIX-tyylisen rajapinnan, ja ilmeisesti kykenee jopa ajamaan SysV ABIa (ELF) käyttäviä binäärejä sellaisenaan.

Actor. Choruksessa sana actor tulkitaan kokoelmana resursseja. Sen voidaan katsoa olevan samansukuinen olento prosesseille, mutta itsessään actor ei kuitenkaan pidä sisällään minkäänlaista execution tracea. Koodin suorittaminen actorin sisällä vaatii säikeen, joita voi olla ajossa useita samanaikaisesti yhden actorin sisällä.

Port. Port on Choruksessa kommunikaation päätepiste. Portit kuuluvat aina tietylle actorille, mutta niitä voidaan siirtää actoreiden välillä, jopa toisella sitellä olevalle actorille. *Messaget*, joita porttien välillä lähetellään, ovat aina osoitettuna portille eikä actorille, joka omistaa portin. Tämä mahdollistaa viestien menemisen oikeaan paikkaan, vaikka portti olisikin muuttanut omistajaa välillä.

6.2 Choruksen skedulointi

Choruksen nucleus tukee monia skedulointipolitiikkoja, kuten staattinen time slicing ja prioriteetin degradaatioon perustuva skedulointi. Reaaliaikanäkökulmasta katsottuna paras vaih-

toehto luultavasti kuitenkin on staattiseen prioriteettiin perustuva pre-emptiivinen skedulointi, joka pistää aina ajoon korkeimman prioriteetin omaavan CPU:ta haluavan säikeen.

6.3 Johtopäätökset

Solariksen tavoin myös Chorus tukee lukitusrakenteita, joiden avulla on mahdollista välttää priority inversion. Tämä on toteutettu samaan tapaan kuin Solariksessa eli käyttämällä priority inheritance -protokollaa.

Chorus näyttäisikin olevan pitkälti samalla rintamalla Solariksen kanssa: isohko ja ainakin täydessä lastissa varsin raskas järjestelmä, joka kykenee suoriutumaan kaikista reaaliaikajärjestelmiltä vaadittavista tempuista. Tosin jäi hieman sellainen näppituntuma, että Chorus olisi modulaarisempi ja helpommin karsittavissa oleva kuin Solaris.

Hajautettujen ominaisuuksiensa ja niiden tuomien hyötyjen ansiosta Chorus soveltuu erinomaisesti vikasietoisuutta vaativiin järjestelmiin.

7 Esimerkkijärjestelmä: UROS

UROS (Ultimate Realtime Operating System) on Urosv Platisevn kirjoittama reaaliaikakäyttöjärjestelmä, joka on tarkoitettu Atmelin 8-bittiselle AVR RISC-prosessorille. Tästä järjestelmästä mielenkiintoisen tekee se, kuinka pienillä resursseilla sen pitää tulla toimeen: muistia löytyy maksimissaan muutama kilo, eikä flashilläkään mässäillä kuin muutamien hassujen kymmenien kilotavujen verran. UROS on Choruksen tavoin vapaasti haettavissa verkosta lähdekoodeineen.

Järjestelmän kirjoittamisen alkuperäinen motivaatio oli toimia jonkinlaisena kirjastona AVR-sovelluskehittäjille, jotta heidän ei tarvitsisi kehittää kaikkia laittestonläheisiä rutiineja itse. Lisäksi käyttöjärjestelmän olemassaolo muutenkin helpottaa sovelluskehittäjän elämää, koska hänen ei tarvitse miettiä käyttöjärjestelmälle perinteisesti kuuluvia asoita, kuten moniajoa.

Pienestä koostaan huolimatta (700 tavua flashiltä kaikki käännösoptiot päällä ja memory footprintiltaan alkaen kymmenestä tavusta[3]). UROS tarjoaa tuttuja ominaisuuksia: skedulointi, IPC ja debuggaustuki. Kuten aiemmin jo mainittiin, kaupan päälle tulee myös laitteistoläheiset rutiinit.

7.1 Skeduleri

UROS tarjoaa kaksi vaihtoehtoista mallia skedulointiin: *time slicing* sekä *event mode*.

Time slicing -tilassa UROS toimii kuten tuikinormaali moniajava käyttöjärjestelmä, joka läpinäkyvästi vaihtaa sovelluksesta toiseen aina saatuaan tietyn määrän kellokeskeytyksiä.

Event-tilassa kellokeskeytys on pois päältä, ja on sovelluksen vastuulla pyytää skedulointia, silloin kun sille sattuu sopimaan. Tämä strategia tunnetaan myös yleisesti koopera-

[3]Tämä ei pidä sisällään yhtään ajossa olevaa sovellusta. Sovellusten tuoma memory overhead on joitain kymmeniä tavuja per sovellus. Lisäksi mukaan pitää tietenkin laskea sovelluksen oma koodi.

tiivisena moniajona.

7.2 Johtopäätökset

UROSin reaaliaikaominaisuuksia perustellaan sillä, että koska workload on järjestelmässä aina staattinen, voimme tietää etukäteen, kuinka usein joku tietty prosessi on ajossa, ja täten varmuudella sanoa pystyykö se vai eikö se pysty suorittamaan asiansa ajoissa.

Ikävä puoli tässä staattisessa moniajossa on kuitenkin se, ettei järjestelmä millään tavalla takaa ulkoisten keskeytysten latenssi- ja prosessointiaikaa. Jos keskeytys tulee ikävästi juuri sen käsittelyä harrastavan sovelluksen oltua ajossa, saattaa kulua luvattoman pitkä aika, ennen kuin se käsitellään.

Lähteet

Giorgio C. Buttazzo. Hard Real-Time Computing Systems – Predictable Scheduling Algorithms and Applications. ISBN 0-07-114243-6. Kluwer Academic Publishers, Second Printing, 2000, ISBN 0-7923-9994-3.

C. M. Krishna, Kang G. Shin. Real-Time Systems. McGraw-Hill Computer Science Series, 1997.

Alan Burns, Andy Wellings. Real-Time Systems and Programming Languages. ISBN 0-201-40365-X. Addison Wesley Longman Ltd., Second Edition, 1997.

Jim Mauro, Richard Mc Dougall. Solaris Internals – Core Kernel Architecture. ISBN 0-13-022496-0. Pearson Education, First Edition, 2000.

M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the Chorus distributed system. Technical Report CSTR -90-25, Chorus Systemes, F-78182 St. Quentin-enYvelines Cedex, France, 1991.

Uroš Platišěn: Ultimate Real-time Operating System I. Versions 1.1, 2.

Reaaliaikatyökalut

Oskar Kohonen
Timo Lilja

1. Johdanto

Järjestelmä jossa on merkittävää millä hetkellä tietty tehtävä suoritetaan, eikä vain missä järjestyksessä tehtävät suoritetaan, kutsutaan yleisesti *reaaliaikajärjestelmiksi*. Reaaliaikajärjestelmiä on pääasiassa kahdenlaisia pehmeitä ja kovia. Pehmeä reaaliaikaisuus tarkoittaa että jos järjestelmä myöhästyy aikarajastaan niin suorituskyky laskee merkittävästi. Esimerkkejä pehmeistä reaaliaikajärjestelmistä ovat äänentoistolaitteistot ja pesukoneiden ohjaimet. Kovassa reaaliaikaijärjestelmässä aikarajasta myöhästyminen merkitsee että järjestelmä toimii virheellisesti, pahimmassa tapauksessa siten että se johtaa vahinkoihin reaalimaailmassa. Kovia reaaliaikajärjestelmiä ovat lentokoneiden sekä joidenkin työkoneiden ohjaimet.

Monella sovellusalueella on tarvetta järjestelmälle joka tekee tiettyä hyvin määriteltyä tehtävää joka vaati totetutukseensa tietokonetasoista logiikkaa. Yleensä tehtävän toteutus perinteisellä tietokoneella olisi joko liian vaikeaa tai kallista, joten tarvitaan erityisesti tehtävään soveltuva laitteisto ja ohjelmisto. Näitä järjestelmiä joissa ohjelmisto sulautuu laitteistoonsa kutsutaan *sulautetuiksi järjestelmiksi*. Sulautetun järjestelmän toteutukseen liittyy monia haasteita, toiminnallisuutta voi toteuttaa useilla eri yhdistelmillä laitteistoa ja ohjelmistoa, järjestelmät ovat usein luonteeltaan sellaisia että ne vaativat hyvin suurta luotettavuutta ja yleisiä ovat myöskin reaaliaikaisuusvaatimukset.

Viime vuosina prosessoritekniikka on kehittynyt siten että muistin käsittelyn tehostamiseksi käytetään yhä enemmän nopeita välimuisteja mikä on sallinut huomattavan suorituskyvyn lisäyksen pöytätietokoneissa ja palvelimissa. Sulautetuissa ja reaaliaikajärjestelmissä on kuitenkin huomattavan hankala käyttää välimuistillisia prosessoreita koska välimuistin käyttö vaikeuttaa ohjelmiston aika-analyysia.

Tämän paperin sisällön fokus on työkaluissa jotka helpottavat sulautettujen reaaliaikajärjestelmien kehitystä välimuistillisilla prosessoreilla, mutta katsauksemme kattaa myöskin hieman yleisempään käyttöön tarkoitettuja reaaliaikajärjestelmissä käytettäviä työkaluja. Tutkimme työkaluja neljästä ryhmästä: *WCET-analyysityökaluja*, eli työkaluja joilla analysoidaan ohjelman pahimman tapauksen suoritusaikoja, *mallintarkastustyökaluja*, joilla kehitettävästä järjestelmästä voidaan luoda formaaleja malleja joiden eri ominaisuuksia voidaan tutkia, *sulautustyökaluja* jotka helpottavat sulautusprosessia sekä *asiantuntijajärjestelmiä* jotka voivat ennakkoon ohjelmoidun tietämyksen perusteella antaa ohjeita ja ratkaisuja esitettyihin ongelmiin.

1.1 Välimuistit

Välimuistin perusideana on että koska nopea muisti on kallista, ja muistiin viitataan useimmin lähekkäin oleviin osoitteisiin, voidaan hitaamman muistin toimintaa tehostaa lisäämällä prosessorin ja keskusmuistin välille pienikokoinen nopea muisti joka säilyttää keskusmuistin eniten viitattujen osoitteiden sisällön. Kun viitataan tiettyyn muistipaikkaan tarkistetaan ensin jos kyseinen muistipaikka on jo tallennettu välimuistiin. Jos se löytyy välimuistista, eli tuli välimuistiosuma (cache hit) käytetään välimuistissa olevaa tietoa, jos ei, eli tuli ”välimuistihuti” (cache miss) pitää hakea keskusmuistista tieto välimuistiin josta sitä sen jälkeen käytetään. Välimuisteihin liittyy keskeisesti kolme parametria: välimuistin koko, lohkon koko sekä välimuistin assosiatiivisuus. Välimuistin koko viittaa kapasiteettiin, eli miten paljon tietoa välimuistiin mahtuu. Lohkon koko taas kertoo miten suurissa lohkoissa muistia luetaan välimuistiin. Assosiatiivisuus on tietolohkon mahdollisten sijoittautumisten lukumäärä välimuistissa. Välimuistin sanotaan olevan *set-associative* mikäli muistin jokainen lohko kuvautuu jonkin funktion mukaan tiettyyn osaan välimuistia [8].

2. WCET-työkalut

WCET-analyysi pyrkii löytämään konservatiivisen ja tarkan ylärajan ohjelman suoritusajalle. Konservatiisuus tarkoittaa sitä että ohjelman suoritusaika on aina enintään laskettu raja muttei koskaan sitä pidempi, tarkkuus tarkoittaa että laskettu raja on mahdollisimman lähellä oikeaa suoritusaikaa. Raja lasketaan hyödyntäen ohjelman vuoanalyysiä. Vuoanalyysi tutkii staattisesti miten kontrolli ja tieto siirtyy ohjelman suorituksen aikana, ja sitä tehdään tyypillisesti ohjelmointikielten kääntäjissä. Vuoanalyysissä ohjelma jaetaan peruslohkoiksi, jotka ovat ohjelman osia joiden käskyt suoritetaan aina peräkkäin, eli toisin sanoen peruslohkot eivät sisällä hyppykäskyjä lohkon sisään tai peruslohkosta ulos. Hyppykäskyt muodostavat peuslohkojen välille yhteyksiä ja peruslohkojen ja niiden yhteyksien summana muodostuu graafi jota kutsutaan (kontrolli)vuokaavioksi. Kontrollivuokaavio kuvaa miten kontrolli voi siirtyä ohjelman suorituksen aikana. Tietovuoanalyysi tutkii tarkemmin miten tieto siirtyy muuttujasta toiseen peruslohkoissa. WCET-analyysissa yritetään löytää ohjelman jokaisen peruslohkon vaatima aika, ja sitten vuokaavion avulla yhdistää nämä ajat että saadaan koko ohjelman suoritusaika. Vanhemmilla mikrokontrollereilla kuten Z80 ja 8051 kaikki käskyt ovat vakioaikaisia joten aika voitiin laskea yksinkertaisesti yhteenlaskulla. Nykyprosessorien liukuhihnat ja välimuistit tekevät laskennasta huomattavasti monimutkaisemman [1].

2.1 Bound-T

Bound-T on Space Systems Finlandilla kehitetty WCET-analyysiin keskittyvä työkalu. Sille syötetään käännetty ja linkitetty ohjelma, joka analysoidaan staattisesti ja käyttäen vuoanalyysia se löytää ylärajat ohjelman suoritusajalle sekä pinon käytölle. Analyysin sivutuotteena syntyvistä vuokaavioista voidaan generoida kuvia AT&T Bell Labsin dot ohjelman avulla. WCET-analyysin ratkaisu suoritetaan etsimällä jokaiselle peruslohkolle pahin mahdollinen suoritusaika, sekä silmukoiden suorituskerroille ylärajat. Bound-T löytää ylä-

rajat suorituskerroille sellaisille silmukoille, joiden toistokerrat riippuvat laskurimuuttujasta. Muille silmukoille ohjelmoija voi spesifioida suorituskertojen määrän käyttäen erillistä assertiotiedostoa, jossa voidaan myös määritellä kokonaisten aliohjelmien suoritusaikoja esimerkiksi jos ne eivät ole valmiita tai niitä ei jostain syystä haluta analysoida. Peruslohkojen ja silmukoiden luomien rajoitteiden avulla ohjelmasta luodaan Integer Linear Programming ongelma, joka kuvaa ohjelman polkujen pituuksia. Maksimoimalla tätä käyttäen *lp-solve*-nimistä ohjelmapakettia löydetään yläraja WCET-ajalle. Bound-T on tarkoitettu pieniin ja keskisuuriin sulautetuihin prosessoreihin ja siksi siitä puuttuu välimuistianalyysi, mikä johtaa siihen että tulokset ovat turhan pessimistisiä välimuistillisilla prosessoreilla. Bound-T on myöskin riippuvainen kohdealustasta, ja tukee Intel 8051, SPARC v7 ERC32, sekä DSP prosessoria ADSO-21020, ja tuki SHARC (21060) prosessorille on tulossa[10].

2.2 Cinderella

Cinderellan lähestymistapa WCET analyysiin jakaa ongelman kahteen osaan:

- Ohjelman polkuanalyysi
- Mikroarkkitehtuurin mallinnus

Ohjelman polkuanalyysi viittaa perinteiseen vuoanalyysiin ja perustuu vuokaavion määräämiin ohjelman *rakenteellisiin rajoituksiin* miten eri polkuja voidaan suorittaa. Tämän lisäksi analysoidaan ohjelman *toiminnallisia rajoituksia*, jotka liittyvät muuttujien arvoihin ja ovat siksi saatavissa tietovuoanalyysista. Cinderellassa on kuitenkin arvioitu että tiedot ovat tehokkaammin saatavissa ohjelmoijalta, jonka tiedon pohjalta voidaan laskea tarkkoja rajoja silmukoiden suorituskertoihin [3]. Mikroarkkitehtuurin mallinnus koostuu prosessorin liukuhihnan ja sekä käsky- että datavälimuistien malleista. Välimuistien oletetaan olevan set associative ja niiden koko voi vaihdella. Datavälimuistin analyysi on huomattavasti vaikeampaa kuin käskyvälimuistin, koska datan osoitteet voivat muuttua suorituksen aikana. Tämän takia Cinderellassa on rajoitettu ohjelmia niin että dynaamista dataa ei sallita. Tästä rajoituksesta huolimatta ei aina voida selvittää datan osoitteita staattisesti, jolloin yksinkertaisesti oletetaan ettei minkään muistiviittauksen tulokset löydy välimuistista. Analyysia varten vuokaavion peruslohkot on jaettava osiksi, joiden välimuistikäyttäytyminen on sisäisesti samanlainen, eli kuuluvat kaikki samaan käskyvälimuistin osaan. Näitä peruslohkon osia kutsutaan rivilohkoiksi. Rivilohkot jotka käyttävät samaa välimuistiosaa ovat *konfliktissa*. Rivilohkojen konfliktit muodostavat *välimuistin konfliktikaavion*. Välimuistin sisällön muuttuminen kun kontrolli siirtyy eri solmuihin konfliktikaaviossa on kuvattu *välimuistin tilasiirtymäkaaviossa* ja sen analyysista voidaan laskea välimuistin osumat ja hudit.

2.2.1 Cinderella käytännössä

Cinderellalle syötetään valmiiksi käännetty ja linkitetty ohjelma josta se luo vuokaavion, sekä välimuistin konfliktikaavion ja tilasiirtymäkaavion. Lisäksi Cinderella tulostaa tiedostoja, joissa lähdekoodiin on lisätty tieto peruslohkojen rajoista ja Cinderella pyytää käyttäjältä silmukoiden suorituskerroille rajoja. Tämän jälkeen Cinderella alkaa laskea WCET:tä. Käyt-

täjä voi halutessaan antaa lisää toiminnallisia rajoituksia, jotta voidaan saavuttaa tarkempi WCET-estimaatti. Cinderella ei vaadi parikseen erikoiskääntäjää luomaan sen tarvitsemia kaavioita, mutta sen sijaan se on riippuvainen kohdelaitteistostaan. Cinderellassa on valmiit mallit Intel i960KB ja Motorola MC68000 -prosessoreille.

3. Mallintarkastustyökalut

Mallintarkastustyökaluilla tarkistetaan järjestelmän *oikeellisuusominaisuuksia*. Tarkistettavat ongelmat ovat tyypillisesti rinnakkaisohjelmointiin liittyviä aikarajoite-, reiluus-, deadlockin välttämis-, ym. ongelmia.

Erilaisten mallintarkastustyökalujen toimintaperiaatteet ovat hyvin samanlaiset: kullakin työkalulla on *kuvauskieli*, jolla ongelma esitetään. Mallit ymmärretään yleensä äärelliseksi tilakoneiksi. Kun mallintarkastustyökalulle on määritelty ongelma, luo työkalu *saavutettavuusgraafin*, joka kuvaa järjestelmän kaikkia mahdollsia tiloja ja tilasiirtymiä.

Mallintarkastus tarkoittaa saavutettavuusgraafin analysointia niin, että tutkitaan, päteekö haluttu ominaisuus kaikissa tiloissa. Esimerkiksi niin, että järjestelmä ei saa missään tilanteessa joutua sellaiseen silmukkaan, josta ei pääse eteen päin (deadlock).

Turvallisuusominasuuksia mallinnetaan yleensä *lineerisella temporaalilogiikalla* (LTL). LTL on tavallisen propositiologiikan laajennus, joka sisältää seuraajatilan käsitteen eli operaattorit, joilla voidaan sanoa: *jossain* seuraavassa tilassa pätee logiikan lause X tai *kaikissa* seuraavissa tiloissa pätee X.

3.1 UPPAAL

UPPAAL on työkalu jolla voidaan mallintaa, simuloida sekä verifioida reaaliaikajärjestelmiä. Työkalu soveltuu järjestelmille jotka voidaan kuvata joukkona epädeterministisiä prosesseja joilla on äärelliset kontrollirakenteet, reaaliaikaiset kellot ja kommunikoivat joko kanavien tai yhteisten muuttujien avulla. Keskeisimmät sovellukset ovat reaaliaikakontrollerit sekä protokollat joissa ajoitus on tärkeää. Työkalu analysoi pääasiassa mallin invarianssiominaisuuksia sekä saavutettavuusominaisuuksia. [2].

3.1.1 Rakenne

UPPAAL koostuu kolmesta osasta:

Kuvauskieli jota käytetään järjestelmien suunnitteluun ja mallinnukseen.

Simulaattori jolla voidaan tutkia nopeasti yhden mahdollisen suorituksen vaikutuksia

Mallitarkastaja jolla saadaan kaikki suoritukset kattavia tuloksia

Järjestelmän voi kuvata sekä graafisella että tekstipohjaisella kuvauskielellä. Järjestelmän teoreettinen perusta on *ajoitettu automaatti* [5]. Ajoitettu automaatti on tila-automaatin laajennus, jossa tila-automaattiin on lisätty kellomuuttujat. UPPAAL kuvaa järjestelmän verk-

kona ajoitettuja automaatteja, jotka on käytettävyyden parantamiseksi laajennettu tavallisimmilla muuttujatyypeillä (esim. boolean ja integer). Ajoitetut automaatit esittävät järjestelmän eri prosesseja.

Mallitarkastajan pääasiallinen tehtävä on tarkastella, onko jokin konfiguraatio (eli yhdistelmä kontrollisolmuja automaateissa sekä data- ja kellomuuttujien yhdistelmä) mahdollinen saavuttaa alkutilasta. UPPAAL tulostaa diagnostiikkajäljen, joka kertoo miksi tietty ominaisuus on tai ei ole voimassa. Tätä jälkeä voidaan myös visualisoida simulaattorin avulla.

UPPAAL työkalua on sovellettu muun muuassa audio/video sekä lähiverkkojen törmäysten välttämisprotokollien kehityksessä sekä vaihdelaatikkokontrollerissa.

3.2 SPIN

SPIN [9] on asynkronisten prosessien mallintarkastusjärjestelmä. SPIN on suunniteltu erityisesti *prosessien välisten interaktioiden* mallintamiseen ja tarkastamiseen. Ongelmat määritellää PROMELA-nimisellä kuvauskielellä. PROMELA on Algol-tyyppinen ohjelmointikieli jota on täydennetty rinnakkaisuusprimitiiveillä. Oikeellisuusvaatimukset esitetään *lineaarisella temporaalilogiikalla* (LTL).

SPIN verifioi mallin oikeellisuuden konstruoimalla *saavutettavuusgraafin* ja tarkistamalla että annettu LTL-kaava toteutuu kaikissa saavutettavissa olevissa tiloissa. Jos näin ei ole, antaa SPIN vastaesimerkin.

Periaattessa SPINiä voi käyttää minkä tahansa rinnakkaisongelman mallintamiseen. Paperissa [9] esimerkkinä mainitaan asynkronisen kahden prosessin skedulointi: Asiakasprosessi käyttää resurssia ja jos sitä ei ole saatavilla, menee nukkumaan. Palvelinprosessi jakelee resurssia ja herättää asiakkaan tarvittaessa. Molemille prosesseille määritellään tietyt synkronointiehdot ja toteutaan määrittely PROMELA-kielelllä. Tämän jälkeen SPIN luo saavutettavuusgraafin ja etsii mahdollisia *deadlock*-tilanteita.

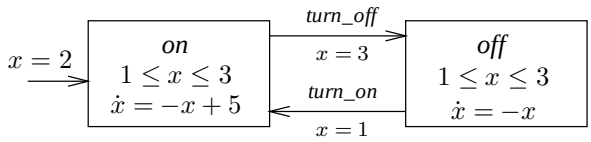
Vähänkin monimutkaisemman protokollan tai prosessien interaktioiden ollessa kyseessä, saavutettavuusgraafin tila kasvaa hyvin suureksi. SPINissä on tätä varten tehty monenlaisia optimointeja, joiden takia voidaan käsitellä hyvinkin suuria malleja kohtuullisessa ajassa ja tilassa.

3.3 HYTECH

HYTECH [7] on mallintarkastusjärjestelmä *hybrideille järjestelmille*. Hybridit järjestelmät tarkoittavat järjestelmiä, joissa on sekä diskreettejä että jatkuvia komponentteja. Esimerkiksi termostaatti, joka pyrkii säilyttämään huoneen lämpötilan tietyllä arvoalueella, voidaan ymmärtää hybridinä järjestelmänä: huoneen lämpötila on jatkuva ja lämmityksen päälläolo diskreetti suure.

Hybridejä järjestelmiä kuvataan *hybridien automaattien* avulla. Hybridiautomaatti on tavallinen äärellinen automaatti (tilakone), jota on laajennettu käsittelemään jatkuvia suureita. Käytännössä tämä tarkoittaa sitä, että hybridin automaatin tilasiirtymärelaatioihin voi sisällyttää reaalilukuihin perustuvia rajoitteita ja siirtymäehtoja.

Kuvassa 1 esitelty hybridiautomaatti kuvaa sekä termostaatin että ympäristön tilaa. Muut-



Kuva 1: Yksinkertainen termostaatti hybridinä automaattina

tuja x kertoo termostaatin lämpötilan. Termostaatti lähtötila on $x = 2$ eli kaksi astetta. Termostaatilla on kaksi tilla *on* ja *off*, jotka kertovat onko lämmitys päällä vai ei. Kun termostaatti on päällä (tilassa *on*), pysytään tässä tilassa eli pidetään lämmitystä päällä, kun lämpötila on välillä $1 \leq x \leq 3$ astetta. Kun lämmitys on päällä, termostaatti lämpenee differentiaaliyhtälön $\dot{x} = -x + 5$ mukaisesti. Lämmitys katkaistaan, kun lämpötila saavuttaa kynnysarvon $x = 3$. Vastaavat määrittelyt *off*-tilalle kuvaavat miten menetellään, kun termostaatti ei ole päällä.

HYTECH-työkalulla voidaan määritellä edellä kuvatun kaltaisia hybridisiä automaatteja ja tarkistaa niistä *turvallisuus*-ominaisuuksia (*safety properties*). Turvallisuusominaisuus on rinnakkaisohjelmoinnista tuttu käsite ja tarkoittaa sitä että prosessin suorituksen aikana ei päädytä tilaan, jossa ei-haluttu ominaisuus tulee todeksi. HYTECH suorittaa turvallisuusanalyysin konstruoimalla ns. *saavutettavuusgraafin* ja tutkimalla että lähtötilasta ei voida joutua sellaiseen tilaan, missä *turvaton* (*unsafe*) ominaisuus pätee.

HYTECHille annetaan siis tutkittavan järjestelmän kuvaus hybridinä automaattina[4] ja tutkittava turvallisuusominaisuus. HYTECH luo saavutettavuusgraafin ja mikäli turvallisuusominaisuutta rikkova tila löytyy, tulostetaan *suorituspolku* lähtötilasta turvallisuusominaisuutta rikkovaan tilaan. Mikäli tällaista tilaa ei löydy, todetaan järjestelmä turvalliseksi kyseisen ominaisuuden suhteen.

Koska HYTECH pystyy hybridien automaattien avulla käsittelemään sekä diskreettejä että jatkuvia suureita, soveltuu se monien sulautetuissa järjestelmissä esiintyvien ongelmien formulointiin ja verifiointiin. Reaaliaikaominaisuuksia HYTECHissa ei ole.

4. Sulautustyökalut

Sulautustyökaluja käytetään sulautusprosessien helpottamiseksi. Sulautettujen järjestelmien erikoispiirre on se, että laitteisto ja ohjelmisto ovat niin läheisesti toistensa kanssa tekemisissä että laitteiston valinta riippuu pitkälti ohjelmistosta, ja päinvastoin. Tästä johtuen olisi parasta suunnitella sekä laitteisto että ohjelmisto samanaikaisesti ja saman viitekehyksen sisällä. Esimerkkinä tällaisesta lähestymistavasta on alla esitetty COSYMA-työkalu. Muunkinlaisia sulautustyökaluja on olemassa mutta aikarajoituksen puitteissa niitä ei tähän katsaukseen voitu sisällyttää.

[4] HYTECHissä käytetään *lineaarisia* hybridejä automaatteja, jotka ovat hieman rajoittuneempi versio yllä esitellystä automaatista.

4.1 COSYMA

COSYMAn (COSYnthesis for EMbedded micro Architectures) lähtökohta sulautuksen ongelmiin on generoida kuvauksen perusteella sekä ohjelmiston koodi että laitteiston kuvaus. Tätä kutsutaan yleisesti *kosynteesiksi*.

Järjestelmä ohjelmoidaan prosessiprimitiiveillä laajennetulla C-kielellä esimerkkialustalle (SPARC 33MHz, liukulaskentayksikkö ja suunnittelijan lisäämät oheislaitteet) [4].

4.1.1 COSYMAN käyttö

Kun järjestelmä on kuvattu sille suoritetaan kontrolli- ja tietovuoanalyysi, sekä ajoitustiedon saamiseksi jokaiselle peruslohkolle simulointi, tai vaihtoehtoisesti symbolinen analyysi. Tämän jälkeen skeduloidaan järjestelmän prosessit. Skedulointi on mahdollista suorittaa kahdella eri tavalla: Scalable Performance Scheduling (SPS) etsii mahdollisen ja minimaalisen tavan suorittaa prosesseja sarjallisesti, jolloin prosessit voidaan käsitellä kuin kyseessä olisi vain yksi prosessi. Toinen vaihtoehto yhdistää skeduloinnin laitteiston ja ohjelmiston rajaukseen.

Korkean tason komponenttien synteesiin COSYMAssa käytetään Braunschweig Synthesis System:ä (BSS). BSS luo kaavion skedulointiaskeleista, funktioyksiköistä sekä muistin käytöstä jonka avulla järjestelmän pullonkaulat voidaan löytää. Laitteiston synteesiin käytetään Synopsys Design Compileria ja ohjelmiston synteesiin standardi C-kääntäjää.

Viimeiseksi suoritetaan laitteiston ja ohjelmiston rajaus jossa jokaiselle peruslohkolle valitaan, toteutetaanko se ohjelmistolla vai laitteistolla. COSYMA aloittaa konfiguraatiosta jossa kaikki on toteutettu ohjelmistolla ja optimoi konfiguraatiota niin että reaaliaikavaatimukset täyttyvät ja laitteistokustannukset minimoituvat. Ongelma on johdettavissa skedulointiongelmaan joka on tunnetusti NP-kova ja siksi optimointi on totetettu simuloidulla jäähdytyksellä. COSYMA tuottaa seuraavat tulokset:

- Prosessien skedulointi
- Gantt kaavio ja prosessikaavio
- Lista SW- ja HW-komponenteista
- Prosessien kommunikaatiokäskyt C-koodissa
- Schedulointikaavio jossa operaatiot ja muistiviittaukset joka kontrolliaskeleella

5. Asiantuntijajärjestelmät

Asiantuntijajärjestelmien idea on esittää kaikki ongelmaan liittyvä tieto yhdessä paikassa ja tarjota menetelmät, joilla relevantin tiedon saanti on helppoa.

5.1 CAISARTS

CAISARTS eli *Conceptual, Analytical and Implementation Scheduling Advice for Real-Time Systems* [6] on sääntöpohjainen asiantuntija-järjestelmä (*expert-system*). Tietämys eli

säännöt ja ongelma esitetään kuvauskielellä. CAISARTS käy tietämystietokantantansa läpi ja etsii tehtävälle ”ratkaisun” perusteluineen.

CAISARTS koostuu käyttöliittymästä ja päättelyohjelmasta. Kaikki käyttäjän kommunikointi tapahtuu graafisen käyttöliittymän kautta. Ohjelmalle annetaan rinnakkais- tai reaaliaikaongelma, joka yleensä koostuu useammasta prosessista, joilla on tiettyjä määräaikarajoitteita. Lisäksi CAISARTSille kerrotaan, millainen ympäristö on käytössä, minkä jälkeen CAISARTS etsii omasta ennakkoon ohjelmoidusta tietokannasta ongelmaa vastaavan tietämyksen ja tulostaa tämän käyttäjälle.

Ohjelman antama tuloste voi sisältää lähdekoodia, viittauksia standardeihin tai tunnettuihin periaatteisiin. Tämän lisäksi annetaan jonkinlainen kuva prosessien skeduloitavuudesta eli suoritetaan laskenta, jolla osoitetaan toteuttaako järjestelmä annetut aikarajoitteet.

Tietämys ja ongelma kuvataan syntaksiltaan LISPiä muistuttavalla CLIPS-kielellä. Esitystapa on hieman logiikkaohjelmointia muistuttava IF–THEN-rakenne: mikäli ehdot täyttyvät, säännön runkoa sovelletaan. Runko voi sisältää useita erilaisia toimintoja kuten neuvo, kommentti, perustelu tai lähdekoodi. Kun CAISARTS löytää säännön, jota voidaan soveltaa, se toteuttaa säännön rungossa määritellyt toiminnot. Esimerkiksi perustelu-kohta antaa käyttäjälle perustelun, miksi juuri tämä ratkaisu on järkevä.

Ohjelman mukana tulee laaja tietokanta erilaisista rinnakkais- ja reaaliaikaohjelmointiin liittyvästä tietämyksestä. Itse varsinainen ongelma ja tietämys sisällytetään säiliöihin (Container):

Task Group Container Sisältää kuvauksen skeduloitavista prosesseista

Time Constraint Container kuvaa prosessien aikarajoitteet

Hardware kuvaa raudan rajoitteet ja ominaisuudet

Sched Algorithm Container määrittelee käytettävät skedulointi-algoritmit

Tämän lisäksi CAISARTS sisältää muitakin säiliöitä. Erityisesti kannattanee mainita POSIX Implementation Approach Container, joka kuvaa tietämyksen siitä, miten POSIX-palveluja käytetään implementoimaan skedulointialgoritmeja. Käyttäjä määrittelee Task Group Container ja Time Constraint Containeriin oman ongelmansa taskit ja aikarajoitteet.

CAISARTS tukee sekä yleisiä että erityisiä sääntöjä eli ongelmaan voidaan antaa hyvin yksilöllisiä tai yleisiä neuvoja. CAISARTS osaa myös käsitellä ristiriitaisia sääntöjä[5]: mikäli ongelman ratkaisu johtaa ristiriitaisten sääntöjen soveltamiseen, annetaan käyttäjälle molemmat säännöt ja mahdollsuus mitä ristiriitaisista säännöistä sovelletaan. Käyttäjä voi lisätä itse melko vapaasti sääntöjä tietokantaan.

6. Yhteenveto

WCET-analyysissä etsii suoritetulle ohjelmalle karkean ylärajan. Mallintarkastustyökaluilla voidaan tutkia erilaisia rinnakkaisohjelmoinnin ongelmia. Asiantuntijajärjestelmät pyrkivät

[5] Ristiriitaiset näkemykset ovat paperin kirjoittajan mukaan erittäin yleisiä alan tutkijoiden keskuudessa.

tuomaan kaiken tarvittavan tiedon yhteen paikkaan ja tarjoamaan menetelmät relevantin tiedon helpolle löytämiselle.

Reaaliaika-, rinnakkaisuus- ja sulautustyökaluja on monenlaisia. Ongelmaan sopivan työkalun löytäminen saattaa olla hankalaa kuin myös itse työkalun käyttö. Ongelman formuloiminen työkalun vaatimalla tavalla voi olla työlästä ja erota paljonkin varsinaisen toteutuksen tavasta.

Viitteet

[1] Engblom J., Ermedahl A., Sjödin M., Gustavsson J., Hansson H. *Towards Industry Strength Worst-Case Execution Time Analysis*. ASTEC Technical Report Number 99/02
`http://www.docs.uu.se/astec/Reports/Reports/9902.pdf`

[2] Larsen K. G., Pettersson P., Yi W. *UPPAAL in a Nutshell*.
`http://www.docs.uu.se/docs/rtmv/papers/lpw-sttt97.ps.gz`

[3] Li Y.S., Malik S., Wolfe A. *Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches*. Proceedings of the IEEE Real-Time Systems Symposium, December, 1996
`ftp://ftp.ee.princeton.edu/pub/yauli/rtss96.ps.gz`

[4] Österling A., Benner Th., Ernst R., Herrmann D., Scholz Th., Ye W. *The COSYMA System*.
`http://www.ida.ing.tu-bs.de/research/publications/ps/OBE+97:HardwSoftwCoDesig.ps.gz`

[5] Alur R., Dill D., *Automata for Modelling Real-Time Systems*. Theoretical Computer Science, 126(2):183-236 April 1994.

[6] Humphrey M., Stankovic J. A. *CAISARTS: A Tool for Real-Time Scheduling Assistance*. Scheduling Assistance, Proceedings of the Second IEEE Real-Time Technology and Applications, Brookline, MA, June, 1996.

[7] Henzinger T. A., Ho Pei-Hsin, Wong-Toi H. HYTECH*: A Model Checker for Hybrid Systems*. Software Tools for Technology Transfer 1:110-122, 1997.
`http://www-cad.eecs.berkeley.edu/~tah/Publications/hytech.html`

[8] Hill M. D., *Evaluating Associativity in CPU Caches*. IEEE Transactions on Computers, Vol. 38 No. 12, December 1989.

[9] Holzmann, G. J., *The Model Checker* SPIN. IEEE Transactions on Software Engineering, Vol 23, No. 5, May 1997.

[10] Anon, *Bound-T - The best tool for the worst case*. Space Systems Finland advertisement, July 2002

Mikroprosessoreita ja -kontrollereita staattisen analyysin näkökulmasta

Antti-Pekka Liedes
Jussi Rautio

1. Johdanto

Tämä tutkielma on yleiskatsaus neljään mikroprosessoriin ja -kontrolleriin staattisen analyysin näkökulmasta. Tarkastelemme neljää erilaista prosessoria: Intel 8051, Atmel AT91-perhe, Motorola Coldfire ja MIPS R3000.

Toisessa luvussa kerrotaan yleisiä asioita staattisesta analyysista. Kolmannessa luvussa käydään läpi prosessorit, ja kerrotaan niiden staattiseen analyysiin vaikuttavista ominaisuuksista, kuten käskykannoista. Neljännessä luvussa pohditaan prosessorien staattista analysoituvuutta sekä esitetään niistä jo tehtyjen analyysejä, jos sellaisia on.

2. Yleistä staattisesta analyysista

Tässä tutkielmassa käydään läpi vain tärkeimmät staattiseen analyysiin liittyvät termit ja käsitteet. Tarkempi kuvaus staattisesta analyysistä yleisesti löytyy tutkielmasta [1].

2.1 Termistöä

Staattinen analyysi: käännösaikana tapahtuva analyysi.

Polkuanalyysi (path analysis): Polkuanalyysi valitsee kaikista mahdollisista suorituspoluista sen, jonka suorittaminen vie eniten aikaa.

Mikroarkkitehtuurin mallintaminen: minkä verran aikaa pisin polku vie? Mallinnettavia toimintoja ovat prosessorin välimuisti ja liukuhihnat.

Peruslohko (basic block): jokin lopullisen konekielelle käännetyn koodin osio, johon pääsee vain yhtä kautta ja josta poistutaan vain yhtä kautta.

Suoritusaika: Peruslohkon pahin mahdollinen suoritusaika $\times$ peruslohkon pahin mahdollinen suorituskertojen määrä.

3 Prosessorit

3.1 Intel 8051

Intel 8051 on yksi ensimmäisistä mikrokontrollereista. Sen valmistus on aloitettu vuonna 1977, eli vain muutama vuosi ensimmäisten mikroprosessorien jälkeen. Arkkitehtuuri on edelleenkin hyvin suosittu. Prosessorin hinta liikkuu muutamissa dollareissa (vuonna –97 $3.50) ja on arvioitu, että jopa joka toinen maailman mikrokontrollereista olisi 8051-pohjainen. Nykyään perus-8051:tä valmistaa Philips, ja siitä ovat tehneet klooniversionsa mm. Atmel, Dallas Semiconductor ja Siemens. 8051:lle on toteutettu useampia kääntäjiä, mm. `gcc`. Lisäksi sille löytyy ainakin BASIC- ja Forth-tulkit.

3.1.1 Käskykanta

8051 on käytännössä pinokone. “Yleiskäyttöisiä” rekisterejä on yksi, nimeltään akkumulaattori (`ACC`). Kaikki tärkeimmät operaatiot hoituvat akkumulaattorin ja jonkin toisen rekisterin välillä paitsi kerto- ja jakolaskut, jotka on pakko tehdä akkumulaattorin ja `B`-nimisen rekisterin välillä (jota ei sitten käytetä oikeastaan mihinkään muuhun). Suurin osa operaatioista jättää tuloksensa automaattisesti akkumulaattoriin. Operaation toinen osapuoli voi olla paljas luku, rekisteri, muistiosoite tai epäsuorasti viitattu muistiosoite.

8051:ta on kehuttu erityisesti sen yksinkertaisesta keskeytysmallista. Keskeytyksiä on 5–6, ja niiden käsittelijät ladataan erilliseen nopeaan muistiin.

3.1.2 Muisti

8051:n muistimalli on monimutkainen. Prosessorissa on kaksi 256 tavun kokoista nopeaa muistia, joista toiseen voi viitata vain epäsuorasti (käytännössä varattu pinolle). Varsinainen koodi pidetään erillisellä EPROMilla, jonka koko on 64 kilotavua, ja sen käyttämä muisti erillisellä RAMilla, jonka koko on myös 64 kB. Yhteensä muistia on siis hieman yli 128 kB. Eri muistiviittaukset ovat vaativuudeltaan erilaisia ja vievät kukin eri määrän syklejä.

3.1.3 Reaaliaikaominaisuuksia

8051:ssä ei ole liukuhihnaa eikä välimuistia, mitkä ominaisuudet tekevät siitä periaatteessa helposti analysoitavan. Eri käskyjen erilaiset koot ja muistihakujen erilaiset nopeudet tekevät analyysista vaikeampaa mutta tuskin mahdotonta.

3.1.4 Kloonit

8051:n kloonit eroavat toisistaan monessa suhteessa. Uudemmissa versioissa on mm. tukea 16- ja jopa 32-bittisille kokonaisluvuille sekä liukuluvuille, nopeampia kellotaajuuksia, alhaisempaa virrankulutusta ja kaikenlaista eksoottista tavaraa, kuten sisäänrakennettuja BASIC-tulkkeja. Kalleimmista ja eksoottisimmista versioista saakin sitten maksaa viisikymmentä dollaria kappaleelta.

3.1.5 Käyttökohteet

8051 on perus-“pesukoneprosessori”. Verkosta (mm. [3]) löytyy vapaata koodia mm. infrapunakaukosäätimiin, termostaatteihin, herätyskelloihin ja valomainoksiin.

3.2 Atmel AT91–perhe

Atmelin AT91-perhe perustuu pääosin ARM7TDMI ARM Thumb prosessoriytimeen, jonka ympärille on rakennettu erilaisia mikrokontrollereille tyypillisiä palveluita, kuten ohjelmoitavat I/O-linjat, ajastimet/laskimet ja USARTit. Atmelin mukaan AT91 tarjoaa eniten MIPSejä per watti. AT91-perhe toimii 1–33MHz kellotaajuudella.

3.2.1 Käskykanta

ARM7TDMI tukee kahta käskykantaa: normaali ARM-kanta ja 16-bittinen THUMB-kanta, joka on tarkoitettu pieniin järjestelmiin. Molemmat kannat ovat ovat RISC:mäisiä: käytössä on kolmisenkymmentä rekisteriä, käskyt ovat vakiomittaisia ja LOAD/STORE-muistiosoitus.

THUMBilla koodatut ohjelmat ovat parhaimmillaan kooltaan 65% ja suorityskyvyltään 160% vastaavista ARM-käskykannalla koodatuista ohjelmista 16-bittisessä ympäristössä. Toiminta käskykantaa välillä voidaan vaihtaa ajoaikaisesti.

3.2.2 Muisti

Atmelin AT91-kontrollereissa on piirillä jonkin verran SRAMia, esim. 8kB. Tämä muisti on käytettävissä välittömästi yhdellä kellojaksolla ilman odotusta.

Ulkoista muistia voidaan käyttää EBI (External Bus Interface) -liitynnän kautta. EBI:n taakse voidaan kytkeä esim. FLASH:ia ohjelmakoodia varten. Myös EBI kykenee yhden kellojakson toimintoihin.

3.2.3 Käskysuoritus ja liukuhihna

ARM7TDMI suorittaa käskyt yhdessä tai useammassa syklissä, käskystä riippuen. Käskyjen tarvitsemat syklit on myös jaettu neljään eri ryhmään sen perusteella, mitä ulkoisia liityntöjä ne mahdollisesti tarvitsevat: non-sequential cycle (load/store ei riipu edeltävän syklin osoitteesta), sequential cycle (loadin/storen osoite on joko edellisellä syklillä käytetty tai siitä +2 tai +4 tavua. Lisäksi on täysin sisäinen internal cycle ja coprocessor register transfer, joka riippuu apuprosessorista.

ARM7TDMI lataa käskyt suoritusyksikölle liukuhihnaa pitkin. Liukuhihna joudutaan tyhjentämään, jos käskysuorituksessa tapahtuu ehdollinen hyppy.

3.2.4 Reaaliaikaominaisuuksia

ARM7TDMI:ssä on erityinen FIQ (Fast Interrupt Request) -keskeytys, jolla on käytössä riittävä määrä paikallisia rekistereitä, jotta keskeytyksessä ei tarvitse tehdä rekisterien tallen-

nusta tai latausta ollenkaan. Näin saadaan erityisen nopea keskeytys, FIQ:n maksimilatenssi on 28 sykliä.

3.2.5 Käyttökohteet

Atmelin mukaan AT91 soveltuu erityisesti reaaliaikasovelluksiin, ja sen käyttökohteita ovat mm. teollisuusautomaatio, mp3/wma-soittimet, datankeräyslaitteet, terveydenhuolto sekä GPS ja verkkolaitteet.

Atmel ei kerro AT91:n hintaa suoraan, mutta se lienee samaa luokkaa Motorola Coldfire 3:n kanssa tai hieman halvempi, eli n. 10–15 dollaria kappaleelta.

3.3 Motorola Coldfire

Motorola Coldfire -perhe on jatkoa suositulle MC68k-perheelle. Coldfirestä on neljä eri versiota, joista tarkastellaan lähinnä versioita 3 (5307) ja 4 (5407).

3.3.1 Coldfire 3

Coldfire 3:a on saatavilla 66 ja 90MHz versioina. Prosessorissa on joitakin mikrokontrollerimaisia piirteitä, kuten kaksi UART:ia ja yleiskäyttöinen 16-bittinen I/O-portti, mutta oleellisesti kyseessä on jo enemmänkin “oikea” prosessori kuin mikrokontrolleri.

3.3.2 Käskykanta

Coldfiren käskykanta on oleellisesti MC68k-kanta muutamin yksinkertaistuksin. Suurimpana erona monimutkaisista osoitusmuodoista on luovuttu, ja kaikki osoitukset käsittelevät vain yhtä osoitetta per käsky. Coldfire on Motorolan mukaan RISC, mutta käskykannassa on myös CISC:mäisiä piirteitä. Käskyt eivät ole vakiomittaisia, vaan käskystä riippuen joko 16-, 32-, tai 48-bittisiä. Rekistereitä on käytettävissä kahdeksan data- ja kahdeksan osoiterekisteriä.

Vanhoja MC68k-ohjelmia voidaan suorittaa Coldfirellä joko kääntämällä koodi lennosta tai etukäteen. Työkalut tähän ovat vapaasti saatavilla.

3.3.3 Muisti

Coldfiressä on 4kB SRAM itse piirillä. Lisäksi ulkoista muistia varten on DRAM-kontrollerit SDRAM:lle ja EDO:lle. Ulkoisen muistin kanssa käytössä on 8kB jaettu välimuisti, joka käyttää pseudo round-robin korvausta.

3.3.4 Käskysuoritus ja liukuhihna

Coldfiressä on nelivaiheinen käskynlautausliukuhihna, joka syöttää käskypuskurin läpi kaksivaiheista käskynsuoritusliukuhihnaa. Liukuhihnassa voi tapahtua tyhjentäminen ainakin ehdollisen hypyn seurauksena.

Coldfiren käskynlautasliukuhihna käyttää yksinkertaista staattista hyppyennustusta: ehdollisia hyppyjä eteenpäin ei oteta, taaksepäin otetaan.

Käskynsuoritusliukuhihna suorittaa yksinkertaiset rekisteristä-rekisteriin-käskyt yhdellä läpiviennillä eli yhdellä kellojaksolla. Muistista-rekisteriin-käskyt joudutaan viemään liukuhihnan läpi kahdesti, joten niiden suorittamiseen tarvitaan aina kaksi sykliä liukuhihnan puolesta. Rekisteristä-muistiin-käskyen suoritusvaiheita voidaan rinnakkaistaa niin, että ne saadaan liukuhihnasta läpi kerralla ja siis yhdessä syklissä.

3.3.5 Coldfire 4

Coldfire 4 on Coldfire 3:a uudempi ja huomattavasti tehokkaampi versio. Coldfire 4 toimii jopa 220MHz kellotaajuudella. Kellotaajuutta on pystytty lisäämään pidentämällä käskynsuoritusliukuhihna viisivaiheiseksi. Lisäksi Coldfire 4 käyttää Harvard-muistiarkkitehtuuria, eli käsky- ja datamuisti on erotettu toisistaan. Käskymuistille on 16kB ja datamuistille 8kB välimuisti. Coldfire 4 on myös rajoitetusti superskalaari ja pystyy osittain suorittamaan enemmän kuin yhden käskyn syklissä.

3.3.6 Käyttökohteet

Coldfireä käytetään vaativissa sovelluskohteissa, mm. AIRBUSin lentokoneissa. Lisäksi Motorolan mukaan Coldfireä käytetään erilaisissa kannettavissa laitteissa ja verkkolaitteissa. Coldfiren eräs versio, MCF5282, on maailman ensimmäinen sulautetulla ethernet-kontrollerilla varustettu mikrokontrolleri.

Yli tuhannen kappaleen erissä Coldfire 3 maksaa 13–17 dollaria kappaleelta ja Coldfire 4 22–27 dollaria kappaleelta.

3.4 MIPS R3000

MIPS R3000 on kehitetty alunperin vuonna 1988 tavalliseksi RISC-mikroprosessoriksi, joka toimi 20 MHz:n kellotaajuudella. Nykyinen lähes samaa käskykantaa käyttävä mikrokontrolleri MIPS32 4Kc toimii 200 MHz:n kellotaajuudella. Myös MIPS-arkkitehtuuri on levinnyt oman yhtiönsä ulkopuolelle: esimerkiksi AMD tekee nykyään myös MIPS-klooneja.

3.4.1 Käskykanta

R3000 on 32-bittinen RISC-prosessori. Yleiskäyttöisiä rekisterejä on 32, joista osa on varattu erityistarkoituksiin. Operaatioita voidaan tehdä vapaasti rekisterien välillä. Muistihaku ja -sijoitus ovat erillisiä operaatioita. Kukin käsky vie saman verran tilaa ja aikaa. “Hallintalogiikan vähentämiseksi ja nopeuden lisäämiseksi käskykannassa on ainoastaan 73 erilaista käskyä” [4].

3.4.2 Muisti

R3000:n muistinhallinnasta vastaa prosessoriin integroitu 64:n osoittimen TLB. Toisin kuin myöhemmissä MIPS-prosessoreissa, alkuperäisessä R3000:ssa ei ole sisäänrakennettua välimuistia. Osoitusmalli sallii 30-bittiset osoitteet, eli enimmillään 4 GB ulkoista muistia. Koska R3000:ta ei ole suunniteltu reaaliaikaprosessoriksi, ongelmaksi nousee välttämättä joko prosessorin blokkautuminen jokaisen hitaan muistihaun yhteydessä (jos ei käytetä ulkoista välimuistia) tai arvaamaton käyttäytyminen ja mahdolliset pitkään kestävät välimuistin päivitykset (jos ulkoista välimuistia käytetään).

3.4.3 Liukuhihna

R3000:ssa on viisiportainen liukuhihna. Kääntäjät tukevat yleisesti mm. *branch-delay* -vaatimusta, joka estää liukuhihnan jumiutumisen ehdollisen hyppykäskyn yhteydessä. Dataongelmia (*data hazards*) pyritään välttämään viivyttämällä tarvittaessa käskyjen suoritusta ajon aikana.

3.4.4 Käyttökohteet

MIPSin sulautettuja prosessoreja löytyy mm. kämmenmikroista, pelikonsoleista (Nintendo 64 ja PlayStation), tulostimista, WebTV-suorittimista ja satelliittitelevisioista. Vuodelta -98 peräisin oleva MIPSin oma esite kehuu heidän omaa prosessoriaan markkinajohtajaksi “tehokkaiden sulautettujen RISC-prosessorien” alalla. Nykyään tilanne ei ole enää niin selkeä, mutta MIPS pysyy edelleen varteenotettavana vaihtoehtona myös sulautettujen prosessorien puolella.

4 Loppupäätelmiä

4.1 Prosessorien mallintaminen

Prosessorien mallintaminen staattista analyysia varten jakautuu lähinnä kahteen osaan: muistiarkkitehtuurin (lähinnä välimuistin) ja liukuhihnan mallintamiseen. Nämä molemmat lisäävät prosessorien keskimääräistä nopeutta, mutta tekevät prosessoreista myös vaikeammin ennustettavia. Pessimistinen “pahin tapaus aina” -analyysi johtaa jopa monikymmenkertaisesti (välimuistin tapauksessa n. 30-kertaisesti) liian suureen ohjelman maksimiin suoritusaikaan.

Prosessorien mallintamista on tehty lähinnä eurooppalaisessa DAEDALUS-projektissa [5]. Projektissa on mukana useita teollisia ja akateemisia kumppaneita, kuten AIRBUS France, AbsInt GmbH ja Saarlandin yliopiston Compiler Design Laboratory. DAEDALUS-projektin paperit eivät valitettavasti ole saatavilla suoraan Internetistä, eikä niitä voida analysoida tarkemmin tässä katsauksessa.

4.2 Muistiarkkitehtuuri

Muistiarkkitehtuurin kannalta AT91 ja Coldfire ovat helppoja niin kauan, kun pysytään prosessorin sisäisellä SRAM:lla. Sisäisen muistin käyttäminen onnistuu aina yhdellä kellojaksolla, eli se on “ilmaista”. 8051 on periaatteessa myös helppo, kunhan ottaa huomioon erityyppisten muistihakujen erilaiset (vakio-) kustannukset. R3000:ssa taas ei ole rekisterien lisäksi lainkaan sisäistä muistia, joten sen mallintaminen on täysin ulkoisen muistin mallintamista.

Jos ja kun joudutaan menemään prosessorin ulkoiseen muistiin, tilanne hankaloituu huomattavasti. AT91 ja R3000 tarjoavat ulkoisen muistin käyttöön parhaimmillaan yhden syklin pääsyn, mutta lopullinen nopeus riippuu luonnollisesti käytetystä ulkoisesta muistista ja siihen mahdollisesti liitetystä välimuistista. Koska prosessorit eivät itse tarjoa välimuistia ulkoiselle muistille, ei sille löydy “standardeja” tuloksia.

Coldfire 3:n välimuistia on mallinnettu onnistuneesti DAEDALUS-projektissa [6]. Coldfiren välimuistin hankaluuksina ovat mm. seuraavat piireet:

- Jaettu välimuisti: ohjelmakoodi häiritsee dataa ja päinvastoin.
- Pahin tapaus ei aina ole cache miss.
- Välimuisti käyttää hankalaa round robin korvausalgoritmia; globaalia round robin -laskurin arvo voi muuttua keskeytyskäsittelijöissä.

Coldfire 4 voi tässä suhteessa olla jopa hieman yksinkertaisempi, koska siinä on Harvard-muistiarkkitehtuuri ja siis erotettu käsky- ja datamuisti. Coldfire 4:n välimuistista ei löytynyt valmiita analyyseja.

4.3 Liukuhihna

Liukuhihnan analysointia hankaloittaa muun muassa se, että keskeytyksiä voi reaaliaikajärjestelmässäkin periaatteessa tulla missä vaiheessa tahansa. Keskeytys tyhjentää liukuhihnan.

8051:ssa ei ole liukuhihnaa, joten sen analysoiminen tässä suhteessa on huomattavan helppoa.

AT91:n liukuhihna on yksinkertainen, mutta voi kuitenkin tyhjentyä ainakin ehdollisten hyppäysten johdosta. Muista tyhjenemistilanteita ei ole manuaalissa ainakaan suoraan selitetty. Myöskään Coldfire 3:n liukuhihnan tyhjentymistilanteita ei ole selitetty manuaalissa.

Sekä AT91:n että Coldfire 3:n liukuhihna on mallinnettu DAEDALUS-projektissa ja niille on tehty abstraktiin tulkintaan perustuva analyysityökalu. Jopa Coldfire 3:n liukuhihnaa pidettiin analyysin kannalta monimutkaisena, vaikka se onkin vain 4 + 2 (käskylataus + käskysuoritus) vaihetta FIFO-puskurilla erotettuina. Työkalu yhdistää välimuisti- ja liukuhihna-analyysin.

Myös Uppsalan yliopistossa on tehty liukuhihnojen analyysiin liittyvää tutkimusta. Jakob Engblom on tutkinut liukuhihnoja yleisellä tasolla [7], ja tulokset ovat sovellettavissa niin ARM7:aan ja Coldfireen kuin moneen muuhunkin prosessoriin.

Liukuhihnansa puolesta Coldfire 4 on vielä huomattavasti 3:a hankalampi analysoitava, koska käskysuoritus on pidennetty viisivaiheiseksi. Coldfire 4:n liukuhihnasta ei löytynyt valmiita analyyseja.

R3000:n liukuhihnaa on tutkittu ja analysoitu hyvin laajalti. Sen tutkimiseen on olemassa useita valmiita simulaattoreita. Liukuhihna on vanha ja periaatteiltaan yksinkertainen ja sellaisena kohtuullisen helposti analysoitavissa.

Viitteet

[1] Juha Tukkinen, Mikko Reinikainen. Ohjelmistojen välimuistin suorituskyvyn arviointi staattisilla analyysimenetelmillä. Tämä julkaisu.

[2] 8051 microcontroller FAQ. Internet FAQ Consortium.
URL:<http://isc.faqs.org/faqs/microcontroller-faq/8051/>

[3] 8051 Microcontroller. Reynolds Electronics.
URL:<http://www.rentron.com/8051.htm>

[4] MIPS Architecture. EDN Access.
URL:<http://www.e-insite.net/ednmag/archives/1998/092498/32_r3000.htm>

[5] Cousot. DAEDALUS, Validation of critical software by static analysis and abstract testing.
URL:<http://www.di.ens.fr/čousot/projects/DAEDALUS/>

[6] Wilhelm. Worst Case Execution Time Prediction. Angewandte Informatik GmbH.
URL:<http://www.di.ens.fr/čousot/projects/DAEDALUS/public/seminar/industrial_day_absint_usaar.pdf>

[7] Engblom: Processor Pipelines and Static Worst-Case Execution Time Analysis, PhD Thesis, University of Uppsala, 2002.
URL:<http://user.it.uu.se/j̃akob/publications/engblom-emsoft-2002.pdf>

Ohjelmistojen välimuistin suorituskyvyn arviointi staattisilla analyysimenetelmillä

Mikko Reinikainen
Juha Tukkinen

1. Johdanto

Tässä katsauksessa tutustutaan ohjelmistojen välimuistin suorituskyvyn arviointiin staattisilla analyysimenetelmillä. Katsauksessa on ensin lyhyt selostus välimuisteista (kohta 2). Tämän jälkeen tarkastellaan ohjelmistojen staattisia analyysimenetelmiä (kohta 3) ja välimuistin analysointia niiden avulla. Tarkasteltuja analyysimenetelmiä ovat perinteinen tietovuoanalyysi, abstrakti tulkinta ja Integer Linear Programming. Koska aiheet ovat paljon syventävää tietoa sisältäviä, niin käytetyistä lähteistä on laadittu lyhyet lukuoppaiksi soveltuvat yhteenvedot (kohta 4).

2. Välimuistit

Välimuisti on keskusmuistia selvästi pienempi, nopea muisti, johon prosessorin viimeksi käyttämät data-alkiot talletetaan. Välimuisti tehostaa ohjelman toimintaa ajallisen ja avaruudellisen paikallisuuden avulla [2]. Välimuistien tutkimus ja kehittäminen muodostavat kokonaisen tutkimusalueen.

Tyypillisesti ohjelmassa viitataan toistuvasti samoihin data-alkioihin. Tämä on *ajallista paikallisuutta*. Usein viitatut data-alkiot pyritään pitämään välimuistissa. *Avaruudellinen paikallisuus* tarkoittaa sitä, että ohjelmissa muistiviittaukset kohdistuvat usein edellisiä viittauksia lähellä olevaan muistipaikkaan. Tästä syystä välimuistiin ladataan dataa keskusmuistista kerralla aina kokonaisen välimuistilohkon verran ja toivotaan, että jotkin seuraavista muistiviittauksista kohdistuisivat lohkon sisältämiin data-alkioihin.

Välimuistin *assosiatiivisuus* määrää, mihin kohtaan välimuistia tietty keskusmuistin lohko voidaan ladata. Esimerkiksi 4-assosiatiivisessa välimuistissa tietty lohko voidaan jakaa neljään eri paikkaan.

Välimuistin käytön tehokkuuta rajoittavia tekijöitä ovat välimuistin koko sekä assosiatiivisuus. Koko määrää sen, kuinka monta lohkoa välimuistiin mahtuu kerralla ja assosiatiivisuus sen, kuinka helposti välimuistiin ladattu lohko osuu jonkin aiemmin ladatun lohkon päälle. Lisäksi välimuistin suorituskykyyn vaikuttavat lohkon koko, prosessorille ja muistille menevien väylien nopeus sekä yhden välimuistilohkon lataamiseen kuluva aika.

Perinteisesti erilaisten välimuistien suorituskykyä on mitattu simuloimalla todellisen ohjelman toimintaa ohjelmasta talletetun muistijäljen avulla. Muistijälki sisältää kaikki ohjelman suorituskyvyn kannalta olennaiset muistioperaatiot jossakin muodossa.

Välimuistin suorituskykyä on mahdollista arvioida myös staattisen analyysin menetelmillä.

3 Staattinen analyysi

Ohjelmien staattinen analyysi tarkoittaa ohjelmaa suorittamatta tapahtuvaa ohjelman rakenteen ja toiminnan analysointia, joka voi tapahtua esimerkiksi käännösaikana. Sen avulla voidaan arvioida ohjelman dynaamisia ominaisuuksia kuten suorituskykyä: huonoimman tapauksen suoritusaikaa (WCET), muistiviittausten määrää ja välimuistin ohi tapahtuvien muistiviittausten määrää tai niiden osuutta kaikista muistiviittauksista.

Täydellistä tietoa ohjelman toiminnasta ei ole mahdollista saada ajamatta sitä kaikilla mahdollisilla syötteillä. Esimerkiksi ohjelman päättymisongelma on ratkeamaton. Siksi staattinen analyysi pyrkii tuottamaan ohjelman suorituskyvystä sellaisen approksimaation, joka on mahdollista laskea järjellisessä ajassa. Jotkut staattisessa analyysissä käytettävät menetelmät tarvitsevat syötettä käyttäjältä, jotta ne pystyvät tuottamaan mielekkäitä arvioita.

3.1 Tietovuoanalyysi

Ohjelman tietovuoanalyysi tarkoittaa yleensä analyysiä siitä, mikä ohjelman suorituksen tila voi olla missäkin kohdassa ohjelmakoodia. Ohjelman tilaa ilmaistaan erilaisilla joukoilla, joita ovat esimerkiksi kääntäjätekniikassa usein käytetyt yltävät määrittelyt ja saatavissa olevat lausekkeet.

Ohjelman tilaa esittäviä joukkoja lasketaan tietovuoyhtälöiden avulla. Yhtälöt määrittelevät, miten jokin ohjelman käsky muuttaa ohjelman tilaa.

Analyysiä varten varten ohjelma jaetaan peruslohkoihin, jotka muodostavat ohjelman kontrollivuokaavion. Yhden peruslohkon sisällä ohjelman suoritus etenee lineaarisesti ja peruslohkon jälkeen hyppää jonkin peruslohkon alkuun. Ohjelman mahdolliset suoritukset ovat polkuja kontrollivuokaaviossa.

Ohjelman muistiviittausten konkreettinen osoite on mahdollista tuntea jo käännösaikana. Tähän perustuen on tehty työkaluja, jotka laskevat ohjelman kontrollivuokaavion, sijoituslauseiden ja tietorakenteiden perusteella tietoa muistiviittausten välimuistikäyttäytymisestä [9].

3.2 Abstrakti tulkinta

Abstrakti tulkinta eroaa perinteisestä tietovuoanalyysistä siten, että siinä ei tutkita ohjelman konkreettisia tiloja vaan abstrakteja tiloja. Yksi abstrakti tila kuvaa yleensä joukon konkreettisia tiloja. Perinteinen tietovuoanalyysi toimii usein myös samalla tavalla, mutta konkreettisten tilajoukkojen esitystä ei ole vastaavalla tavalla formalisoitu kuin abstraktissa tulkinnassa. Abstrakteilla tiloilla laskettaessa on mahdollista saada äärellisessä ajassa tuloksia, jotka kertovat ohjelman toiminnasta kaikilla mahdollisilla syötteillä.

Abstraktia tulkintaa käytetään laajalti esimerkiksi optimoivien kääntäjien tietovuoanalyysissä. Abstraktilla tulkinnalla saadut tulokset ovat myös aina turvallisia, koska abstraktin tulkinnan periaate takaa oikeellisen kuvautuvuuden konkreettisiin tiloihin.

Analyysin turvallisuus tarkoittaa sitä, että tulokset voivat olla epätarkkoja, mutta niihin voidaan aina luottaa. Esimerkiksi jos boolen muuttuja on joskus tosi, niin sen arvo on turvallisesti määritelty sanomalla “en tiedä”, mutta ei “epätosi”.

Täysin assosioituvien välimuistien semantiikka voidaan määritellä seuraavasti [4]: Välimuisti on joukko välimuistirivejä $L = \{l_1, ...l_n\}$ ja päämuisti on joukko muistilohkoja $S = \{s_1, ..., s_m\}$. Jotta voisimme kertoa, että välimuistirivi ei ole yhtään muistilohkoa tarvitsemme tähän uuden elementin I, joten välimuistin määritellään sisältävän elementtejä $S' = S \cup \{I\}$.

Välimuistin (konkreettinen) tila on funktio $c : L \rightarrow S'$. Välimuistin toiminta periaate pitää myös kuvata. Esimerkiksi välimuistin LRU-pohjainen toiminta voidaan kuvata siten, että välimuistirivin indeksi x välimuistijoukon esityksessä $c(l_x) = s_y$ kertoo muistilohkon suhteellisen iän LRU:n mukaan (ei fyysistä sijaintia). Kaikkien välimuistitilojen joukko on kaikkien funktioiden c joukko, jota merkitään C_c.

Jos välimuistin tila tunnetaan ennen jonkin käskyn suoritusta, niin tila käskyn jälkeen voidaan laskea välimuistin päivitysfunktiolla. Välimuistin päivitysfunktio on muotoa $U : C_c \times S \rightarrow C_c$. Välimuistin päivitys toimii siten, että jos viitattu muistilohko s_x on jo välimuistissa, niin kaikkien samassa välimuistijoukossa olevien lohkojen, joita on käytetty s_x:n edellisen käytön jälkeen, ikää lisätään yhdellä. Jos s_x ei ole jo välimuistissa, niin kaikkien muiden muistilohkojen ikää lisätään ja vanhin poistetaan välimuistista. Uusimmaksi kummassakin tapauksessa tulee tietenkin lohko s_x.

Ohjelmat esitetään kontrollivuoverkkoina, joiden solmut kuvaavat peruslohkoja. Välimuistin toiminta voidaan kuvata peräkkäisissä muistiviittauksissa ketjuttamalla päivitysfunktio U:ta sisäkkäin. Alkutilassa c_1 välimuistin kaikki rivit ovat I ja käyttämällä tähän U:ta kontrollivuopolun mukaan saamme välimuistin tilan.

Jos välimuistin abstrakti tila tunnetaan ennen jotain käskyä ohjelmakoodissa, niin asbtrakti tila käskyn jälkeen voidaan laskea välimuistin päivitysfunktiolla. Abstrakti välimuistin tila $\hat{c} : L \rightarrow 2^S$ yhdistää välimuistirivit muistilohkoihin. Kaikkien abstraktien välimuistitilojen joukkoa merkitään $\hat{C}$. Must-analyysi määrittää muistilohkojen joukon, jotka ovat välimuistissa kaikissa tilanteissa, vastaavasti May-analyysi kaikki muistilohkot jotka saattavat olla välimuistissa. Persistence-analyysillä saadaan selville muistilohkot, joita ei ikinä poisteta välimuistista sinne lataamisen jälkeen. Abstrakti päivitysfunktio $\hat{U}$ on U:n vastine abstrakteille tiloille. Jos kontrollivuosolmussa on vähintään kaksi edeltäjää, yhdistefunktiota käytetään yhdistämään abstraktit välimuistitilat. Silmukoiden ja rekursiivisten funktioiden analyysi on kiinnostava, koska ohjelmat viettävät suuren osan suoritusajastaan niissä. Silmukat muunnetaan käsiteltäviksi kuten proseduurit. Silmukoiden ja rekursiivisten kutsujen ensimmäinen suoritus erotetaan lopuista (unroll).

3.3 Integer Linear Programming

Ohjelmien suoritusajan analyysissä täytyy ohjelman osien aiheuttamien välimuistitilojen lisäksi selvittää niiden esiintyminen, eli selvittää ohjelman mahdolliset suorituspolut. Käyttämällä Integer Linear Programming -menetelmää (ILP) ohjelman rakenteeseen liittyvät suorituspolut voidaan kuvata implisiittisesti.

ILP:ssä joukko rajoitteita kuvaa ohjelman kokonaisrakenteen ja näiden rajoitteiden ratkaiseminen antaa tuloksia ohjelman suorituspoluista. On huomattava, että jos mallinnetaan monimutkaisia komponentteja kuten välimuisteja tai liukuhihnoja, yhtälöiden ratkaiseminen on hyvin vaikeaa. Theiling ja Ferdinand [4] ovat käyttäneet abstraktia tulkintaa mallintamaan mikroarkkitehtuurin käyttäytymisen ja ILP:ia löytämään huonoimman tapauksen ohjelmapolut.

Ongelma, joka on muotoiltu ILP:lla koostuu arvotusfunktiosta ja sen muuttujien rajoitteista. Arvotusfunktio kuvaa ääritapaukseen (worst case) käytettyjä CPU-kellojaksoja. Jokainen muuttuja arvotusfunktiossa kuvaa yhden peruslohkon suoritusmäärän ja sitä painotetaan sen lohkon suoritusajalla. Lisäksi käytetään muuttujia, jotka vastaavat kontrollivuoverkon polkujen kulkumääriä. Pääperiaate on, että peruslohkojen suoritusmäärä on sama, kuin sen sisään tulevien ja ulos menevien polkujen summa. ILP-menetelmä ottaa kantaa vain peruslohkojen suorituskertoihin, se ei sisällä tietoa siitä, missä järjestyksessä lohkot suoritetaan. Se siis hukkaa tiedon ohjelman tilasta ja siksi sitä ei yksinään voi käyttää välimuistianalyysin menetelmänä.

ILP-yhtälöryhmien ratkaisuja voidaan laskea automaattisesti, mutta käyttäjän syötettä tarvitaan esim. löytämään hitain ohjelmapolku sekä silmukoiden ja rekursioiden rajat. Tämä on myös ILP-menetelmän etu, sillä käyttäjä voi kuvata tarkoin termein lähes kaiken mitä hän tietää ohjelmasta. Yhtälöryhmien ratkaiseminen on NP-täydellinen ongelma, joten monimutkaisten ohjelmien analysointi tällä menetelmällä ei ole tehokasta. Yksi heuristiikka yhtälöryhmien ratkaisemiseen on branch-and-bound.

Theiling ja Ferdinand saivat tällä menetelmällä hyviä WCET-arvioita käyttämällä ideaalista virtuaalista laitteistoa, jossa oli yhden kilotavun kokoinen nelitasoinen käskyvälimuisti, jonka rivikoko oli 16 tavua. Yksinkertaistetusti välimuistiosuma vei yhden kellojakson ja ohiviittaus kymmenen kellojaksoa. He analysoivat noin 9500 riviä C++ -koodia, jossa oli Lispin kaltaisella kielellä määritelty silmukoiden ja rekursioiden rajat kuten muita lisärajoitteita. Analysoija teki analyysin suoraan binääristä, testialgoritmeja oli kertoman laskemisesta aina JPEG:in diskreetistä kosinimuunnoksesta DES-salaukseen. Mielenkiintoinen huomio oli, että ennustus ei aina huonone käytettäessä kääntäjän optimointeja, vaikka tällöin käytettävät rajoitteet ovat varmasti liian pessimistisiä. Kuitenkin optimointi kutistaa koodin määrää ja välimuistianalyysi paranee.

4 Kirjallisuutta

Seuraavassa on lyhyt yhteenveto tärkeimmistä aiheeseen liittyvistä julkaisuista, joihin tutustuimme selvityksen aikana. Yhteenvetoja voi käyttää lukuoppaina, jos aiheesta haluaa opiskella lisää.

4.1 Christian Ferdinandin väitöskirja

Christian Ferdinandin väitöskirja [3] käsittelee yleisesti välimuistianalyysiä. Yleinen teoria on kappaleissa 1–4. Kappaleessa 1 (johdanto) puhutaan yleisesti abstraktista tulkinnasta ja siitä, miksi sen avulla lasketaan WCET-arvoja. Kerrotaan myös reaaliaikajärjestelmistä ja kahdesta tavasta ennustaa niiden suorituskykyä: arkkitehtuurimallintaminen ja ohjelman polkuanalyysiä. Välimuistit vaikeuttavat ennustamista. Esitellään must- ja may-joukot.

Kappaleessa 2 (Abstract Interpretation) kerrotaan abstraktista tulkinnasta. Luku 2.1 ja siitä eteenpäin on melko matemaattista notaation kuvausta.

Kappaleessa 3 (välimuistit) kerrotaan välimuistien toiminnasta. Esitetään Hennessyn ja Pattersonin nyrkkisääntö: ohjelmat käyttävät 90% suoritusajastaan 10%:iin koodia. Kerrotaan virtuaalivälimuisteista ja siitä miten ne vaikeuttavat reaalivälimuistien toimintaa ja miten ongelmat voidaan välttää.

Kappaleessa 4 käsitellään välimuistien semantiikkaa. Kappaleessa 5 on aiheena silmukoiden ja rekursiivisten proseduurien analyysi. Se sisältää muun muassa must- ja may-analyysin. Kappaleessa 6 tutkitaan välimuistin ohiviittauksien ylärajoja. Seitsemäs kappale keskittyy data- ja yhdistettyyn välimuistiin. Kappale 8 on liukuhihnojen semantiikkaa. Väitöskirjan lopussa eli kappaleissa 9–11 käsitellään käytännön kokeita ja aiheeseen liittyviä muita töitä sekä tehdään yhteenveto.

4.2 Nielsonin, Nielsonin ja Hankinin oppikirja

Kirja [5] on perusteos ohjelma-analyysistä. Sen mukaan ohjelma-analyysin sovelluksia ovat redundantin laskennan välttäminen (esim. laskenta tehdään useita kertoja), tarpeettoman laskennan välttäminen ((laskennan arvoja ei tarvita, tai ne tiedetään jo käännösaikana), ohjelmiston validointi ja formaali välimuistimallinnus.

Perusasiat ovat luvuissa 1 ja 2. Luvussa 1 (johdanto) käsitellään yltäviä määrittelyjä (mihin ohjelman kohtiin jokin tietty sijoituslause voi yltää), ekvationaalisen ja rajoitteisiin perustuvan lähestymistavan eroja sekä abstraktia suorittamista. Luvussa käsitellään myös muita menetelmiä, jotka on jätetty tämän julkaisun ulkopuolelle.

Luvussa 2 käsitellään tarkemmin tietovuoanalyysiä, ja esitetään menetelmät saatavilla olevien lausekkeiden (available expressions), yltävien määrittelyjen (reaching definitions), erittäin ruuhkaisten lausekkeiden (very busy expressions) ja elävien muuttujien (live variables) laskemiseen.

4.3 Sebekin raportti

Teknisessä raportissaan[6] Sebek käsittelee aihetta kattavasti. Luvussa yksi on hyvä johdanto välimuisteihin ja niiden kehitykseen, tässä on paljon uudehkoakin asiaa, kuten Pentium 4-prosessorin Trace-cache. Myös mm. ohjelmistolla ja raudalla toteutettua prefetch:iä on käsitelty. Tarkemmin on käsitelty käytännön prosessoreista Pentium III:a, Motorolan PowerPC 750:tä (eli G3:a) ja StrongARM SA-1110:ä, ei kuitenkaan reaaliaikamielessä.

Toisessa luvussa on esittelyä reaaliaikajärjestelmistä yleensä, skeduloinnista, suoritusai-

ka-analyysistä, välimuisteista reaaliaikajärjestelmissä. Yleensäkin käsitellään ennustamattomasti käyttäytyvää laitteistoa reaaliaikajärjestelmissä kuten TLB:tä ja virtuaalimuistia, liukuhihnoja konflikteineen sekä käyttäytymistä välimuistin kanssa, DMA:ta, keskeytyksiä ja dynaamisen muistin päivitystä.

Kolmas kappale käsittelee reaaliaikajärjestelmien välimuistianalyysiä mm. ILP-menetelmällä (Integer Linear Programming). Menetelmän taustat selvitetään perusteellisesti. Ennen ILP:iä Alt, Ferdinand, Martin ja Wilhelm ovat Saksassa käyttäneet menetelmää ennustamaan joukkoassosioituvan käsky- ja datavälimuistin käyttäytymistä. Heidän menetelmänsä kategorioi peruslohkojen joko ehkä (may) tai varmasti (must) olevan välimuistissa. Tästä johdetaan kaksi turvallista tilaa; “aina osuma” ja “aina ohiviittaus” josta analyysi määrittelee $WCET_C$:n joka peruslohkolle. Tämä luonnollisesti antaa silmukoille hyvin pessimistisen arvion, koska peruslohko silmukassa on aina “ehkä” (aina ohiviittaus). Ensimmäisellä iteraatiollahan tämä voi olla totta, mutta koska seuraavilla kierroksilla käskyt ja mahdollisesti dataakin käytetään uudestaan, voisi arviota parantaa. Ongelma ratkaistaan levittämällä silmukkaa auki yksi askel, jotta välimuisti voidaan tavallaan alustaa ennen menemistä itse silmukkaan. Alustuslohko voi olla “ehkä”, mutta itse silmukan peruslohko saakin “aina osuma”-määrityksen.

Floridassa tutkimusryhmä on lähestynyt ongelmaa toisesta näkökulmasta. He tuottavat ajoitusdataa datavuoanalyysistä staattisella välimuistisimuloinnilla. Käännetystä C-ohjelmasta muodostetaan kontrollivuoverkko, sitä analysoimalla tunnistetaan mahdolliset käskyt jotka kilpailevat samasta paikasta välimuistista ja lopuksi jokainen käsky luokitellaan käyttäytymisensä mukaan. Menetelmä ei osaa analysoida rekursiota eikä epäsuoria kutsuja. Välimuistisimuloinnin jälkeen ajoitusanalyysi tehdään, jotta $WCET_C$:iä voitaisiin rajoittaa. Työkalu kysyy käyttäjältä joka silmukan maksimisuorituskerrat joita kääntäjä ei voinut automaattisesti määritellä. Seuraavaksi analysoija muodostaa ajastusanalyysipuun ja worst-case välimuistisuoritusaika arvoidaan jokaiselle puun silmukalle. Myöhemmin tätä menetelmää on laajennettu lisäämällä malliin suoraan kuvautuvat datavälimuistit.

4.4 Isabelle Puautin artikkeli

Artikkelissa [7] verrataan kahta menetelmää reaaliaikaisen moniajojärjestelmän välimuistin huonoimman tapauksen suorituskyvyn ennustamiseen skedulointia varten: staattista välimuistianalyysiä ja välimuistin lukitsemista.

Välimuistin lukitseminen varaa tietyille taskeille tietyn osan välimuistista, jolloin ne eivät häiritse toisiaan ja välimuistin käytön arvointi on helpompaa.

Artikkeli sisältää hyviä viitteitä tuoreisiin julkaisuihin.

4.5 Sebek ja Gustafssonin raportti

Raportissa [8] esitellään staattinen menetelmä huonoimman tapauksen välimuistin ohiviittausten suhteen arviointiin. (Worst Case Cache Miss-Ratio). Tämä vaikuttaa olennaisesti järjestelmän virrankulutukseen.

Menetelmä toimii seuraavasti: Lasketaan ohjelman kullekin käskylle “paikallinen miss-

ratio". Tuotetaan binääripuu ohjelman mahdollisista suorituspoluista. Käydään puu läpi ja haetaan polku joka tuottaa suurimman miss-ration.

Menetelmä ei ota huomioon ajallista paikallisuutta (olettaa että kaikki viittaukset uuteen välimuistiriviin (cache line) ovat huteja). Ohjelman kaikki mahdolliset polut talletetaan binääripuuhun, jonka koko kasvaa suureksi.

Viitteet

[1] Cousot Patrick. *Semantics and Abstract Interpretation*. 2000. [WWW-dokumentti]. <http://www.di.ens.fr/~cousot/abstract_interpret.shtml>.

[2] Chilimbi Trishul M., Hill Mark D., Larus James R. *Making Pointer-Based Data Structures Cache Conscious*. Päiväys tuntematon. [WWW-dokumentti]. <http://citeseer.nj.nec.com/324091.html>.

[3] Ferdinand Christian. *Cache Behacior Prediction for Real-Time Systems*. Phd Thesis, Saarland University, 1999.

[4] Theiling Henrik, Ferdinand Christian. *Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis*. 1998. Universität des Saarlandes / Fachbereich Informatik.

[5] Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*.Springer-Verlag, 1999, 450 pages.

[6] Sebek Filip. *The state of art in Cache Memories and Real-Time Systems*. 2.10.2001. [WWW-dokumentti]. <http://www.idt.mdh.se/fsk/docs/sota.pdf>.

[7] Puaut Isabelle. *Cache Analysis Vs Static Cache Locking for Schedulability Analysis in Multitasking Real-Time Systems*. 2002. [WWW-dokumentti]. <http://citeseer.nj.nec.com/534615.html>.

[8] Sebek Gustafsson. *Determining the Worst Case Instruction Cache Miss-Ratio*. 2002. [WWW-dokumentti]. <http://citeseer.nj.nec.com/sebek02determining.html>.

[9] White Randall T., Mueller Frank, Healy Christopher A., Whalley David B., Harmon Marion G. *Timing Analysis for Data and Wrap-Around Fill Caches*. 1999. <http://citeseer.nj.nec.com/white99timing.html>.

Mikroprosessorien liukuhihnat staattisen analyysin näkökulmasta

Antti Kantee
Antti-Pekka Liedes

1 Johdanto

Reaaliaikaohjelmistojen tehokkuuden analyysissa tarvitsee käytetystä prosessorista mallintaa kaksi monimutkaista, mutta tärkeää asiaa: välimuisti ja liukuhihnat. Tässä paperissa esitellään liukuhihnojen mallintamiseen liittyviä ongelmia ja ratkaisuja, ja pyritään valottamaan sitä, miksi liukuhihnat ovat niin monimutkaisia kuin ovat.

Paperin ensimmäinen osuus käsittelee liukuhihnoja yleisesti. Osuudessa käydään läpi yleisiä liukuhihnojen piirteitä ja hieman sitä, miten näihin on päädytty. Lisäksi selvitetään liukuhihna-arkkitehtuurin ongelmia ja miten näitä on eri prosessoreissa pyritty lieventämään tai ratkaisemaan.

Paperin toisessa osuudessa käsitellään tarkemmin kahta nykyisin käytettyä liukuhihnaa. MIPS R3000 on esimerkki yksinkertaisesta, mutta silti modernista liukuhihnasta, jossa näkyy jo monien tämän hetken huippuprosessoreiden piirteitä. PowerPC taas on esimerkki yhdestä tämän hetken tehokkaimmista prosessoriarkkitehtuureista.

Paperin lopussa on vielä tiivistelmä ensimmäisestä ja toisesta osuudesta sekä yhteenveto.

2 Yleistä liukuhihnoista

Mikroprosessorien perustyypit ovat *yksisykliset (single cycle)*, *monisykliset (multi-cycle)* ja *liukuhihnalliset (pipeline)*. Näistä yksisykliset ovat yksinkertaisimpia. Niissä yhden käskyn suoritus etenee alusta loppuun yhdessä kellojaksossa, kulkien kaikkien tarvittavien suoritusyksiköiden (osoitteenlaskenta, rekisteripankki, aritmetiikka jne.) läpi kerrallaan.

Yksisyklisen toiminnan ongelmana on kuitenki erittäin pitkä signaalipolku. Signaalit pitää viedä kaikkien yksiköiden läpi, myös niiden, joita käskyn suorittamiseen ei edes tarvita. Näin prosessorin suurimman mahdollisen kellotaajuuden määrää pahin mahdollinen suorituspolku ja siinä tarvittava aika lopullisen tulossignaalin stabiloitumiseen. Tyypillinen esimerkki yksisyklisestä prosessorista on Intel 8051.

Ratkaisuksi kehitettiin aikoinaan monisykliset prosessorit. Niissä käskyn suoritus etenee suoritusyksikkö kerrallaan, yhden yksikön per kellojakso. Näin yhdellä kellojaksolla edettävä signaalipolku saadaan pienemmäksi ja kellotaajuutta vastaavasti nostettua. Eri käskyt vievät erimäärän kellojaksoja riippuen siitä, kuinka monta suoritusyksikköä niiden suorittamiseen tarvittiin. Esimerkiksi Motorolan alkuperäinen MC68000 on monisyklinen.

Monisyklisissä prosessoreissa on kuitenkin se ongelma, että vain pieni osa prosessorista tekee työtä kerrallaan. Jos käskyä suoritetaan aritmetiikkayksikössä, ei osoitteenlaskuyksikkö yleensä voida käyttää yhtään mihinkään samalla kellojaksolla. Niinpä syntyivät liukuhihnat, jotka suorittavat käskyt monessa kellojaksossa kuten monisykliset prosessorit, mutta käskysuoritus etenee aina tietyssä järjestyksessä yksiköltä toiselle, ja uusi käsky aloitetaan liukuhihnan alusta jokaisella kellojaksolla. Näin saadaan sekä lyhyt signaalipolku, joka mahdollistaa korkean kellotaajuuden, että suoritettua yksi käsky jokaisella kellojaksolla, ainakin periaatteessa. MIPS R3000 on koulukirjaesimerkki suoraviivaisesta, viisivaiheisesta liukuhihnasta.

2.1 Termistöä

Superliukuhihna: pitkiä, yli n. 10 vaihetta sisältäviä liukuhihnoja kutsutaan superliukuhihnoiksi.

Superskalaari: useampi liukuhihna rinnakkain, jolloin voidaan suorittaa enemmän kuin yksi käsky kellojaksossa.

Pipeline stall: käsky ei pääse etenemään seuraavaan vaiheeseensa liukuhihnalla, ja liukuhihna "pysähtyy", jotta tarvittavat operaatiot saadaan suoritettua ja käsky pääsee jatkamaan matkaansa. Stall on ei-toivottu tilanne, sillä se hidastaa kokonaissuoristusta.

Hasardi: tapahtuma, joka haittaa liukuhihnan toimintaa. Hasardit jaetaan yleensä kolmeen pääkategoriaan: rakenne-, data- ja kontrollihasardeihin. Rakennehasardi syntyy, kun samaa suoritusyksikköä tarvitsee kaksi käskyä samaan aikaan. Datahasardi puolestaan tarkoittaa sitä, etta dataa seuraavassa käskyssä ennen kuin se on ehtinyt valmistua jostain edellisestä käskystä ja kontrollihasardi on vastaava tapaus, jossa emme tiedä minne hypätä, koska laskennan tulos on vielä epäselvillä.

2.2 Liukuhihnojen suoritusvaiheet

Liukuhihnojen yksityiskohdat vaihtelevat prosessorista toiseen, mutta yleensä niistä löytyy seuraavanlaisia vaiheita:

Käskynhaku: käsky haetaan muistista (käskyvälimuistista).

Dekoodaus: käsky/käskyt dekoodataan ja eri osat ohjataan eteenpäin oikeille suorituspoluille.

Ryhmittely: käskyt jaetaan liukuhihnoihin (vain superskalaari).

Suoritus: varsinainen käskysuoritus, jossa operandien arvot haetaan rekistereistä ja suoritetaan aritmetiikka. Tässä vaiheessa voidaan myös laskea LOAD- tai STORE-käskyn käyttämä muistiosoite.

Muistinkäyttö: kirjoitetaan tai luetaan haluttu muistiosoite välimuistista. Muistinkäyttöön liittyy useasti monta vaihetta, joista yhdessä tehdään virtuaaliosoitteesta oikeaksi osoitteeksi muunnos MMU:lla ja L1 välimuistista haku, toisessa (jos tarvitaan) tehdään L2 välimuistista haku, jne. Vaiheiden määrästä ja koneen muistiväylästä riippuen jopa ulkoinen muistihaku on periaatteessa mahdollista tehdä näiden suoritusvaiheiden aikana, mutta yleensä cache miss tässä kohtaa aiheuttaa liukuhihnan pysähtymisen.

Kokoaminen: rinnakkain suoritetut käskyt kootaan takaisin yhteen.

Kirjoitus: käskyn tulos kirjoitetaan rekistereihin.

Joitakin vaiheita voi olla useampia peräkkäin, tyypillisesti juuri muistinkäyttöön liittyviä, jotta liukuhihna voidaan pitää synkronissa muistinkäyttöön liittyvien operaatioiden kanssa. Toisaalta synkronointia voi tapahtua myös erityyppisten liukuhihnojen välillä. Lisäksi myös pitkät aritmeettiset operaatiot, kuten kertolasku, voivat vaatia vaiheiden lisäämistä liukuhihnaan.

2.3 Liukuhihnojen suunnitteluperiaatteista

Liukuhihnojen pääperiaate on pitää yksittäiset vaiheet mahdollisimman pieninä, jotta pisin yhdessä kellojaksossa kuljettu signaalipolku olisi mahdollisimman pieni ja kellotaajuus saataisiin näin mahdollisimman korkeaksi. Mitään vaihetta ei kuitenkaan kannata tehdä myöskään huomattavasti lyhyemmäksi kuin muita, koska tällöin kyseisen vaiheen tulossignaali muodostuu “liian nopeasti” ja sit joudutaan ylläpitämään seuraavan kellojakson alkamista varten erikseen. Joissain tapauksissa välituloksia voidaan joutua jopa puskuroimaan.

Pisin yksittäinen vaihe muodostaa prosessorissa ns. kriittisen polun, joka määrää koko prosessorin maksimikellotaajuuden. Tämän vaiheen lyhentämiseksi sen suoritusta voidaan joko rinnakkaistaa tai jakaa se useampaan peräkkäiseen vaiheeseen. Rinnakkaistusta voidaan tehdä yllättävissäkin paikoissa, esimerkiksi 64-bittinen adder voidaan tehdä kahdella (tai useammalla) 32-bittisellä adderilla, jolloin signaalin läpimenoaikaa saadaan pienennettyä. Ylempien bittien adderin tulos riippuu tietenkin alempien bittien adderin carrysta, mutta tehdään ylemmille biteille kaksi adderia, toinen laskee kuin carry olisi 0 ja toinen kuin carry olisi 1. Näistä valitaan oikea tulos sitten, kun tiedetään alempien bittien adderin carry.

Jos vaihetta ei pystytä rinnakkaistamaan, voidaan se yleensä kuitenkin jakaa moneen peräkkäiseen vaiheeseen. Näin yksittäisen vaiheen läpimenoaika saadaan pieneksi ja kellotaajuutta nostettua vastaavasti, mutta liukuhihna pitenee. Yleisestikin pidemmät liukuhihnat mahdollistavat suuremmat kellotaajuudet, mutta liukuhihnan pidentäminen aiheuttaa muita ongelmia, joista lisää myöhemmin. Nykyiset hihnat voivat olla jopa 20 vaiheen mittaisia (Pentium 4).

Liukuhihnan lisäksi kellotaajuuteen liittyy kiinteästi myös välimuisti, koska välimuistista täytyy pystyä lataamaan yksi tai useampikin käskysana yhdessä kellojaksossa. Muutoin paraskin liukuhihna pyörii tyhjää, koska suoritettavaa ei ole.

2.4 Rinnakkaiset liukuhihnat

Yksinkertaisemmat mikrokontrollerit poislukien useimmissa nykyprosessoreissa on useampi rinnakkainen liukuhihna. Liukuhihnoja on myös erilaisia, kuten kokonaislukuhihna, LOAD/STORE-hihna, haarautumishihna ja liukulukuhihna. Nykyisissä prosessoreissa erityyppisiä liukuhihnoja voi olla rinnakkain yhteensä jopa kymmenkunta.

Hihnojen alun käskynlautaukseen liittyvät vaiheet ovat yleensä samat kaikille hihnoille. Käskyn dekoodauksen paikkeilla tai jälkeen käskyt jaetaan erityyppisille hihnoille käskyn tyypin mukaan. Erilliset hihnat voivat olla erimittaisia (esim. Alpha 21064), jolloin tarvitaan lisää kontrollia käskyjen keräämisessä yhteen, koska eri hihnoilta valmistuvat käskyt tulevat väärässä järjestyksessä. Hihnat voivat olla myös yhteennivottuja (esim. SPARC), jolloin niiden pituudet ovat samoja (jokin hihna voi jopa sisältää ylimääräisiä vaiheita muiden hihnojen odottelua varten) ja esim. liukulukuliukuhihna voi aiheuttaa kokonaislukuhihnan pysähdyksen. Toisaalta tällainen toimintatapa on suoraviivaisempi ja helpompi toteuttaa.

Prosessorien erikoiskäskykannoille, kuten Pentiumien MMX ja SSE sekä SPARCien VIS, on yleensä omat liukuhihnansa. Nämä voivat kuitenkin käyttää joitakin suoritusyksiköitä, kuten liukulukuyksikköä, yhdessä muiden hihnojen kanssa, joten niiden käskysuoritus ei välttämättä ole täysin rinnakkaista muiden hihnojen kanssa.

Nykyiset prosessorit pystyvät aloittamaan yhdestä neljään käskyä rinnakkain yhdessä kellojaksossa. Rinnakkaisuudelle on yleensä kuitenkin joitain rajoituksia, kuten että pystytään aloittamaan korkeintaan kaksi liukulukukäskyä, vaikka neljä olisi käytettävissä.

2.5 Liukuhihnojen ongelmia

Liukuhihnoissa on myös omat ongelmansa. Koska käskyjen suoritus aloitetaan jo ennen kuin edellisten käskyjen tulokset ovat valmiita, voi käskyjen välisistä riippuvuuksista syntyä ns. hasardeja. Yleisen tällainen tilanne on luku-ennen-kirjoitusta, jossa jonkin paikan (rekisteri, muistipaikka) sisällön lukee jokin käsky j, joka suoritetaan käskyn i jälkeen. Käsky i on kirjoittamassa samaan paikkaan, mutta ei ole vielä ehtinyt tehdä kirjoitustaan, jolloin j saa lukiessaan paikan väärän arvon. Paikka on yleensä rekisteri, mutta se voi periaatteessa olla jopa itse käsky j muistissa. Vaikka itsemuuntuvaa koodia ei sallittaisikaan, voisi tällainen tilanne tulla kuitenkin vastaan esim. uutta ohjelmakoodia ladattaessa.

Pahimmat hasardit syntyvät ehdollisten hyppyjen seurauksena, kun hypyn kohdetta ei vielä tiedetä ja liukuhihnaa pitäisi ruokkia joillain käskyillä.

Hasardien tuloksena liukuhihna joudutaan nykyprosessoreissa tyhjentämään (pipeline stall). Joissakin vanhoissa prosessoreissa, kuten IBM System/360:ssä, pystyttiin jatkamaan hihnalla hasardin saaneen käskyn jälkeen olevia käskyjä normaalisti [4], mutta tällaista tekniikkaa ei käytetä nykyisissä prosessoreissa. Nykyisin pyritään lähinnä minimoimaan hasardien määrää erilaisilla tekniikoilla, joita kuvataan hieman seuraavassa.

2.5.1 Rekisterinohitus

Rekisterinohitusta (data bypassing, bypass path) käytetään, kun hihnalla jo oleva käsky tarvitsee pidemmällä hihnalla olevan käskyn tulosta [4]. Tällöin tulos tuodaan suoraan rekiste-

ripankin ohi sitä tarvitsevalle käskylle, jolloin säästetään rekisteripankin käyttökustannus.

Tätä tekniikkaa on käytetty pitkään erityisesti liukulukuoperaatioissa (esim. alkuperäinen Pentium). 70-lukulainen IBM 360/91 sisältää erityisen hienon sisäisen väylän (common data bus, CDB), joka nimeää liukuhihnalla olevien käskyjen operandeja ja välittää tuloksia suoraan varsinaisen rekisteripankin ohi käskyltä toiselle [4].

2.5.2 Rekisterien uudelleennimeäminen

Turhia hasardeja voidaan välttää nimeämällä rekistereitä uudelleen eri käskyjen välillä. Tämä tulee vastaan erityisesti epäjärjestyssuorittamisen yhteydessä, jolloin rekistereiden nimeämisellä voidaan “liikkua ajassa” käskyjen tulosten suhteen [3].

Yksi tapa toteuttaa rekistereiden uudelleennimeäminen tapahtuu niin, että laitteistolla on käytössään oikeasti huomattavasti suurempi määrä rekistereitä, kuin mitä se antaa (export) käyttäjälle. Näitä rekistereitä sidotaan rekistereiden nimiin (siis esim. "r1") aina tarpeen mukaan. Esimerkiksi jos käsky 1 käyttää rekisteriä r5 argumenttina, ja käsky 2 kirjoittaa tuloksensa rekisteriin r5, voimme välttää hasardin käyttämällä jälkimmäisen käskyn r5:na jotain aivan muuta rekisteriä, mutta vain kutsumalla sitä nimellä r5.

2.5.3 Kontrollihasardit ja haarautumisen ennustaminen

Ohjelmakoodin haarautumiseen (hyppyihin) liittyviä ongelmia on kahdenlaisia: ehdolliset hypyt (kontrollirakenteet) ja hypyt tuntemattomaan (funktio-osoittimet). Kummassakaan tapauksessa ei voida olla varmoja, mitä käskyjonoa pitäisi suorittaa ennen kuin suoritus on jo pitkällä liukuhihnalla ja pahimmassa tapauksessa jono oli väärä, eli kaikki tehty työ meni hukkaan ja liukuhihna täytyy tyhjentää.

Väärän käskyjonon suorittaminen on monessa suhteessa ongelmallisin hasardityyppi; sitä vastaan ei voida taistella esim. rekisterien uudelleennimeämisellä. Lisäksi etäisyys virheen tapahtumisesta virheen havaitsemiseen liukuhihnalla on suuri, jolloin menetetään paljon työtä. Lisäksi voi syntyä vielä käskyvälimuistin kanssa ongelmia. Normaalisti käskyvälimuisti esihakee käskyjonoa, jotta liukuhihnaa pystyttäisiin ruokkimaan mahdollisimman tehokkaasti (täyttä suoritustehoa varten käskyvälimuistista pitää pystyä lataamaan neljä tai enemmänkin käskysanaa per kellojakso), joten huti käskyvälimuistissa voi olla melkoinen katastrofi.

Näiden kontrollihasardien välttämiseksi ja niiden ongelmien lievittämiseksi on joitakin konsteja:

- Jos käytössä on useampi rinnakkainen liukuhihna, voidaan kahta eri käskyjonon haaraa suorittaa rinnakkain varmuuden vuoksi. Kun haarautumisen tulos tiedetään, oikeaksi osoittunut käskyjono jatkaa normaalisti ja väärän tulokset hylätään. Tämä vaatii huomattavaa ohjauslogiikkaa, koska käskysuoritus jakautuu kahteen täysin erilliseen ja toisistaan riippumattomaan haaraan. Lisäksi ainakin teoreettinen nopeus puolittuu operaation ajaksi, joskin on oletettavaa, että kahta jonoa ajettaessa syntyy vähemmän muita hasardeja, koska toisistaan riippuvia käskyjä pitäisi olla piipuissa vähemmän. Hyvänä puolena tietty minimiteho säilyy aina.

- Hyppyjä voidaan ennustaa. Yksinkertainen staattinen hyppyennustus (branch prediction) valitsee taaksepäin otettavan ehdollisen hypyn otettavaksi (silmukan toistaminen) ja eteenpäin otettavan ehdollisen hypyn ei-otettavaksi (silmukasta poistuminen) siitä yksinkertaisesta syystä, että silmukat yleensä suoritetaan useasti.

- Dynaamisessa hyppyennustuksessa käytetään erillistä puskuria (branch target buffer, BTB), johon tallennetaan hyppykäskyjen otettuja kohdeosoitteita. Jos hyppy tuntemattomaan havaitaan käskyn dekoodausvaiheessa, sen oletetaan olevan puskurissa olevaan osoitteeseen, jos kyseinen käsky löytyy puskurista. Esimerkiksi alkuperäisessä Pentiumissa on 256 paikan neljätieassosiatiivinen BTB [2].

- Ehdollisten hyppyjen dynaamiseen ennustamiseen käytetään mm. PowerPC:ssä laskurirekisteriä (count register) silmukoiden tapauksessa [2]. Näiden avulla pyritään päättelemään hypyt vanhasta suoritushistoriasta.

- Ehdollisen hypyn molemmat mahdolliset kohteet voidaan esihakea käskyvälimuistiin, jolloin vältytään ainakin käskyvälimuistin hudeilta.

3 Esimerkkejä liukuhihnoista

Seuraavassa tarkastellaan kahta oikeaa liukuhihnatoteutusta ja sitä, miten em. seikat heijastuvat niihin. MIPS R3000 toimii eräänlaisena johdantona nykyaikaisiin liukuhihnoihin. Vaikka sen päivät työasemakäytössä ovatkin jo ohi, sitä käytetään edelleen sulautetuissa järjestelmissä. Lisäksi sen liukuhihna on hyvin suoraviivainen ja sen toiminta on jopa kohtuudella ymmärrettävissä, joten sen avulla on helppo hahmottaa kohtalaisen yksinkertaisella tasolla liukuhihnojen toimintaa ja niiden ongelmia.

PowerPC taas on yksi tämän hetken tehokkaimmista työasema- ja palvelinprosessoreista. Desktopilla PowerPC on i386-arkkitehtuurin vakavin haastaja Applen käyttämänä ja esimerkiksi Suomen CSC:n uusin IBM:n superkone on POWER4-pohjainen. PowerPC on tässä esitelty alan state-of-the-artina, josta löytyy paljon tämän hetken parhaita ratkaisuja.

3.1 MIPS R3000

MIPS R3000 [7] on tyyppillinen “your first pipeline”, joka tungetaan opiskelijoille päähän opintojen ensimmäisenä vuonna. Kyseessä on yksikerroksinen liukuhihna, jonka jokainen vaihe suorittuu tasan yhdessa kellojaksossa. Koska vaiheita on viisi, liukuhihnan latenssi on tasan viisi kellojaksoa ja läpäisy on tasan yksi käsky per kellojakso. Poikkeuksena on kertolaskuyksikkö, joka ei toimi liukuhihnan sisällä. Oma vaikutuksensa on myös FPU:lla, mutta koska se on erillinen apuprocessori (coprocessor 1, CP1), se jätetään huomiotta [6].

[6] Asiassa on menty niin pitkälle, että normaalissa ajossa ensimmäisestä FPU:lle menevästä käskystä heitetään poikkeus, jonka käyttöjärjestelmä voi halutessaan kaapata ja alustaa FPU:n.

3.1.1 Liukuhihnan vaiheet

Vaiheet ovat melkein suoraan samat kuin edellä esitelty yleinen malli. Käskyjen enemmän tai vähemmän viralliset nimet kertovat myös hieman siitä, mitä osaa prosessorista kyseinen liukuhihnavaihe rasittaa.

IF: käskyhaku, luetaan seuraava käsky käskyvälimuistista

RD: käskyn dekoodaus, päätellään mitä pitäisikään tehdä

ALU: suoritus tai muistiosoitteen laskeminen, jos käsky on load/store

MEM: muistista lukeminen/kirjoittaminen

WB: tallennus rekistereihin

3.1.2 Liukuhihnan ominaisuudet

Liukuhihnan yksinkertaisuudesta johtuen liukuhihna tuottaa kaksi ulospäin ohjelmoijalle näkyvää vaikutusta. Ensinnäkin ehdollisen haarautumiskäskyn jälkeinen käsky, *branch delay slot*, suoritetaan aina haarautumisen tuloksesta riippumatta. Toiseksi, koska mitään hienoa kuten käskyjen uudelleenjärjestyä ei suoriteta enää sisäisesti, loadin on tulos nähtävissä vasta yhden käskyn loadin jälkeen johtuen siitä, että RD haluaa jo käyttää dataa, joka ladataan vasta MEM-vaiheessa. Tätä nimitetään viivettä *load delay slotiksi*, johon yleensä kääntäjä pyrkii laittamaan jonkin käskyn, mutta jos muu ei auta, pitää se täyttää **nop**illa.

Yksi mielenkiintoisuus on se, että kerto- ja jakolaskuja ei suoriteta varsinaisesti liukuhihnan sisällä, koska ne vaativat useamman kuin yhden kellojakson suoriutuakseen, jopa 80 kellojaksoa parhaimmillaan. Näille operaatioille on varattu erilliset tulosrekisterit **hi** ja **lo**. Toisin kuin normaalit kokonaislukurekisterit, nämä rekisterit on myös *interlockattu* [7] , eli jos niitä yritetään käyttää ennen kuin niihin kirjoittava käsky on valmistunut, pysäytetään koko CPU odottamaan käskyn valmistumista. Kertolaskuyksikkö ei sisäisesti ole liukuhihnoitettu, vaan suorittaa vain yhtä käskyä kerrallaan.

3.2 PowerPC

PowerPC [5] [6] on IBM:n, Applen ja Motorolan yhteistyössä kehittämä vuonna 1991 julkaistu arkkitehtuuri. Se on nykypäivänä käytössä muunmuassa tavanomaisissa Applen pöytäkoneissa ja lisäksi sulautetuissa sovelluksissa erityisesti automaailmassa. Perusominaisuuksiltaan arkkitehtuuri on joustava, sallien sekä 32- että 64-bittisen tilan ja toimien kummassakin tavujärjestyksessä.

Suorittamisen kannalta olennaiselta rakenteeltaan PowerPC on jaettu kolmeen erilliseen ja samanaikaiseen suoritukseen kykenevään yksikköön: haarautumis-, kokonaisluku- ja liukulukuyksikköön. Yhtä tyyppiä olevaa yksikköä on myös mahdollista rakentaa useita kappaleita samaan prosessoriin. Tämä mahdollistaa kolme käskyn käynnistämisen ja valmistu-

[7] MIPS: Microprocessor without Interlocked Pipeline Stages.

misen samalla kellojaksolla. PowerPC on siis aiemmin yleisellä tasolla kuvattujen rinnakkaisten liukuhihnojen omaavien prosessorien kaltainen eli superskalaarinen.

Lisäksi muita yksiköitä ovat käskynlataus-, käskynvalmistumis- ja load/store-yksikkö. Liukuhihnan yleisen toiminnan kannalta on oleellista tuntea käskynlatausyksikön toiminta. Jokaisella kellojaksolla yksikkö hakee käskyvälimuistista kaksi käskyä lisää kahdeksan käskyn pituiseen tauluunsa. Jokaisella kellojaksolla yksikkö tutkii neljää alinta käskyä, ja aloittaa niistä korkeintaan kolme. Käskyjen suorittamisjärjestyksen valinnalla on suuri merkitys sen kannalta, joudutaanko myöhemmin liukuhihnateknisiin ongelmiin (tuleeko hasardeja).

PowerPC:ssä on kaksi eri rekisteripuskuria käytettävissä suorituksen aikana: yksi puskuri yleiskäyttöisille rekistereille ja toinen puskuri liukulukuyksikön käyttöön. Käskylle allokoidaan suorituksen alkaessa ns. rename rekisterit, joihin käsky kirjoittaa tuloksensa. Jos jonkun toisen käskyn pitää käyttää vielä suorituksessa olevan käskyn tulosta, se saa sen (jos tulos on jo valmistunut) uudelleennimeämispuskurista.

Edellä mainitulla puskurilla on myös toinen funktio. Voimme helposti "peruuttaa"jo valmistuneiden käskyjen tulokset vain olemalla siirtämättä arvoja uudelleennimeämispuskurista rekisteripankkiin. Esimerkiksi jos hyppyennustus meni pieleen, väärästä paikasta suoritettujen käskyjen tuloksia ei haluta talteen. PowerPC:n käskynvalmistumisyksikkö toimii auktoriteettina asian suhteen ja päättää koska käsky oikeasti valmistuu siirtäen samalla tulokset puskurista rekisteripankkiin.

PowerPC:n liukuhihna vaihtelee yksikköittäin. Eri yksiköt sisältävät eripituiset liukuhihnat, ja niiden toiminta vaihtelee. Kaikille yksiköille sama ensimmäinen vaihe on käskyn hakeminen käskynlatausyksikköön, jota ei oikeastaan edes suoriteta yksikön sisällä, ja sen takia se onkin jätetty allaolevista listauksista pois. Myöskin jokaiselle käskylle pitää tehdä käskyn dekoodaaminen, joten tätäkään ei ole listattu. Sekaannuksien välttämiseksi on myös hyvä huomata, ettei seuraavsaassa yksi alakohta aina suinkaan vastaa yhtä kellojaksoa, vaan vaiheet on jaoteltu tällä kertaa enemmänkin niiden loogisen funktion mukaan.

3.2.1 Liukuhihna – kokonaislukuyksikkö

Kokonaislukuyksikön liukuhihna on jo tutuksi käyneen mallinen:

- suoritus
- kirjoitus rekisteripankkiin

Useimmat käskyt pystyvät suorittamaan suoritusvaiheensa yhdessä kellojaksossa, mutta eräät kerto- ja jakolaskut voivat viedä yhteensä kahdesta 37:een kellojaksoon suoritusvaiheessa.

Yksikön latenssi on normaalisti kolme kellojaksoa.

3.2.2 Liukuhihna – load/store

Load/store-yksikön liukuhihnan toiminta vaihetelee hieman sen mukaan, ollaanko suorittamassa loadia vai storea. Loadille:

- osoitteenlaskenta

- muistihaku
- kirjoitus rekisteripankkiin

Store toimii hieman eri tavalla, koska ennen kuin se voi tallentaa muistiin, on sen juostava MMU:n kautta tarkistamassa, saako muistiin tallettaa.

- osoitteenlaskenta
- tarkistus
- tallennus

Yksikön latenssi on kolme kellojaksoa.

3.2.3 Liukuhihna – liukulukuyksikkö

Liukulukuyksikkö kokonaislukuyksikön tapaan sisältää liukuhihnan, joissa useimmat käskyt kuten single-precision yhteenlasku valmistuvat kellojakson välein, mutta double-precision operaatiot ja jako- ja kertolasku vievät huomattavasti kauemmin.

Liukuhihnan vaiheet ovat seuraavat:

- suoritus1: kertolasku
- suoritus2: yhteenlasku
- suoritus3: tuloksen normalisointi
- tallennus

Yksikön latenssi on kolmesta neljään kellojaksoa riippuen laskutarkkuudesta.

4 Yhteenveto

Liukuhihnat ovat muodostuneet viimeisen n. 10 vuoden aikana kaikkien tehokkaiden prosessorien de-fakto-toteutustavaksi. Kehitystä on ajanut eteenpäin jatkuva tarve lisäteholle, jota on saatu kellotaajuuden nostamisella, ja ainoa tapa toteuttaa prosessori nykyisillä kellotaajuuksilla on liukuhihna. Samalla on vakiintunut tiettyjä trendejä nykyaikaisissa liukuhihnoissa, kuten:

- Pitkät superliukuhihnat, nykyiset liukuhihnat voivat olla jopa 20 askeleen mittaisia (Pentium 4), jotta ne mahdollistaisivat nykyiset kellotaajuudet (yli 3GHz).
- Rinnakkaista suoritusta lisätään ja nykyisissä prosessoreissa on jopa kymmenkunta rinnakkaista liukuhihnaa. Monet prosessorit pystyvät maksimissaan neljän käskyn päättämiseen kellojaksossa.
- Liukuhihnatyyppejä on useita. Yleiset tyypit ovat kokonaislukuliukuhihna ja liukulukuhihna, jotka suorittavat prosessoreiden perinteisimmät käskyt. Lisäksi haarautumis/hyppylogiikalle ja muistioperaatioille on nykyisin usein omat/omia liukuhihnat/hihnoja. Ja tähän vielä päälle erikoiskäskykannat ja niiden omat liukuhihnansa.

- Vanhoissa liukuhihnoissa saatettiin hasardin sattuessa pystyä peruuttamaan vain itse hasardin aiheuttanut käsky. Nykyisin tällaista logiikka ei kuitenkaan näe käytetyn, vaan hasardeja pyritään lähinnä estämään eri keinoin, kuten rekisterien uudelleennimeämisellä, rekisterien ohituksella ja haarautumisen ennustamisella. Itse hasardista, jos/kun sellainen sattuu, on taas tullut erittäin vakava ongelma.

- Suurin osa toiminnoista on pyritty liukuhihnoittamaan, mutta edelleenkin on tiettyjä vaikeita toimintoja, joiden latenssit ovat todella pitkiä. Pahimpia ovat yleensä liukulukujakolasku ja -neliöjuuri, jotka saattavat kestää jopa kymmeniä kellojaksoja, näin häiriten prosessorin toimintaa huomattavasti, varsinkin jos seuraava laskenta riippuu näiden tuloksista.

Liukuhihnoista on tullut monimutkaisia; ne suorittavat monia erilaisia toimintoja rinnakkain, mutta toimintojen täytyy kuitenkin olla synkronoituja keskenään oikeellisuuden takaamiseksi. Lisäksi suoritus on tiiviissä yhteydessä välimuistin toimintaan käskylatauksen ja LOAD/STORE-toimintojen muodossa, joten liukuhihnojen täytyy olla synkronoituja myös välimuistin suuntaan.

5 Johtopäätöksiä

Kuten välimuistikin, liukuhihna on erinomainen keskimääräisen tehon kannalta, mutta erittäin huono pahimmassa tapauksessa. Liukuhihna muodostaa yhden suuren epävarmuustekijän arvioitaessa järjestelmän tehoa. Liukuhihnat eivät kuitenkaan ole muodostuneet vahingossa sellaisiksi, kuin ne ovat. Pohjimmiltaan liukuhihnat pyrkivät ratkaisemaan ongelman "kuinka suorittaa eri käskyt mahdollisimman pienissä vaiheissa (jotta kellotaajuutta voitaisiin nostaa), suorittaen mahdollisimman monta vaihetta rinnakkain (jotta käytetty piin pinta-ala saataisiin hyödynnettyä mahdollisimman tehokkaasti)?"

Lisäksi liukuhihnojen suunnittelua rajoittaa ja monimutkaistaa huomattavasti prosessorien käskykannat ja niihin liittyvät tekijät, kuten ansat (trap). Siinä missä välimuisti on suunniteltu tutkimalla keskimääräisten ohjelmien toimintaa ja virittämällä välimuisti toimimaan mahdollisimman tehokkaasti niiden kanssa, prosessorin käskykanta on enemmän tai vähemmän ennalta-annettu ominaisuus, ja liukuhihna vain pitää suunnitella suorittamaan kyseinen kanta mahdollisimman tehokkaasti. Näin joudutaan esimerkiksi yhdistämään triviaali kahden rekisterin välinen looginen operaatio yhteen kahden liukuluvun jakolaskun kanssa, ja tulos on väistämättä monimutkainen.

Kyse ei siis ole vain reaaliaikaohjelmoijien elämän tekemisestä hankalaksi. Kilpailu ajaa prosessorimarkkinoita voimakkaasti ja nykyiset kellotaajuudet ovat mahdollisia vain monimutkaisilla liukuhihnoilla. Monia nykyajan reaaliaikamaailman ongelmia ei pystyisi ratkaisemaan vanhanaikaisilla prosessoreilla, koska ne ovat yksinkertaisesti liian hitaita. Tilanteeseen siis on sopeuduttava, ja liukuhihnoihin staattisen analyysin yhteydessä on kehitetty metodeja ja ratkaisuja [1] [8].

Viitteet

[1] Engblom: Processor Pipelines and Static Worst-Case Execution Time Analysis, PhD Thesis, University of Uppsala, 2002.

[2] Murphy, MacDonald: Survey of High Performance RISC Microprocessors, 1994

[3] Palacharla, Jouppi, Smith: Complexity-Effective Superscalar Processors

[4] Ramamoorthy, Li: Pipeline Architecture, ACM Computing Surveys Vol. 9, No. 1, March 1977

[5] Allen, Becker, Moore et al: The PowerPC 601 Microprocessor, IEEE Micro, October 1993

[6] Burgess, Ullah, Van Overen, Ogden: The PowerPC 603 Microprocessor, The Communication of ACM Vol. 37, No. 6, June 1994

[7] Sweetman: See MIPS Run, Morgan Kaufman Publishers 1999

[8] Rautio Jussi, Tukkinen Juha: Ohjelmistojen liukuhihnan suoritusajan arviointi staattisilla analyysimenetelmillä.

Ohjelmistojen liukuhihnan suoritusajan arviointi staattisilla analyysimenetelmillä

Jussi Rautio
Juha Tukkinen

1 Johdanto

Liukuhihna-analyysi on tekniikka, jossa tietovuoanalyysin pohjalta tehdään päätelmiä liukuhihnan käyttäytymisestä. Tätä varten tarvitaan mahdollisimman täsmällinen malli liukuhihnasta. Ohjelmakoodista pitää pystyä ratkaisemaan liukuhihnan tila, mikä on vaikeaa. Tämä tehdään usein staattisesti abstraktilla tulkinnalla. Liukuhihna-analyysiä käytetään usein yhtenä menetelmänä osana ohjelman huonoimman mahdollisen suoritusajan (WCET) arviointia. Tässä kirjallisuustutkimuksessa käsitellään muutamia keskeisiä julkaisuja, jotka liittyvät liukuhihna-analyysiin. Alussa on lisäksi käsitelty liukuhihnoja ja niihin liittyviä analyysitekniikoita yleisesti.

Toisessa luvussa kuvataan lyhyesti liukuhihnajärjestelmän tärkeimmät periaatteet. Enemmän yleistä asiaa liukuhihnoista löytyy artikkelista [6]. Kolmas luku on varsinainen katsaus: siinä käydään läpi tekniikoita julkaisuittain. Neljäs luku sisältää yhteenvedon.

2 Yleistä liukuhihnoista

Liukuhihnoissa käskyn suorittaminen on jaettu useaan vaiheeseen, joten useita käskyjä voidaan suorittaa rinnakkaisesti. Jotkut liukuhihnat voivat aloittaa useamman kuin yhden käskyn jaksolla. Käytännön ongelmia aiheuttavat välimuistin ohiviittaukset ja liukuhihnojen *hasardit*.

Liukuhihnat toimivat yleensä hyvin. Kääntäjä voi vähentää hasardeja käskyskeduloinnilla. Toisalta liukuhihnoja on vaikeaa ennustaa, niiden pysähtymiset riippuvat suoritushistoriasta ja välimuistin sisällöstä. WCET-oletuksina käskyn suoritusta ei voi päällekäistää, lisäksi jos hasardia ei voida turvallisesti sanoa olevan tapahtumatta, sen on oletettava tapahtuvan. Haarautumiset ovat pullonkaulana liukuhihnoissa, ennustaminen voi perustua kohdeosoitteeseen, hypyn historiatauluun tai staattiseen haarautumisen ennustamiseen. Väärä ennustaminen on hyvin kallista, koska tällöin menetetään käskyvälimuistin data.[1]

3 Liukuhihna-analyysi

Liukuhihna-analyysin tavoitteena on laskea kaikki mahdolliset liukuhihnan tilat tietyssä ohjelman pisteessä. Yksi menetelmä on suorittaa syklimäinen elinkaari liukuhihnasta eli mää-

ritellään kaikki mahdolliset liukuhihnan seuraavat tilat. Tähän tarvitaan formaali malli liukuhihnasta, sen tiloista ja niiden välisestä kommunikaatiosta. Lopputuloksena saadaan WCET peruslohkoille. Liukuhihnan abstrakti tila on joukko konkreettisia liukuhihnan tiloja. Joukot ovat pienehköjä, koska liukuhihna ei ole kovin pitkälle riippuvainen suoritushistoriasta.

Ongelmallista liukuhihna-analyysissä on mm. se, että liukuhihnoista täytyy tehdä asbtrakti malli monen vaiheen kautta. Monimutkaisimmat liukuhihnat ovat lisäksi huonosti dokumentoituja ja turvallisten tulosten saamiseksi malli täytyy jo heti aluksi tehdä pessimistiseksi. Tuloksen tekee vielä enemmän pessimistiseksi se, että liukuhihnojen tilojen kuvausta täytyy yksinkertaistaa.[1]

Abstrakti liukuhinan päivitysfunktio kuvastaa mitä tapahtuu kun liukuhihna etenee yhden askeleen. Se ottaa huomioon nykyisen liukuhihnatilajoukon, erityisesti resurssien varaukset, esinoutojonon (prefetch queue) sisällön, käskyjen ryhmittämisen ja muistiviittausten luokittelun osumiksi tai ohiviittauksiksi välimuistiin.

3.1 Polku- ja ajoitusanalyysin yhdistäminen

Lundqvist ja Stenström [2] käsittelevät tekniikkaa, jolla voidaan tarkasti ennustaa WCET-suorituskykyä yhdistämällä polku- ja ajoitusanalyysin. Tällä menetelmällä voidaan jättää huomioimatta monta mahdotonta ohjelmapolkua ja sillä voidaan laskea polkuinformaatiota, kuten rajoja silmukoiden suoritusmäärälle ilman käsin tehtäviä lisäyksiä. Lisäksi menetelmä mallintaa liukuhihnan ja välimuistin käyttäytymistä ja niiden aiheuttamia ajoitusominaisuuksia. Tekijät väittävät, että tämä yhdistetty analyysi antaa eksaktit arviot huonoimmasta suoritusajasta kuudelle seitsämästä ohjelmasta jota he ovat analysoineet.

WCET-arvio on usein turhan pessimistinen kahdesta pääsyystä. Arviointitekniikka voi ottaa mukaan ohjelman polut, joita ei koskaan suoriteta riippumatta syötedatasta. Toinen syy on se, että ajoitusarviot ovat liian yksinkertaistettuja, joka johtaa WCET:in yliarviointiin. Ensimmäinen ongelma voidaan ratkaista joko ohjelmoijan koodiin tekemillä polkuinformaatiolisäyksillä tai automaattisilla polkuanalyysimenetelmillä. Toiseen ongelmaan on ratkaisuna useita staattisia ajoitusanalyysimenetelmiä. Lundqvist ja Stenström esittävät uuden staattisen WCET-analysointimenetelmän, joka yhdistää polkuanalyysin tarkkaan ajoitusanalyysiin. He käyttävät arkkitehtuurillisia simulaatiomenetelmiä, jotka simuloivat jokaisen polun raudan syklitason ajoitusmalliissa. Lisäksi he laajentavat näitä tekniikoita mahdollistamaan ohjelmien symbolisen suorittamisen, kun syötedataa ei tunneta. Käskysemantiikka lisätään käsittelemään myös tuntemattomat syötedata-arvot. Näin saadaan automaattisesti staattisesti tietoon polkuinformaatiota kuten silmukoiden rajat mielivaltaisen monimutkaisista lauseista ja lisäksi voidaan poistaa mahdottomia polkuja analyysistä.

3.2 Vaiheistettu analyysi

Artikkeli [1] esittelee modulaarisen lähestymistavan, jossa analyysi jaetaan vaiheisiin. Polku- ja liukuhihna-analyysin lisäksi otetaan huomioon välimuisti. Lähestymistavassa käytetään kiinteää järjestystä, jossa ensin käsitellään välimuistiin liittyvät asiat. Tämän jälkeen lasketaan liukuhihnan vaikutukset. Laskettaessa voidaan ottaa huomioon välimuisti ja sen

aiheuttamat liukuhihnan tyjentymiset. Viimeiseksi tehdään polkuanalyysi ja lasketaan lopullinen WCET käyttämällä hyväksi liukuhihna-analyysin tuottamia suoritusaikoja.

Tällainen vaiheistaminen toimii, jos prosessorin rakenteesta ei seuraa syklisiä riippuvuuksia eri vaiheiden välille. Lisäksi vaiheiden kiinteä järjestys aiheuttaa sen, että arvion laatu saattaa kärsiä. Oikeellisuudesta ei tietenkään voida tinkiä. Koska välimuistianalyysi tehdään ennen polkuanalyysia, analyysin on oletettava mahdolliseksi kaikki mahdolliset suorituspolut. Jos polkuanalyysi tehtäisiin ensin, se pystyisi karsimaan pois mahdottomat polut ja mahdollistaisi ehkä paremman arvion välimuistin käytön viemästä ajasta. Se ei kuitenkaan itse pystyisi hyötymään välimuistianalyysin tuomasta uudesta informaatiosta. Ongelma muistuttaa kääntäjien suunnittelun vaiheistusongelmia.

Aluksi käännetystä koodista muodostetaan CRL-välikielellä oleva kontrollivuoverkko. Sitä seuraavassa arvoanalyysissä selvitetään muuttujien mahdollisia arvoja ajon aikana. Tässä pyritään päättelemään epäsuorat viittaukset ja silmukoiden suorituskerrat. Välimuistianalyysi luokittelee muistiviittaukset kolmeen luokkaan: niihin, jotka ovat välimuistissa aina, joskus tai ei koskaan. Silmukat ja rekursiiviset aliohjelmat tutkitaan erikseen VIVU-menetelmää käyttäen. Menetelmä irrottaa silmukoista ensimmäiset toistot ja käsittelee niitä erikseen, koska ensimmäinen silmukan toisto aiheuttaa usein paljon enemmän välimuistin päivityksiä. Liukuhihna-analyysi käyttää abstraktia esitystapaa mallintamaan liukuhihnan toimintaa. Esitystapa on tavallaan kaikkien mahdollisten hidastavien tekijöiden superpositio. Tuloksena on suoritusaika jokaiselle käskylle jokaisessa eri suoritusyhteydessä. Lopuksi ILP[5]-generaattori mallintaa ohjelman kontrollivuon ja suorittaa polkuanalyysin. Kokonaislukulineaariohjelma ratkaistaan lp_solve:lla. Tällä tavalla saadaan lopulta suoritusmäärät joka peruslohkolle ja käyntimäärät joka reunalle.

3.3 Parempiin tuloksiin ajoitusvaikutuksia laskemalla

Varsinkin sulautetuissa reaaliaikajärjestelmissä staattinen WCET-analyysi on osoittautunut lupaavaksi tekniikaksi, ja siitä on jo toteutettu useita erilaisia muunnelmia. Jakob Engblomin väitöskirjassaan [3] esittämä tekniikka perustuu analyysin modulaarisuuteen. Kaikki moduulit käyttävät yhteisiä rajapintatietorakenteita ja siksi yksittäinen moduuli on helppo korvata toisella esimerkiksi kohdeprosessoria vaihdettaessa.

Laskettaessa koko ohjelman suoritusaikaa peruslohkojen suoritusaikojen perusteella yksinkertainen yhteenlasku tuottaisi liian konservatiivisen arvion. On otettava huomioon myös nk. *ajoitusvaikutukset*, sellaiset muutokset suoritusaikaan, jotka syntyvät suoritettaessa peräkkäin tietyt peruslohkot. Jos ajoitusvaikutus syntyy kahdesta peruslohkosta, puhutaan pareittaisesta vaikutuksesta, muuten pitkäaikaisesta. Ajoitusmalli on rajapintatietorakenne, joka sisältää tiedot sekä peruslohkojen suoritusajoista että kaikista mahdollisista ajoitusvaikutuksista. Kaikki mahdolliset vaikutukset on otettava huomioon siksi takia, että jotkin pitkäaikaiset ajoitusvaikutukset voivat olla negatiivisia. Mallia käytetään yhdessä polkukaavion kanssa laskettaessa ohjelman lopullista pahinta mahdollista suoritusaikaa.

Ajoitusmallin luomiseen tarvitaan tarkempaa tietoa liukuhihnan toiminnasta. Ajoituksen kuvaamiseen matalalla tasolla käytetään eräänlaista rajoitemallia. Malli kuvaa eri käskyjen välisiä yhteyksiä ja muistuttaa ajoitusmallia käskytasolle. Se ottaa huomioon mahdolliset

hasardit ja anomaliat ja sitä voidaan käyttää myös moniliukuhihnajärjestelmien kanssa. Rajoitemallin luominen on monimutkainen tapahtuma ja sen yksityiskohdat riippuvat mallinnettavasta laitteistosta.

Ajoitusmalli pystytään luomaan suoraan rajoitemallin perusteella. Tämän *ajoitusanalyysin* toteuttava moduuli on riippumaton laitteistosta. Analyysissa käydään läpi ensin kaikki peruslohkot ja niiden mahdolliset parit. Sen jälkeen päätellään pitkäaikaiset ajoitusvaikutukset *laajennusehtojen* avulla.

Analyysiä kokeiltiin kahdella RISC-suorittimella, NEC V850E:lla ja ARM9:lla. Ajoitusvaikutusten huomioonottaminen tiukensi WCET-arviota keskimäärin noin 20%:lla. Tällä tavalla saadut WCET-arviot olivat lähes yhteneviä todellisen suoritusajan kanssa.

3.4 WCET-analyysi

WCET-analyysi koostuu:

- Vuoanalyysistä, joka analysoi ohjelman lähdekoodia ja/tai objektikoodia ja määrittää ohjelman mahdolliset vuot.
- Globaalista matalan tason analyysistä, joka määrittää globaaleista laiteriippumattomista tekijöistä, kuten välimuisteista, johtuvat vaikutukset ohjelman ajoitukseen. Parempien tulosten saamiseksi voidaan käyttää myös vuoanalyysin tuloksia.
- Paikallisesta matalan tason analyysistä, joka määrittää koneriippuvaisista tekijöistä johtuvat vaikutukset ohjelman ajoitukseen. Voi käyttää edellisiä menetelmiä hyväkseen parantaakseen tuloksia.
- Laskennasta, joka yhdistää kolmen muun komponentin tulokset ja tuottaa todellisetn WCET-arvion.

3.4.1 Vuoanalyysi

Vuoanalyysi koostuu vuon määrittämisestä, vuon kuvaamiseta ja laskentaan valmistautumisesta.

Vuon määritteleminen tarkoittaa itse koodin analysointia. Vuoinformaatio voidaan laskea manuaalisesti ja lähdekoodiin täytyy lisätä annotaatioita tai antaa tämä informaatio koodin ulkopuolella. Tähän on käytetty erilaisia menetelmiä, mm. laajentamalla C:n syntaksia, käyttmällä $\#pragma$-määrittelyjä, käyttämällä säännöllisiin lausekkeisiin perustuva IDL-kieltä tai lineaarisia rajoitteita. Engblom on käyttänyt vuofaktakieltä kuvaamaan ohjelmavuota objektikooditasolla, missä rajoitteet ovat lokaaleja pienelle osalle ohjelmaa.

Vuon määrittelemiseen automaattisesti tarvitaan esimerkiksi symbolista suorittamista, abstraktia tulkintaa tai symbolista käskytason objektikoodin simulointia. Tämäkin menetelmä toimii paremmin käytäjän antamalla vihjeistyksellä.

Lähdekooditasolta saatu ohjelmavuoinformaatio on vaikeaa sovittaa objektikooditasolle, koska kääntäjä voi tehdä radikaaleja muutoksia koodille, kuten avata silmukoita, tehdä

inline:ä funktioille ja duplikoida koodia. Engblom[3] tutki tähän ongelmaan ratkaisua diplomityössään vuonna 1997. Muita tapoja on esim. käyttää *gcc*:n sisäistä debug-informaation leviämispalvelua tähän tarkoitukseen tai olettaa, että kääntäjä säilyttää nimiöt (label) jotka määrittävät olennaiset paikat ohjelmakoodissa ja tehdä tunnistus näiden avulla. Myös tutkimalla sitä, miten koodi muuttaa tiettyä muuttujaa, on mahdollista tunnistaa tietyt osat koodista. Helpointa olisi tehdä analyysi kääntäjän sisällä.

3.4.2 Matalan tason analyysi

Ohjelmakoodia on analysoitava, jotta saataisiin oikea ajoituskäyttäytyminen selville. Tätä kutsutaan matalan tason analyysiksi (low-level analysis).

Globaalissa matalan tason analyysissä tarvitaan arvioita raudan toiminnasta, koska tarkka analyysi on tavallisesti mahdotonta. Arvion on oltava aina turvallinen. Käskyvälimuistit ovat helppoja analysoida, koska käskyn haku (IF)-käyttäytyminen voidaan määritellä ohjelmavuosta. Datavälimuistit ovat monimutkaisempia analysoida, koska datan käyttöä on vaikeaa määrittää staattisesti.

Lokaalissa matalan tason analyysissä käsitellään ajoitusvaikutuksia (timing effects), jotka johtuvat yksittäisestä käskystä ja sen välittömistä naapureista. Lähes kaikki lokaalin matalan tason analyysi keskittyy liukuhihnojen analysointiin. Arvioita tarvitaan tälläkin tasolla usein, mutta paikalliset seuraukset ovat usein yksinkertaisempia, joten arvioiden tarkkuus on parempi.

3.5 Mittaus vaihtoehtona staattiselle analyysille

Julkaisu [4] lähestyy pahimman mahdollisen suoritusajan ongelmaa täysin toisella tavalla. Sen mukaan nykyiset WCET-ongelmat ovat kompleksisuudeltaan sitä tasoa, että niitä on hyvin vaikea ratkaista staattisella analyysillä. Tämän takia arvioita ei pitäisikään tehdä staattisen analyysin avulla vaan suoraan mittaamalla suoritusajat.

Kääntäjältä pyydetään CFG (kontrollivuokaavio), jota se käytti optimoidessaan koodia. Koska kaikkien mahdollisten suoritusten mittaaminen on yleisesti liian hankalaa, kontrollivuokaaviosta pyritään karsimaan pois suurin osa mahdottomista poluista *kontrollivuoanalyysillä*. Jos kaavio on edelleen liian monimutkainen, se joudutaan jakamaan useampaan palaseen, jotka käsitellään itsenäisesti. Tämä heikentää arvion laatua melkoisesti.

Menetelmässä otetaan lisäksi huomioon käyttöjärjestelmän toiminnan ja häiriöiden mahdollisuudet muokkaamalla lopullista aikaa *turvakertoimen* avulla. Menetelmän hyvä puoli on nopea siirrettävyys. Testialustana on käytetty REAR-moniprosessorijärjestelmää, jossa on erikseen reaaliaikayksikkö (MIPS R4600) ja suoritusnopeusyksikkö (Intel Pentium II). Tulokset ovat olleet melko tarkkoja.

4 Yhteenveto

Kuvatuissa liukuhihna-analyysin menetelmissä on hyvin paljon yhtäläisyyksiä mutta myös eroja. Jonkinlainen polku- tai kontrollivuoanalyysi lienee pakollinen jokaisessa analyysime-

nettelyssä. Lisäksi yhteistä kaikissa järjestelmissä oli modulaarisuus ja laitteistoriippumattomuus: hyvin tehty analyysiohjelma on helppo muokata tarvittaessa käsittelemään toisenlaista prosessoria.

Eroavaisuudet eri menetelmien välillä ovat lähinnä käsittelyjärjestyksessä ja eri osaalueiden priorisoinnissa. Keskinäisistä tehokkuuseroista on vaikea todeta mitään. Käsitellyt julkaisut eivät valitettavasti viittaa juurikaan toisiinsa, eivätkä ne sisällä keskinäisiä tehokkuusvertailuja. Polkuanalyysin ja mittauksen yhdistelmää vaihtoehtona staattiselle analyysille kannattaa luultavasti harkita silloin, kun halutaan saada melko tarkkoja tuloksia vähällä vaivalla, tai silloin, kun staattinen analyysi on liian vaikea toteuttaa.

Viitteet

[1] Ferdinand Christian, Heckmann Reinhold, Langenbach Marc, Martin Florian, Schmidt Michael, Thesing Stephan. *Reliable and Precise WCET Determination for a Real-life Processor*. 2001. Proc. First International Workshop on Embedded Software (EMSOFT 2001), LNCS 211. Springer-Verlag.

[2] Lundqvist Thomas, Stenström Per. *An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution*. 1999. Real-Time Systems, 17(2/3):183-207, 1-27. Kluwer Academic Publishers, Boston.

[3] Engblom Jakob. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. 2002. ACTA Universitatis Upsaliensis.

[4] Petters Stefan, Färber G. *Making Worst-Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible*. 1999. Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99). IEEE Computing Society Press.

[5] Theiling Henrik, Ferdinand Christian. *Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis*. 1998. Universität des Saarlandes / Fachbereich Informatik.

[6] Liedes Antti-Pekka, Kantee Antti. *Mikroprosessorien liukuhihnat staattisen analyysin näkökulmasta*. Tämä julkaisu.

Sulautetun järjestelmän energiankulutus ohjelmiston näkökulmasta

Oskar Kohonen
Sampo Vuorinen

1 Johdanto

Sulautettua tietotekniikkaa sisältävien kannettavien laitteiden nopea yleistyminen on tehnyt energiankulutuksesta tärkeän näkökohdan digitaalisten järjestelmien suunnittelussa. Suoritamme lyhyen katsauksen energiankulutukseen ja sen hallintaan liittyviin tekniikoihin. Tutustumme mihin virtaa kuluu, mitkä komponentit kuluttavat eniten ja millaisia akkuja on käytettävissä. Tämän jälkeen tutkimme, miten näitä tekijöitä voidaan hallita ohjelmiston avulla. Tarkastelemme miten käyttöjärjestelmän toteutus vaikuttaa, kääntäjiä jotka optimoivat ohjelmia energiankulutuksen suhteen ja menetelmiä, joilla ohjelmistojen energiankulutusta voidaan analysoida.

2 Perustekniikka

2.1 Elektroniikka – Miten ja miksi virtaa kuluu

Nykyaikaiset mikroprosessorit, muistit ja muut integroidut digitaaliset piirit perustuvat pääosin CMOS-teknologiaan (Complementary Metal Oxide Semiconductor), jossa piialustalle tuotetaan kanavatransistoreita (MOSFET) litografia- ja metallihöyrystystekniikalla. Transistorit toimivat jänniteohjattuina kytkiminä. Digitaalitekniikan peruskomponentti, looginen invertteri voidaan muodostaa kahdesta kanavatransistorista, ja niitä yhdistelemällä saadaan aikaan muut loogiset portit. CMOS-teknologia on valmistusmenetelmineen hyvin tunnettu, ja se on tämänhetkisistä IC-teknologioista tehokkain sekä energian- että tilankäytön suhteen. [1, luku 13.1] Nämä ominaisuudet ovat erityisen toivottavia sulautetuissa järjestelmissä, joilta vaaditaan usein kykyä toimia verkkovirrasta riippumatta.

Loogisen invertterin tehonkulutus jakautuu *staattiseen* osaan, johon sisältyvät erilaiset piirin tilanmuutoksista riippumattomat häviöt, ja *dynaamiseen* osaan, joka vastaa piirin tilanmuutokseen suoranaisesti tarvittavaa energiaa. Yksi CMOS-teknologian huomattava piiritekninen etu on, että sen staattinen kulutus on teoreettisesti nolla ja se voidaan käytännössäkin saada huolellisella suunnittelulla merkityksettömän pieneksi. [1, luku 5.8] CMOS-invertterin dynaamiselle tehonkulutukselle pätee

$$P_D = fCV_{DD}^2$$

missä f on tilanmuutosten taajuus (”kellotaajuus”), C piirin sähköisistä ominaisuuksista johtuva kuormakapasitanssi ja V_{DD} käyttöjännite. Kuormakapasitanssi vastaa sähköisesti

piiriin kytkettyä kondensaattoria. Kun invertterin sisääntulo vaihtaa tilaansa korkeasta jännitetasosta matalaan, ulostulon jännite nousee vastaavasti. Tällöin teholähteen energiaa kuluu ”virtuaalisen” kondensaattorin varaamiseen. Toiseen suuntaan tapahtuva muutos taas purkaa kondensaattorin varauksen, jolloin purkausvirta maadoittuu toisen transistorin kautta ja sen energia muuttuu hukkalämmöksi.

Kaavasta havaitaan, että minkä tahansa parametrin arvon pienentäminen vähentää piirin tehonkulutusta. Käyttöjännite on tässä suhteessa erityisen houkutteleva kohde, koska tehonkulutus riippuu siitä neliöllisesti eikä toimintataajuutta ja sitä kautta suorituskykyä tarvitse välttämättä alentaa. Käytännössä kuitenkin taajuutta joudutaan usein alentamaan käyttöjännitteen mukana, koska jännitteen laskiessa kuormakapasitanssin varaaminen vaatii enemmän aikaa ja signaalien nousunopeus hidastuu.

2.2 Prosessoreiden ja komponenttien energiankulutus

Komponenttien energiankulutuksen keskinäinen suhde on tärkeä huomio, koska jos esim. onnistumme säästämään 50 % muistin käyttämästä energiasta, mutta sen suhteellinen osuus järjestelmämme energiankulutuksessa on alle prosentin, niin kokonaissäästömme on hyvin marginaalinen. Kannettavien tietokoneiden komponenttien suhteellista energiankulutusta on tutkittu jonkun verran. Lorch [18] mittasi vuonna 1995 Apple Macintosh -kannettavia. Koneet joiden spesifikaatiot ovat alla olevassa taulukossa ovat nykymittapuun mukaan vanhanaikaisia.

Machine	Features
Duo 230	68030 processor; 33 MHz top speed; supports 16 MHz operation; internal hard drive either 80 MB, 120 MB, or 160 MB; 9” backlit supertwist monochrome display with 16 levels of gray; optional internal modem; trackball; keyboard; 4MB RAM expandable to 24 MB
Duo 270c	68030 processor with 68882 math co-processor; 33 MHz top speed; supports 16 MHz operation; internal 240 MB hard drive; 8.4” backlit active-matrix color display; optional internal modem; trackball; keyboard; 4 MB RAM expandable to 32 MB
Duo 280c	68LC040 processor; 33 MHz top speed; internal 320 MB hard drive; 8.4” backlit active-matrix color display; optional internal modem; trackball; keyboard; 4 MB RAM expandable to 40 MB

Alla on taulukko yllä mainittujen kannettavien tietokoneiden komponenttien energiankulutuksesta kun kaikki virransäästöominaisuudet ovat poissa käytöstä.

Component	Duo 230		Duo 270c		Duo 280c	
Processor	1.15 W	(14.67%)	1.15 W	(11.41%)	3.33 W	(27.57%)
Hard drive	0.96 W	(12.24%)	1.22 W	(12.10%)	1.48 W	(12.25%)
Backlight	2.38 W	(30.36%)	3.21 W	(31.85%)	3.40 W	(28.15%)
Display	0.20 W	(2.55%)	0.75 W	(7.44%)	0.75 W	(6.21%)
Modem	0.58 W	(7.40%)	0.51 W	(5.06%)	0.58 W	(4.80%)
Sound	0.19 W	(2.42%)	0.19 W	(1.88%)	0.19 W	(1.57%)
FPU	N/A	(0.00%)	0.46 W	(4.56%)	N/A	(0.00%)
Video	0.35 W	(4.46%)	0.46 W	(4.56%)	0.46 W	(3.81%)
Power management	0.12 W	(1.53%)	0.12 W	(1.19%)	0.12 W	(0.99%)
Memory	0.06 W	(0.77%)	0.06 W	(0.60%)	0.06 W	(0.50%)
Reducible miscellaneous	1.19 W	(15.13%)	1.23 W	(12.24%)	0.41 W	(3.37%)
Nonreducible miscellaneous	0.66 W	(8.47%)	0.72 W	(7.11%)	1.30 W	(10.79%)
Total (Type III battery life)	7.84 W	(2.16 hr)	10.08 W	(1.68 hr)	12.08 W	(1.40 hr)

2.3 Akkutekniikka

Akkujen rajoittunut kapasiteetti on ollut yksi tärkeä tekijä ohjaamaan kehitystä kohti energiaa säästeliäämmin käyttäviä laitteita. Samaan aikaan kun tietotekniikka on kehittynyt nopeasti, akkujen kehitys on ollut huomattavasti hitaampaa. Lorch et al. [12] raportoivat vuonna 1998 parhaaksi akkukapasiteetiksi per tilavuus 380 Wh/l litium-ionitekniikalla toteutetulla akulla. Vanhempien nikkeli-metallihydridiakkujen kapasiteetti on noin puolet tästä. Kapasiteetin lisäksi akun suorituskykyyn liittyy suuresti *relaksaatioaika* virtapulssien välillä [2]. Akku kestää paremmin jos sitä kuormitetaan pienellä kertakuormituksella (*duty cycle*) mutta suurella taajuudella.

3 Käyttöjärjestelmän rooli energianhallinnassa

Energianhallintaa voi tehdä monella tasolla:

- Komponenttitasolla, eli yksittäiset komponentit hoitavat itse energianhallintansa
- Käyttöjärjestelmätasolla
- Sovellustasolla
- Käyttäjätasolla

Ideaalista olisi, että energianhallinta sijoitettaisiin mahdollisimman korkealle tasolle, koska matalammilla tasoilla on vähemmän tietoa järjestelmän yleiskuormasta. Useimmat energianhallintaan liittyvät tehtävät ovat sopimattomia käyttäjän suoritettavaksi, koska käyttäjällä ei ole tarvittavaa tietoa hoitaakseen niitä, eikä käyttäjä pysty reagoimaan millisekuntien viiveillä. Sovellustason ongelma on, että sovellukset toimivat itsenäisesti, ja niiltä puuttuu tiettyä kriittistä tietoa, jonka käyttöjärjestelmä sulkee sisäänsä. Näistä syistä johtuen suurin osa energianhallinnassa tehdään käyttöjärjestelmätasolla käyttäjän tehdessä korkean tason päätöksiä ja sovellusten yrittäessä käyttää vähemmän komponentteja. On olemassa malleja joissa sovellukset antavat tietoa käyttöjärjestelmälle resurssitarpeistaan; täten voidaan yhdistää sovellustason että käyttöjärjestelmätason energianhallinnan edut. [12]

3.1 Skedulointi

Eräs käyttöjärjestelmän perustehtävistä on jakaa laitteiston resurssit eri prosessien välillä. Yleensä skedulointi optimoidaan mahdollisimman suuren kapasiteetin tai mahdollisimman pienen latenssin saavuttamiseksi, eikä oteta huomioon tehonkulutusta, vaikka olisi mahdollista säästää tehoa huomattavasti eri skeduloinnin osa-alueilla.

3.1.1 Prosessoriskedulointi

On olemassa pääosin kaksi tapaa vähentää prosessorin energiankulutusta: Prosessorin saattaminen tehonsäästötilaan ja kellotaajuuden, ja sitä kautta käyttöjännitteen laskeminen. Tehonsäästötilassa vain osa prosessorin osista pidetään aktiivisina. Toteutus on kompromissi energiansäästön ja tilamuutoksen aiheuttaman viiveen välillä. Sovelluksissa, joissa pieni viive on ensiarvoisen tärkeää, tehonsäästötiloja on vaikea hyödyntää. Kellotaajuuden dynaaminen laskeminen on houkuttelevaa, koska se sallii jännitteen laskemisen, ja tehonkulutus on suhteellinen jännitteen neliöön. Kellotaajuuden muuttaminen aiheuttaa kuitenkin sekä viivettä että ylimääräistä tehonkulutusta, joten sen soveltamisessa on oltava varovainen. Reaaliaikajärjestelmässä kellotaajuuden laskeminen muuttaa tietysti skedulointiongelmaa, kun prosessorin nopeus ei ole vakio.

On mahdollista laajentaa *Fixed Priority Scheduling* -algoritmia siten, että samat reaaliaikaisuusvaatimukset pätevät, vaikka energiaa säästyykin. Fixed Priority Scheduling on skedulointimenetelmä jossa on kaksi jonoa: Run-jono, jossa prosessit jotka ovat valmiita suoritukseen odottavat, sekä Delay-jono, jossa jo ajaneet prosessit ovat odottamassa seuraavan ajojaksonsa alkamista. Run-jono on prioriteettijono ja algoritmi takaa, että korkeimman prioriteetin omaava prosessori saa prosessorin käyttöönsä. Skeduloitavuusanalyysiin käytetään *Rate Monotonic Analysis* -menetelmää, jossa prosessien suoritusajoista ja deadlineista muodostuu jaksoja jotka ovat keskenään samanlaisia, jolloin riittää analysoida yksi jakso. *Low Power Fixed Priority Scheduling* sitä että prosessit usein eivät käytä skeduloitavuusanalyysissa käytettävää *Worst-Case Execution Time* -aikaa. Tämän seurauksena skedulointijakson loppuun jää aikaa jolloin prosessori on inaktiivinen. Low Power Fixed Priority Scheduling havaitsee kun jakson viimeinen prosessi on vuorossa ja laskee kellotaajuutta niin että prosessi ehtii juuri valmistua ennen seuraavan jakson alkua. Jos kaikki prosessit ehtivät valmistua, prosessori siirretään energiansäästötilaansa, kunnes seuraava jakso alkaa. Menetelmää sovellettiin neljään todelliseen reaaliaikajärjestelmään, ja se saavutti näissä 10–60 % energiansäästön. [13]

3.1.2 Tiedostojärjestelmäskedulointi

Perustasolla tiedostojärjestelmäskeduloinnin tehonsäästö tapahtuu siten, että kiintolevy pysäytetään, kun se on ollut poissa käytöstä tietyn aikaa. Parempi ratkaisu olisi suunnitella tiedostojärjestelmä siten, että se ottaa huomioon mahdollisuuden, että levy on pysähtynyt. Esimerkiksi skeduleri voisi puskuroida matalan prioriteetin levykyselyjä kunnes levy on jo käynnissä. [2] Monissa järjestelmissä on myöskin mahdollista korvata kiintolevy flash-muistilla.

3.1.3 Tietoliikenneskedulointi

Matalan tason protokollat voidaan suunnitella vähän energiaa kuluttaviksi, pääosin siten että ne yrittävät pitää verkkolaitteiston päällä mahdollisimman lyhyen ajan. Laitteet jotka voidaan kytkeä moneen erilaiseen verkkoon voi optimoida siten, että ne skeduloivat vain korkeimman prioriteetin yhteyksiä kalliisiin verkkoihin (esim. matkapuhelinverkko), kun taas matalamman prioriteetin asiat odottavat kunnes laite kytketään halvempaan verkkoon (esim. lähiverkko). [2]

4 Kääntäjätekniset ratkaisut

Kääntäjäteknisillä ratkaisuilla voidaan vaikuttaa huomattavasti energiankulutukseen. Hsu ja Kremer [15] ovat tehneet kääntäjän, joka laskee prosessorin kellotaajuutta muistirajoitetuissa osissa ohjelmasta. Prosessorin kellotaajuutta ja sen mukana jännitettä muuttavia algoritmeja kutsutaan DVS-algoritmeiksi (Dynamic Voltage Scaling). DVS-algoritmeista on käännösaikaisten off-line-versioiden lisäksi olemassa on-line-versiot, jotka toimivat ajonaikaisesti. Off-line-algoritmeja käytetään pääosin reaaliaikajärjestelmissä, kun taas on-line-algoritmeja interaktiivisten tehtävien energiankulutuksen optimointiin. Muistirajoitetussa ohjelman osassa suurinta osaa prosessorin kellojaksoista ei voida hyödyntää, koska prosessorin on odotettava muistipiirin toimintaa. Tästä johtuen voidaan laskea prosessorin kellotaajuutta näissä osissa suorituskyvyn juuri lainkaan kärsimättä. Hsu ja Kremer raportoivat simuloinnin tuloksina, että *SPECfp95*-benchmark kuluttaa koodimuunnoksen jälkeen 24 % vähemmän energiaa ja vain 2,7 % enemmän aikaa. Vastaava tulos mitattuna todellisella laitteistolla kannettavalla Athlon 600 MHz–1,2 GHz tietokoneella on 9,17–55,65 % energiansäästö kun aikaa kuluu 0,69–6,14 % enemmän. [16]

Yang et al. [17] ovat tutkineet käskyjen skeduloinnin vaikutusta energiankulutukseen. Monissa prosessoreissa käytetään *clock gating* -tekniikkaa, joka tarkoittaa että funktionaaliset yksiköt jotka eivät ole käytössä sulkevat input-kellonsa säästääkseen energiaa. Tällaisissa prosessoreissa on mahdollista vähentää tehonkulutusta siten, että skeduloidaan ne käskyt, jotka eivät ole kriittisellä polulla tai eivät käytä kriittistä resurssia, niin että ne käyttävät mahdollisimman harvoja funktionaalisia yksiköitä. Tuloksena saadaan koodia joka on yhtä nopeaa kuin ei muunnettu koodi, mutta kuluttaa huomattavan paljon vähemmän sähköä. Kaikki funktionaaliset yksiköt kuluttavat kuitenkin inaktiivisinakin jonkun verran sähköä, ns. hukkakulutusta. Tästä kulutuksesta voidaan päästä eroon käyttämällä *power supply gating*-tekniikkaa. Power supply gating tarkoittaa, että virtalähde yksinkertaisesti sulkee jonkun funktionaalisen yksikön kokonaan. Ongelmana on kuitenkin se, että yksikön tilanmuutos inaktiivisen ja aktiivisen tilansa välillä vie huomattavan ajan, usein monta sataa tai jopa tuhansia kellojaksoja. Siksi on hyödyllistä tehdä staattista analyysia jolla saadaan koko ohjelmaa, tai suuren osan ohjelmaa koskevia tietoja eri funktionaalisten yksiköiden aktiivisuudesta. Käskyjen skedulointiongelma voidaan muuntaa ILP (Integer Linear Programming) -ongelmaksi, joka ratkaisemalla saadaan aikaan tarvittava koodimuunnos. Kokeellisina tuloksina (perustuvat simulointiin) mainitaan parhaimmillaan 15,4 % (keskimäärin 8,5 %) dynaamisen energiankulutuksen väheneminen liukulukuyksiköissä NASAn kerneleillä ilman

että suorituskyky laskee. Jos olisi käytettävissä täydellinen power supply gating -mekanismi, ILP:n tuottamilla koodimuunnoksilla voitaisiin saavuttaa jopa 51,5 % vähennys hukkakulutukseen (keskimäärin 31,8 %).

5 Ohjelma-analyysitekniikat

Ohjelmistojen energiankulutus on verraten uusi tutkimusalue. Sen tarpeellisuus alkoi ilmetä 1990-luvun alkupuolella sulautettujen järjestelmien nopean esiinmarssin myötä. Vuonna 1994 Tiwari et al. [3] julkaisivat ensimmäisen nimenomaan ohjelmiston näkökulmasta tehdyn tutkimuksen. Tätä ennen kulutusta oli tarkasteltu pelkästään piiritekniikan tasolta. Loogisten porttien tai yksittäisten transistoreiden tasolla toimivia analyysityökaluja oli olemassa, mutta ne eivät mahdollistaneet järkevällä tavalla ohjelmistokomponentin tarkastelua.

Tiwarin et al. jälkeen ala alkoi kehittyä nopeasti, ja analyysimenetelmät jakautuivat kahteen päälinjaan eli mittaus- ja simulaatiopohjaisiin. [5] Tässä luvussa esitetään esimerkkejä kummastakin; mittauspohjaiset menetelmät jaetaan vielä abstraktiotasonsa perusteella konekäsky- ja funktiotason menetelmiin.

5.1 Konekäskytason analyysi

Yksittäisten konekielikäskyjen tasolla tapahtuva analyysi on menetelmistä vanhin, eräänlainen ”perustaso”. Siinä pyritään mallintamaan prosessorin eri käskyjen energiankulutus mahdollisimman yleispätevästi ja käyttämään tätä mallia perustana staattiselle ohjelma-analyysille, jonka tulos on periaatteessa konekielisessä syöteohjelmassa käytettyjen käskyjen energiankulutusten summa.[8] Menetelmän huomattavana etuna on, että prosessorin rakenteesta ei tarvita tarkkoja piiritason tietoja, vaan malli voidaan laatia kokeellisesti mittausten perusteella. Verrattuna esim. simulointiin perustuviin menetelmiin se on myös laskennallisesti kevyt ja soveltuu hyvin integroitavaksi erilaisiin ohjelmankehitystyökaluihin. Menetelmän ongelmat liittyvät pitkälti yksittäisten käskyjen näkökulman kapeuteen järjestelmän tilan esittäjänä; nykyaikaisissa prosessoreissa on rakenteita (esim. välimuistit, liukuhihnat), joiden vaikutuksesta lähekkäiset käskyt saattavat vaikuttaa toistensa tehonkulutukseen (*inter-instruction effects*), eikä tietyn käskyn kulutus ole piirin erilaisen sisäisen tilan vuoksi välttämättä aina sama (*circuit state effects*). Näiden ilmiöiden huomioon ottaminen muodostaa merkittävän haasteen mallin laatimiselle.

5.1.1 Peruskustannusmalli

Tiwari et al. [3] mittasivat kahden yleisen prosessorin (Intel 486DX2 ja Fujitsu SPARClite 934) kuluttamaa virtaa lyhyillä testiohjelmilla, joissa kulloinkin tutkittavia käskyjä ajettiin

[8]Staattisen ohjelma-analyysin yleisiä tekniikoita kuten peruslohkojen muodostamista ja kontrolli- ja silmukkarakenteiden seuraamista ei ole tarkoituksenmukaista käsitellä tässä lähemmin. Niitä on tietenkin sovellettava, jotta ohjelman kontrollivuo (ts. mitkä käskyt suoritetaan, missä järjestyksessä ja kuinka monta kertaa) saadaan selville ja tekstissä mainittu summa pystytään laskemaan. Ohjelma-analyysin perusteita käsitellään kääntäjätekniikan oppikirjoissa, esim. [14].

silmukassa stabiilin mittaustuloksen saamiseksi. Näin kullekin käskylle saatiin virtamääräinen *peruskustannus* (yksikkö mA). Kun piirin käyttöjännite tunnetaan, teho saadaan jännitteen ja virran tulona ja energia vastaavasti tehon ja käskyn suoritusajan tulona (käskyjen suoritusajat ilmaistaan datalehdissä usein kellojaksoina). Yksinkertaisimmillaan ohjelman energiankulutus voitaisiin arvioida summaamalla käskyjen energiaksi muutetut peruskustannukset. Näin saatu arvio on kuitenkin epätarkka edellä mainittujen riippuvuusilmiöiden vuoksi. Malliin kuuluukin eräitä korjaustekijöitä (*lisäkustannuksia*), joilla merkittävät epätarkkuuden aiheuttajat pyritään ottamaan huomioon.

Käskyjen keskinäisvaikutusten tutkimiseksi mitattiin peräkkäisten käskyjen virrankulutusta kahden käskyn ryhmissä ja tuloksia verrattiin yksittäisten käskyjen peruskustannuksiin. Tällä tavoin N käskyn käskykannalle tehty täydellinen kustannustaulukko sisältäisi N^2 arvoa, mikä on varsin epäkäytännöllistä. Havaittiin kuitenkin, että esimerkiksi 486DX2:n tapauksessa keskinäisvaikutusten aiheuttama peruskustannusten ylitys vaihteli varsin kapeissa rajoissa 15 mA:n ympärillä. Riittäväksi korjaukseksi osoittautui siten lisätä 15 mA ohjelmalle laskettuun keskimääräiseen virrankulutukseen.

Liukuhihnan pysähdyksistä (*stalls*) johtuvia kustannuslisiä tutkittiin testiohjelmilla, joiden tiedettiin aiheuttavan pysähdyksiä liukuhihnan eri vaiheissa. Erityyppisille pysähdystilanteille mitattiin keskimääräiset virrankulutukset kellojaksoa kohti. Vastaavasti mitattiin välimuistin ohitusten (*cache misses*) aiheuttama kustannus, joka esim. 486DX2:lle oli keskimäärin 216 mA/jakso.

Varsinainen konekielisten ohjelmien analysointi tapahtuu tarkoitukseen tehdyllä ohjelmistolla. Analyysi alkaa peruslohkojen etsinnällä. Peruslohkoille lasketaan peruskustannukset[9] ja niistä etsitään liukuhihnan pysähtymisiä aiheuttavat kohdat[10], joiden aiheuttamat lisäkustannukset lisätään peruskustannuksiin. Peruslohkojen suorituskertojen määrä selvitetään koodin profiloinnilla, ja tästä tiedosta lasketaan koko ohjelman yleiskustannus. Välimuistin ohitusten määrä arvioidaan välimuistisimulaation avulla, ja lopullinen tulos saadaan lisäämällä välimuistin lisäkustannukset edellä laskettuun yleiskustannukseen.

Menetelmän keskimääräiseksi tarkkuudeksi todetaan vertailevien mittausten perusteella noin 3 %, jota on pidettävä huomattavan hyvänä tutkimuksen pioneeriluonteeseen nähden. Merkittävästi suurempiin tarkkuuksiin ei toistaiseksi ole päästy millään menetelmällä; joissakin myöhemmissä tutkimuksissa (esim. [6]) saavutetut tarkkuudet ovat olleet selvästi huonompia. Toisaalta on huomattava, että mainittu 3 % tarkkuus koskee (ilmeisesti) vain prosessoria. Keskusmuistin energiankulutusta on vaikeampi arvioida staattisen analyysin avulla, ja tässä suhteessa tulosten annetaan ymmärtää olevan lähinnä suuntaa-antavia.

[9] Tämä tapahtuu suoraviivaisesti summaamalla käskyjen peruskustannukset. Tarkkaan ottaen kullekin peruslohkolle lasketaan keskimääräinen peruskustannus (keskiarvo), josta saadaan lohkossa kuluva energia kertomalla se tarvittavalla suoritusajalla (kellojaksot).

[10] "*The number of stall cycles is estimated through a traversal of the program code*"; lähteessä ei selosteta tarkemmin liukuhihnan tapahtumien analysointiin käytettävää menetelmää.

5.1.2 Kulutusyhtälömalli

Lee et al. [5] kritisoivat peruskustannusmallin tyyppisiä, keskimääräiseen virtaan perustuvia arviointimenetelmiä siitä, että ne paljolti sivuuttavat prosessorin kulutuskäyttäytymisen yksityiskohdat. He laativat mallin, jossa energiankulutus abstrahoidaan lineaarisen yhtälön muotoon. Prosessorista valitaan tarkasteltavaksi kohteita (*resursseja*), joiden oletetaan vaikuttavan energiankulutukseen ja joiden tila on helppo ilmaista numeerisessa muodossa (esim. opkoodit, rekisterit). Resurssit valitaan ”lohkokaaviotasolta”, josta tietoa on julkisesti saatavilla prosessorien käsikirjoista; yksityiskohtaisia piiritason tietoja prosessorin rakenteesta ei siis tarvita. Resurssien tilat sijoitetaan riippumattomiksi muuttujiksi lineaariseen yhtälöön, jonka arvoksi määrätään tietyn kellojakson aikana kuluva energia. Liukuhihnan vaikutusten huomioon ottamiseksi yhtälö huomioi senhetkisen sekä edellisen kellojakson aikana kussakin liukuhihnan vaiheessa suorituksessa olleet käskyt. Laatimalla sopivia testiohjelmia, joiden vaikutus valittujen resurssien tiloihin tunnetaan, ja mittaamalla todellinen energiankulutus niitä ajettaessa voidaan yhtälön vakiokertoimet selvittää lineaarisen regressioanalyysin avulla.

Koska yhtälön riippumattomien muuttujien arvot ovat suoraan peräisin testiohjelmista ja siten etukäteen tiedossa, vakiokertoimet voidaan selvittää osajoukko kerrallaan laatimalla testiohjelmia, joissa esimerkiksi käskyhaut tehdään eri osoitteisiin tai jotka käyttävät eri rekistereitä, toiminnallisuuden pysyessä muutoin samana. Näin voidaan tutkia tietyn resurssin vaikutusta energiankulutukseen muusta mallista irrallaan ja laatia malli pienemmissä osissa, mikä helpottaa regressioanalyysin tekoa.

Kun prosessorikohtainen kulutusyhtälö on valmis, ohjelma-analyysin suorittaminen on verraten yksinkertaista. Analyysiohjelmisto selvittää konekielisen syöteohjelman kontrollivuon kulun staattisen ohjelma-analyysin perustekniikoita käyttäen ja laskee ohjelman käskyjen energiankulutukset sijoittamalla ohjelmakoodista saatavat resurssimuuttujien arvot yhtälöön.[11] Koska yhtälö antaa tulokseksi yhtä kellojaksoa koskevan kulutusarvion, koko ohjelman prosessointi vaatii varsin runsaasti iterointia. Laskut ovat kuitenkin suhteellisen yksinkertaisia ja koko analyysiohjelmisto lienee huomattavasti vähemmän kompleksinen kuin esimerkiksi Tiwarin et alin peruskustannusmallissa, koska erillistä liukuhihna- ja välimuistisimulaatiota ei tarvita.

Menetelmää on kokeiltu RISC-tyyppiselle ARM7TDMI-prosessorille ja saavutetuksi keskimääräiseksi tarkkuudeksi ilmoitetaan 2,5 %, joka edustaa alan kärkeä. Menetelmä vaikuttaa hyvin sopivalta ohjelmankehitystyökaluihin integroitavaksi, koska yhtälö tarjoaa siistin, abstraktin ja helposti käytettävän rajapinnan laitteiston käyttäytymiseen, eikä raskasta simulointia vaadita. Valitsemalla yhtälön muuttujat (resurssit) sopivasti voidaan lisäksi keskittyä tarkastelemaan kulutuskäyttäytymisen eri osia. Tarkastelua voitaisiin esimerkiksi siirtää käskytasolta arkkitehtuuritasolle valitsemalla resursseiksi prosessorin sisäisiä komponentteja (liukuhihnan eri osat, ALU:t, osoiterekisterit, jne), jolloin liukuhihnan ja muisti-

[11] ”*The analyzer – – extracts the values of the model variables by examining the instruction sequence*”; lähteestä ei selviä, kuinka paljon laskentaa muuttujien arvojen esillesaaminen syöteohjelmasta vaatii, mutta prosessi on luultavasti melko suoraviivainen, koska muuttujien edustamat resurssit ovat ohjelmoijalle näkyviä prosessorin osia.

järjestelmän vaikutusta energiankulutukseen voitaisiin arvioida tarkemmin.[12] Tämä ei kuitenkaan ole välttämättä tarkoituksenmukaista, mikäli menetelmää halutaan käyttää nimenomaan ohjelma-analyysiin. Tämänhetkinen tutkimus keskittyykin selvittämään, miltä abstraktiotasolta malli tulisi laatia, jotta se palvelisi parhaiten ohjelmien energiankulutuksen analysointia ja optimointia.

5.1.3 ”Quick’n’Dirty”

Russell ja Jacome [6] kokeilivat mahdollisimman yksinkertaista mallia, jossa prosessorin kaikkien käskyjen tehonkulutusta edustaa yksi ainoa parametri, keskimääräinen teho yhtä kellojaksoa kohden. Tällaisella mallilla staattinen ohjelma-analyysi on erityisen helppoa: syöteohjelmasta tarvitaan vain kellojaksoina ilmaistu arvio ohjelman suoritusajasta, jonka jälkeen arvio energiankulutuksesta saadaan kertomalla suoritusaika keskimääräisellä teholla.

Tällaisen mallin käyttökelpoisuus riippuu luonnollisesti sekä prosessorin ominaisuuksista että siitä, kuinka hyvin prosessorin tehonkulutusta todellisilla ohjelmilla kuvaava keskimääräinen teho onnistutaan määrittämään. Tutkimuksessa käytettiin Intelin RISC-tyyppisiä i960-prosessoreita, joiden tehonkulutuksen vaihtelu eri käskyjen välillä havaittiin tilastollisesti merkityksettömäksi. Näillä prosessoreilla mallin keskimääräiseksi tarkkuudeksi saatiin 8 %, joka jää selvästi jälkeen muista tässä esitellyistä malleista; esimerkiksi Tiwarin et alin mittausten perusteella tarkkuuden voisi olettaa olevan huomattavasti huonompi Intelin x86-perheen prosessoreilla, joiden tehonkulutus vaihtelee paljon suuremmissa rajoissa kuin i960:n.

Koska kulutusominaisuuksiltaan i960:n tapaisilla prosessoreilla ohjelman suorituksen vaatima energia riippuu pääasiassa ohjelman suoritusajasta, on suorituskyvyn suhteen optimoitu ohjelma optimaalinen myös energiankulutuksen suhteen. Tämä on sekä ohjelmoijan että kääntäjän tekijän kannalta mieluisa seikka. Yleisestikin ottaen näyttää pätevän, että nopea ohjelma on myös energiatalouden kannalta tehokas, mutta nämä ominaisuudet eivät välttämättä ole täysin yhtäpitäviä, mikäli järjestelmään on tehty erityisiä virransäästötiloja tai muita kulutusoptimointeja. [4]

5.2 Funktiotason analyysi

Koska käytännön ohjelmissa on yleensä varsin suuri määrä käskyjä, syöteohjelman tutkiminen käsky kerrallaan saattaa olla laskennallisesti raskasta. Toisaalta, kuten edellä on kerrottu, luotettavien konekäskytason mallien laatiminen on vaikeaa, koska prosessoreissa on runsaasti tehonkulutukseen vaikuttavaa sisäistä tilatietoa, joka ei käy ilmi yksittäisistä käskyistä.

Qu et al. [7] laativat konekäskytason menetelmien pohjalta analyysimenetelmän, jossa tarkastelu tapahtuu yksittäisten käskyjen sijasta kokonaisten funktioiden tasolla. Mene-

[12] ARM7TDMI-kokeilussa käytetyssä mallissa liukuhihnaa ja välimuistia ei ilmeisesti huomioitu mitenkään erikseen (vrt. [3]). Näyttää siltä, että prosessorin energiankulutus voidaan mallintaa yhtälömuodossa niin hyvin, ettei erityisjärjestelyitä liukuhihnan jne. suhteen tarvita.

telmä pohjautuu siihen, että suurin osa ajettavasta koodista kuuluu ohjelmoijan määrittelemiin funktioihin, ja jos näiden funktioiden aiheuttama tehonkulutus analysoidaan kerran ja ohjelma-analyysissä ”katetaan” analysoitava koodi tunnetuilla funktioilla, voidaan saavuttaa huomattava ajansäästö. Myös ohjelma-analyysin tarkkuus paranee, kun käskyjen keskinäisvaikutukset tulevat kokonaisia funktioita mitattaessa huomioiduiksi paremmin kuin yksittäisiä käskyjä tutkimalla.

Menetelmässä konekielinen koodi jaetaan neljään osaan: valmiisiin kirjastofunktioihin (toimitettu kääntäjän, käyttöjärjestelmän tms. mukana), sovelluskohtaisiin funktioihin, pääohjelmaan (*main()*) ja muihin (esim. ohjelman ajoympäristön alustava runtime-moduuli). Menetelmän ydin on tehotietokanta (*power data bank*), joka sisältää tiedot valmiiden kirjastofunktioiden ja prosessorin eri käskyjen tehonkulutuksesta. Käytännön toteutuksissa tietokannan luontevin toimittaja lienee prosessorin valmistaja. Tietokanta voidaan luoda mittauspohjaisesti tai simuloinnin avulla; käyttäjän näkökulmasta asialla ei ole väliä. Sovelluskohtaiset funktiot analysoidaan osittamalla ne kirjastofunktioiden kutsuihin (joiden tehonkulutus saadaan tietokannasta) ja omaan koodiin, jonka kulutus arvioidaan kuten konekäskytason menetelmissä.[13] Jos pääohjelma on hyvin yksinkertainen eikä sisällä toistorakenteita, se voidaan jättää ohjelma-analyysissä huomiotta; monimutkainen pääohjelma käsitellään kuten tavallinen sovelluskohtainen funktio. Muun koodin (runtime-moduulit jne.) osuus on useimmiten pieni ja sen osuutta ohjelman energiankulutukseen voidaan kuvata sopivalla vakiolla.

Ohjelma-analyysi alkaa syöteohjelman kääntämisellä lähdekoodista konekielelle. Kääntäjä tunnistaa kirjasto- ja sovelluskohtaiset funktiot symbolitaulusta ja laatii riippuvuuksia kuvaavan graafin, jossa sovelluskohtaiset funktiot ovat solmuja ja niiden väliset kutsut kaaria. Graafi järjestetään topologiseen järjestykseen, jossa pääohjelma (*main()*) on nielu (”viimeinen” solmu). Toisessa vaiheessa käännetyn ohjelman ajonaikainen käyttäytyminen (kuinka monta kertaa eri funktioita kutsutaan, jne.) selvitetään koodin profiloinnilla. Viimeisessä vaiheessa muodostetaan arvio ohjelman energiankulutuksesta läpikäymällä alussa muodostettu graafi. Solmuina olevat sovelluskohtaiset funktiot analysoidaan sovittamalla niiden rakennetta tehotietokannan tietoihin. Pääohjelmaa kuvaava solmu käsitellään viimeisenä, ja sen tuloksena saadaan arvio koko ohjelman kuluttamasta keskimääräisestä tehosta ja suoritusajasta.

Menetelmää on kokeiltu MIPS R3000A -prosessoriin perustuvalla sulautetulla Toshiba TX39-ytimellä, jonka kanssa on päästy keskimäärin 3,5 % tarkkuuteen. Tarkkuutta arvellaan voitavan vielä parantaa hienosäätämällä mallia. Lähteessä ei kerrota, kuinka nopeasti ohjelma-analyysi tapahtuu käytännössä. Mikäli menetelmä pitää suorituskyvystä annetut lupaukset, se voi olla hyvä valinta ohjelmankehitystyökaluihin liitettäväksi. Sidonnaisuus tiettyyn joukkoon kirjastofunktioita (ts. valmistajakohtaisuus) saattaa rajoittaa sen käyttökelpoisuutta itsenäisenä, erikseen hankittavana työkaluna.

[13] Tämä tapahtuu ”lennossa” ohjelma-analyysin aikana. Hieman epäselväksi jää, miten luvattu tarkkuusetu konekäskytason menetelmiin nähden säilytetään, jos koodin osia joudutaan analysoimaan käskytasolla. (”*We suggest to measure each user-defined function, at least those being frequently used, separately based on the structure of the function and the ’power data bank’.*”)

5.3 Simulointi

Simulointi muodostaa toisen merkittävän tehon- ja energiankulutuksen arviointimenetelmien luokan. Simuloinnin etuna on, että tarkkuus voidaan valita vapaasti tarpeen mukaan simulaation yksityiskohtaisuutta muuttamalla. Lisäksi se tarjoaa mittauspohjaisia menetelmiä suoraviivaisemman tavan ottaa huomioon sellaisten kohteiden vaikutus, joiden mallintaminen analyyttisesti on vaikeaa, kuten väylät ja muistijärjestelmät. Toisaalta simulointi on laskennallisesti raskasta ja hidasta verrattuna yksinkertaistettuja malleja käyttäviin mittauspohjaisiin menetelmiin, simulaattorit ovat itsessään laajoja ja monimutkaisia järjestelmiä, ja niiden laatiminen vaatii yksityiskohtaisia laitteistotason tietoja, joita laitevalmistajat eivät välttämättä julkaise. Mielestämme on perusteltua sanoa, että mittaus- ja simulaatiopohjaiset menetelmät eivät suoranaisesti kilpaile samoista käyttökohteista, vaan ne soveltuvat eri tarkoituksiin: edelliset ovat parhaimmillaan loppukäyttäjän (sovellusohjelmoijan) työkaluina ja jälkimmäiset tutkimuskäytössä.

Seuraavassa esitellään lyhyesti joitakin erilaisia simulointityökaluja. Esiteltävät ratkaisut on valittu antamaan käsitys sovellusalueen laajuudesta; paljon muitakin työkaluja on olemassa, eikä otokseen ole millään muotoa pyritty valitsemaan niistä ”tärkeimpiä”, mikä olisikin vaikeaa, koska työkaluista suurin osa on yhä akateemisen tutkimuksen asteella.

SoftWatt [8] on laaja ja massiivinen simulaattori, joka simuloi kokonaista tietokonetta prosessoreineen, muisteineen ja levyjärjestelmineen. Prosessorisimulaatio kattaa erilaisia MIPS-tyylisiä suorittimia (R4000, R10000). Simuloitu tietokone ajaa tavanomaista, ”oikeaa” käyttöjärjestelmää (IRIX 5.3), jossa puolestaan voidaan ajaa mitä tahansa kyseiselle käyttöjärjestelmälle saatavia ohjelmia. SoftWattin avulla voidaan tarkastella kokonaisen laitteistosta ja ohjelmistosta (ml. käyttöjärjestelmä) koostuvan järjestelmän tehonkulutuskäyttäytymistä. Ympäristönä SoftWatt on selvästi suunnattu sulautettujen järjestelmien sijaan ”täysimittaisten” tietokoneiden suuntaan. Yhdeksi kaavailluksi käyttökohteeksi esitetään Web-palveluntarjoajien järjestelmien kehittämistä vähemmän energiaa kuluttaviksi. Asialla on huomattavaa taloudellista merkitystä, koska tuhansia palvelimia ylläpitävän webhostingyrityksen tehonkulutus voi jäähdytysjärjestelmineen nousta kymmeniin megawatteihin.

Wattch [9] on tarkoitettu arkkitehtuuritason tehonkulutussimulointiin. Se ei itsessään ole varsinainen simulaattori, vaan erilliseen yleiskäyttöiseen arkkitehtoniseen simulaattoriin liitettävä lisäosa. Lähdeartikkelissa kerrotaan sen käytöstä *SimpleScalar*-simulaattoriin [11] liitettynä, mutta muitakin vastaavia alustoja voidaan käyttää. Arkkitehtuuri- eli ”lohkokaaviotaso” on valittu siksi, että simulaatio saadaan paljon nopeammaksi kuin piiritasolla (Wattchin luvataan olevan jopa yli tuhatkertaisesti olemassaolevia piiritason simulaattoreita nopeampi); myös simulaattorin sovittaminen uusille arkkitehtuureille on helpompaa, kun prosessorin rakenteen yksityiskohtia ei tarvitse tuntea. Wattchia voidaan käyttää monipuolisesti niin laite- kuin ohjelmistosuunnittelunkin apuna. Sen keräämiä metriikkatietoja ovat teho, energia, suorituskyky (kellojaksoja / ohjelma) ja ns. *Energy-Delay Product*, energiankulutuksen ja suoritusajan yhdistävä metriikka, jonka avulla voidaan karsia pois sellaiset ratkaisut, jotka joko kuluttavat vähän energiaa mutta ovat hitaita, tai ovat nopeita mutta paljon kuluttavia. Simulaatio keskittyy ainakin toistaiseksi prosessorin arkkitehtonisesti tärkeimpiin osiin, joten saavutetut tarkkuudet ovat pikemminkin riittävää kuin erinomaista tasoa

(n. 10 %).

JouleTrack [10] on StrongARM- ja Hitachi SH-4-prosessoreille sovitettu simulaattori, joka käyttää energiankulutuksen arviointiin hyvin yksinkertaista mallia. Näillä prosessoreilla kulutuksen vaihtelu eri käskyjen välillä on vähäistä, joten noin 7 % tarkkuuteen voidaan päästä kertomalla ohjelman suoritusaika keskimääräisellä teholla. (Vrt. [6].) Tätä kutsutaan ensimmäisen kertaluvun malliksi (first order model). Voidaan käyttää myös tarkempaa toisen kertaluvun mallia, jossa prosessorin käskyt on ryhmitelty muutamaan eri luokkaan keskimääräisen tehonsa suhteen. Lisäksi simulaatio mallintaa prosessorissa tapahtuvia vuotovirtailmiöitä (CMOS-tekniikan staattista tehonkulutusta). JouleTrackissa on Web-pohjainen käyttöliittymä, johon käyttäjä voi ladata ohjelmansa C-kielisenä lähdekoodina. Järjestelmä kääntää ohjelman ja ajaa sen ARM-simulaattorissa, joka tuottaa ajonaikaista statistiikkatietoa. Tämä tieto annetaan erilliselle ”arviointimoottorille” (estimation engine), joka laatii arvion ohjelman energiankulutuksesta simulaattorista saadun statistiikan perusteella.[14]

Viitteet

[1] A. S. Sedra, K. C.Smith. *Microelectronic Circuits*. Oxford University Press, 1998. Fourth Edition. ISBN 0-19-511690-9.

[2] Paul J. M. Havinga, Gerard J. M. Smit. *Design techniques for low power systems*. University of Twente, Department of Computer Science, 2000.
`<http://wwwhome.cs.utwente.nl/~havinga/papers/energy.design.techniques.jsa.pdf>`

[3] Vivek Tiwari, Sharad Malik, Andrew Wolfe. *Power Analysis of Embedded Software: A First Step Towards Software Power Minimization*. Princeton University, Department of Electrical Engineering, 1994.
`<ftp://ftp.ee.princeton.edu/pub/vivek/tvlsi94.ps>`

[4] Vivek Tiwari, Sharad Malik, Andrew Wolfe. *Instruction Level Power Analysis and Optimization of Software*. Princeton University, Department of Electrical Engineering, 1996.

[5] Sheayun Lee, Andreas Ermedahl, Sang Lyul Min, Naehyuck Chang. *Statistical Derivation of an Accurate Energy Consumption Model for Embedded Processors*.

[6] Jeffry T. Russell, Margarida F. Jacome. *Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors*. University of Texas at Austin, Department of Electrical and Computer Engineering.

[14] Järjestelmä muistuttaa kuvauksen perusteella enemmän edellisissä kappaleissa kuvattuja mittauspohjaisia, ts. ennalta laadittuihin kulutusmalleihin perustuvia työkaluja. Energiankulutusta ei siis simuloida ”suoraan”, vaan simuloinnilla selvitetään ohjelman ajonaikainen käyttäytyminen varsinaista energia-analyysiä varten; mittauspohjaisissa järjestelmissä kyseinen työvaihe suoritetaan staattisella ohjelma-analyysillä.

[7] Gang Qu, Naoyuki Kawabe, Kimiyoshi Usami, Miodrag Potkonjak. *Code Coverage-based Power Estimation Techniques for Microprocessors*. Journal of Circuits, Systems, and Computers, Vol. 11, No. 5 (2002) 1–18.

[8] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, N. Vijaykrishnan, Mahmut Kandemir. *Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach*. Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02), IEEE, 2002.

[9] David Brooks, Vivek Tiwari, Margaret Martonosi. *Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*. ISCA 2000, ACM, 2000.

[10] Amit Sinha, Anantha P. Chandrakasan. *JouleTrack – A Web Based Tool for Software Energy Profiling*. DAC 2001, June 18.–22.2001, ACM, 2001.

[11] D. Burger, T. M. Austin. *The SimpleScalar Tool Set, Version 2.0*. Computer Architecture News, p. 13–25, June 1997.

[12] Jacob R. Lorch, Alan Jay Smith. *Software Strategies for Portable Computer Energy Management*. University of California, IEEE Personal Communications Magazine vol 5 nr 3 p 60-73 Jun 1998

[13] Youngsoo Shin, Kiyoung Choi. *Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems*. School of Electrical Engineering, Seoul National University, Proceedings of Design Automation Conference, pp.134–139, 1999.

[14] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998. ISBN 0-521-58388-8.

[15] Chung-Hsing Hsu, Ulrich Kremer: *Compiler-Directed Dynamic Voltage Scaling Based on Program Regions*. Department of Computer Science, Rutgers University, Lecture Notes in Computer Science, Vol 2008, pp.65, November 2001

[16] Chung-Hsing Hsu, Ulrich Kremer. *Compiler-Directed Dynamic Voltage Scaling for Memory-Bound Applications*. Department of Computer Science, Rutgers University Technical Report DCS-TR498, August 2002

[17] Hongbo Yang, R. Govindarajan, Guang R. Gao, George Cai, Ziang Hu *Exploiting Schedule Slacks for Rate-Optimal Power-Minimum Software Pipelining*. Proceedings of The 3rd workshop on Compilers and Operating Systems for Low Power pp.5-1 2002

[18] Jacob Lorch *A complete picture of the energy consumption of a portable computer*. Masters thesis, Computer Science, University of California at Berkeley, 1995

Tehokas käännetty prosessorisimulaatio

Timo Lilja
Mikko Reinikainen

1 Johdanto

Prosessorisimulaattori on ohjelma, joka simuloi olemassa olevan tai olemattoman prosessoriarkkitehtuurin toimintaa. Prosessorisimulaattoreita tarvitaan prosessoriarkkitehtuurin suunnitellussa ja kehittämisessä: simulaattori mahdollistaa arkkitehtuurin tutkimisen ennen kuin ensimmäistäkään koeversioita prosessorista on tehty. Hyvä prosessorisimulaattori on nopea, mahdollistaa statistiikkojen keräämisen (kuten esimerkikksi käskyjen käytön jakauman), on laajennettavissa helposti uusilla käskykannoilla ja on helppo saada toimimaan muiden työkalujen (debuggerin, laitesimulaattorien) kanssa.

Simulaattorit voidaan jakaa kolmeen erilliseen ryhmään: käskykantasimulaattorit (*Instruction Set Architecture Simulator, ISA Simulator*) simuloivat vain prosessorin käskykantaa, logiikkasimulaattorit (*logic simulator*) simuloivat varsinaista laiteistototeusta, syklisimulaattorit (*cycle accurate simulator*) ovat jotain tältä väliltä: ne simuloivat käskykannan ja pystyvät tuottamaan tilastoja käskysykleistä [2].

Prosessorisimulaattorit voidaan myös jakaa seuraavasti: *tulkkaamiseen, staattiseen kääntämiseen* ja *dynaamiseen kääntämiseen perustuvat*. Tulkkaamiseen perustuvissa simulaattoreissa ohjelma tulkitsee jokaisen käskyn erikseen ja suorittaa käskyä vastaavat toiminnot joka käskylle erikseen. Staattiseen kääntämiseen perustuvissa tekniikoissa simulaattori ja simuloitava ohjelma käännetään etukäteen simulaatiota ajavan isäntäkoneen ymmärtämään muotoon, minkä jälkeen ohjelma ajetaan isäntäkoneella. Tällöin vältetään dynaamisesta tulkkaamisesta tuleva kuorma. Dynaamisella kääntämisellä tarkoitetaan tekniikkaa, jossa käännös tapahtuu osittain ajon aikana. Yksi esimerkki dynaamisesta kääntämisestä on ”Just-In Time” -kääntäminen, jossa kunkin käskyn käännös tehdään ajonaikana vasta kun kyseistä käskyä suoritetaan. Ensimmäisellä käskyn suorituskerralla hävitään staattiselle käännökselle, mutta seuraavat suorituskerrat ovat yhtä nopeita kuin staattisessa käännöksessä [3].

Tässä työssä tarkastellaan käskykanta- ja syklisimulaattorien taustaa ja nykyisiä tutkimusongelmia. Tutkitaan myös, missä tällä hetkellä mennään ja mitkä ovat nykyiset trendit ja huipputulokset. Eräänä erityiskysymyksenä selvitetään, miten simulaattorit soveltuvat välimuistin ja liukuhihnan toiminnan simulointiin.

2 Simulaattorien toteutustekniikat

Tämän paperin pääpaino on *käännetyissä* käskykanta- ja syklisimulaatioissa. Logiikkasimulaattorit jätetään tarkastelun ulkopuolelle.

Seuraavaksi esitellään tärkeimmät simulaation toteutustavat, jotka ovat *tulkkaus* ja *kääntäminen*.

2.1 Tulkkaus

Perinteinen tapa toteuttaa prosessorisimulaattori on rakentaa erillinen simulaattoriohjelma joka lukee suoraan simuloitavan koneen konekieltä. Tämä simulaattoriohjelma ajetaan *isäntäkoneessa* (*host machine*) ja se simuloi *kohdekoneen* (*target machine*) käskykantaa. Tulkkaukseen perustuvissa simulaattoreissa jokainen käsky *noudetaan* (*fetch*), *dekoodataan* (*decode*) ja *suoritetaan* (*execute*) jokaisella suorituskerralla erikseen. Tästä suhteellisen raskaasta käskynkäsittelyprosessista aiheutuu huomattava kuormitus: yhden kohdekoneen käskyn suorittaminen voi vaatia satoja tai jopa tuhansia isäntäkoneen käskyjä.

Tulkkaukseen perustuvien simulaattorien suurin ongelma on siis *hitaus*. Toisaaalta tulkkauspohjaiset simulaattorit pystyvät suorittamaan dynaamisesti ladattua/generoitua koodia, mikä mahdollistaa käyttöjärjestelmien tai dynaamiseen linkitykseen perustuvien ohjelmien ajamisen tällaisissa simulaattoreissa.

2.2 Kääntäminen

Monissa sovelluksissa simulaattorin tulee olla nopea, joten tulkkaavien simulaattorien lisäksi on kehitetty ns. *käännettyjä simulaattoreita*, joissa simulaattori ja simuloitava ohjelma käännetään isäntäkoneen ymmärtämään muotoon (joko suoraan konekieleksi tai yleensä C-kieliseksi). Tällöin vältetään tulkkauksesta aiheutuva kuormitus ja simulaatiosta tulee nopeampaa. Joissain tilanteissa tämä on melkein yhtä nopeaa kuin natiivikoodista.

Suoraan simuloitavan koodin ja simulaattorin kääntävissä järjestelmissä ei yleensä voi ajaa dynaamisesti generoitua koodia, kuten dynaamisten kirjastojen lataamista tai käyttöjärjestelmän userlandin prosesseja. Tällaisia tilanteita varten on käytetty *dynaamisia käännöstekniikoita*, kuten JIT (Just In-Time) [6].

Perinteisessä staattisessa kääntämisessä ohjelma käännetään kohdekoneen ymmärtämään muotoon tai jollekin välikielelle, josta lopulta saadaan yhdistetty simulaattori ja simuloitava ohjelma. Tulkissa esiintyvää simulaattorin ja simuloitavan ohjelman välistä jakoa ei ole. Dynaamiseen käännöstekniikkan perustuvissa simulaattoreissa on erillinen simulaattoriohjelma, jolle syötetään simuloitavan kohdekoneen konekieltä. Simulaattori kääntää ”lennossa” kohdekoneen koodin simuloitavan isäntäkoneen koodiksi. Voidaan ajatella että käännöksestä aiheutuva kuorma jaetaan (amortoidaan) käskyn jokaiselle suorituskerralle: ensimmäisellä kerralla käskyn kääntäminen ja suoritus on hitaampaa kuin vastaavan tulkatun käskyn, mutta seuraavilla kerroilla tuntuvasti nopeampaa.

Toinen hyvä puoli dynaamisen kääntämisen tekniikoissa on se, että niillä dynaamisesti generoidun koodin käyttäminen on helppoa: käännetäänhän kaikki koodi ainakin kertaalleen läpi, joten ajonaikana luodun/ladattavan koodin käsittely ei eroa mitenkään alussa ladatusta koodista.

Erityisen nopea geneerinen kääntämiseen perustuva simulaatio esitellään paperissa [3]. Kääntämisen ja tulkkaamiseen perustuvia simulaattoreita käsitellään tarkemmin paperis-

sa [4].

3 Simulaation tehokkuudesta

Erilaisten simulaattorien tehokkuusvertailu on hankalaa koska testialusta ja simuloitava ympäristö vaihtelevat hyvin paljon. Testialustan nopeus ja arkkitehtuuri vaihtelee jonkin verran. Valtaosassa tässä käsitellyistä paperissa kokeet suoritettiin PC-alustalla. Simulaation kohde vaihteli yleiskäyttöisista RISC-prosessoreista yksinkertaisiin DSP-prosessoroihin. On selvää, että yksinkertaisen prosessorin simuloinnissa päästään huomattavasti parempiin tuloksiin kuin monimutkaisen.

Myös simulaation abstraktiotaso vaikuttaa simuloinnin nopeuteen. Mitä matalammalla (laiteläheisemmällä) ja tarkemmalla tasolla simulaatiota tehdään, sen enemmän operaatioita joudutaan suorittamaan.

Seuraavassa on esitelty muutamia prosessorisimulaattoreita ja niiden suorituskykyä.

3.1 LISA

Simulaatioympäristö LISA esitellään paperissa [6]. LISA on geneerinen käskykantasimulaattori eli siinä on mallinnuskieli, jolla voidaan kuvata simuloitavan prosessorin käskykanta. Paperissa esitetyissä testeissä simulaation kohdekoneena oli sulautetut DSP-prosessorit Advanced Risc Machines ARM7 ja ST Microelectronics ST2000. Testeissä suoritettiin jpeg2000 codec-operaatioita. Suorituskyvyssä yllettiin parhaimmillaan n. 7 MIPSiin. Testiympäristönä oli PC Athlon 1200MHz, 768MB RAM. (Paperissa vertailtiin tulkattua, JIT-käännettyä ja staattisesti kännettyä simulaatiota keskenään, joten fokus ei ollut nopean simulaation tuottamisessa.)

3.2 Chang, Hu

Syklitarkkaa simulaatiota käsitellään paperissa [2]. Koska syklin tarkkuudella mallintaminen on huomattavasti raskaampaa kuin pelkkä käskykannan mallintaminen, on ymmärrettävää että simulaatiossa ei päästä kovinkaan hyviin tehokkuuslukemiin. Testiympäristönä oli PC Pentium II 733MHz, 256KB välimuistilla. Kohdeympäristönä oli MIPS R2000. Parhaimillaan päästiin 240 000 simuloituun sykliin/sekuntti.

Simulaattorissa mallinnitteen myös liukuhihnaa, mikä omalta osaltaan aiheuttanaa performanssihäviötä. Simulaattori on kuitenkin huomattavasti nopeampi kuin geneeriset logiikkasimulaattorit, koska se on tarkoitettu nimenomaan prosessorien mallintamiseen.

3.3 JACOB

JACOB-simulaattori kuvataan paperissa [4]. JACOB on tarkoitettu lähinnä DSP-piirien simulointiin ja tukee sekä tulkattua että käännettyä simulaatiota. Testiympäristönä oli SPARCstation 20, jossa oli 256 MB RAM-muistia. Tulkkattavassa simulaatiossa päästiin 115 000

käskyn sekuntinopeuteen. Käännetty simulaatio ylsi jopa 300 000:n simuloituun käskyyn sekunnissa. Tekijät huomattavat, että yksinkertaisemmilla arkkitehtuureilla voidaan yltää jopa 500 000 käskyn sekunttinopeuteen.

3.4 Zhu, Gajski

Erityisen nopea geneerinen simulaattorijärjestelmä on kuvattu paperissa [3]. Simulaattori perustuu staattiseen kääntämiseen, mikä osaltaan selittänee sen nopeuden. Testialustaa ei paperissa kuvattu ja simuloitavasta kohteestakin mainittiin vain että se on SPARC, mutta testitulosten perusteella simulaattori yltää jopa 100 MIPS:in nopeuteen, mikä on muutamaa kertaluokkaa enemmän kuin millään muulla tässä työssä käsitellyistä simulaattoreista. Koska kyseessä on staattiseen kääntämiseen perustuva simulaattori, ei sillä voi ajaa dynaamista koodia generoivia ohjelmia, mikä ei välttämättä ole DSP-ympäristössä kovinkaan suuri puute.

4 Käskykannan kuvaaminen

Yksi simulaattorien tärkein piirre on *geneerisyys* eli kuinka saada simulaattorista sellainen, että sillä on helppo simuloida erilaisia käskykanta-arkkitehtuureja. Kun tähän vielä yhdistetään vaatimukset tehokkuudesta, ei ongelma ole mitenkään triviaali.

Geneerisyyteen pyritään kuvaamalla simuloitava prosessoriarkkitehtuuri jollakin kuvauskielellä. Osa kuvauskielistä on korkean tason kieliä, joilla pystytään yksinkertaisesti kuvaamaan monimutkaisiakin prosessorin toiminnallisuuksia (esimerkiksi liukuhihnan tai välimuistin toiminta). Osa kielistä on taas matalan tason kieliä, jotka kuvaavat prosessorin sisäistä toimintaa logiikka- tai komponenttitasolla.

Seuraavassa on esitetty joitakin konkreettisia esimerkkejä kummankin tyypin kielistä.

4.1 VHDL

VHDL (Very High Speed IC Hardware Description Language) on IEEE:n standardoima digitaalielektroniikan suunnitteluun käytettävä kieli. VHDL:ssä mallia on mahdollista tarkastella useammalla tasolla, joita ovat piirin toiminnallinen taso, peruskomponenttien välisten yhteyksien rakenteellinen taso ja laitteiston toteuttamien loogisten funktioiden taso. Kielen primitiivejä ovat esimerkiksi rekisterit, laskurit ja multiplekserit. [5]

VHDL soveltuu konkreettisten piiritoteutusten suunnitteluun. Pelkkään prosessorin käskykannan kuvaamiseen kieli on turhan matalalla tasolla, mutta silti myös joissakin käskysimulaattoreissa käytetään VHDL:n kaltaisia kieliä käskykannan kuvaamiseen.

4.2 MIMOLA

Leupers et. al. käyttävät simuloitavan käskykannan kuvaamiseen MIMOLA HDL -kieltä, joka on VHDL:n kaltainen kieli, jossa käskyjen semantiikka kuvataan rekisterisiirtojen ja

yksinkertaisten laskuoperaatioiden avulla. MIMOLA:ssa kuvataan kuitenkin erikseen suunniteltavan yksikön rakenteet ja toiminta. Rakenteet mallinnetaan laitteistokomponenttien netlisteinä rekisterisiirtokielellä (RTL, Register Transfer Language) ja toiminta puolestaan Pascalin kaltaisella korkean tason ohjelmointikielellä. Kieli on hierarkinen, eli tietyn yksikön kuvaus voi koostua pienempien yksiköiden kuvauksista. [1]

MIMOLA on kehitetty käytettäväksi erilaisissa prosessorin suunnittelu-, synteesi- ja simulaatiotyökaluissa. Kielen soveltuvuudesta erityisesti muistisimulaatioon ei ole selvää tietoa.

4.3 Sim-nML

Rajesh ja Moona ovat kehittäneet korkean abstraktiotason hierarkkisen kielen Sim-nML simuloitavan käskykannan kuvaamiseksi. Vaikka kyseessä on korkean tason kieli, se on riittävän ilmaisuvoimainen prosessorin käskykannan kuvaamiseen. Sim-nML-kieli perustuu nML-nimiseen formalismiin, joka kuvaa prosessorin käskykannan attribuuttikieliopin avulla, mutta ei sisällä kontrollivuorakenteita. Sim-nML:ään kontrollivuorakenteet on lisätty. [7]

4.4 Chang, Hu

Chang ja Hu ovat kehittäneet kielen, jolla voidaan määritellä liukuhihan sisältäviä prosessorimikroarkkitehtuureita. Määrittelyn perusteella voidaan tuottaa automaattisesti syklintarkka prosessorisimulaattori kohdearkkitehtuurille. Määrittelykielen abstraktiotaso on melko korkea: sen avulla simulaattorille voidaan kertoa esimerkiksi ohjelman suoritusjärjestyksestä, forwardingista ja liukuhihnan sakkauksesta. Koska kieli on suunniteltu nimenomaan prosessorien kuvaamiseen, on prosessoriarkkitehtuurin kuvaaminen sillä paljon selkeämpää ja tiiviimpää kuin yleisellä laitteistosuunnittelukielellä, kuten VHDL:llä. Tekijät ovat kuvanneet MIPS:in kokonaislukuytimen, jossa on liukuhihnan 300 määrittelykielen rivillä. [2]

4.5 LISA

LISA on DSP-arkkitehtuureille tarkoitettu kuvauskieli, jolla on mahdollista kuvata prosessoriarkkitehtuuri tehokasta käännettyä käskysimulaatiota varten. LISA-kielessä arkkitehtuuri kuvataan osaksi rakenteellisella, osaksi toiminnallisella tasolla. Rakenteellisen tason määrittely käsittää prosessorin resurssit, kuten rekisterit, muistit ja liukuhihnat. Prosessorin käskyt ja niiden ajoitukset kuvataan LISA:ssa yksinkertaisen rekisterisiirtokielen operaatioilla. [6]

5 Välimuisti ja liukuhihna

Perinteiset välimuistisimulaattorit perustuvat simuloitavasta ohjelmasta tuotettuun *muistijälkeen* (*memory trace*). Uhlig ja Mudge ovat tehneet vuonna 1997 kattavan katsauksen [9]

muistijälkeen perustuvista muistisimulaattoreista, mutta se alkaa olla jo hieman vanhentunut.

Seuraavassa on kerrottu jotain edellä mainittujen simulaattoreiden kyvystä simuloida välimuistin ja liukuhihnan toimintaa.

5.1 Ravindran, Moona

Ravindran ja Moona ovat kehittäneet kääntävän käskyntarkan kohdekoneriippumattoman välimuistisimulaattorin [8]. Siinä simuloitava käskykanta kuvataan Sim-nML-kielellä [7].

Välimuisti voi olla monikerroksinen ja se kuvataan konfiguraatiotiedostossa, jossa sille määritellään erilaisia parametreja. Kustakin simuloitavan ohjelman käskystä tulee yksi C-kielisen simulaatorin funktiokutsu. Simulaattori tuottaa tilaston välimuistiosumista sekä eri tyyppisistä välimuistin ohi tapahtuvista viittauksista. Tekijät ovat kehittämässä käskysimulaattorin pohjalta syklintarkkaa simulaattoria.

5.2 LISA

LISA-simulaatiotyökalu tukee sellaisen prosessoriarkkitehtuurin simulointia, johon kuuluu liukuhihna. Työkalun tekijät ovat käyttäneet sitä ARM7- ja ST Microelectronics ST2000 -arkkitehtuureiden mallintamiseen. [6]

Paperissa ei kerrota, mitä tilastotietoa työkalu tuottaa simulaatiosta.

5.3 Chang, Hu

Changin ja Hun simulaattori tukee liukuhihnan sisältäviä prosessoriarkkitehtuureita. Tekijät ovat simuloineet sillä MIPS:in kokonaislukuytimen toiminnan. [2]

Paperissa ei kerrota, mitä tilastotietoa työkalu tuottaa simulaatiosta.

6 Johtopäätökset

Prosessorisimulaattorit ovat tärkeä työkalu prosessoriarkkitehtuurin suunnittelussa. Käskykantasimulaattorit eivät tuota yhtä tarkkaa tietoa laitteiston toiminnasta kuin syklintarkat simulaattorit tai logiikkasimulaattorit, mutta niiden suorituskyky on yleensä parempi.

Prosessoriarkkitehtuurin suunnittelussa on myös tärkeää, että käytössä oleva simulaattori on tarpeeksi yleinen ja joustava, jotta kaikki mahdolliset uudet arkkitehtuurin ominaisuudet voidaan kuvata simulaattorille. Arkkitehtuurin kuvaaminen on yksinkertaisempaa korkeamman tason kielellä, mutta sellaisella ei aina pystytä määrittelemään alla olevan raudan yksityiskohtia riittävällä tarkkuudella, jotta simulaatio olisi täysin realistista. Simulaation abstraktiotaso määrää myös simulaation nopeuden. Korkeamman tason käskykantasimulaatio on nopeampaa kuin matalan tason sykli- tai logiikkasimulaatio.

Prosessorisimulaattoreiden suorituskykyä voidaan parantaa erilaisilla tekniikoilla, joista tärkein on kääntäminen. Kääntäminen voidaan tehdä joko staattisesti, kun simulaattoria tuo-

tetaan, tai lennossa (Just-In-Time), kun ohjelmaa simuloidaan. Just-In-Time-kääntäminen mahdollistaa myös dynaamisen koodin simulointia, joka ei onnistu staattisesti käännetyllä simulaattorilla.

Välimuistin ja liukuhihnan simulointi vaikeuttaa käskysimulaatiota. Ilman välimuistia ja liukuhihnaa käskysimulaation ei tarvitse huolehtia käskyjen ajoituksista. On kuitenkin olemassa joitakin simulaattoreita, jotka pystyvät välimuistin tai liukuhihnan toiminnan simulointiin. Mielenkiintoisimpia näistä ovat ne simulaattorit, joissa prosessoriarkkitehtuuri kuvataan korkean tason kuvauskielellä ([2], [6]).

Käännetty prosessorisimulaatio on verraten uusi tutkimusala, eikä ongelmiin ole vielä valmiita ratkaisuja. Useimmat simulaattorit ovat joko kehitysasteella tai niissä on liikaa rajoituksia tuotantokäyttöön. Uusien menetelmien, kuten Just-In-Time-kääntäminen, soveltaminen tehokkaamman käännetyn prosessorisimulaation tuottamiseksi tarjoaa hyvän suunnan tutkimukselle.

Viitteet

[1] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The mimola language - version, 1994.

[2] Felix Sheng-Ho Chang and Alan J. Hu. Fast specification of cycle-accurate processor models. pages 488–492, 2001.

[3] Daniel D. Gajski Jianwen Zhu. A retargetable, ultra-fast instruction set simulator.

[4] R. Leupers, J. Elste, and B. Landwehr. Generation of interpretive and compiled instruction set simulators, 1999.

[5] G. Meister. A Survey on Parallel Logic Simulation. Technical report, 1993.

[6] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation.

[7] V. Rajesh and R. Moona. Processor modeling for hardware software codesign, 1999.

[8] R. Ravindran and R. Moona. Retargetable cache simulation using high level processor models. In *Australasian Computer Systems Architecture Conference, Goldcoast, Queensland Australia*, pages 113–120. IEEE Computer Society Press, 2000.

[9] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, 1997.