VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Kompiliatorių optimizavimas IA-64 architektūrai

Compilers optimization for IA-64 architecture

Magistro baigiamasis darbas

Atliko:	Tadas Valiukas	(parašas)
Darbo vadovas:	Asist. Giedrius Noreikis	(parašas)
Recenzentas:	Doc. dr. Antanas Mitašiūnas	(parašas)

Vilnius – 2009

Santrauka

Tradicinės x86 architektūros spartinimui artėjant prie galimybių ribos, kompanija Intel pradėjo kurti naują IA-64 architektūrą, paremtą EPIC – išreikštinai lygiagrečiai vykdomomis instrukcijomis vieno takto metu. Ši pagrindinė savybė leidžia vykdyti iki šešių instrukcijų per vieną taktą. Taipogi architektūra pasižymi tokiomis savybėmis, kurios leido efektyviai spręsti su kodo optimizavimu susijusias problemas tradicinėse architektūrose. Tačiau kompiliatorių optimizavimo algoritmai ilgą laiką buvo tobulinami tradicinėse architektūrose, todėl norint išnaudoti naująją architektūrą, reikia ieškoti būdų tobulinti esamus kompiliatorius. Vienas iš būdų – kompiliatoriaus vidinių parametrų atsakingų už optimizacijas reikšmių pritaikymas IA-64. Būtent toks yra šio darbo tikslas, kuriam pasiekti reikia išnagrinėti IA-64 savybes, jas vėliau eksperimentiškai taikyti realaus kodo pavyzdžiuose bei įvertinti jų įtaką kodo vykdymo spartai. Pagal gautus rezultatus nagrinėjami kompiliatoriaus vidiniai parametrai ir su specialia kompiliatorių testavimo programa randamas geriausias reikšmių rinkinys šiai architektūrai. Vėliau šis rinkinys išbandomas su taikomosiomis programomis. Gauto parametrų rinkinio reikšmės turėtų leisti generuoti efektyvesnį kodą IA-64 architektūrai.

Raktiniai žodžiai: IA-64 architektūra, Itanium, predikacija, išankstinis duomenų užkrovimas, ciklų optimizavimas, valdymo spėjimas, išreikštinai lygiagretus instrukcijų vykdymas.

Summary

After performance optimization of traditional architectures began to reach their limits, Intel corporation started to develop new architecture based on EPIC – Explicitly Parallel Instruction Counting. This main feature allowed up to six instructions to be executed in single CPU cycle. Also this architecture includes more features, which allowed efficient solution of traditional architectures code optimization problems. However for long time code optimization algorithms have been improved for traditional architectures only, as a result those algorithms should be adopted to new architecture. One of the ways to do that – exploration of internal compilers parameters, which are responsible for code optimizations. That is the primary target of this work and in order to reach it the features of the IA-64 architecture and impact to execution performance must be explored using real-life code examples. Tests results may be used later for internal parameters selection and further exploration of these parameters values by using special compiler performance testing benchmarks. The set of those new values could be tested with real life applications in order to prove efficiency of IA-64 architecture features.

Keywords: IA-64 architecture, Itanium, EPIC – Explicitly Parallel Instruction Counting, VLIW – Very Long Instruction Word, predication, control and data speculation, software pipelining, branch prediction, prefetching, RSE – rotating register engine, gcc optimization.

Turinys

1.Įvadas

Gana ilgą laiką Intel aktyviai tobulino savo x86 architektūrą, kuri tapo plačiai pripažintu standartu IT industrijoje. Tačiau praėjusio dešimtmečio viduryje Intel pastebėjo, kad šios architektūros našumas artėja prie fizinių galimybių ribos. Todėl kartu su HP buvo pradėta tyrinėti nauja RISC (reduced instruction set computer) architektūra, kurios pagrindinė savybė yra EPIC (Explicitly Parallel Instruction Computing – išreikštinai lygiagretus instrukcijų vykdymas). Taip buvo sukurta IA-64 architektūra, pasižyminti VLIW (Very Long Instruction Word – labai ilgas instrukcijos žodis).

Šios architektūros pagrindinis principas yra perkelti visą optimizavimo darbą programuotojui arba kompiliatoriui, taip atsisakant sudėtingo ir brangaus instrukcijų vykdymo optimizacijų realizavimo aparatūroje. Architektūra paremta tuo, kad kompiliatoriui arba programuotojui suteikiamos galimybės programos kodą užrašyti taip, kad jis būtų vykdomas sparčiausiai. Toks požiūris yra žymiai geresnis, lyginant su tradicine x86 architektūra, kadangi tiek kompiliatorius, tiek programuotojas gali žvelgti į programos kodą žymiai plačiau, negu procesorius, numatantis vos po kelias instrukcijas. Taip pat vykdydamas optimizacijas programos kompiliavimo metu, kompiliatorius yra žymiai mažiau apribotas laike, bei gali išsamiau taikyti įvairias kodo analizės priemones.

Kompiliatoriuose naudojami kodo optimizacijos algoritmai yra gerai pritaikyti tradicinėms architektūroms, tačiau dažnai pilnai neišnaudoja IA-64 architektūros teikiamų privalumų, nes reikalingi nauji optimizavimo algoritmai skirti konkrečiai architektūrai bei įvairios vidinių optimizacijas valdančiųjų parametrų reikšmės nėra optimalios būtent IA-64 architektūrai.

Tikslas ir uždaviniai

Magistrinio darbo pagrindinis tikslas - geresnių parametrų reikšmių rinkinio paieška, kuris leistų kompiliatoriui generuoti geriau optimizuotą kodą, bei žymiai efektyviau išnaudoti IA-64 architektūros teikiamas savybes. Tikslui pasiekti reikalingi du pagrindiniai uždaviniai:

- Ištirti IA-64 architektūros savybių įtaką kompiliuojamo kodo efektyvumui. Kiekviena savybė pritaikoma realaus kodo pavyzdžiuose ir matuojamas laiko skirtumas tarp pradinio ir optimizuoto kodo.
- Eksperimentiškai ištirti vidinių kompiliatoriaus parametrų reikšmių įtaką. Remiantis pirmojo uždavinio gautais rezultatais pasirinkti reikalingus kompiliatoriaus parametrus ir juos išnagrinėti, naudojant tiek specialias kompiliatoriaus testavimo, tiek taikomąsias programas.

Darbo struktūrą sudaro kelios pagrindinės dalys:

- Pirmojoje dalyje pristatoma IA-64 architektūra. Taip pat nagrinėjamos optimizuojančios

savybės, kurios susiję su geresnio kodo generavimu, bei aptariamos optimizavimo problemos, kurioms spręsti taikomos tos savybės.

- Antroje darbo dalyje prieš tai išnagrinėtos savybės tiriamos eksperimentiškai, nagrinėjant tipinius C bei asemblerio kodo pavyzdžius bei pritaikant minėtas savybes. Taipogi yra tiriama jų įtaka kodo vykdymo spartai bei pateikiami eksperimentų rezultatai ir apibendrinimai.
- Trečioji darbo dalis yra skirtas kompiliatoriaus vidinių parametrų tyrimui. Kaip kompiliatoriaus realizacija pasirinktas gcc (Gnu C compiler collection) įrankių rinkinys. Kadangi antroje darbo dalyje padaryta išvada jog svarbiausios IA-64 savybės yra instrukcijų vykdymas lygiagrečiai, bei laikinųjų atmintinių valdymas, nagrinėjami tie parametrai, kurie yra susiję su minėtomis savybėmis. Parametrai tiriami keičiant jų reikšmes bei kompiliuojant ir vykdant testinę programą. Pagal programos vykdymo skaitliukų statistiką nustatomos geriausios parametrų reikšmės. Pasirinktų reikšmių rinkinys bandomas su keliomis realiomis programomis, bei įvertinamas kodo našumas.

Numatomas darbo rezultatas - surastas geresnis optimizacijas valdančių parametrų reikšmių rinkinys, su kuriuo kompiliatoriaus generuojamas kodas bus našesnis bei efektyvesnis IA-64 architektūroje.

2.Itanium architektūra

2.1 Istorija

1989 metais HP (Hewlett-Packard) kompanija nustatė, kad RISC (Reduced Instruction Set Computer) architektūros artėja prie skaičiavimo našumo ribos, vykdant tik vieną instrukciją per taktą. HP mokslininkai pradėjo tirti naują architektūrą, vadinamą EPIC (Explicitly Parallel Instruction Computing). Pagrindinė šios architektūros savybė buvo labai ilgas instrukcijos žodis (Very Long Instruction Word - VLIW), į kurį galima sutalpinti keletą lygiagrečiai vykdomų instrukcijų. Kompiliatorius tuomet iš anksto nustato, kokios instrukcijos gali būti vykdomos tuo pačiu metu, ir atitinkamai sugeneruoja instrukcijų žodžius. Procesoriui, vykdančiam tokį kodą, nereikia sudėtingų mechanizmų, sprendžiančių, kurias instrukcijas galima vykdyti lygiagrečiai. Tokie mechanizmai yra įdiegti pavyzdžiui IA-32 architektūroje.

HP nusprendė, kad neefektyvu tokiai kompanijai kaip ji pati gaminti mikroprocesorius, ir 1994 metais kartu su Intel sukūrė naują IA-64 architektūrą, kuri paveldėjo visas EPIC savybes. Pirmojo šios architektūros procesoriaus kodinis pavadinimas buvo Merced. Kūrimo metu tiek Intel, tiek HP tikėjosi, kad nauja architektūra dominuos tiek darbo stočių, tiek serverinių sistemų rinkoje. Programinės įrangos gamintojai pradėjo kurti operacines sistemas šiai architektūrai, tačiau 1997 buvo pastebėta, kad įgyvendinti pačią architektūrą bei sukurti jai skirtus kompiliatorius bus žymiai sunkiau, nei manyta. Susidurta su tokiomis techninėmis kliūtimis, kaip labai didelis tranzistorių skaičius, reikalingas ilgiems instrukcijų žodžiams bei laikinų duomenų saugykloms (cache). Taip pat atsirado problema su kompiliatoriais, kadangi savybių, kuriomis rėmėsi IA-64, dar niekas nebuvo juose panaudojęs. Merced buvo pirmasis šios architektūros procesorius, todėl kūrėjų komanda, susidūrusi su nenumatytomis tokio tipo kliūtimis, nusprendė atidėti procesoriaus išleidimą ir papildomai skirti laiko tyrinėjimams.

1999 metais Intel suteikia oficialų vardą mikroprocesoriui – Itanium, kurio eksploatacinės savybės bei pardavimai rinkoje nepateisino lūkesčių, kokių buvo tikėtasi. Vis dėl to tai buvo pirmasis tokios architektūros procesorius, kuriame stengiamasi atsisakyti sudėtingų optimizavimo sprendimų pačiame procesoriuje. Tas darbas buvo atiduotas kompiliatoriui, procesoriui suteikiant tokias savybes kaip registrų pervadinimas (register renaming), predikacija (predication), spėliojimas (speculation), bei būsimų vykdymo šakų numatymas (branch prediction), taip pat kelių lygių laikinosios atmintinės valdymas (cache control) [Int04], [Int06b].

2.2 Pagrindinės savybės

IA-64 (toliau – Itanium) architektūra buvo kuriama sujungiant svarbius aspektus, leidžiančius

pasiekti labai aukštą efektyvumo lygį bei plečiamumą ateities tobulinimams [HMR00+]. Pagrindiniai Itanium architektūros principai yra šie:

- Aiškiai išreikštas lygiagretumas instrukcijų lygyje (ILP – instruction level parallelism) – kompiliatoriaus generuojamos instrukcijos gali vienu metu lygiagrečiai naudotis dideliais procesoriaus resursais (registrais, vykdymo bei atminties pakrovimo konvejeriais) [SA00].
- Savybės, kurios padidina ILP – išankstinis užkrovimas, leidžiantis „paslėpti“ duomenų skaitymo iš atminties užlaikymus, predikacija – leidžianti atsisakyti vykdomo kodo išsišakojimų, bei pačių išsišakojimų prognozė (branch prediction), sumažinanti perėjimo poveikį procesoriaus darbo spartai.
- Skirtas didelis dėmesys programinės įrangos efektyvumui – speciali programinės įrangos išskaidymo galimybė, spartus darbas su slankaus kablelio skaičiais, bei specialios daugiaterpinės programinės įrangos palaikymo instrukcijos.
- Išskirtinės galimybės kompiliatoriui valdyti procesoriaus laikinąją atmintį, perėjimų prognozes ir t. t.

2.3 Registrų modelis

Itanium architektūra apibrėžia didelį skaičių registrų, leidžiančių atlikti daugybę skaičiavimų be dažno kreipimosi į atmintį [Int06a]:

1. 128 64 bitų dydžio bendrojo naudojimo registrai r0-r127, kiekvienas jų dar turi papildomą 65 bitą, vadinamąjį NaT (Not A Thing), kuris naudojamas kylančioms išskirtinėms (exception) situacijoms saugoti, kai naudojamas išankstinis užkrovimas. Registras r0 visada yra lygus nuliui ir prieinamas tiktai skaitymui.
2. 128 82 bitų dydžio slankaus kablelio registrai f0-f127, kurie naudojami įvairioms operacijoms su slankiuoju kableliu. Registras f0 visada lygus 0, o registras f1 visada lygus 1 ir yra prieinami tiktai skaitymui.
3. 64 1 bito predikatiniai registrai p0-p63 skirti saugoti loginių išraiškų rezultatams. Registras p0 visada lygus 1 arba true ir yra prieinamas tik skaitymui.
4. 8 64 bitų dydžio atšakų registrai b0-b7, kurie naudojami nurodyti tikslo adresams įvairiuose netiesioginiuose perėjimuose (indirect branching).
5. 128 įvairaus dydžio specialaus naudojimo registrai ar0-ar127, skirti sisteminėms reikšmėms bei pačiam procesoriui valdyti ir t.t.
6. IP – instrukcijų skaitliukas (Instruction Pointer). Registras rodantis į šiuo metu vykdomą instrukcijų pluoštą.

7. CFM – esamo lango žymeklis (Current Frame Marker). Skirtas valdyti registrų stekui (rotacija, bei registrų steko valdymas procedūrų kvietimo metu).

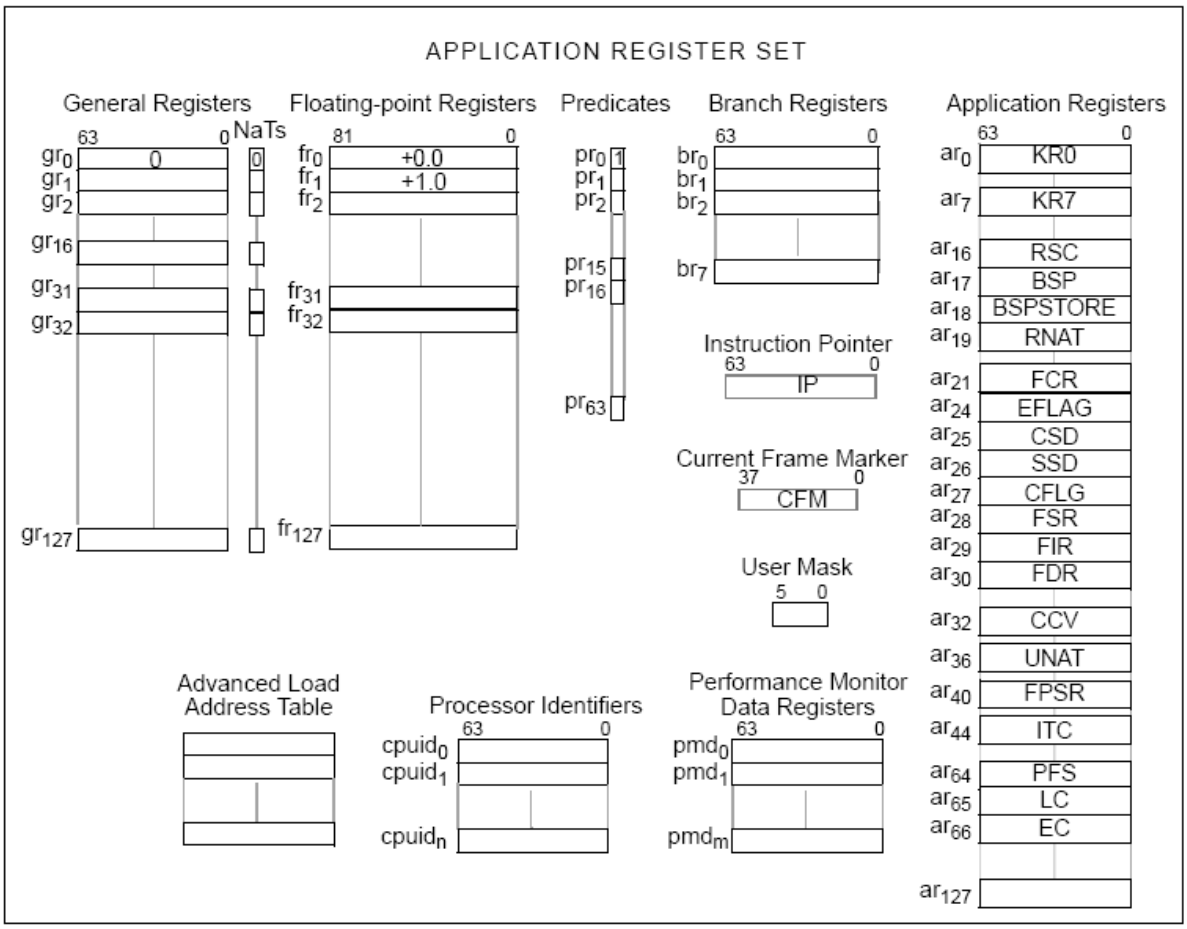
8. ALAT – išankstinio užkrovimo adresų lentelė (Advanced Load Address Table), naudojama išankstinio duomenų užkrovimo instrukcijų.

9. Processor identifiers – registrai skirti įvairiai informacijai apie procesorių gauti (realizuotos savybės, laikinosios atminties valdymo ypatumai ir t.t.).

10. Performance Monitor Data Registers – registrai skirti procesoriaus darbo efektyvumo kontrolei.

11. User Mask – įvairūs procesoriaus parametrai, kontroliuojantys jo darbą programos vykdymo metu.

Bendras registrų modelis pavaizduotas 1 pav. [Int06a]:



1 pav. Itanium procesoriaus registrai

2.4 Instrukcijų vykdymas

Itanium architektūroje programinį kodą sudaro dideli 128 bitų žodžiai dar kitaip vadinami paketais arba pluoštais (angl. Bundle) (VLIW – very long instruction word), kuriuose mašininės instrukcijos yra grupuojamos po tris (2 pav.):

2 pav. Itanium instrukcijų paketas

Instrukcijų pluoštą sudaro 5 bitų pluošto šablonas, apibrėžiantis kokiam vykdymo konvejeriui yra priskiriama kiekviena instrukcija, bei trys 41 bito vietos trims instrukcijoms. Kadangi instrukcijos grupuojamos po tris, tai Intel architektūroje mažiausias programos kodo vienetas yra pluoštas – programos vykdymas negali prasidėti pluošto viduryje bet kokiu atveju, instrukcijų skaitiklio saugomas adresas visada yra vykdomojo pluošto adresas. Taipogi instrukcijų pluošte gali būti vadinamieji stop bitai kurie pasako procesoriui, kad instrukcijos, einančios prieš stop bitą, ir instrukcijos, sekančios po stop bito, turi tam tikrų priklausomybių ir negali būti vykdomos vieno takto metu. Iš esmės stop bitas procesoriui pasako baigti esamą taktą, atnaujinti ir sinchronizuoti procesoriaus resursus bei būseną ir pradėti naują taktą.

Instrukcijos pagal tipą dar yra skirstomos į šešias grupes M, I, A, F, B, L. M – atminties skaitymo arba rašymo instrukcijos, A – sveikų skaičių aritmetikos instrukcijos, I – specialiosios daugialypės terpės instrukcijos, F – slankaus kablelio instrukcijos, B – atšakų instrukcijos, L – praplėstosios instrukcijos, kai vienas iš operandų įdedamas į sekančios instrukcijos vietą. Taipogi procesoriaus branduolio vykdomieji konvejeriai (Execution Units) yra 5 tipų – M, I, F, B, L. Tam tikro tipo instrukcijos gali būti vykdomos tik tam tikro vykdomojo konvejerio, o kadangi vieno tipo konvejerių procesoriuje yra keli (pvz, M tipo – atminties skaitymo rašymo – yra du konvejeriai) tai šablono tipas bei instrukcijos vieta pluošte tiksliai apibrėžia, kokio tipo instrukcija ir kuriam konvejeriui priskiriama [SA00].

Instrukcijų grupė yra instrukcijos, kurios gali būti vykdomos vieno takto metu. Kadangi procesorius netikrina, ar vienos grupės instrukcijos turi tarpusavyje neleistinų priklausomybių (apie jas vėliau), tai yra išskirtinai programuotojo (kompiliatoriaus reikalas), kaip teisingai jas sugrupuoti. Priešingu atveju procesoriaus būsena, įvykdžius neleistinai suformuotą instrukcijų grupę, tampa neapibrėžta. Instrukcijų grupės yra atskiriamos vadinamaisiais stop bitais, kurie yra pluošto šablono dalis ir kurie gali būti po antros, pirmos arba nulinės instrukcijos (šablono gale). Taip pat instrukcijų grupę gali užbaigti tam tikros instrukcijos, kurios reikalauja, kad procesoriaus būsena būtų pilnai atnaujinta ir sinchronizuota, prieš vykdant sekančią instrukciją.

Procesorius instrukciją vykdo keliais etapais:

- Instrukcijos skaitymas iš atminties (Fetch).
- Procesoriaus būsenos ir resursų nuskaitymas jeigu reikia (Read).

- Nurodytos operacijos įvykdymas (Execute).
- Procesoriaus būsenos ir resursų atnaujinimas (Update).

Jeigu instrukcijos priklausančios vienai grupei atitinka keliamus nepriklausomybės viena nuo kitos reikalavimus, tai netgi įvykdžius lygiagrečiai visas instrukcijas vienu metu, iš išorės atrodys, jog jos buvo vykdomos nuosekliai viena po kitos pagal aukščiau išdėstytus vykdymo etapus. Priešingu atveju procesoriaus būsena tampa neapibrėžta. Instrukcijoms keliami reikalavimai [Int06a]:

- Registrų priklausomybės. Nėra leidžiama RAW (Read-after-write), tai reiškia, kad vienos grupės instrukcijoms draudžiama iš pradžių rašyti į tą patį registrą, o po to jį nuskaityti. Taipogi nėra leidžiama WAW (write-after-write) – vienoje grupėje keli rašymai į tą patį registrą. WAR (Write after Read) priklausomybė - kuomet toj pačioj grupėj iš pradžių registro reikšmė yra nuskaitoma, o vėliau pakeičiama. Tokia priklausomybė leidžiama, kadangi neprieštarauja instrukcijų vykdymo etapams. Taipogi leidžiama RAR (Read-after-Read) priklausomybė – to pačio registro skaitymas grupėje kelis kartus.
- Atminties priklausomybės. Atminčiai leidžiamos visos RAW, WAR ir WAW priklausomybės.

Žinoma reikia atkreipti dėmesį, jog yra tam tikrų išimčių, kurios liečia specialius atvejus ir specialias instrukcijas [Int04], [Int06c].

Paprasčiausio programinio kodo pavyzdys (3 pav.):

```
{ .mii
1.          ld4 r28=[r8]
2.          add r9=2,r1
3.          add r30=1,r1
}
{.m_mi
4.          ld4 r29=[r40];;
5.          st4 [r30]=r9
6.          sub r7=1,r8
}
```

3 pav. Itanium asemblerio fragmentas

Programinį kodą sudaro du pluoštai instrukcijų, atskirti skliausteliais. Po atsidarančio skliaustelio yra nurodomas pluošto šablonas. MII reiškia, jog pluošte yra viena M (memory arba atminties) instrukcija, bei dvi I (integer arba skaitinės) instrukcijos. Dvigubas kabliataškis po ketvirtos instrukcijos reiškia stop bitą antrajam pluošte. Vadinasi kodą sudaro dvi instrukcijų grupės – pirmoji sudaryta iš keturių pirmųjų instrukcijų, antroji – iš likusių dviejų. Noriu pabrėžti, jog instrukcijų skirstymas į grupes bei skirstymas į pluoštus tarpusavyje visiškai nesusiję. Aiškumo

dėlei tolimesniuose kodo pavyzdžiuose, skliaustelius bei pluošto šablono nurodymus praleisiu. Taipogi matome, jog dėl RAW priklausomybės antra ir penkta instrukcijos negali būti vienoj grupėj [Int00].

Intel architektūra praktiškai neriboja instrukcijų skaičiaus grupėje, tačiau jų įvykdymas per vieną taktą priklauso nuo esamos procesoriaus realizacijos – vykdymo konvejerių skaičiaus, pavyzdžiui, esant kelioms atminties skaitymo instrukcijoms, gali tiesiog nelikti laisvo procesoriaus konvejerio atsakingo už atmintį. Tuomet, esant nepakankamam reikalingų resursų kiekiui, procesoriui gali tekti pradėti naują taktą ir taip nukentės darbo sparta.

Taipogi labai svarbu prisilaikyti priklausomybių ribojimų instrukcijų grupėje. Vienas iš pagrindinių kompiliatorių optimizavimo aspektų ir yra kuo efektyvesnis instrukcijų grupavimas ir išdėstymas [KKL00+].

3.Kodo optimizavimas Itanium architektūrai

3.1 Predikacija

Pradėjus rinkti įvairią statistiką apie programų vykdomąjį kodą, pastebėta, kad didžiąją programos vykdymo laiko dalį sudaro sąlyginis kodo vykdymas, kuomet pagal tam tikrą sąlygą procesorius keičia programos vykdymo eigą (paprasčiausias if sakinys C kalboje). Įvairios programos atšakos labai sulėtina programos vykdymą, nes atsiranda įvairūs užvėlavimai dėl būsimos atšakos neteisingo prognozavimo. Intel architektūra sprendžia šią problemą pateikdama predikacijos idėją. Predikacija – tai dinaminis instrukcijų „įjungimas“ arba „išjungimas“ programos vykdymo metu, leidžiantis atsisakyti sudėtingų sąlyginių atšakų. Pateikime paprastą pavyzdį (4 pav.):

```
If(a!=0) {
    If(b==5){
        C=4;
    }
}else{
    C=1;
}
```

C kodo sąlyginis sakinys

```
Cmp.ne p1,p2=0,a
(p1)cmp.eq p1=5,b
(p1)mov r3=4
 (p2)mov r3=1
Mov c=r3
```

Tas pats kodas IA-64 instrukcijomis

4 pav. If sakinys ir jo asemblerio kodas

Matome kaip galima if sakinį su dar vienu if viduje transformuoti į Itanium architektūros instrukcijų seką, kuri bus įvykdyta per tą patį taktą [KKL00+]. Trumpai paaiškinsime kodą. Pirma eilutė nustato pirmąją sąlyga ar „a“ lygu nuliui. Po šios komandos įvykdymo p1 predikate bus reikšmė vienetas, jeigu „a“ nelygu nuliui, ir nulis, jei „a“ lygu nuliui. Antrasis komandos parametras predikatas p2 visada įgaus reikšmę priešingą pirmajam. Priešais sekančią komandą skliausteliuose nurodytas predikatas p1, priklausomai nuo reikšmės, įjungia arba išjungia komandos vykdymą. Todėl visos komandos, prieš kurias yra predikatas p1 bus vykdomos tada, kai „a“ reikšmė nelygi nuliui, ir priešingu atveju visos komandos prieš predikatą p2 bus vykdomos tada, kai „a“ reikšmė bus lygi nuliui. Išjungtos komandos yra pateikiamos vykdymo konvejeriams lygiai taip pat kaip ir įjungtos – skirtumas tik tas, jog komandos 4 vykdymo etape neatnaujina procesoriaus būsenos. Labai svarbu yra tai, jog gali būti visos įmanomos priklausomybės tarp p1 ir p2 predikatais pažymėtų komandų (nes bet kokiu atveju bus vykdomos arba vienos arba kitos komandos). Tokiu būdu predikacija įgalina procesorių vienu taktu įvykdyti abi sąlyginio sakinio šakas kaip paprastą nepriklausomų instrukcijų seką. Taip galima statistiškai pašalinti iki 50% vykdomų atšakų bei

poreikį jas prognozuoti, ir tai smarkiai padidina programos vykdymo efektyvumą bei leidžia žymiai geriau išlygiagretinti patį kodą [BCC00+]. Jeigu prieš instrukciją nenurodomas joks predikatinis registras, tai reiškia, kad pagal nutylėjimą imamas predikatinis registras p0 (kuris visada lygus vienetui) ir komanda vykdoma visada.

3.2 Išankstinis duomenų užkrovimas (speculation)

Spekuliacija arba išankstinis užkrovimas - tai duomenų užkrovimo principas, leidžiantis paslėpti vėlavimus susijusius su atminties skaitymu. Yra skiriamos dvi išankstinio duomenų užkrovimo rūšys – valdymo spekuliacija (control speculation) ir duomenų spekuliacija (data speculation) [LLC03+].

3.2.1 Valdymo spėjimas (control speculation)

Valdymo spekuliacija tai metodas iškelti priklausomą nuo sąlygos duomenų pakrovimą virš esančios sąlygos tikrinimo į prieš tai esančią vieną iš kelių grupių, taip paslepiant skaitymo iš atminties vėlavimus. Pavyzdžiui kodas, kuris nuskaito reikšmę tam tikru adresu ir atlieka veiksmus, prieš tai patikrinęs ar adresas nelygus *null* (5 pav.):

```
(p1) br.cond addr_is_null // 0
taktas
Ld8 r5=[r3];; // 1 taktas
Add r4=r5,2 // 3 taktas
```

Pradinis kodas

```
Ld8.s r5=[r3] //keletas taktų
atgal
.....
(p1) br.cond addr is_null //0
taktas
Chk.s r5, recovery // 1 taktas
Add r4=r5,2          // 1 taktas
```

Kodas su išankstiniu pakrovimu

5 pav. Išankstinis valdymos spėjimas

Pradinį kodą įvykdyti reikia 4 taktų: vienas taktas sąlyginei atšakai (jei adresas ne nulis), du taktai (skaitymo iš atminties užlaikymas – du taktai) skaitymui iš atminties, gale būtinai turi būti stop bitas, nes tarp antros ir trečios eilutės yra RAW priklausomybė ir instrukcijos negali būti vienoj grupėj. Komanda *ld8.s* užkraunanti reikšme iš duoto adreso iškeliama į viršų – į artimiausią įmanomą prieš tai ėjusią grupę taip, kad būtų paslėpti du taktai, atsirandantys dėl skaitymo iš atminties užlaikymo. Modifikatorius s nurodo, jog vykdant šią komandą, visos išimtinės situacijos (puslapiavimo klaidos ir t. t.) turi būti atidėtos ir iškart neapdorotos. Jei adresas pasiekiamas - duomenys tiesiog užkraunami į registrą, o jei kyla išimtinė situacija, į registro NaT bitą įrašomas vienetas ir duomenys nenuskaitomi iš atminties, tačiau išimtinės situacijos apdorojimo kodas nėra

vykdomas. Tokia išimtinė situacija vadinama atidėtąja (deferred exception). Instrukcija *chk.s* patikrina NaT bitą ir jei jis lygus vienetui – vykdoma atšaka į recovery žymę, o jei ne – instrukcija tiesiog nieko nedaro, nes išankstinis užkrovimas pavyko. Recovery žymėje turi būti kompiliatoriaus sugeneruotas atstatymo kodas, kuris pakartotinai užkrauna duomenis į registrą jau po sąlygos patikrinimo. Kadangi nebeliko RAW žymių tarp *chk.s* ir *add* instrukcijų – jos gali būti įvykdomos per vieną taktą. Taigi, naudojant išankstinį užkrovimą, modifikuotą kodą tapo įmanoma vykdyti tik per du procesoriaus taktus. Išankstinio užkrovimo atveju galima iškelti ne tik patį duomenų skaitymą iš atminties bet ir kitas, su tais duomenimis susijusias, operacijas, tačiau tuomet didėja recovery kodas, kurį prisireiks vykdyti nesėkmės atveju. O jeigu kilo atidėtoji išimtinė situacija, bet vykdoma sąlyginė null_pointer šaka, tuomet kodas vykdomas, tarsi išimtinės situacijos net nebūtų buvę. Taipogi svarbus dalykas tas, jog atliekant veiksmus su registrais, kurių NaT bitas lygus vienetui, visi registrai, kuriuose saugomas veiksmų rezultatas, taip pat įgyja NaT reikšmę lygią vienetui. Tai įgalina vienu metu iškelti kelių registrų užkrovimus, po sąlygine atšaka įvykdyti norimus veiksmus su jais, o vėliau tiesiog su *chk.s* instrukcija patikrinti rezultatų registrą ir, jei bus rasta atidėtoji išimtis, vykdyti atstatomąjį kodą [BCC00+].

3.2.2 Išankstinis duomenų užkrovimas (data speculation)

Išankstinis duomenų užkrovimas tai registro užkrovimo pernešimas, prieš kito registro reikšmės išsaugojimą, kai nėra aišku ar atminties adresai persidengia. Pateiktame asemblerio kodo pavyzdį (6 pav.):

```
St8 [r55]=r45  // 0 taktas
Ld8 r3=[r5];;   // 0 taktas
Shr r7=r3,r87 // 2 taktas
```

```
Ld8.a r3=[r5] //išankstinis užkrovimas
…                //kitos instrukcijos
St8 [r55]=r45  // 0 taktas
Ld8.c r3=[r5]   // 0 taktas
Shr r7=r3, r87 // 0 taktas
```

Modifikuotas kodas su užkrovimu prieš išsaugojimą

6 pav. Išankstinis duomenų užkrovimas

Pradiniame kode nėra aišku, ar adresai registruose r55 ir r5 nepersidengia, todėl vykdant šias komandas kaip vieną instrukcijų grupę kyla neapibrėžtumas. Taip pat tarp antros ir trečios instrukcijos vėl gi yra RAW priklausomybė ir reikalingas stop bitas, bei dėl atminties skaitymo užlaikymo sunaudojamas papildomas procesoriaus taktas. Problemą galima spręsti pasitelkus ld8.a ir ld8.c instrukcijas. Pirmoji instrukcija yra iškeliama keletą taktų į viršų, paslepiant skaitymo iš

atminties užlaikymą, o modifikatorius a reiškia, kad įvykdžius šią komandą, registro numerį, atminties adresą bei nuskaitytą baitų kiekį reikia išsaugoti ALAT lentelėje. Bet koks rašymas į atmintį visada yra sutikrinamas su adresais ir registrais saugomais šioje lentelėje ir jei registrai arba atminties vietos sutampa arba persidengia – įrašas išimamas. Operacija ld8.c, o tiksliau jos modifikatorius c patikrina ALAT adresų lentelę ir jei randamas įrašas, atitinkantis tą patį registrą bei atminties vietą, nieko nedaro. Priešingu atveju įvykdomas pakartotinas užkrovimas su tam tikru užlaikymu [LCC03+].

Išankstiniai užkrovimai, iškeliant užkrovimus tiek virš sąlyginių šakų vykdymo, tiek virš duomenų rašymo į atmintį nevienareikšmiškais adresais, yra viena iš kertinių Itanium architektūros optimizavimo galimybių ir įgalina kompiliatorių generuoti labai efektyviai dirbantį ir gerai lygiagretinamą kodą, kadangi kompiliatoriui suteikiama laisvė laisvai kilnoti problematiškas instrukcijas tarp skirtingų instrukcijų grupių. Vienas iš aspektų ribojančių tokio tipo optimizacijas yra pailgėjęs registrų „gyvybingumas“ (liveness). Tai reiškia, kad kuo anksčiau į norimą registrą pakrauname reikalingus duomenis, tuo ilgiau jis yra užimtas iki pat veiksmų su juo pabaigos, o registrų skaičius procesoriuje anaiptol nėra begalinis [KIF99].

3.3 Atšakų prognozavimo žymės

Nors Itanium architektūra ir leidžia pilnai išnaudoti predikacijos teikiamas galimybes, eliminuojant atšakas, tačiau būna atvejų, kai jų panaudojimas tiesiog būtinas. Todėl, norėdamas išvengti atšakų užlaikymo efektų, procesorius bando atspėti, kurią šaką vykdyti (branch prediction). Toks būdas pasiteisina ir sumažiną nereikalingą užlaikymą sėkmės atveju. Tačiau priešingu (misprediction) atveju neteisingo sprendimo sukelti užlaikymai tampa dar didesni, nes procesorius tuomet priverstas išvalyti visas instrukcijas iš laikinųjų atminčių, atlaisvinti tarpinių rezultatų užimamus resursus, bei pereiti į teisingą atšakos vykdymo kelią. Dažniausiai programos vykdymo metu procesorius turi labai ribotą „langą“ tiek laiko, tiek programos kodo atžvilgiu ir ne visada sėkmingai gali nuspėti programos vykdymo kelią. Kompiliatorius, priešingai, turi praktiškai neribotą laiką (procesoriaus atžvilgiu) bei žymiai platesnį kodo vaizdą, todėl Itanium architektūros galimybės leidžia kompiliatoriui su tam tikrais požymiais padidinti procesoriaus šansus teisingai atspėti tolesnį programos vykdymo kelią [KKL00+]. Šie požymiai yra tiek atšakų instrukcijų, tiek ciklų modifikatoriai:

1. Sptk – nekintamas spėjimas, paimta (static prediction, taken). Atšaka tikrai bus vykdoma, neskirti jokių resursų statistikos apie ją rinkimui.
2. Spnt – nekintamas spėjimas, nepaimta (static prediction, not taken). Atšaka nebus vykdoma, neskirti jokių resursų statistikos apie ją rinkimui.

3.Dptk – dinaminis spėjimas, paimta (dynamic prediction, taken). Atšaka greičiausiai bus paimta, tačiau rinkti statistinę informaciją apie šios šakos vykdymą arba nevykdymą ir pasinaudoti sekančio spėjimo metu.

4.Dpnt – dinaminis spėjimas, nepaimta (dynamic prediction, not taken). Tas pats kaip ir dptk atveju tik šaka greičiausiai nebus vykdoma.

Šie keturi modifikatoriai suteikia procesoriui informacijos kaip elgtis prognozuojant atšaką. Didžiausią tikimybę dėl atšakos vykdymo kompiliatorius nurodo naudodamas static prediction modifikatorių, mažesnę – su dynamic prediction. Taip pat visiems keturiems modifikatoriams dar galima nurodyti, kiek instrukcijų procesorius galėtų pakrauti į savo instrukcijų laikinąją atmintį. Tai būtų many (daug) ir few (keletą). Many atveju procesorius gali pakrauti visą kešuojančią atmintį, few atveju – keletą instrukcijų. Šie modifikatoriai taip pat susiję su atšakos vykdymo tikimybe. Visų šių modifikatorių realizacija iš tikrųjų nėra privaloma IA-64 architektūros tipo procesoriui, kadangi jų buvimas ar nebuvimas niekaip neįtakoja programos kodo vykdymo teisingumo [Int06a]. Kitas dalykas yra tas, jog šių modifikatorių priskyrimas šakoms gali būti atliekamas nebūtinai kompiliavimo metu. Dauguma kompiliatorių (pvz. gcc – Gnu Compiler Collection) turi profiliavimo galimybę, kuomet programa sukompiliuojama su raktu, liepiančiu generuoti programos vykdymo statistiką. Tuomet programa paleidžiama, o surinkta informacija apie įvairius bazinių blokų tiek vykdymus, tiek vykdymo skaičių gali būti panaudojama dvejetainio optimizatoriaus, kuris analizuoja jau sukompiliuotos programos mašininį kodą ir pagal profilio informaciją sudėlioja atšakų spėjimo modifikatorius [KIF99]. Nors šie modifikatoriai dažniausiai skiriami ciklų instrukcijoms, bei sąlyginiams perėjimams, jie taip pat tinkami procedūrų kvietimo bei grįžimo komandoms. Atskiru atveju su specialia instrukcija netgi galima nurodyti kvietimo tikimybę adresui saugomam viename iš sąlyginio vykdymo registrų (branch registers).

3.4 RSE – Registrų steko variklis ir parametrų perdavimas procedūroms

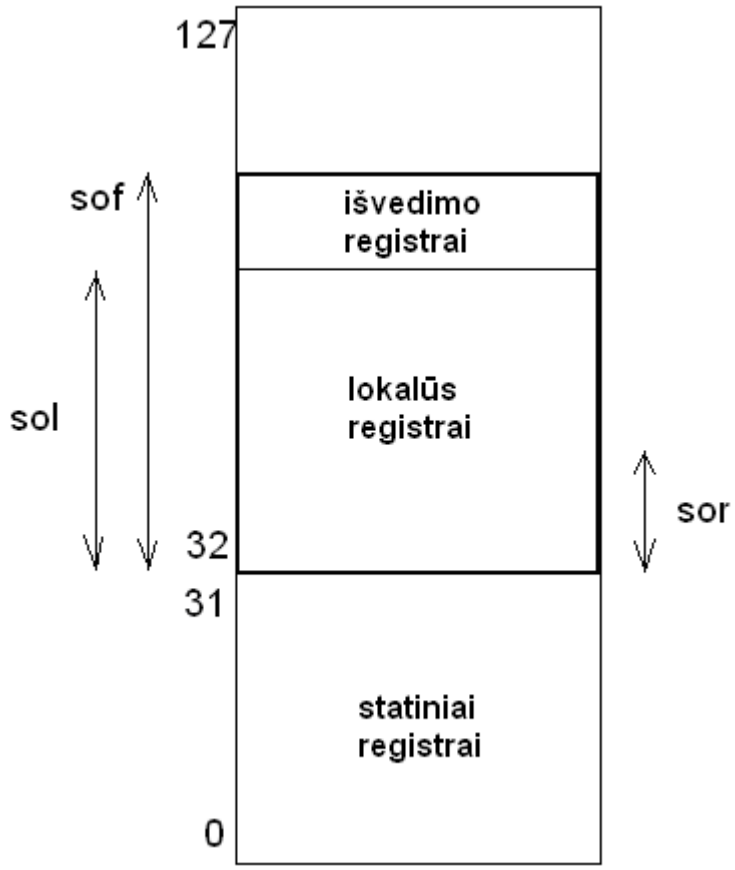
Itanium architektūroje visi bendrojo naudojimo registrai yra suskirstyti į dvi grupes. Pirmoji grupė yra statinė – tai registrai r0-r31. Antroji grupė yra dinaminė tai visi likę r32-r127 registrai, kurie formuoja registrų steką, automatiškai valdomą registrų steko variklio procedūrų iškvietimo ar grįžimo metu. Stekas arba registrų langas (register stack frame) yra padalintas į dvi dinamiškai keičiamas dalis – pirmoji lokaliems ir įvedimo registrams, antroji išvedimo registrams. Procedūros iškvietimo metu registrai automatiškai pervadinami taip, jog kviečiančiosios procedūros išvedimo registrai formuoja iškviečiamos procedūros registrų lango pradžią. Grįžtant iš procedūros, registrų langas atstatomas į prieš tai buvusią kviečiančiosios procedūros būseną. *Alloc* instrukcija [Int06c]

naudojama dinaminiam registrų lango valdymui (tiek pačio lango, tiek abiejų regionų dydžių keitimui) iškviečiamos procedūros viduje. Nėra reikalaujama iš procedūros naudoti *alloc* instrukciją, jei ji neketina nieko rašyt į savo registrų steką. Ji dar gali naudoti registrus, kurie kviečiančiojoje procedūroje buvo pažymėti kaip išvedimo, tačiau negali nieko rašyti į bet kokį lango registrą, prieš tai nepanaudojusi *alloc* instrukcijos. Šios taisyklės nepaisymas sukeltų išimtį. Itanium architektūra aparatūriškai neskirsto registrų į įvedimo ir lokalius. Kviečiančiosios procedūros išvedimo registrai automatiškai tampa visu kviečiamosios procedūros registrų steku, jei nenaudojama *alloc* instrukcija. Architektūra numato, jog iki aštuonių išvedimo registrų yra naudojama parametrų perdavimui.

Už procedūros registrų langą yra atsakingas CFM registras, kuris automatiškai išsaugomas atstatomas procedūros iškvietimo grįžimo metu [KIF99]. CFM registro formatas (7 pav.):

7 pav. CMF registro formatas

Taip atrodo procedūros registrų stekas (8 pav.):



8 pav. Registrų steko struktūra

CFM registro laukų reikšmės:

- Rrb yra bendra laukų grupė, pagal kurią apskaičiuojamas atitikmuo tarp besisukančių ir fizinių registrų. Pr; fr ir gr yra atitinkamai predikatų registrų, slankaus kablelio registrų ir bendrųjų registrų bazės.
- Sor – besisukančių registrų lango dydis (size of rotating registers).
- Sol – procedūros lango lokalių ir įvedimo registrų dydis (size of locals).
- Sof – procedūros lango dydis (size of frame).

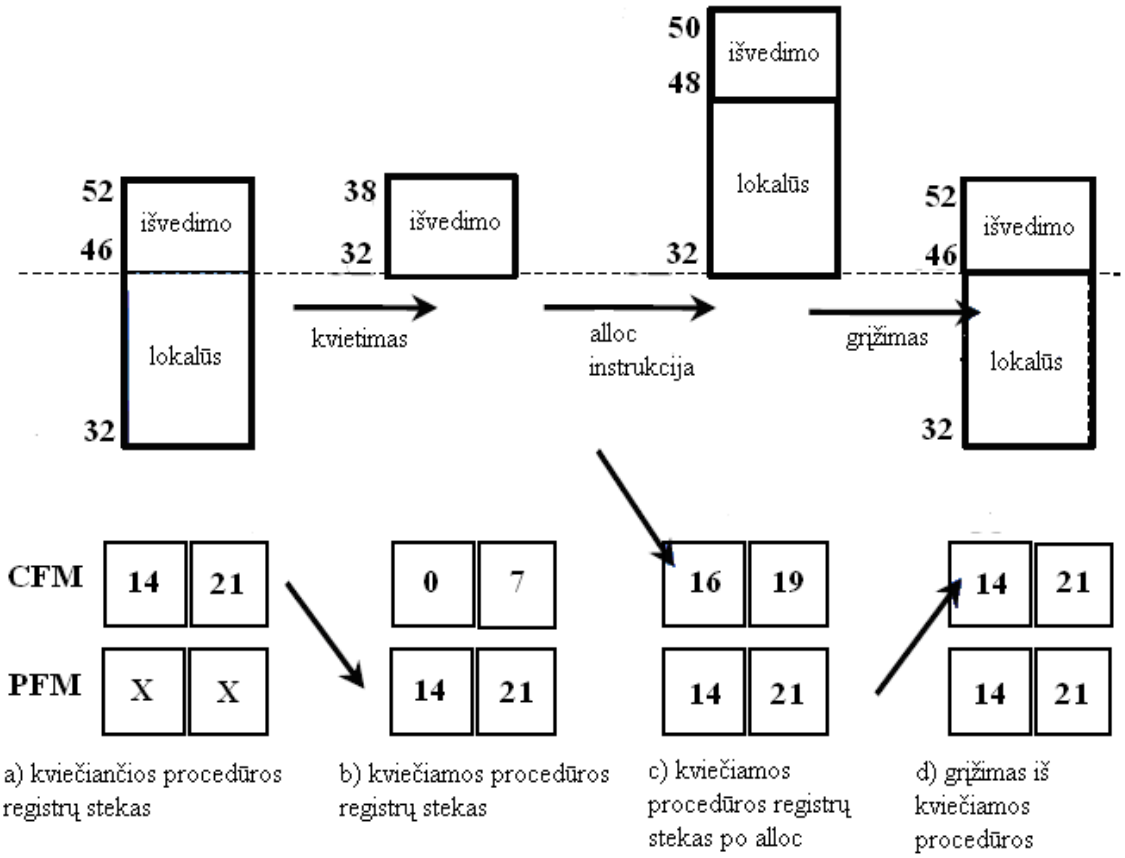
Besisukantys registrai realizuoti yra taip. Tarkim rbr.gr reikšmė yra nulis, o sor reikšmė yra 5. Vadinasi besisukančių registrų langas yra 5 registrų dydžio. Registrų rotacijos principas pateiktas 9 paveikslėlyje. Pradinės procedūros lango registrų reikšmės pavaizduotos atveju a), fizinių registrų reikšmės pavaizduotos c):

a)	b)	c)
R36 -> 4	R36 -> 0	R36 -> 4
R35 -> 3	R35 -> 4	R35 -> 3
R34 -> 2	R34 -> 3	R34 -> 2
R33 -> 1	R33 -> 2	R33 -> 1
R32 -> 0	R32 -> 1	R32 -> 0

9 pav. Registrų rotacija

Padidiname rbr.gr reikšmę vienetu. Dabar kreipiantis į „virtualų“ registrą R32, fizinio registro adresas skaičiuojamas prie virtualaus pridedant atitinkamą rbr lauko reikšmę, šiuo atvejų vienetą. Gauname fizinį registrą R33, todėl b) atveju atrodo, kad padidinus rbr.gr reikšmę vienetu, virtualių registrų reikšmės nusileido per vieną registrą žemyn, o pirmojo registro reikšmė atsidūrė paskutinio registro vietoje. Taip sukuriamas besisukančių registrų efektas. Verta paminėti, jog visos instrukcijos operuoja tiktai virtualiais registrais, o CFM registras architektūriškai nėra tiesiogiai prieinamas. Analogiškai gauname ir su predikatiniais, bei slankaus kablelio besisukančiais registrais.

Registrų lango pavyzdys vienos procedūros iškvietimo iš kitos metu (10 pav.):



10 pav. Registrų steko langas kviečiant procedūras

Kaip matome a) atveju kviečiančioji procedūra turi 14 lokaliu registrų ir 7 išvedimo registrus, atitinkamai laukuose sol 14 ir sof 21 (14 lokalių + 7 išvedimo). Iškvietus kitą procedūrą b) atveju CFM registro reikšmės patenka į specialių registrų masyvą, išimtinai valdomą procesoriaus PFM (Previous Frame Markers). O naujos CFM laukų sol ir sof reikšmės atitinkamai tampa 0 ir 7, nes lokalių registrų kol kas nulis, o registrų lango dydis atitinka kviečiančios procedūros išvedimo registrų dydį. Tarkim kviečiama procedūra įvykdo komandą alloc, su kuria nurodo, jog reikia 16 lokalių registrų bei 3 išvedimo. Tuomet c) atveju į sol ir sof laukus atitinkamai įrašomos reikšmės 16 ir 19. Grįžtant iš šios procedūros, CFM laukai atstatomi iš PFM registrų ir taip kviečiančioji procedūra vėl naudoja savo registrų langą – atvejis d).

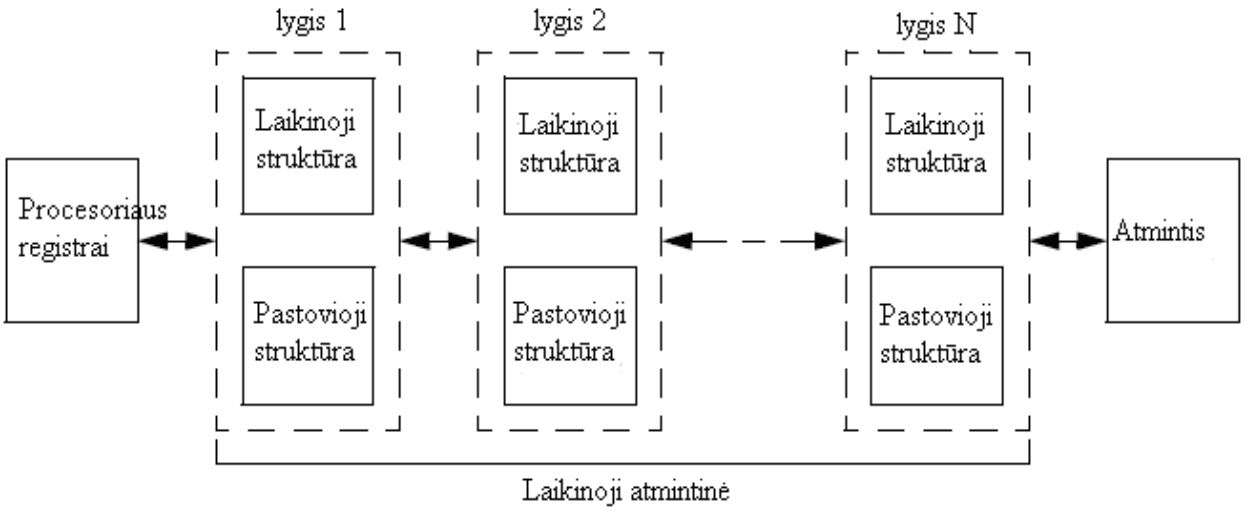
Toks registrų valdymo principas yra labai svarbus, nes leidžia atsisakyti parametrų perdavinėjimo tarp procedūrų per atminties steką, bei leidžia išvengti bereikalingo registrų kopijavimo, perduodant parametrus. Taip pat registrų pervadinimas arba „apsukinėjimas“ (rotating) leidžia aparatūriškai įgyvendinti vieną iš ciklų optimizacijos technikų vadinamą software pipelining – apie ją vėliau. Žinoma fiziškai registrų stekas nėra begalinis, nors programuotojui tokia iliuzija ir suteikiama. Pritrūkus fizinių registrų, procesorius automatiškai jų reikšmes gali iškelti į operatyviąją atmintį, o grįžtant iš procedūrų – automatiškai atstatyti kviečiančiosios procedūros kontekstą. Nors

šiuo atveju ir panaudojama operatyvioji atmintis, bet procesorius gali efektyviai paslėpti išsaugojimo bei atstatymo užvėlavimus, neturėdamas arba turėdamas tik minimalią įtaką programos vykdymo spartai.

3.5 Spartinančiosios atmintinės (cache) valdymas

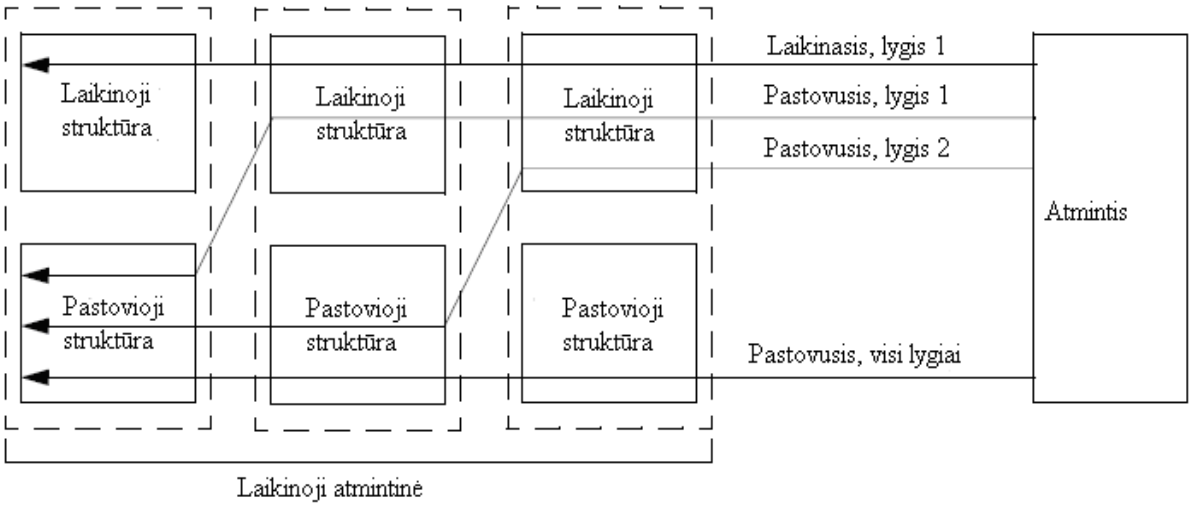
Itanium architektūra leidžia kompiliatoriui valdyti ne tik atšakų spėliojimą, bet su komanda *lfetch* ir tam tikrais modifikatoriais suteikia galimybę nurodyti kokius duomenis ir į kurią iš trijų spartinančiųjų atminčių (cache) įkrauti [Int04]. Modifikatoriai tinka ir bet kokiai skaitymo iš atminties komandai, tačiau komanda *lfetch* yra skirta išskirtinai duomenų pakrovimui iš operatyviosios atminties į laikinąją. Kaip jau buvo minėta visi šie nurodymai (hints) nėra privalomi Itanium mikroarchitektūros realizacijai, nes neturi įtakos programos vykdymo korektiškumui.

Spartinančiosios atminties hierarchija atrodo šitaip (11 pav.):



11 pav. Kelių lygių spartinančioji atmintinė

Matome keletą spartinančiosios atminties lygių (priklausomai nuo mikroarchitektūros realizacijos), kiekvienas lygis dar yra dalinamas į dvi dalis laikinąsias (temporal) struktūras ir pastoviąsias (non-temporal). Laikinosios struktūros saugo duomenis, kurie galimai artimai susiję laiko atžvilgiu (temporal locality), priešingai pastoviosioms struktūroms. Yra keturi instrukcijų modifikatoriai, kurie nurodo, į kurį spartinančiosios atmintinės lygį geriausia patalpinti skaitomą duomenų seką. Jeigu seka jau yra esamame arba aukštesniame atmintinės lygyje – nevyksta joks duomenų judėjimas tarp lygių. Įmanomi duomenų patalpinimo atmintinėje keliai (12 pav.):



12 pav. Duomenų judėjimas kelių lygių spartinančiojoje atmintinėje

Nurodant, kad duomenys bus laikinojoje struktūroje, daroma prielaida, kad jie bus panaudoti netolimoje ateityje, ir priešingai – nelaikinojoje struktūroje duomenys greičiausiai nebus reikalingi iškart. Pirmąjį duomenų saugojimo kelią atitinka modifikatorius „none“. Tai reiškia, kad skaitomi duomenys yra patalpinami visose trijuose atmintinės lygiuose. Antrąjį kelią atitinka modifikatorius „nt1“, atitinkamai trečiąjį ir ketvirtąjį kelius atitinka modifikatoriai „nt2“ bei „nta“. Paprasčiausias optimizavimo pavyzdys, panaudojant anksčiau išvardintas galimybes, galėtų būti toks – tarkime turime ciklą, kuris suskaičiuoja visų masyvo elementų sumą. Tarkim elemento dydis – vienas baitas. Tuomet ciklo vykdymo metu, skaitant kiekvieną elementą, tektų laukti kol iš atminties bus nuskaitytas baitas. Panaudojant instrukciją *lfetch*, užtektų kas 8 ciklo iteraciją į spartinančiąją atmintį užkrauti 8 baitų duomenų bloką, taip sumažinant sekančių septynių iteracijų duomenų skaitymo kaštus.

3.6 Ciklų optimizavimas

Dažnai programos darbo efektyvumas priklauso nuo to kaip gerai ciklai esantys kodo viduje išnaudoja procesoriaus teikiamus resursus. Tačiau labai dažnai juose nebūna pakankamai nepriklausomu instrukcijų, kad būtų įmanoma paslėpti duomenų užkrovimo užlaikymus, pavyzdžiui šitas kodo pavyzdys yra labai sunkiai lygiagretinamas (13 pav.):

```
L1:ld4    r4 = [r5],4;; // 0 taktas
  add     r7 = r4,r9;;  // 2 taktas
  st4     [r6] = r7,4   // 3 taktas
  br.cloop L1;;         // 3 taktas
```

13 pav. for ciklo iteracija

Šiame pavyzdyje iteracija X+1 pradeda vykdyti tik tada, kai visos instrukcijos iš iteracijos X jau yra įvykdytos. Darant prielaidą, jog atminties adresai tiek skaitymui, tiek rašymui skiriasi, procesoriaus resursų išnaudojimas gali būti patobulintas, iškeliant nepriklausomas instrukcijas iš iteracijos X+1 į iteraciją X. Tokiu būdu vykdymo atžvilgiu abi iteracijos dalinai viena kitą perdengia. Yra du bendri metodai iteracijų perdengimams, kurie abu priverčia augti kodo dydį tradicinėse architektūrose – ciklo išvyniojimas (loop unrolling) bei software pipelining. Itanium architektūra įgyvendina savus metodus spręsti kodo padidėjimo problemai [RDG04+].

3.6.1 Ciklo išvyniojimas (loop unrolling)

Ciklo išvyniojimas tai technika, kuomet ciklo kūnas kopijuojamas kelis kartus, ir kopijos sujungiamos sudarant naują kūną. Tokiu būdu viena nauja iteracija įvykdo kelias originalaus ciklo iteracijas. Kadangi sujungiamos kelios nepriklausomos iteracijos tai galima bandyti ciklo kūno instrukcijas lygiagretinti. Pradiniam pavyzdžiui pritaikę dvigubą „išvyniojimą“, gauname (14 pav.):

```
L1:ld4    r4 = [r5],4;;   // 0 taktas
  ld4     r14 = [r5],4;;  // 1 taktas
  add     r7 = r4,r9;;    // 2 taktas
  add     r17 = r14,r9    // 3 taktas
  st4     [r6] = r7,4;;   // 3 taktas
  st4     [r6] = r17,4    // 4 taktas
  br.cloop L1;;           // 4 taktas
```

a) dvigubai išvyniotas ciklas

```
  add     r15 = 4,r5
  add     r16 = 4,r6;;
L1:ld4    r4 = [r5],8         // 0 taktas
  ld4     r14 = [r15],8;;     // 0 taktas
  add     r7 = r4,r9          // 2 taktas
  add     r17 = r14,r9;;      // 2 taktas
  st4     [r6] r7,8           // 3 taktas
  st4     [r16] = r17,8       // 3 taktas
  br.cloop L1;;               // 3 taktas
```

b) dvigubai išvyniotas ciklas su indukcinio kintamojo praplėtimu į du registrus

14 pav. Ciklo „išvyniojimas“

Pirmuoju a) atveju matome, kad netgi dabar nelabai galime sumažinti reikalingų taktų skaičių, kadangi dvi pirmos ir dvi paskutines operacijos naudoja du tuos pačius registrus (r5 saugomas skaitomos atminties adresas, r6 – rašomos). Tačiau pastebime, kad iteracijos metu keičiasi (didinamos arba mažinamos ketvertu – komandos ld ir st [Int06c]) abiejų šių registrų reikšmės. Tokie kintamieji, kurie keičiami ciklo iteracijos metu (didinami arba mažinami), vadinami indukcijos kintamaisiais. Jeigu abu šiuos registrus „praplėstume“ – tai yra, skaitymo atveju registro r5 reikšmė rodytų į ptr, o registro r15 reikšmė rodytų į sekantį elementą ptr+4, bei analogiškai su registru r6. Tokiu atveju tiek abi skaitymo tiek abi rašymo operacijos galėtų būti vykdomos iš karto – atvejis b). Matome, kad šio ciklo ir originalaus ciklo iteracija vykdoma per tiek pat taktų, nors b) atveju per vieną iteraciją įvykdome dvi originalaus ciklo iteracijas – darbo efektyvumas

padvigubėja. Analogiškai ciklas galėtų būti išvyniojamas keturis kartus, bet tada kartu didėja ir ciklo kodo dydis.

3.6.2 Kodo „kanalas“ (software pipelining)

Software-pipelining tai ciklų optimizavimo technika, kurios veikimo principas panašus į vykdomojo konvejerio. Kiekviena iteracija yra padalinama į vieno takto ilgio fazes, kurias sudaro nulis arba daugiau instrukcijų. Paimkime prieš tai nagrinėtą pavyzdį (15 pav.):

```
1:ld4 r4 = [r5],4
2:---
3:add r7 = r4,r9
4:st4 [r6] = r7,4
```

15 pav. Ciklo iteracijos suskirstymas fazėmis

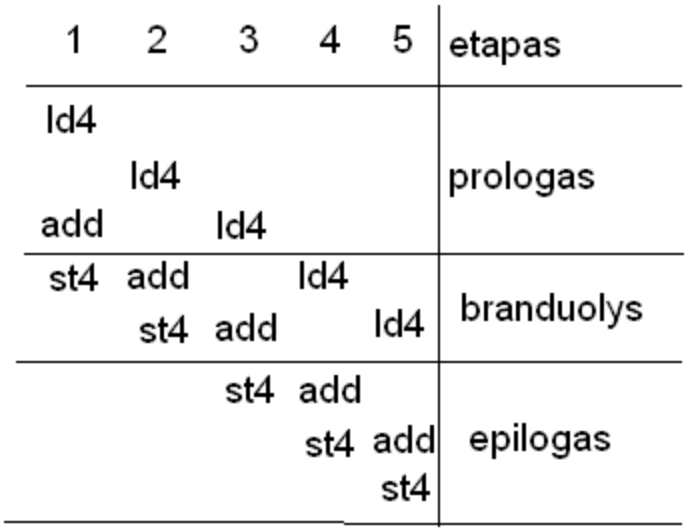
Pirmos fazės metu vyksta duomenų pakrovimas iš atminties, antra fazė yra tuščia – nes reikalingi du taktai (juos atitiks dvi iteracijos) duomenims nuskaityti. Trečios fazės metu atliekami skaičiavimai. Ketvirta fazė rašo duomenis į atmintį. Taip atrodo penkios perdengtos iteracijos (16 pav.):

```
  1    2    3    4    5          iteracija
-------------------------------------
  ld4                              X
      ld4                         X+1
  add      ld4                    X+2
  st4 add      ld4                X+3
      st4 add      ld4            X+4
          st4 add                 X+5
              st4 add             X+6
                  st4             X+7
```

16 pav. Iteracijų perdengimas

Taktų skaičius tarp dviejų viena po kitos einančių iteracijų pradžių vadinamas inicijavimo intervalu (Initiation Interval - II). Šiuo atveju II lygus vienam, o kiekviena fazė yra II taktų ilgio. Taip optimizuoti ciklai turi tris vykdymo etapus: prologą, branduolį (kernel) ir epilogą. Prologo etapo metu kiekviena iteracija pradedama kas II taktų, norint užpildyti ciklo „kanalą“ (pipeline). Per pirmą prologo taktą, vykdoma pirma pirmos iteracijos fazė, per antrąjį – antra pirmos iteracijos fazė, bei pirma antros iteracijos fazė ir t. t. Prieš prasidedant branduolio vykdymo etapui, ciklo „kanalas“ pilnai užpildytas. Branduolio etapo vykdymo metu, kas II taktų skaičių pradedama nauja

iteracija ir kita pabaigiama. Epilogo etapo metu naujos iteracijos nepradedamos, o baigiamos vykdyti tos, kurios buvo pradėtos anksčiau – taip ciklo „kanalas“ ištuštinamas. Žemiau pavaizduotos visi ciklo etapai (17 pav.):



17 pav. Ciklo vykdymas etapais

Taip optimizuoto ciklo kodas labai skiriasi nuo originalaus. Šiame pavyzdyje matome, jog antros iteracijos nuskaitymo komanda yra įvykdoma žymiai anksčiau, negu duomenų išsaugojimo operacija pirmojoje iteracijoje. Todėl kiekvienos iteracijos duomenų pakrovimas reikalauja skirtingų registrų, siekiant išvengti reikalingų reikšmių perrašymo, dar nespėjus jų panaudoti. Taip pat tradicinėse architektūrose toks ciklų optimizavimo būdas reikalauja ciklų išvyniojimo bei programinio registrų pervadinimo. Be to kiekvienas etapas taip pat reikalauja kodo generavimo, ko pasekoje ciklo kodas ir instrukcijų skaičius smarkiai padidėja [ASU01].

Itanium architektūros teikiamos galimybės laisvai leidžia išvengti aukščiau išvardintų ciklų optimizavimo bėdų. Automatiškai pervadinami registrai (rotating registers) registrai panaikina būtinybę programiškai juos pervadinti. Taip pat specialios ciklų komandos, automatiškai vykdydamos registrų pervadinimą bei naudodamos predikaciją, leidžia išvengti būtinybės generuoti papildomą kodą prologo bei epilogo etapams. Kaip jau buvo minėta, registrų pervadinimas arba rotacija vyksta, registro numerį pridedant prie registro pervadinimo bazės (rrb). Speciali ciklo vykdymo instrukcija po kiekvienos ciklo iteracijos sumažina bazę vienetu – taip suteikiamas įspūdis, jog registro R reikšmė atsidūrė registre R+1. Jeigu R yra didžiausias leidžiamas registro numeris – R reikšmė atsiduria žemiausiam registre (r32). Registrų pervadinime dalyvauja dinamiškai valdoma bendrųjų registrų (r32-rN) dalis, bei nekintančios slankaus kablelio (f32-f128) ir predikatinių registrų (p16-p64) dalys. Pastarųjų predikatinių registrų pervadinimas tarnauja dviems tikslams. Pirmasis – būtinybė išvengti predikatinės registro reikšmės perrašymo sekančios

iteracijos metu, kai ta reikšmė vis dar reikalinga. Antras tikslas – predikatiniai registrai naudojami užpildant bei ištuštinant ciklo „kanalą“. Norėdamas tai pasiekti, programuotojas priskiria po predikatinį registrą kiekvienai ciklo fazei, tam kad galėtų kontroliuoti tos fazės instrukcijų vykdymą. „For“ ciklams arba kitaip ciklams, kurių iteracijų skaičius iš anksto žinomas, registras p16 architektūriškai yra priskiriamas pirmajai fazei, p17 – antrajai ir t. t. (18 pav.):

```
1:(p16)ld4 r4 = [r5],4
2:(p17)---
3:(p18)add r7 = r4,r9
4:(p19)st4 [r6] = r7,4
```

18 pav. Predikatų susiejimas su fazėmis

Registrų pervadinimas vykdomas kiekvienos ciklo iteracijos metu. Todėl iš pradžių yra vykdoma tik pirma fazė, nes kol kas tik registras p16 yra vienetas. Po ciklo instrukcijos įvykdymo p16 reikšmė pasislenka į p17 – taip sekančios iteracijos metu bus įjungiama pirma ir antra fazės ir t. t. Tokiu būdu visi trys ciklo vykdymo etapai sutampa ir tik nuo predikacinių registrų priklauso koks ciklo etapas yra duotu laiko momentu vykdomas. Norėdamas sukurti efektyvų kodo „kanalą“ kompiliatorius turėtų atsižvelgti į vykdymo konvejerių užlaikymą. Taip yra pirmojoje fazėje (skaitymas iš atminties du taktai), todėl antra fazė (sekantis taktas) yra palikta tuščia. Taip atrodytų pilnas pradinio pavyzdžio ciklo kodas (19 pav.):

```
 mov lc = 199                              //1
 mov ec = 4                                //2
 mov pr.rot = 1<<16;;                      //3
L1:
(p16)ld4 r32  = [r5],4   // 0 taktas      //4
(p18)add r35  = r34,r9   // 0 taktas      //5
(p19)st4 [r6] = r36,4    // 0 taktas      //6
 br.ctop L1;;            // 0 taktas      //7
```

19 pav. Ciklo vykdymas konvejerio principu

Tarkime ciklas turi 200 iteracijų. Pirmoje eilutėje į specialų registrą lc (loop counter) įrašome vienetu mažesnių iteracijų skaičių (nes viena papildoma iteracija įvykdoma „kanalo“ užpildymo metu). Antroje eilutėje į registrą ec (epilog counter) įrašome epilogo fazių skaičių + viena. Trečioje eilutėje inicializuojame predikatinių registrų masyvą – registre p16 vienetas, visuose kituose – nuliai. Ciklo kūnas pakeistas taip, kad visi naudojami indukciniai kintamieji turi būti saugomi statiniuose registruose (nesisukančiuose). Taip pat matome, jog išnyko registrų rašymo ir skaitymo

priklausomybės (būtent šios optimizacijos efektas), todėl visas ciklo kūnas gali būti įvykdomas per vieną taktą. Pateiksime lentelę su registrų reikšmėmis ciklo vykdymo metu (20 pav.):

Taktas	Vykdymo konvejeris/instrukcija				Būsena prieš br.ctop instrukciją					
	M	I	M	B	p16	p17	p18	p19	LC	EC
0	ld4			br.ctop	1	0	0	0	199	4
1	ld4			br.ctop	1	1	0	0	198	4
2	ld4	add		br.ctop	1	1	1	0	197	4
3	ld4	add	st4	br.ctop	1	1	1	1	196	4
...	...	...	...	...	...	...	...	...	...	...
100	ld4	add	st4	br.ctop	1	1	1	1	99	4
...	...	...	...	...	...	...	...	...	...	...
199	ld4	add	st4	br.ctop	1	1	1	1	0	4
200		add	st4	br.ctop	0	1	1	1	0	3
201		add	st4	br.ctop	0	0	1	1	0	2
202			st4	br.ctop	0	0	0	1	0	1
...					0	0	0	0	0	0

20 pav. Predikatų bei registrų reikšmės ciklo vykdymo metu

Iš čia galime matyti kaip yra užpildomas „kanalas“, bei kaip vėliau vykdomas epilogo etapas, kontroliuojamas predikatinių registrų. Br.ctop instrukcija iš pradžių tikrina LC registrą, jeigu jis nelygus vienetui – sumažina jo reikšmę vienetu, įrašo į patį viršutinį predikatini registrą p63 vienetą (ši reikšmė po registrų apsukimo atsidurs p16 ir – ciklas toliau tęsiamas), sumažina visus tris rrb laukus vienetu (atliekama registrų rotacija), bei peršoka į žymę L1 (pradedama nauja iteracija – branduolio arba prologo etapas). Jeigu LC reikšmė nulis – toliau tikrinama EC reikšmė. Jei ji daugiau už vienetą, į viršutinį predikatinį registrą įrašoma nulis (po registrų pervadinimo jis atsidurs p16 ir taip paeiliui išjungs visų II fazių instrukcijas), sumažinama EC reikšmė bei visi rrb laukai vienetu, vykdoma atšaka į žymę L1 – taip ištuštinamas „kanalas“. Jeigu EC reikšmė lygi vienetui – ciklas baigtas ir peršokimas į žymę L1 nesuveikia. Egzistuoja specialus atvejis kai EC yra 0 tuomet reiškia, kad ciklas buvo vykdomas kaip specialus „išvyniotas“ ciklas.

Ciklai, kurių iteracijų skaičius iš anksto nežinomas (while ciklai), vykdomi panašiai, tiktai vietoj iteracijų skaitliuko naudojamas dar vienas sąlygos predikatas; kuris apsprendžia ciklo vykdymo sąlygą ir yra skaičiuojamas kiekvienos iteracijos metu. Apskritai „kanalo“ kūrimas šių ciklų atveju yra labai panašus į „for“ ciklų.

Apibendrinant galima teigti, kad Itanium sukurta architektūra suteikia labai plačias galimybes generuoti efektyvų kodą. Čia apžvelgtos architektūros savybės bei pavyzdžiai demonstruoja nemažą dalį būdų, kurie gali būti ir yra pritaikomi kompiliatorių generuojamo optimizavimui:

- Aiškiai išreikštas lygiagretumas instrukcijų lygyje leidžia išnaudoti visus procesoriaus vykdymo konvejerius vieno takto metu. Taip grupuojant instrukcijas, pasiekiama labai aukšta mašininio kodo vykdymo sparta.
- Predikacija leidžia atsisakyti iki 50% atšakų, suteikdama galimybę procesoriui vienu metu vykdyti abi nuo sąlygos priklausomas kodo atšakas.
- Išankstinis duomenų užkrovimas leidžia paspartinti procesoriaus darbą, sumažinant priklausomybes tarp problematiškų instrukcijų, susijusių su atmintimi. Tokiu būdu instrukcijos gali būti laisvai kilnojamos tarp skirtingų grupių, taip paslepiant atminties skaitymo užlaikymus.
- Atšakų prognozavimo žymės leidžia kompiliatoriaus sukauptą informaciją apie atšakų vykdymą, kuri paprastai būna žymiai išsamesnė negu procesoriaus, perteikti procesoriui, taip iki minimumo sumažinant nepageidaujamas prastovas dėl neteisingai nuspėtos kodo atšakos.
- Lankstus spartinančiosios atmintinės valdymas suteikia galimybę labai efektyviai paslėpti skaitymo iš atminties delsimą ir yra ypatingai gerai išnaudojamas cikluose.
- Parametrų perdavimas nenaudojant steko, bei registrų pervadinimas leidžia išvengti bereikalingo parametrų kopijavimo, bei smarkiai paspartinti procedūrų iškvietimus ir sugrįžimus iš jų.
- Predikacija, registrų pervadinimas bei specialios ciklų vykdymo komandos leidžia galingą ciklo optimizavimo techniką – software pipelining perkelti į procesoriaus lygmenį, išvengiant tokių problemų kaip papildomo kodo generavimas bei ribotas registrų kiekis, kurios yra tipinės tradicinių architektūrų kodo optimizavime.

Itanium architektūra suteikia kompiliatoriui galimybę žymiai geriau valdyti procesorių ir generuoti efektyvesnį kodą, panaudojant aukščiau išvardintas savybes.

4. IA-64 savybių tyrimas

4.1 Tyrimo aplinka ir priemonės

Šnekant apie programos kodo pertvarkymą optimizavimo tikslais, visada siekiama vieno iš dviejų dalykų – programos apimties sumažėjimo arba programos vykdymo paspartėjimo. Kadangi IA-64 visos anksčiau minėtos savybės yra skirtos būtent programos vykdymo spartinimui, tirsime kaip kaip jos įtakoja programos vykdymo laikus.

Norint tiksliai išmatuoti laiko skirtumą tarp optimizuotos ir neoptimizuotos programos, neužtenka vien tiktai rasti laiko skirtumus tarp programų vykdymo pradžios ir pabaigos. Pirma šiuolaikinėse architektūrose bei operacinėse sistemose, programa labai retai vykdoma nepertraukiamai iki pat jos pabaigos. Vien dėl daugiagijiškumo programa labai dažnai yra pertraukiama, norint sureaguoti į kitus įvykius, kuriuos privalo apdoroti moderni operacinė sistema.

Antras dalykas pačios savybės leidžia optimizuoti kodą labai lokaliai (keleto instrukcijų lygyje), ir gaunama nauda keliomis dešimtimis ar šimtais taktų. Šiuolaikiniai Itanium 2 procesoriai dirba 1,5 Ghz dažniu – tai reiškia, kad per sekundę sugeneruojama iki pusantro milijardo taktų. Išmatuoti keliolikos taktų skirtumo tarp dviejų programos versijų praktiškai neįmanoma.

Todėl laiko matavimui pasirinktas Linux pagalbinis įrankis *time* [ISN08]. Šios programos išvedami duomenys pavaizduoti 21 pav.

```
tava2834@sgi01:/scratch/tava2834/perfTest> time ./fTest for opt
Test: for
Optimize: 1
Success

real    0m9.416s
user    0m9.404s
sys     0m0.000s
tava2834@sgi01:/scratch/tava2834/perfTest>
```

Pav. 21 programa *time*

Pirmas parametras „real“ yra testuojamos programos vykdymo laikas nuo programos darbo pradžios iki pabaigos, įskaičiuojant ir laiką, kuris buvo reikalingas apdoroti operacinės sistemos įvykius. Trečias programos parametras „sys“ apibrėžia laiką, kuris buvo praleistas būtent kernelio aplinkoje (reaguojant į įvykius). Mus domina būtent vidurinysis parametras „user“, kuris tiksliai nusako, kiek laiko tiksliai programa vykdė savo kodą. Šiuo parametru ir remsimės matuodami testavimo programų vykdymo laikus.

Antrajai problemai spręsti reikia pačią testuojamąją programą pertvarkyti taip, jog dominantis kodo fragmentas būtų vykdomas pakankamai ilgą laiką – kodo fragmentą įdedant į ciklą su pakankamai ilgu iteracijų kiekiu. Tarkime neoptimizuoto kodo fragmento vykdymo laiką pažymėsime Tn, optimizuoto – To. Jei programa šiuos fragmentus vykdytų po vieną kartą, šie laikai būtų labai maži ir su operacinės sistemos galimybėmis praktiškai neišmatuojami. Tačiau, jei juos panaudosime cikle, tuomet iš 22 pav. matyti, kad santykis Tn/To artėja prie testuojamų programų vykdymo laikų santykio, kai iteracijų skaičius k yra pakankamai didelis. Neoptimizuotos programos vykdymo laikas pažymėtas raide Pn, optimizuotos – Po. M raide pažymėta konstanta, reiškianti vykdymo laiko pastoviąją dalį (kintamųjų inicializavimas ir t. t.).

$$\frac{Pn}{Po} = \frac{M + Tn \times k}{M + To \times k} \approx \frac{Tn}{To}$$

22 pav.

Toliau tiriamos savybės bus vertinamos pagal programų vykdymo laiko santykį su skirtingais iteracijų skaičiais. Pats savybių tyrimas vyks tokių principu – su C kalba parašomas dažnai įvairiose programose pasitaikantis kodas (įvairūs ciklai, reiškinių skaičiavimas ir panašiai), tuomet programa sukompiliuojama su kompiliatoriumi be jokių papildomų optimizacijų, taip gaunamas pradinis neoptimizuotas programos variantas. Pritaikant konkrečią savybę, programos C kodas dar sykį perleidžiamas per kompiliatorių tik šį kartą transliuojant jį į asemblerio kodą, kuris atitinkamai modifikuojamas ir transliuojamas į mašininį kodą. Taip gaunamos dvi programos versijos, kurias jau galima lyginti.

Norint pradėti tirti IA-64 savybes, visų pirma reikalinga aplinka (tiek architektūra, tiek operacinė sistema) testinių programų paleidimui. Visi bandymai buvo atliekami, naudojantis Vilniaus Universiteto Informacinių technologijų centro suteikta prieiga prie IA-64 architektūros serverio, turinčio šešiasdešimt keturis Itanium 2 dvigubo branduolio procesorius, veikiančius 1,5 Ghz dažniu. Serveryje įdiegta Suse Linux operacinė sistema, programų kompiliavimui buvo naudojama naujausia gcc (Gnu C compiler collection – Gnu c kompiliatoriaus įrankių rinkinys) 4.3.3 versija.

4.2 EPIC – išreikštinai lygiagretus instrukcijų vykdymas

Pagrindinė IA-64 architektūros savybė yra gebėjimas vykdyti kelias nepriklausomas instrukcijas vieno takto metu. Žinoma instrukcijų grupės, vykdomos per vieną taktą, turi tenkinti tam tikras sąlygas, kitaip procesoriaus būsena, o kartu ir vykdomos programos rezultatai tampa

neapibrėžti. Todėl būtent programuotojas turi užtikrinti taisyklingą šių instrukcijų grupavimą. Žinoma šiuolaikiniai asemblerio kalbos transliatoriai sugeba sekti nepriklausomybes tarp instrukcijų ir pranešti programuotojui apie tokio pobūdžio klaidas. Sąlygos keliamos lygiagrečiai vykdomoms instrukcijoms buvo išsamiai aprašytos pirmoje darbo dalyje, todėl dabar tik trumpai paminėsiu, jog instrukcijoms esančioms vienoje grupėje, draudžiama [Int01]:

- Rašyti, o po to skaityti iš to pačio resurso (RAW – Read After Write priklausomybė).
- Kelis kartus rašyti į tą patį resursą (WAW – Write After Write priklausomybė).

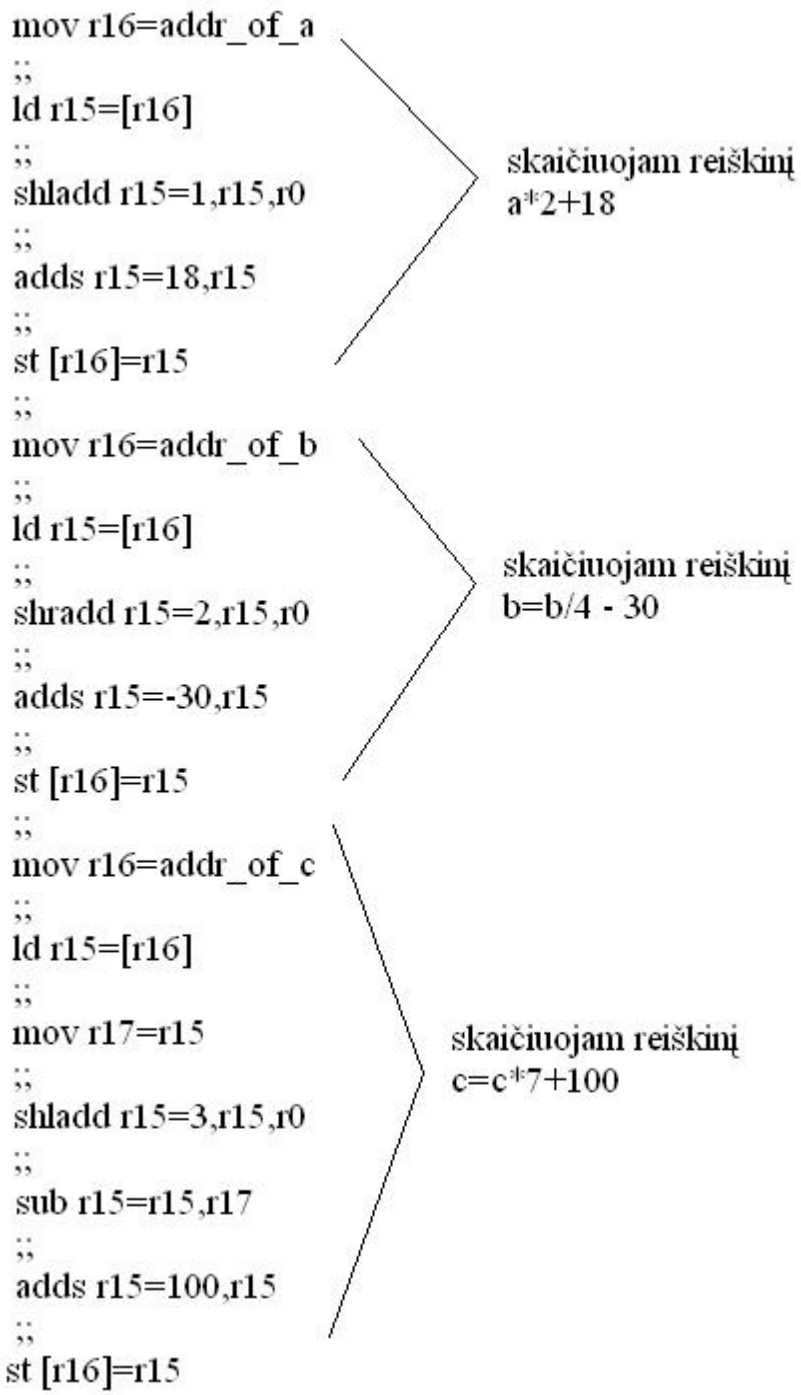
Instrukcijos esančios vienoje grupėje gali kelis kartus skaityti iš to pačio resurso, bei iš pradžių jį nuskaityti, po to rašyti. Čia minimi resursai reiškia IA-64 architektūros procesoriaus registrus, bei tą patį atminties adresą.

Norint gerai išnaudoti šią savybę, kompiliatorius turi gerai mokėti grupuoti sugeneruotas instrukcijas. Tai vadinama planavimo (scheduling) optimizacija. Išsamiau patyrinėkime šios savybės teikiamas galimybes. Tarkime turime tokį C kodo fragmentą (pavaizduotą 23 pav.)

```
int a=5;
a=a*2+18;
int b=7;
b=b/4-30
int c=9;
c=c*7+100;
a=b+c;
```

23 pav. C kodas

Kompiliatorius tokį kodą generuoja gana tiesmukai – visus iš eilės einančius reiškinius paverčia tarpusavyje priklausomomis instrukcijų sekomis (naudojančiomis tuos pačius registrus), todėl nesugrupuojamomis (24 pav.):



24 pav. Sugeneruotas kodas

Dvigubi kabliataškiai tai dalis asemblerio sintaksės, kurie išskirsto instrukcijas į grupes. Sugeneruoto kodo fragmento instrukcijoms įvykdyti reikia 17 taktų. Iš karto pastebime, jog instrukcijų grupuoti negalime, nes visur naudojami tie patys keli registrai. Tačiau mūsų nagrinėjama architektūra pasižymi tuo, jog turi didžiulį registrų failą. Tuomet kiekvienam reiškiniui skaičiuoti paskyrę po atskirus registrus, gautą kodą pertvarkome šitaip (25 pav.)

```
mov r16=addr_of_a
mov r18=addr_of_b
mov r20=addr_of_c
;;
ld r15=[r16]
ld r17=[r18]
ld r19=[r20]
;;
shladd r15=1,r15,r0
shradd r17=2,r15,r0
mov r21=r19
;;
adds r15=18,r15
adds r17=-30,r17
shladd r19=3,r19,r0
;;
st [r16]=r15
st [r18]=r17
sub r19=r19,r21
;;
adds r19=100,r19
;;
st [r20]=r19
```

25 pav. Optimizuotas kodas

Iš karto pastebime, jog šiam kodui įvykdyti reikės žymiai mažiau taktų – tiktais septynių taktų. Teoriškai turėtume gauti ((17 – 7)/17) * 100 = 58% greitesnę programą. Tačiau rezultatai rodo šiek tiek ką kitą:

Iteracijos	Pn	Po	Paspartėjimas
100000000	0 min 11,400 sek.	0 min 6,706 sek.	41,17%
200000000	0 min 22,250 sek.	0 min 13,056 sek.	41,32%

Vadinasi kažkur apytiksliai prarandam apie 1-2 taktus. Įsigilinus į IA-64 architektūros aprašymą, paaiškėja, jog instrukcijų vykdymas per vieną taktą tai pat priklauso ir nuo procesoriaus turimų vykdymo konvejerių (Execution unit) skaičiaus įvairiems instrukcijų tipams [Int04]. Kadangi Itanium 2 procesorius turi tik du skaitymo iš atminties instrukcijų vykdymo konvejerius, antroji optimizuoto kodo instrukcijų grupė vykdoma ne per vieną taktą, o per du [BT08b]. Architektūriškai procesoriui trūkstant laisvų vykdymo konvejerių įvyksta vadinamoji prastova (stall), kuri gali užimti vieną ir daugiau taktų. Tą patvirtina ir eksperimentai, kuomet pašalinus

tarkim pirmojo reiškinio skaičiavimus, optimizuoto kodo instrukcijos užima tiek pat taktų, tačiau antroje instrukcijų grupėje nebelieka trečios atminties skaitymo operacijos. Todėl likusiems dviem skaitymams įvykdyti pakanka ir vieno takto:

Iteracijos	Pn	Po	Paspartėjimas
100000000	0 min 11,400 sek.	0 min 4,694 sek.	58,82%
200000000	0 min 22,250 sek.	0 min 8,920 sek.	59,91%

Apibendrinant, galima pasakyti, kad instrukcijų grupavimas yra labai efektyvi optimizacija. Žinoma ji nebūtų įmanoma didelio registrų failo, kuomet skirtingų reiškinių rezultatus galima saugoti skirtinguose registruose, taip išvengiant pastovaus skaitymo ir rašymo į atmintį, bei mažinant priklausomybes tarp instrukcijų. Taipogi grupuojant instrukcijas, jei yra nurodyta kompiliatoriui optimizuoti konkrečiam procesoriui, tai konkretaus mikroprocesoriaus realizacija taip pat tampa reikšminga. Tačiau, bendrai paėmus, net jei su esama realizacija instrukcijų grupė nebus įvykdoma per vieną taktą, trūkstant aparatūrinių galimybių, toks kodas net nemodifikuotas žymiai našiau dirbs su procesoriais, kurie turės didesnes instrukcijų vykdymo lygiagrečiai galimybes.

4.3 Predikacija – instrukcijų išjungimas

IA-64 architektūra suteikia galimybę priklausomai nuo tam tikros sąlygos vykdomąją instrukciją, „išjungti“. Kiekvienai instrukcijai (su pora išimčių) galima priskirti po tam tikrą predikatinį registrą, ir tuomet instrukcija bus vykdoma kai registre reikšmė yra „true“ ir nevykdoma, kai reikšmė tampa „false“. Šitaip galima atsisakyti sąlyginių perėjimų, o pačiam procesoriui nereikia papildomų resursų, planuojant programos vykdymo kelią. Pateiksime paprastą C kodo pavyzdį (26 pav.):

```
if ( i % 3 == 0 ){
   a=i*2;
}else{
   a=i+20;
}
```

26 pav. Sąlyginis sakinys

Tai paprastas sąlyginis sakinys, kurį kompiliatorius paverčia tokia instrukcijų seka:

```
and r16=2,r15
;;
cmp.ne p2,p3=r16,r0
;;
(p2) br.cond .else_saka
  shladd r4=1,r15
br .exit_if
.else_saka :
   adds r4=20,r15
.exit_if :
...
```

27 pav. Sąlyginis perėjimas

Iš kodo matome, jog paprasčiausiam if sakiniui sugeneruoti reikia net dviejų perėjimų, kurių vienas yra sąlyginis ir sunkiai nuspėjamas vykdymo planuotojui. Pasitelkus predikaciją, šį kodą galima modifikuoti į žymiai efektyvesnį. Čia pateiksime tik galutinį rezultatą:

```
and r16=2,r15
;;
cmp.eq p2,p3=r16,r0
;;
(p2) shladd r4=1,r15
(p3) adds r4=20,r15
;;
...
```

28 pav. predikacija

Iš 28 pav. matome, jog kodas tapo paprastesnis – atsikratėme sąlyginių perėjimų. Taip pat sutaupėme vieną taktą, kadangi skirtingais predikatais pažymėtoms instrukcijoms (kuomet tiksliai žinoma, kad vieno predikato reikšmė yra visada priešinga kito reikšmei) negalioja rašymo į tą pati registrą priklausomybė ir jos gali būti sugrupuotos. Atlikus bandymus su programomis taip pat matyti, jog pavyko jas neblogai paspartinti:

Iteracijos	Pn	Po	Paspartėjimas
50000000	0 min 8,576 sek.	0 min 7,776 sek.	10%

Pasinaudojant predikacija, į paprastas instrukcijų sekas galima transformuoti gana sudėtingus sąlyginius sakinius, tačiau reikia atkreipti dėmesį į vieną dalyką. Grupuojant skirtingais predikatiniais registrais pažymėtas instrukcijas, galioja anksčiau minėti instrukcijų vykdymo apribojimai. „Išjungtos“ instrukcijos niekur nedingsta, jos lygiai taip pat yra kraunamos į instrukcijų

buferį, po to patenka į vykdymo konvejerius. Vienintelis skirtumas yra tas, jog instrukcijų įvykdymo rezultatai nėra išsaugomi. Todėl grupuojant tokias instrukcijas nereikėtų pamiršti, kad joms taip pat reikalingi vykdymo konvejerių resursai, o jų kaip žinia kiekis yra ribotas.

4.4 Išankstinis valdymo ir duomenų spėjimas

Šiuolaikiniai procesoriai įvairius skaičiavimus gali atlikti labai sparčiai. Tačiau dažnai procesoriams tenka tuščiai laukti, kol gerokai lėtesnė atminties magistralė pristatys duomenis reikalingus skaičiavimams. Šią problemą galima spręsti, duomenų iš atminties pakrovimo instrukcijas iškeliant gerokai anksčiau, prieš prireikiant duomenų, ir taip kompensuoti prastovas. Bet labai dažnai to padaryti neįmanoma, kadangi susiduriama su dviem problemomis. Pirmoji – labai dažnai duomenų pakrovimas priklauso nuo tam tikros sąlygos – jei sąlyga netenkinama, duomenų pakrauti neįmanoma. Pavyzdžiui, tarkim programa tikrina ar rodyklė nelygi *null*, jei taip tuomet tarkim kažkaip specialiai ją inicializuoja, ir tik tuomet įmanomas to atminties adreso skaitymas ir atliekami skaičiavimai. Savaime suprantama skaičiavimų prieš rodykles tikrinimą iškelti nepavyks, nes tuomet nukentės programos rezultatų korektiškumas. Antras dalykas – labai dažnai yra sunku nustatyti skaitymo rašymo sekoje ar atminties adresai, į kuriuos rašoma ir iš kurių skaitoma – nepersidengia. Jeigu taip tuomet neįmanoma skaitymo instrukcijos iškelti prieš rašymo. Kompiliatorius norėdamas nustatyti galimus persidengimus tradicinėse architektūrose dažnai neturi tam tinkamų priemonių.

Tačiau IA-64 architektūroje yra numatytos priemonės šioms dviems problemoms spręsti. Tai valdymo nuspėjimas (control speculation) ir duomenų pakrovimo nuspėjimas (data speculation).

4.4.1 Valdymo spėjimas

Pirmosios problemos atveju dažnai būna taip, jog programoje yra sąlyginis sakinys, kuris tikrina ar rodyklė *null* ir jei ne atlieka tam tikrus paskaičiavimus. Paimkime paprastą pavyzdį (29 pav.):

```
SomeStruct * ptr;
if(ptr==NULL){
  ptr=alloc_mem_and_init();
}
int SomeRes=ptr->val*a+b;
```

29 pav.

Matome, kad neįmanoma kreipimosi į atmintį iškelti prieš rodyklės tikrinimą, nes tuomet

programos rezultatai bus nekorektiški. Štai tokį kodą sugeneruoja kompiliatorius (30 pav.):

```
    mov r15=some_struct_addr
    ;;
    cmp.eq p(3),p(4)=r15,r0
    ;;
    (p3) br.cond .not_null
    br.call someFunc
    mov r15=r8
    ;;
.not_null :
    ld r16=[r15]
    ;;
    mul r17=r16,r20
    ;;
    adds r17=r17,r18
```

30 pav.

Iš kodo matome, kad problematiška programos vieta yra būtent komanda *ld* – skaitymas iš atminties. Ją norėtų paslėpti aukščiau tarp kitų instrukcijų, ir palikti pakankamai taktų tarp atminties skaitymo ir skaičiavimų. IA-64 architektūra turi specialią žymę komandai *ld*, kuri nurodo, jog bandant skaityt atmintį tuo adresu, bet kokias kilusias išimtis (jeigu rodyklė *null*!) reikia išsaugoti specialiame bite, kurį turi kiekvienas registras [Int06d]. Tokiu atveju jei rodyklė *null* registro reikšmė pažymima, kaip NaT (Not A Thing – bereikšmė), ir bet kokie skaičiavimai, įtraukiant to registro reikšmę, gražina tą patį NaT rezultatą. Pateikiame modifikuotą programos kodą (31 pav.):

```
  mov r15=addr_of_struct    //išankstiniai paskaičiavimai,
  ;;                        //darant prielaidą, kad rodyklė ne null
  ld.s r16=[r15]            // ir kad yra pakankamai instrukcijų
  ;;                        // šiuos skaičiavimus paslėpti naudojantis EPIC
  mul r17=r16,r20
  ;;
  adds r17=r17,r18
  ;;
  cmp.eq (p3),(p4)=r15,r0
  ;;
  (p3) br.cond .not_null
  br.call alloc_and_init
  mov r15=r8
  ;;
.not null:
  chk.s r17,.recovery       //speciali tikrinimo instrukcija, kuri tikrina ar rezultatas
.recovered:                 //NaN - jei taip, pereiti į duomenų atstatymo bloką
....

.recovery:
  ld r16=[r15]
  ;;
  mul r17=r16,r20
  ;;
  adds r17=r17,r18
  br  .recovered
```

31 pav.

Šiuo atveju „spėjame“, kad rodyklė yra pasiekiama, atliekame skaičiavimus, jei vis dėl to ji buvo *null* – tenka skaičiavimus pakartoti su specialiai sugeneruotu atstatymo kodu. Tačiau bandymai parodė, jog taip optimizuoto kodo vykdymas nė kiek nepagreitėjo:

Iteracijos	Pn	Po	Paspartėjimas
250000000	0 min 40,501 sek.	0 min 40,510 sek.	-0,02%

Be abejo buvo pražiopsotas svarbiausias dalykas – tarp atminties skaitymo operacijos ir pirmojo skaičiavimo yra tik vienas taktas. To jokiu būdu neužtenka, kad būtų pakrauti duomenys iš atminties į registrą, todėl susidaro prastova, kuri siekia apie 300 taktų Itanium 2 procesoriui. Žinoma testinėje programoje iškelti atminties skaitymą dar trim šimtais instrukcijų grupių aukštyn nebuvo įmanoma, nes paprasčiausiai neužteko kodo. Tačiau tiek į originalų, tiek į optimizuotą kodą,

pridėjus pakankamai *nop* instrukcijų (emuliuojant didelį vykdomo kodo gabalą), rezultatai pagerėjo išsyk. Matome, jog tinkamai paslėpus atminties skaitymą, galima gerai paspartinti programos veikimą:

Iteracijos	Pn	Po	Paspartėjimas
250000000	1 min 17,576 sek.	0 min 42,534 sek.	45,17%

Iš esmės galima teigti, kad ši optimizacija programose yra svarbi tik tam tikrais specifiniais atvejais, kadangi speciali žymė NaT yra uždedama, tik tuomet kai atmintis yra nepasiekiama (puslapiavimo klaida), arba *null* atveju. Žinoma pateikti išnagrinėto atvejo laikai yra „geriausias variantas“, kuomet rodyklė niekuomet nebūna *null*, tačiau kaip parodė detalesni tyrimai, šią optimizaciją verta naudoti tol kol, tikimybė, jog rodyklė bus *null*, nesiekia 25%.

4.4.2 Išankstinis duomenų užkrovimas

Kita anksčiau paminėta problema yra rašymas ir skaitymas iš persidengiančių atminties adresų, dar kitaip žinoma kaip dviprasmiškų rodyklių (ambiguous pointer) problema. Dažnai būna taip, kad duomenys atmintyje po skaičiavimų būna atnaujinami tam tikru adresu, o vėliau skaitoma kitu adresu. Rašymo į atmintį instrukcijos yra vykdomos be jokių vėlavimų, kadangi procesoriui nesvarbu, kada tie duomenys atsidurs atmintyje ir ar jie liks vienoje iš spartinančiųjų atminčių. O skaitymas iš atminties yra „brangus“ veiksmas, kuri norėtųsi iškelt kuo anksčiau programos vykdymo kelyje. Šis atvejis yra labai dažnas programos kode, todėl net nereikia ieškoti kažkokių specialių C kodo pavyzdžių. Tarkim turime tokį asemblerio kodo fragmentą:

```
....                    //prieš tai esančios instrukcijos
st [r16]=r15            //rašymas į atmintį
;;
ld r4=[r12]             //skaitymas iš atminties
;;
adds r4=10,r4           // veiksmai su nuskaityta reikšme
;;
shladd r4=2,r4,r0
```

32 pav.

Tarkime prieš rašymą į atmintį, turime pakankamai instrukcijų, kur galėtume paslėpti skaitymą iš atminties ir su ta reikšme atliekamus veiksmus. Tačiau tam trukdo neaiškumas ar skaitymo ir rašymo adresai sutampa (r12 ir r16 registrų reikšmės). Tai labai dažna situacija kompiliuojant programos fragmentus, naudojančius keletą rodyklių. IA-64 architektūra suteikia

galimybę išvengti tokio instrukcijų perkėlimo aukštyn apribojimo. Ji leidžia „spėti“, jog adresai nesutampa, atminties skaitymą ir veiksmus su reikšme perkelti aukštyn, o tam kad nebūtų pažeista kodo semantika, į buvusių skaičiavimų vietą įdėti paprastą tikrinimą, kuris reikšmę nuskaitys ir perskaičiuos tik tuomet, jei adresai sutapo. Taip atrodo modifikuotas kodas (33 pav.):

```
  ld.a r4=[r12]          //speciali užkrovimo žymė
  ;;
  adds r4=10,r4
  ;;                     //atliekam skaičavimus
  shladd r4=2,r4,r0
  ...
  st [r16]=r15           //rašymas į atmintį
  chk.a r4, .recover     //speciali instrukcija, kuri patikrina
.back:                   //ar adresai nepersidengė
  ....


.recover:                //"blogiausias" atvejis
  ld r4=[r12]
  ;;
  adds r4=10,r4
  ;;
  shladd r4=2,r4,r0
  br.sptk.many .back
```

33 pav.

Ši Itanium savybė kaip ir anksčiau nagrinėtas valdymo spėjimas leidžia panaikinti tam tikrus nuo sąlygų priklausomus apribojimus ir geriau išdėstyti atminties skaitymo instrukcijas. Tiriant šią savybę gauti tokie rezultatai:

Sąlyga	Pn	Po	Paspartėjimas
Skaitymo rašymo adresai nesutapo	0 min 11,380 sek.	0 min 6,005 sek.	47,23%
Sutapo kas dešimtas	,,	0 min 7,316 sek.	35,71 %
Sutapo visi	,,	0 min 14,078 sek.	-23,71 %

Kaip matome iš lentelės paskutinė eilutė parodo, kiek papildomų ciklų reikia apdoroti atstatymo blokui.

Apibendrinant galima teigti, kad išankstinio duomenų užkrovimo galimybė yra žymiai plačiau

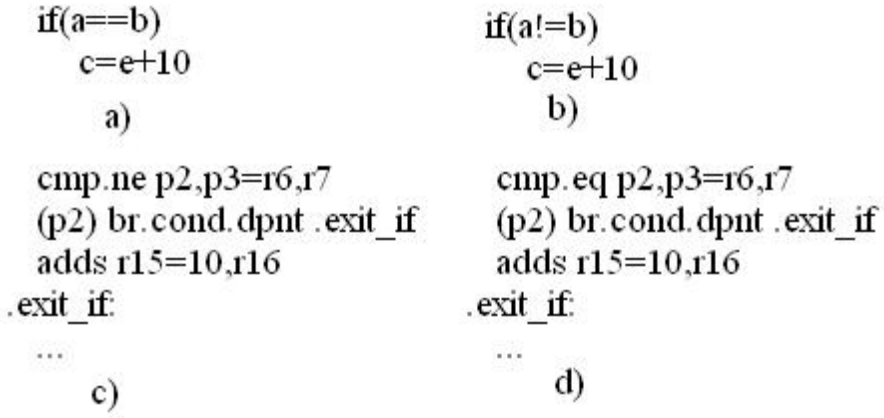
panaudojama ir reikšmingesnė už valdymo spėjimą, kuris yra skirtas tik tam tikriems atvejams. Tačiau šios abi savybės leidžia stipriai sumažinti instrukcijų tarpusavio priklausomybes ir padidina instrukcijų lygiagretinimo galimybes.

4.5 Sąlyginių perėjimų valdymas

IA-64 architektūra suteikia kompiliatoriui galimybę, išanalizavus kompiliuojamą kodą, prie sąlyginių perėjimų sudėti tam tikras žymes, kurios procesoriaus išankstinio instrukcijų pakrovimo planuotojui nurodo kokia tikimybė, jog sąlyginis perėjimas bus vykdomas. Kuo tiksliau nuspėjamas programos vykdymo kelias, tuo greičiau bus vykdoma programa. Kiekvienam sąlyginiam perėjimui galima nurodyti šias žymes:

sptk	Sąlyginis perėjimas bus tikrai vykdomas. Neskirti jokių resursų jo tolimesniam spėjimui
dptk	Sąlyginis perėjimas greičiausiai bus vykdomas, tačiau dinamiškai analizuoti jo galimybes.
dpnt	Sąlyginis perėjimas greičiausiai nebus vykdomas, tačiau skirti resursų jo spėjimui
spnt	Sąlyginis perėjimas nebus vykdomas. Neskirti jokių resursų jo tolimesniam spėjimui

Prie šių žymių dar galima nurodyti, kiek instrukcijų nuskaityti į buferį (*many* – daug arba *few* – mažai), tačiau tai labai priklauso nuo konkrečios procesoriaus realizacijos. Tam tikrais atvejais kompiliatorius naudoja konkrečias žymes, be jokių tolimesnių kodo analizių. Pavyzdžiui tiek procedūros kvietimui, tiek grįžimui iš jos visada naudojama žymė *sptk* [Int00]. Antros kartos Itanium procesoriai netgi ignoruoja žymes prie šių instrukcijų. Kompiliatoriaus generuojamo kodo analizė parodė, jog kompiliatorius visada generuodamas sąlyginį sakinį, perėjimui priskiria *dpnt* žymę (34 pav.):



34 pav.

Apskritai asemblerio kode sąlyginiam perėjimui nepriskyrus jokios žymės, automatiškai yra generuojama dpnt žymė. Šios savybės tyrimas taipogi nebuvo sudėtingas. Tiesiog buvo išmatuota kiek ciklų prarandama, neteisingai sugeneravus sąlyginio perėjimo žymę:

Iteracijos	Neteisingas spėjimas	Teisingas spėjimas	Paspartėjimas
2500000	0 min 17,760 sek.	14,77	16,83%

Kaip matome teisingas atšakų spėjimas neblogai paspartina programos darbą, tačiau teisingas žymių generavimas labai priklauso nuo pačio kompiliatoriaus euristinių kodo analizės algoritmų, ir norint pasinaudoti šia IA-64 savybe reikia tobulinti pačius algoritmus.

4.6 Laikinosios atmintinės valdymas

Kaip jau buvo minėta anksčiau, vienas iš būdų paspartinti programą tai perkelti visas atminties skaitymo operacijas kuo anksčiau ir kuo toliau nuo pačių skaičiavimų. Žinoma tai pavyksta tik tuomet, kai yra pakankamai kitų instrukcijų tarp atminties skaitymo ir reikšmių panaudojimo. Tačiau būna tokių atvejų kai atminties skaitymo instrukcijų iškelt nepavyksta ypač mažuose cikluose susidedančiuose iš keleto ar keliolikos instrukcijų grupių. Pateiksime paprasčiausią C kodo pavyzdį su ciklu, kuris skaičiuoja masyvo elementų sumą:

```
char * buff;
int sum;
for(int i=0;i<size;i++){
   sum+=buff[i];
}
```

35 pav.

Iš sugeneruoto asemblerio kodo (36 pav.) matome, kad didžiausias prastovas sudaro laukimas, kada duomenys iš atminties atsiduria reikalingame registre. Turint omeny, kad atminties skaitymo komandai prireikia apie 300 ciklų, tai tikrai nemažai, lyginant su pačio ciklo dydžiu.

```
.continue :
  cmp p3,p4=r5,r0
  ;;
  (p3) br.cond.dpnt .exit_loop
  ld1 r7=[r6]
  ;;
  add r8=r7,r8
  ;;
  adds r6=1,r6
  adds r5=-1,r5
  br.sptk.many .continue
.exit_loop:
```

36 pav.

Šiame pavyzdyje registre r7 yra kaupiama masyvo elementų suma, r5 registre saugomas ciklo iteracijų skaičius, registre r6 – einamojo elemento adresas. Kaip matome, ciklo kūnas susideda iš vos tris taktus užimančių instrukcijų grupių, todėl papildomi 300 taktų kiekvienos iteracijos metu labai sumažina programos efektyvumą. Jeigu masyvo dalis duomenų jau būtų spartinančiojoje atmintinėje, tai darbas vyktų žymiai sparčiau. IA-64 architektūra turi specialią instrukciją *lfetch* [ISN08b], kuri nurodyto atminties adreso duomenis gali patalpinti vienoje iš trijų spartinančiųjų atmintinių lygių. Mus šiuo atveju domina pirmas lygis. Čia iškart pateiksime optimizuotą ankstesnio kodo pavyzdį (37 pav.):

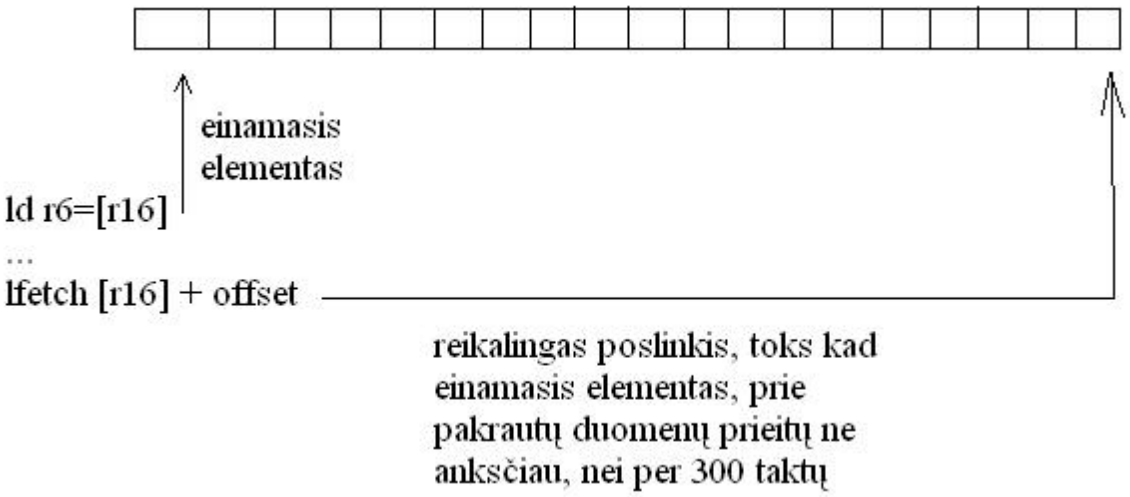
```
.continue :
  cmp p3,p4=r5,r0
  and r9=31,r5
  ;;
  (p3) br.cond.dpnt .exit_loop
  cmp p5,p6=r9,r0
  ld1 r7=[r6]
  ;;
  lfetch [r6]
  add r8=r7,r8
  ;;
  adds r6=1,r6
  adds r5=-1,r5
  br.sptk.many .continue
.exit_loop:
```

37 pav.

Kadangi *lfetch* komanda nuskaito į L1 atmintinę 32 baitus, tai ją reiktų kviesti kas 32 ciklo iteraciją. Bandymai buvo daromi su 5000 baitų dydžio masyvu. Tačiau kaip matyti iš lentelės programų testavimo rezultatai anaiptol ne tokie, kokių tikėjomės:

Iteracijos	Pn	Po	Paspartėjimas
250000	0 min 15,96 sek.	0 min 15,9584 sek.	0,01 %

Paspartėjimas aišku tikrai ne toks kokio tikėjomės, abi programos vykdomos visiškai vienodai (laikų skirtumas atsirado dėl ne idealiai tikslaus skaičiavimo). Detalesnis nagrinėjimas parodė, jog komanda *ld*, skaitydama iš atminties, automatiškai patalpina 32 baitus į L1 atmintinę, todėl *lfetch* komanda jokio papildomo skaitymo tiesiog neatliko. Taip pat reikia nepamiršti, jog duomenų nuskaitymui reikalingi tie patys 300 taktų, tiktai procesorius nelaukia, kol duomenys atsiras L1 atmintinėje. Žinoma pasinaudojus EPIC savybe visus *lfetch* reikalingus atminties adreso skaičiavimus sugebėjome įterpti tarp esamų ciklo komandų, todėl papildomi taktai nebuvo sunaudoti. Vadinasi *lfetch* komanda turėtų krauti ne esamo elemento adresu esančius duomenis, o „užbėgti“ šiek tiek į priekį:



38 pav.

Pats poslinkis atmintyje skaičiuojamas pagal tokią formulę (39 pav.):

Offset = MemLatency * ByteCount / CyclesPerIteration

Offset - poslinkis nuo esamo elemento iki elemento, kurį reikia užkrauti į spartinančiąją atmintį

MemLatency - taktų skaičius reikalingas duomenims iš atminties į registrą pakrauti. Itanium 2 procesoriui 300 taktų

ByteCount - baitų kiekis, apdorojamas per vieną iteraciją.

CyclesPerIteration - kiek taktų sunaudojama vienai ciklo iteracijai vykdyti

39 pav.

Pagal ją gauname, kad mums reikalingas poslinkis yra apie 300 baitų [Lev08]. Šiek tiek patobulinus optimizuojamą programą, gaunami jau labiau tikėtini rezultatai:

Iteracijos	Pn	Po	Paspartėjimas
250000	0 min 15,96 sek.	0 min 8,647 sek.	45,82 %

Apskritai skaitymo iš atminties optimizavimas yra vienas iš kertinių, norint pasiekti maksimalų našumą. Instrukcija *lfetch* puikiai pritaikoma cikluose, ypač dirbant su masyvais. Taipogi gali pasitarnauti tose situacijose, kai neįmanomas skaitymo iš atminties instrukcijų perkėlimas. Tiktai reikia atsižvelgti į vieną dalyką, kad duomenis patalpinti į L1 atmintinę reikia tiek pat laiko kaip ir paprasto skaitymo metu.

4.7 Ciklų optimizacija konvejerio principu

Viena iš plačiai taikomų tradicinių ciklų optimizacijų IA-64 architektūroje yra pritaikoma ypač efektyviai [Int04]. Ši architektūra išsprendžia tokias problemas, kaip nepakankamas registrų kiekis, papildomas kodas, generuojant tiek epilogą, tiek prologą. Paimkime C kodo pavyzdį, kuris skaičiuoja labai paprastą CRC (Cyclic Redudancy Check) funkciją (40 pav.):

```
for(int i=0;i<size;i++){
  sum= sum ^ buff[i];
}
```

40 pav.

Labai neišsiplečiant, galima pasakyti, kad IA-64 architektūra suteikia aparatūrines galimybes vykdyti tiek prologo, tiek epilogo fazes. Tam reikalingi ir besikeičiantys (rotating) registrai, bei predikacija. Optimizacijos esmė – skirtingas kelių iteracijų funkcijas vykdyti „lygiagrečiai“.

Pateiksime originalų, modifikuotą asemblerio kodą (41 pav.):

```
mov ar.lc=buff_size
.loop:
ld1 r5=[r16],1
;;
xor r6=r5, r6
br.sptk .loop
```

a)

```
mov ar.lc=buff_size
mov ar.ec=3
mov pr.rot=1<<16
;;
.loop:
(p16) ld1 r32=[r16],1
(p17) xor r6=r33,r6
br.ctop.sptk .loop:
```

b)

41 pav.

Lygindami originalų a) ir optimizuotą b) kodus, pastebime, jog optimizuoto ciklo kūnas dabar yra įvykdomas per vieną taktą, kadangi išnyko priklausomybės tarp instrukcijų. Vadinasi tokia ciklo optimizacija labai pagerina kodo išlygiagretinimą. Pateiksime programų darbo rezultatus:

Iteracijos	Pn	Po	Paspartėjimas
50000	0 min 11,698 sek.	0 min 9,396 sek.	19,69 %

Aišku programos našumo padidėjimas yra palyginus mažas, tačiau nereikia pamiršti, jog skaitymas iš atminties taip ir liko neoptimizuotas. Tinkamai pritaikius *lfetch* instrukciją gauname:

Iteracijos	Pn	Po	Paspartėjimas
50000	0 min 11,698 sek.	0 min 3,06 sek.	73,84 %

Rezultatai išties neblogi, net ir su sąlyginai mažais duomenų kiekiais. Galime daryti išvada, kad ciklų optimizavimas konvejerio principu gali labai stipriai pagerinti programos darbo našumą.

Eksperimentiškai tiriant IA-64 architektūros savybes ir palyginus programų darbo efektyvumus, pastebime, jog ne visos savybės teikia vienodą našumą ir vienodai lengvai yra pritaikomos. Pavyzdžiui sąlyginių atšakų prognozavimas yra lengvai realizuojamas, tačiau pačių tikimybių apskaičiavimas yra ganėtinai sudėtinga procedūra ir reikalauja pakankamai gerų euristinių kodo analizės algoritmų. Taipogi, pačias savybes galima suskirstyt į dvi kategorijas.

Pagrindinės savybės - tai išreikštinai lygiagretus instrukcijų vykdymas, skaitymo iš atminties optimizavimas, sąlyginių perėjimų valdymas.

Pagalbinės savybės – šių savybių dėka yra maksimaliai panaudojamos pagrindinės savybės.

Predikacija bei valdymo ir duomenų spėjimai leidžia žymiai geriau išlygiagretinti instrukcijas. Ciklų vykdymas konvejerio principu leidžia paslėpti operatyvios atminties skaitymo vėlavimus.

Kaip parodė tyrimai, svarbiausias dalykas optimizuojant kodą - stengtis kuo labiau išlygiagretinti instrukcijas, maksimaliai išnaudojant procesoriaus vykdymo konvejerius. Antras dalykas - skaitymo iš atminties optimizavimas, todėl tolimesnė darbo eiga - kompiliatoriaus parametrų nagrinėjimas, kurie susiję su šias optimizacijas taikančiais kodo pertvarkymais.

5.Vidinių kompiliatoriaus parametrų tyrimas

5.1 Gcc kompiliatorius

Šiuolaikiniai kompiliatoriai be kodo generavimo iš aukšto lygio kalbos į žemo lygio asemblerį sugeba taikyti įvairias kodo optimizacijas. Žinoma, šias optimizacijas galima padalinti į dvi grupes. Pirmajai daliai priklauso tokios optimizacijos, kurios yra taikomos nepriklausomai nuo architektūros, pavyzdžiui CSE (Common Subexpression Elimination) ir panašiai. Antrajai grupei priklauso nuo architektūros priklausomos optimizacijos, kurios naudoja specifines architektūros savybes. Dažniausiai iškyla tokia problema, jog kompiliatoriai, kurie gali generuoti kodą daugeliui platformų, dažnai paprastai slepia nuo architektūros priklausomų optimizacijų valdymą, palikdami tik optimizacijų valdymo bendrus parametrus. Vidiniai parametrai, kurie valdo pačius optimizacijos algoritmus, dažnai yra bendri visoms architektūroms ir kartais nėra patys geriausi konkrečiai architektūrai. Tokia situacija yra su gcc kompiliatoriumi, kurio optimizacijas valdančius parametrus galima išskirti į tris grupes. Pirmoji grupė, tai parametrai įjungiantys tam tikras optimizacijų grupes – O1, O2 ir O3. Šie parametrai yra dažniausiai naudojami, kompiliuojant įvairias programas ir nesigilinant, kokios būtent optimizacijos yra taikomos, ir dažniausiai kasdieniniam darbui to pakanka. Antrai grupė parametrų leidžia įjungti arba išjungti kiekvieną optimizaciją atskirai ir dažniausiai naudojama, kompiliuojant tam tikrą specialią programinę įrangą, pavyzdžiui operacinės sistemos branduolius, bei bibliotekas. Trečioji parametrų grupė yra vidiniai kompiliatoriaus parametrai, kurie yra skirti pačių optimizacijų valdymui ir dažniausiai naudojami norint optimizuoti patį kompiliatorių konkrečiai architektūrai. Būtent gcc kompiliatoriaus parametrų grupė mus ir domina.

Prieš tai buvusioje darbo dalyje padarėme išvadą, jog, optimizuojant kodą Itanium architektūrai, svarbiausi dalykai yra geras instrukcijų išlygiagretinimas, bei operatyviosios atminties skaitymo operacijų optimizacija. Instrukcijų lygiagretinimą tvarko instrukcijų planuotojas (instruction scheduler), todėl nagrinėsime tuos vidinius parametrus, kurie valdo jo darbą. Taip pat tirsime parametrus susijusius su atminties operacijų valdymu.

5.2 Testavimo programa oopack_v1p8.cc

Tiriant kompiliatoriaus generuojamo kodo efektyvumą, reikalingos testinės programos, kurias sukompiliavus ir įvykdžius, galima daryti tam tikras prielaidas apie generuojamą kodą. Gcc kompiliatoriaus testavimui dabar dažniausiai naudojamas didelis testavimo programų rinkinys SPEC2006, turintis platų testinių programų spektrą. Deja šis rinkinys yra mokamas ir nėra laisvai

prieinamas, todėl teko rinktis paprastesnę testinę programą oopack_v1p8.cc, kurią sudaro keli testai, skirti kompiliatoriaus įvairioms optimizacijoms tirti:

- „Iterator“ testas. Skirtas analizuoti tas optimizacijas, kurios susiję su rodyklėmis, bei dinaminių sąrašų panaudojimu.
- „Matrix“ testas. Tai įvairios operacijos su matricomis – sudėtis, atimtis, daugyba ir panašiai. Šio testo kodui taikomos optimizacijos susiję su ciklų išvyniojimu, padalinimu, suliejimu ir taip toliau.
- „Complex“ testas, kurį sudaro įvairūs veiksmai su kompleksiniais skaičiais. Vėlgi tiriamos kompiliatoriaus optimizacijos darbui su įvairiomis duomenų struktūromis atmintyje.
- „Min/Max“ testas skirtas tikrinti, kaip gerai kompiliatoriaus optimizacijos tvarkosi su duomenų srautais (data streams), kaip panaudojama laikinoji atmintinė ir taip toliau.

Gcc vidiniai parametrai tiriami, kompiliuojant šią programą su įvairiomis parametrų reikšmėmis ir tiriant kaip reikšmės įtakoja programos vykdymo laiką.

5.3 Vykdymo statistikos rinkimas

Tiriant programos darbo našumą, pagrindinis ir pats svarbiausias kriterijus yra laikas per kurį ji įvykdo savo užduotis. Tačiau tiriant kompiliatoriaus kodo pertvarkymus vien tuo remtis nepakanka. Dažnai reikia išsiaiškinti, kokie niuansai lemia būtent tokį programos vykdymo paspartėjimą ar sulėtėjimą, pavyzdžiui kaip kito laikinosios atminties panaudojimo statistika, keičiant vieną ar kitą kompiliatoriaus parametrą. Šiuos dalykus teoriškai įmanoma išsiaiškinti, nagrinėjant sugeneruoto asemblerio kodą, tačiau tai yra ne tik labai neefektyvu, bet dažnai ir labai sunku. Tačiau Itanium architektūros procesoriai moka aparatūriškai kaupti įvairią programos vykdymo statistiką, tam pasitelkdami specialius valdymo bei saugojimo registrus, prieinamus programuotojui. Žinoma į patį programos kodą dėti specialias statistikos nuskaitymo bei inicializavimo komandas nėra labai patogu, todėl sukurti įrankiai, kurie suteikia galimybę paleisti bet kurią sukompiliuotą programą ir specialiais parametrais nurodyti kokią vykdymo statistiką rinkti bei atvaizduoti. Vienas iš tokių įrankių yra *pfmon*, sukurtas HP korporacijos. Tirdami gcc kompiliatoriaus vidinius parametrus remsimės jo pateiktais rezultatais [JAR02].

Analizuojant kodo lygiagretinimo optimizacijos našumą, reikia atsižvelgti ne tik į programos vykdymo paspartėjimą, bet ir įvertinti, kiek instrukcijų per taktą procesorius įvykdė. Todėl tirdami parametrus naudosimės šiais skaitliukais:

- CPU_OP_CYCLES_ALL. Labai svarbi charakteristika parodanti, kiek taktų sunaudojo procesorius vykdydamas šią programą.
- CPU_OP_CYCLES_HALTED. Taktų skaičius, kuriuos esama gija išnaudojo laukimo režime (sleep). Dažniausiai šitą skaičių reikia atmesti iš pirmosios charakteristikos.
- IA64_INST_RETIRED_THIS. Įvykdytas programos IA-64 instrukcijų skaičius. Be šio tipo instrukcijų dar gali būti ir emuliuojamos IA-32 instrukcijos, tačiau kadangi mūsų tiriamos programos visos yra IA-64 tipo, tai IA-32 instrukcijų skaičių galime ignoruoti.
- NOPS_RETIRED. Labai svarbus parametras, kuris nusako kiek „tuščių“ instrukcijų (kurios tik užėmė vykdymo konvejerius) buvo įvykdyta. Dažniausiai šios instrukcijos generuojamos laisvose instrukcijų paketų vietose.
- BACK_END_BUBBLE_ALL. Parodo kiek „tuščių“ taktų praleido procesoriaus vykdančioji dalis (back-end), dėl duomenų nebuvimo, vykdymo išimčių ir taip toliau.

Kaip jau minėta anksčiau programos pagrindinis vertinimo kriterijus yra skirtumas tarp CPU_OP_CYCLES_ALL ir CPU_OP_CYCLES_HALTED, kuris atitinka laiką, kuris sunaudotas vykdant realų programos kodą atmetant laiką sugaištą dėl pertraukimų, procesų planuotojo darbo ir t. t. Tolesniuose skyriuose šį skirtumą vadinsime tiesiog *taktų skaičiumi*. Pagrindinis tikslas, keičiant parametrų reikšmes – kuo mažesnis šis skaičius, o kartu ir programos vykdymo laikas.

Antras labai svarbus kriterijus vertinant instrukcijų planuotojo darbą, tai instrukcijų skaičius įvykdytas per vieną taktą (UIPC – Useful Instruction Per Cycle) [JAR02]. Jis apskaičiuojamas iš visų vykdytų instrukcijų atėmus tuščias instrukcijas bei padalinus iš taktų skaičiaus (42 pav.):

$$UIPC = \frac{IA64\_INST\_RETIRED\_THIS - NOPS\_RETIRED}{CPU\_OP\_CYCLES\_ALL - CPU\_OP\_CYCLES\_HALTED - BACK\_END\_BUBBLE\_ALL}$$

42 pav. UIPC skaičiavimas

Šis parametras gali kisti nuo vienos instrukcijos per taktą iki teorinių šešių instrukcijų per taktą.

Vertinant laikinosios atminties panaudojimą, reikės remtis šiais vykdymo statistikos parametrais:

- BE_EXE_BUBBLE_GRALL – prastova procesoriaus vykdymo kanale (Execution pipeline) dėl registro-registro arba registro-atminties neleistinų priklausomybių.

- BE_EXE_BUBBLE_GRGR – prastova procesoriaus vykdymo kanale dėl registro-registro neleistinų priklausomybių.
- BE_L1D_FPU_BUBLE_L1D – prastova procesoriaus vykdymo kanale dėl neparuoštų duomenų pirmo lygio laikinojoje atmintinėje.

Vertinant prastovas dėl duomenų nebuvimo laikinosiose atmintinėse, gauname tokį išvestinį kriterijų (43 pav.):

Cache stalls = BE_EXE_BUBBLE_GRALL - BE_EXE_BUBBLE_GRGR + BE_L1D_FPU_BUBBLE_L1D

43 pav. Pirmo lygio laikinosios atmintinės prastovos

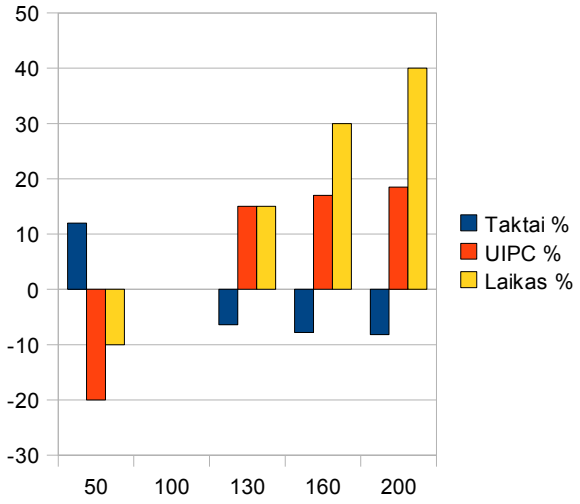
Toliau šį kriterijų vadinsime tiesiog *laikinosios atmintinės prastovos* [JAR02]. Verta pažymėti, jog čia ir visur minimos prastovos yra išreiškiamos taktų skaičiumi, kurį sunaudojo procesorius dirbdamas „tuščiai“. Tiriant parametro reikšmių gerumą, šis kriterijus turi būti kuo mažesnis.

5.4 Vidiniai kompiliatoriaus parametrai

Šiame skyriuje pristatysime vidinius tiek planuotoją, tik atminties optimizaciją valdančius parametrus, pateiksime kodo vykdymo vertinimo kriterijų priklausomybes nuo parametrų reikšmių, bei įvertinsime jų įtaką programos vykdymo laikui. Verta paminėti, kad visi šie vidiniai parametrai tinka tik šiai konkrečiai gcc 4.4.0 versijai, bei vėlesnėse versijose gali keistis tiek jų reikšmės tiek patys parametrų pavaidinimai [GSG08]. Patys parametrai nėra gerai dokumentuoti, o tai apsunkina jų įtakos nagrinėjimą. Visi testai buvo atliekami kaip atskaitos tašką imant testinę programą, sukompiliuotą su maksimaliu optimizacijos lygiu -O3, o grafikuose pateikti procentiniai skirtumai nuo atskaitos taško su tam tikra parametro reikšme.

5.4.1 Instrukcijų planuotojo valdymo parametrai

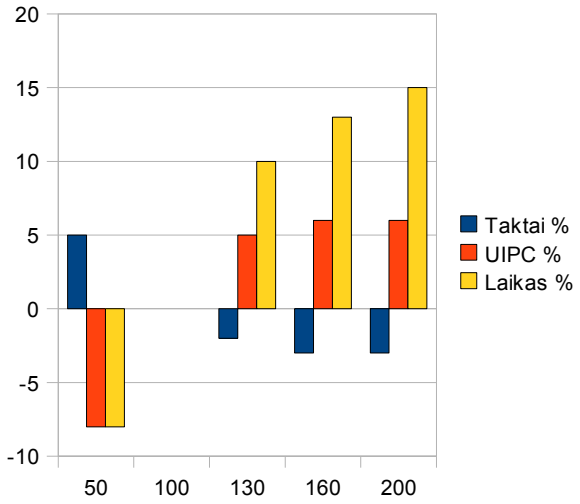
Max-reload-search-insns – parametras, kontroliuojantis analizuojamo kodo srities dydį, kuriame bus ieškoma registro su anksčiau paskaičiuotais reikiamais rezultatais. IA-64 architektūroje tai susiję su išankstinio duomenų užkrovimo spėjimu. Reikšmė pagal nutylėjimą yra 100 instrukcijų kodo srities dydis. Didinant reikšmę, o kartu ir šios srities dydį yra geriau išnaudojamos IA-64 architektūros valdymo bei duomenų spėjimo savybės, tačiau tai kartu stipriai pailgina kompiliavimo laiką su santykinai mažu kodo paspartėjimu. 43 paveikslėlyje pateiktas santykinis taktų skaičiaus, bei UIPC pokytis priklausomai nuo parametro reikšmės:



43 pav. Max-reload-search-insns grafikas

Iš grafiko matome, kad optimaliausia reikšmė šiuo atveju yra kažkur 130 instrukcijų, kadangi smarkiai augant kompiliavimo laikui tiek taktų skaičius tiek UIPC kinta labai mažai.

Max-sched-ready-insns – kiek daugiausiai instrukcijų kompiliatorius turi būti pasiruošęs, per pirmąjį planavimo etapą. Iš esmės šis skaičius susijęs su tuo, jog kompiliatorius pirmojo planavimo metu turi prisirinkti pakankamai nepriklausomų instrukcijų, kurias galėtų panaudoti vėliau pavyzdžiui užpildydamas „tuščius“ taktus tarp atminties nuskaitymo ir vėlesnių reikšmių panaudojimo. Reikšmė pagal nutylėjimą – 100 instrukcijų, vėlgi didinant šią reikšmę didėja kompiliavimo laikas. 44 paveikslėlyje anksčiau minėtų kriterijų priklausomybė nuo šio parametro reikšmės:

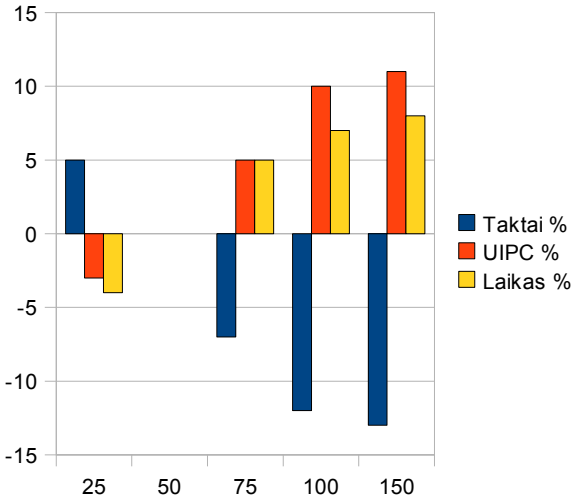


44 pav. Max-sched-ready-insns grafikas

Vėlgi iš grafiko pastebime, kad didinant reikšmę, labai nežymiai padidėja UIPC, tačiau

kompiliavimo laikas priklauso vos ne tiesiškai, taip pat labai nežymiai sumažėja taktų skaičius, todėl šią reikšmę geriausia būtų palikti tokią, kokia ji yra.

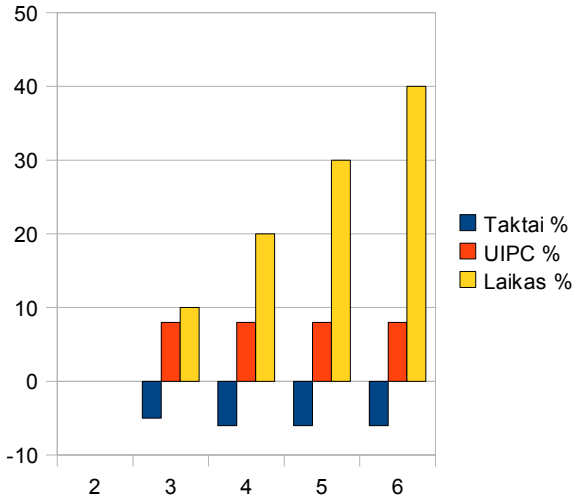
Selsched-max-lookahead parametras apibrėžia antrojo instrukcijų planavimo etapo „užbėgimą į priekį“ (look-ahead). Tai reiškia per kiek instrukcijų į priekį ieškoti reikalingų instrukcijų jas grupuojant. Reikšmė pagal nutylėjimą yra 50 instrukcijų. Iš esmės kuo toliau į priekį žvelgt tuo geriau (45 pav.):



45 pav. Selsched-max-lookahead grafikas

Pagal grafiką optimaliausia reikšmę reikėtų padidinti iki 100 instrukcijų, nes su gana mažai pakitusių kompiliavimo laiku gauname gana neblogą taktų sumažėjimą, bei koeficiento UIPC padidėjimą.

Selsched-max-sched-times parametras apibrėžia antrojo planavimo etapo iteracijų skaičių, per kurias stengiamasi kuo geriau sugrupuoti instrukcijas. Praktiškai tai reiškia kiek maksimaliai kartų instrukcija gali būti perkelta iš vienos vietos į kitą. Pradinė reikšmė yra dvi iteracijos. Eksperimento rezultatai 46 pav.:



46 pav. Selsched-max-sched-times grafikas

Iš grafiko matome, kad geriausi rezultatai yra su trim iteracijom, kadangi vėliau stipriai didėjant kompiliavimo laikui, nelabai kinta nei taktų skaičius nei UIPC.

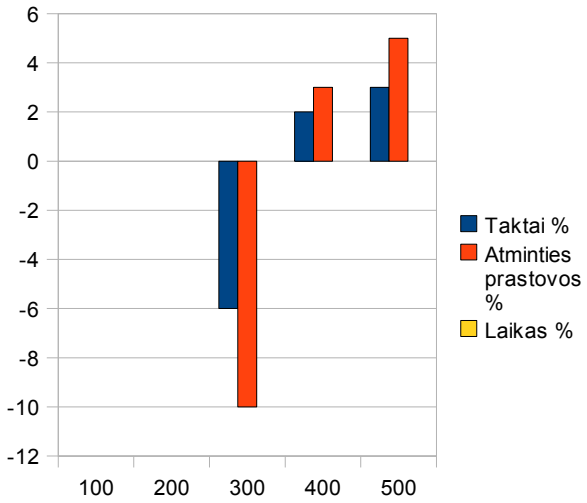
Iš pateiktų keturių parametrų analizės, keičiant pirmo, trečio bei ketvirto parametrų reikšmes gaunami teigiami kriterijų pokyčiai, o trečiojo parametro reikšmę geriausia yra palikti pradinę.

5.4.2 Atminties valdymo optimizacijos parametrai

Nagrinėjant vidinių parametrų įtaką kodo darbui su atmintimi kaip ir planuotojo optimizavimo atveju pirmiausia yra vertinamas taktų skaičius reikalingas programai įvykdyti. Norint tiksliau įvertinti operatyviąją atmintį naudojančio kodo gerumą, taip pat reikia stebėti prastovų, susijusių su duomenų nebuvimu viename iš laikinosios atmintinės lygių, kiekį. Kadangi prastovų dydis tarp laikinosios atmintinės lygių yra skaičiuojamas keletu ar keliolika taktų, o prastovų dydis, kuomet duomenų laikinojoj atmintinėj išvis nėra, skaičiuojamas šimtais taktų, tai tiriant vidinius parametrus, vertinsime tik prastovas, su tiesioginių duomenų užkrovimu iš operatyviosios atminties. Toliau pateiksime nagrinėtus vidinius parametrus, susijusius su atminties panaudojimo optimizavimu.

Prefetch-latency parametras nusakantis, kiek maždaug instrukcijų galima įvykdyti, nuo to momento kai duomenys pradedami skaityti iš atminties iki laiko kai jie tampa pasiekiami procesoriui. IA-64 architektūroje, kurioje vienai instrukcijai įvykdyti reikia vieno takto, šis parametras realiai nusako kiek taktų reikia, kad duomenys iš operatyviosios atminties atsidurtų pirmojo lygio spartinančiojoje atmintinėje (L1 – cache). Šis parametras labai svarbus, apdorojant didelius masyvus cikluose, kadangi tiesiogiai susijęs su poslinkio skaičiavimu, duomenis iš anksto užkraunant į laikinąją atmintinę (plačiau apie tai 4.6 skyriuje). Pagal techninę dokumentaciją tai

turėtų būti apie 300 taktų. Šio parametro pradinė reikšmė yra 200 instrukcijų (arba šiuo atveju taktų). Pateiksime vertinimo kriterijų priklausomybę nuo šio parametro (47 pav.):

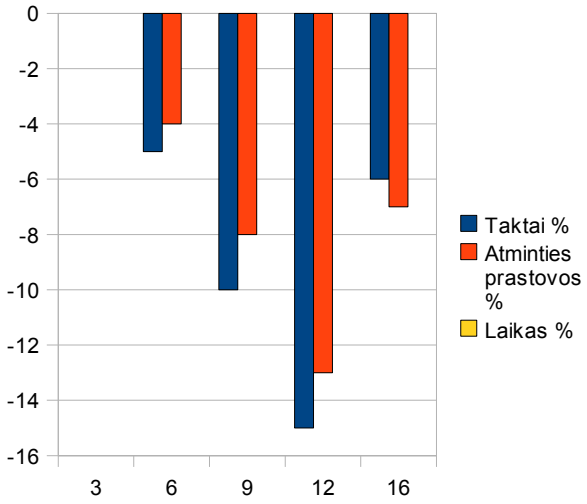


47 pav. Prefetch-latency parametras

Visų pirma iš pateikto grafiko matome, kad šis parametras niekaip neįtakoja kompiliavimo laiko, kadangi tai yra konstanta susijusi su poslinkio skaičiavimu, o ne konkrečios optimizacijos darbu. Antra tiek mažiausias taktų skaičius, tiek prastovų skaičius atitinka būtent techninėje dokumentacijoje numatytą parametro reikšmę. O su didesnėmis ir mažesnėmis reikšmėm šie skaičiai auga arba nesikeičia, lyginant su pradine reikšme. Taip yra todėl, kad mažesnės reikšmės atveju, reikalingi duomenys tiesiog nespėja atsidurti pirmame atmintinės lygyje, ir jokios spartos dėl išankstinio duomenų skaitymo nepavyksta pasiekti. Per daug didinant šią reikšmę išauga poslinkis, o kartu ir taktų skaičius, kuris yra naudojami duomenis skaitant be jokių optimizacijų, todėl yra pastebimas nors ir nežymus tiek prastovų, tiek taktų skaičiaus didėjimas. Taipogi šio parametro reikšmė yra priklausoma ne tik nuo architektūros, bet ir nuo konkrečios procesoriaus realizacijos, todėl šį parametrą, geriausia būtų iškelti į specialų failą, kuriame aprašomi specifiniai konkretaus procesoriaus parametrai, ir kurį vėliau gcc naudoja, kuomet nurodoma generuoti kodą optimizuotą konkrečiam procesoriui.

Simultaneous-prefetches parametras, apibūdina kiek maksimaliai skirtingų išankstinių duomenų užkrovimų galima atlikti vienu metu arba *lfetch* instrukcijų skaičių. Iš esmės tai reiškia, kiek maksimaliai skirtingų duomenų srautų galima apdoroti vienu metu. Šis parametras susijęs su tokiais techniniais dalykais, kaip laikinosios atmintinės dydis ir jau aptartas taktų skaičius, norint duomenis patalpinti iš operatyviosios atminties į pirmą laikinosios atmintinės lygį. Pradinė šio

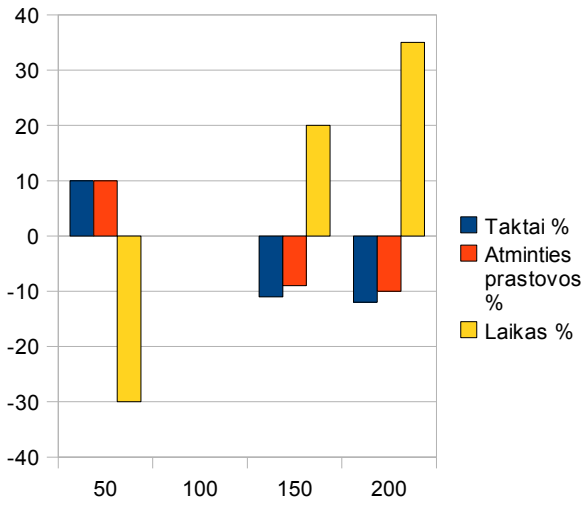
parametro reikšmė yra trys, tai reiškia, kad vienu metu apdoroti galima daugiausia tris skirtingus duomenų srautus. Tiriant kodo vykdymo statistikos priklausomybę nuo šio parametro, pastebėta, jog testinė programa realiai vienu metu apdoroja tik tris masyvus, todėl didinant parametro reikšmę, realiai nesikeičia nei atminties prastovos, nei taktų skaičius. Šiek tiek pamodifikavus testinę programą – į skaičiavimus įtraukus apie dvidešimt duomenų masyvų, gauti tokie rezultatai (48 pav.):



48 pav. simultaneous-prefetches grafikas

Iš grafiko matome, jog didžiausias efektyvumas pasiekiamas, kai parametras lygus 12. Tai reiškia, kad Itanium procesorius efektyviausiai dirba su dvylika skirtingų duomenų srautų [YLW05]. Toliau didinant šį parametrą, paprasčiausiai neužtenka laikinosios atmintinės, sutalpinti pakankamą kiekį iš anksto užkrautų duomenų didesniam masyvų skaičiui. Todėl šio parametro reikšmė turėtų būti tiesiogiai proporcinga skaičiui, kuris gautas, padalinus laikinosios atmintinės dydį iš paskaičiuoto išankstinio duomenų užkrovimo poslinkio. Gauta reikšmė apytiksliai atitiks kiek skirtingų duomenų srautų procesorius gali apdoroti vienu metu.

Max-delay-slot-insn-search šis paskutinis parametras apibrėžia didžiausią instrukcijų skaičių, kuris reikalingas užpildant tarpus tarp duomenų nuskaitymo ir jų panaudojimo (delay slots). Iš esmės šis parametras nusako, kokio dydžio kodo srityje ieškoti reikalingų instrukcijų [GSG08]. Pradinė reikšmė 100 instrukcijų, didinant šią reikšmę vėlgi didėja kompiliavimo laikas, nes kompiliatorius ieškodamas daugiau instrukcijų turi aprėpti didesnę analizuojamojo kodo sritį. 49 pav. pateikti tyrimo rezultatai:



49 pav. max-delay-slot-insn-search grafikas

Iš grafiko matome, kad optimaliausias skaičius yra apie 150 instrukcijų, tačiau šis parametras priklauso ir nuo kompiliuojamo kodo – kuo labiau tarpusavyje susiję skaičiavimai, tuo sunkiau rasti nepriklausomų instrukcijų, kurias galima perkelti. Žinoma reikia nepamiršti, kad šis parametras nurodo maksimalų skaičių, tai yra jį pasiekus kompiliatorius daugiau instrukcijų neieškos, tačiau kartais tiesiog neįmanoma surasti tiek instrukcijų vien dėl instrukcijų tarpusavio priklausomybių ir panašiai. Bet vistiek, bet koks tokių tarpų užpildymas, nors ir dalinis, atneša optimizuojamo kodo paspartėjimą.

Šiame skyriuje apžvelgėme nagrinėtus vidinius gcc kompiliatoriaus parametrus, kurie galimai būtų susiję su geresniu IA-64 architektūros savybių išnaudojimu. Lentelėje pateiksime trumpą parametrų santrauką:

Parametras	Pradinė reikšmė	Optimaliausia reikšmė	Paspartėjimas
max-reload-search-insns	100	130	7 %
max-sched-ready-insns	100	100	0 %
selsched-max-lookahead	50	100	12 %
selsched-max-sched-times	2	3	5 %
prefetch-latency	200	300	8 %
simultaneous-prefetches	3	12	14 %
max-delay-slot-insn-search	100	150	10 %

Iš esmės matome, kad geriausius rezultatus duoda analizuojamųjų kodo sričių išplėtimas, bei papildomų *prefetch* instrukcijų generavimas, tačiau reikia pabrėžti, kad gauti paspartėjimo rezultatai priklauso ne tik nuo parametrų reikšmių keitimo, bet ir nuo kompiliuojamojo kodo, todėl sekančiame skyriuje pateiksiu eksperimentų su realiomis programomis rezultatus.

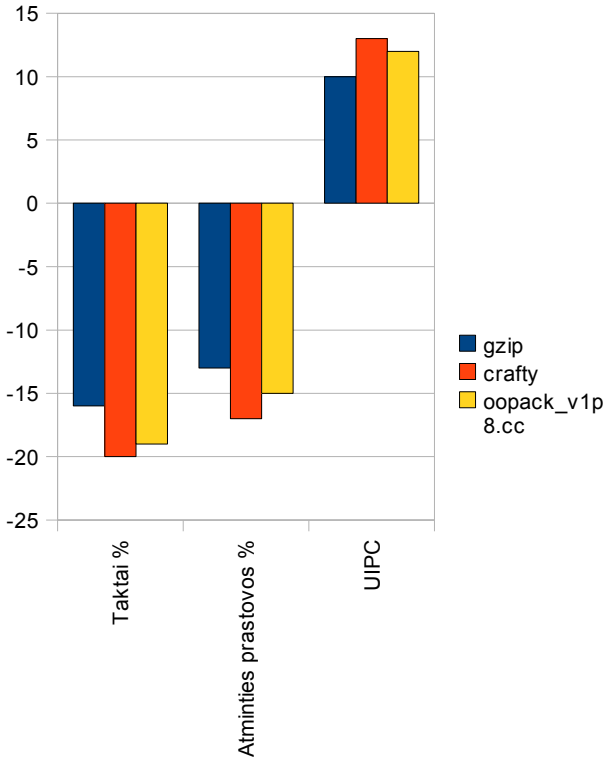
5.5 Taikomųjų programų testai

Ištyrus kompiliatoriaus vidinius parametrus, gautos naujos reikšmės leidžia kodą paoptimizuoti iki penkiolikos procentų, tačiau reikia nepamiršti, jog vidiniai parametrai buvo tiriami, naudojant testinę kompiliatorių programą. Norint susidaryti realų vidinių parametrų įtakos paspartėjimui vaizdą, gautas reikšmes reikėtų išbandyti kompiliuojant taikomąsias programas.

Viena iš tokių programų galėtų būti linux archyvavimo programa gzip. Joje be naudojamųjų suspaudimo algoritmų, taip pat yra darbas su dideliais duomenų blokais, įvairūs skaičiavimai, todėl tiek planuotojo, tiek atminties optimizavimo parametrai turėtų įtakoti sukompiliuoto kodo veikimo spartą. Testuojant pačią gzip programą, užtenka paimti vienos rūšies duomenis – šiuo atvejų bandymai buvo atliekami, archyvuojant pačio gcc kompiliatoriaus išeities tekstus, kadangi lyginame ne gzip algoritmų efektyvumą, o pačio gzip kodo paspartėjimą.

Antroji bandoma programa – konsolinis šachmatų žaidimas „crafty“, taipogi skirtas linux. Ši programa anksčiau buvo į kompiliatorių testavimo aplinką SPEC2000, ir dažniausiai naudojama tiriant būtent kompiliatoriaus darbą su rodyklėmis, įvairiomis duomenų struktūromis, taip pat raktas-reikšmė masyvais ir taip toliau. Programos įvedimo duomenys tai failas, kuriame aprašyta apie penkiasdešimt nuoseklių ėjimų šachmatais. Vėlgi įvedimo duomenys nėra labai svarbus, tiriant optimizuoto kodo paspartėjimą.

Abi šios programos buvo kompiliuojamos kaip ir oopack_v1c8.cc, pradiniu atskaitos taškų pasirenkant vykdymo statistiką, gautą sukompiliavus su raktu -O3 ir programas paleidus. Vėliau perkompiliuojama su visomis geriausiomis anksčiau išvardintų parametrų reikšmėmis. Rezultatai pateikiami 50 paveikslėlyje, tiek taktų skaičius, tiek UIPC ir atminties prastovų pokyčiai pateikiami procentais:



50 pav. gzip, crafty ir oopack_v1c8.cc programų optimizavimo rezultatai.

Iš grafiko pastebime, kad visų trijų programų paspartėjimas daug maž vienodas apie 16-17 procentų, šachmatų žaidimo rezultatai šiek tiek geresni, visų pirma dėl paprastesnio kodo, bei palyginti mažo duomenų kiekio skaitymo tiek iš įvedimo įrenginio, tiek ir laikomo atmintyje. Testavimo programos *oopack* rezultatai šiek tiek prastesni vien dėl to, jog ją sudaro įvairūs testai, kurie skirti konkrečiai optimizacijai tirti ir ne taip lengvai pasiduoda kitoms optimizacijoms. Gzip programos darbą taip pat įtakojo didelis duomenų kiekis iš įvedimo įrenginio (archyvuojamo failo dydis apie 420 megabaitų). Iš esmės galima teigti, kad nagrinėti vidiniai gcc kompiliatoriaus parametrai turėjo gana apčiuopiamą teigiamą įtaką kompiliuojamo kodo spartinimui. Taip pat galima teigti, jog toliau tobulinant patį gcc kompiliatorių, daugiausiai dėmesio reikia skirti būtent atminties operacijų optimizavimui.

6.Rezultatai ir išvados

Pirmoje darbo dalyje buvo pristatyta IA-64 architektūra, pristatytos tipinės tradicinių architektūrų problemos, bei aprašytos Itanium architektūros savybės, kurios leidžia efektyviai spręsti tas problemas. Remiantis šia medžiaga Itanium savybės buvo eksperimentiškai nagrinėjamos toliau, generuojant jas taikantį asemblerio kodą, bei išmatuoti programų vykdymo laikai su jomis ir be jų:

- Išreikštinai lygiagretus instrukcijų vykdymas yra viena iš kertinių IA-64 architektūros savybių. Instrukcijos grupuojamos į grupes pagal tam tikras taisykles, kurios užtikrina korektiškus programos vykdymo rezultatus. Instrukcijų grupė vykdoma per vieną taktą tiktai tuomet, kaip procesorius turi pakankamai vykdymo konvejerių tai instrukcijų grupei. Kitu atveju įvyksta vieno ar keleto taktų prastovos, kol atsilaisvins reikalingas vykdymo konvejeris.
- Predikacija leidžia atsisakyti architektūriškai „brangių“ sąlyginių perėjimų, kiekvienai instrukcijai priskiriant po predikatinį registrą, ir taip efektyviai valdant instrukcijos vykdymą. Svarbu nepamiršti, jog „išjungtos“ instrukcijos taip pat užima procesoriaus vykdymo konvejerius, todėl reikia atkreipti dėmesį į tokių instrukcijų kiekį grupėje.
- Išankstinis valdymo bei duomenų spėjimas leidžia panaikinti apribojimus, kurie atsiranda dėl nuo sąlygos priklausomo atminties nuskaitymo (valdymo spėjimas), bei dėl atminties skaitymo ir rašymo sričių persidengimo (duomenų spėjimas). Šios savybės dėka instrukcijos žymiai geriau grupuojamos.
- Sąlyginių perėjimų prognozavimo galimybės suteikia galimybę perteikti kompiliatoriaus surinktą programos vykdymo kelio analizę procesoriui ir taip žymiai paspartinti programos vykdymą, teisingai nustatant sąlyginių perėjimų atšakų vykdymą.
- Operatyvios atminties skaitymo optimizavimas taip pat yra viena iš svarbiausių sąvybių, leidžianti kompiliatoriui valdyti spartinančiasias atmintines, bei reikalingus duomenis užkrauti iš anksto.
- Ciklų vykdymas konvejerio principu taipogi leidžia paslėpti skaitymo iš spartinančiosios atminties vėlavimus, bei žymiai geriau išlygiagretinti ciklo kūną. Pats ciklų konvejerizavimas yra automatiškai palaikomas IA-64 architektūros be papildomo kodo generavimo.

Eksperimentiškai tiriant IA-64 architektūros savybes ir palyginus programų darbo efektyvumus, pastebime, jog ne visos savybės teikia vienodą našumą ir vienodai lengvai yra pritaikomos. Pavyzdžiui sąlyginių atšakų prognozavimas yra lengvai realizuojamas, tačiau pačių tikimybių apskaičiavimas yra ganėtinai sudėtinga procedūra ir reikalauja pakankamai gerų euristinių kodo analizės algoritmų. Taipogi, pačias savybes galima suskirstyt į dvi kategorijas.

Pagrindinės savybės - tai išreikštinai lygiagretus instrukcijų vykdymas, atminties skaitymo optimizavimas, sąlyginių perėjimų valdymas.

Pagalbinės savybės – šios savybių dėka yra maksimaliai panaudojamos pagrindinės savybės. Predikacija bei valdymo ir duomenų spėjimai leidžia žymiai geriau išlygiagretinti instrukcijas. Ciklų vykdymas konvejerio principu leidžia paslėpti operatyvios atminties skaitymo vėlavimus.

Kaip parodė tyrimai, svarbiausias dalykas optimizuojant kodą - stengtis kuo labiau išlygiagretinti instrukcijas, maksimaliai išnaudojant procesoriaus vykdymo konvejerius. Antras dalykas – atminties skaitymo optimizavimas, todėl sekančioje darbo dalyje nagrinėjami būtent tie septyni kompiliatoriaus vidiniai parametrai, kurie susiję su šiomis dviem savybėmis. Keičiant jų reikšmes, bei naudojantis specialia kompiliatoriaus testavimo programa, buvo renkama įvairi vykdymo statistika, bei rezultatai pateikiami grafikuose. Iš esmės galima daryti išvadą, kad geriausias paspartėjimas buvo gautas, būtent keičiant parametrus susijusius su atminties operacijų optimizavimu – iki 20%.

Gautas naujų parametrų reikšmių rinkinys buvo išbandytas su taikomosiomis programomis. Pirmoji programa – gzip archyvavimo programa. Ją sukompiliavus su minėtų reikšmių rinkiniu, pasiektas vidutinis 16% procentų našumas. Žinoma tam įtakos turėjo sudėtingas programos kodas, bei pakankamai dideli duomenų srautai iš įvedimo įrenginių, bei laikomi atmintyje. Antroji programa “crafty” šachmatų žaidimas su tuo pačiu parametrų reikšmių rinkiniu demonstravo apie 20% didesnį našumą. Tokius rezultatus lėmė sąlyginai paprastesnis programos kodas, bei nedidelis įvedimo duomenų kiekis.

Galima teigti, jog magistrinio darbo tikslas pasiektas – gautas gcc kompiliatoriaus parametrų reikšmių rinkinys leidžia generuoti efektyvesnį kodą, tačiau reikia atkreipti dėmesį, jog nagrinėjama buvo tik dalis visų vidinių parametrų, taipogi kai kurių parametrų reikšmių įtaka kompiliavimo laikui taip pat stipriai jaučiama, todėl tekdavo ieškoti kompromiso tarp šių dviejų kriterijų. Iš gautų darbo rezultatų gautos tokios išvados:

- Pirmas akivaizdus dalykas – būtinybė kiekvienai architektūrai turėti skirtingas tik jai pritaikytas optimizavimo parametrų reikšmes. Tokiu būdu atskirai architektūrai parinktos

parametrų reikšmės leis generuoti tai architektūrai tinkamiausią kodą.

- Antra – reikia atskirti nuo architektūros nepriklausomus kodo optimizavimo algoritmus, nuo tų, kurie skirti konkrečiai architektūrai. Šiuo metu atvirojo kodo kompiliatoriuje gcc IA-64 architektūrai skirtų optimizavimo algoritmų nėra apskritai.
- Taipogi patys parametrai turėtų būti išskirti į dvi grupes – valdantys bendrus optimizavimo algoritmus, bei nuo architektūros priklausomus. Tokiu atveju žymiai supaprastėtų kompiliatoriaus pritaikymas konkrečiai architektūrai.
- Kai kurie parametrai turėtų būti perkeliami iš kompiliatoriaus kodo į specialius procesorių aprašančius failus, kadangi jie valdo specifinius dalykus, priklausomus tik nuo konkretaus procesoriaus mikroarchitektūros realizacijos.

Apibendrinant galima teigti, jog kompiliatorių pritaikymas IA-64 architektūrai nėra toks efektyvus kokio norėtųsi ir yra nemažai problemų, kurias reikėtų spręsti.

Literatūros sąrašas

[ASU01] A. Aho, R. Sethi, J. Ullman.“Compilers Principles, Techinques, And Tools”, Pearson Education, 2001

[BCC00+] Jay Bharadwaj, William Y. Chen, Weihaw Chuang, Gerolf Hoflenhner, Kishore Menezes, Kalyan Muthukumar, Jim Pierce „The Intel IA-64 compiler code generator“ IEEE 2000

[BT08a] Joseph Bissell, Walt Triebel “Recognizing Efficient Use of Caches in Code for the Itanium Processor Family”, 2008. Prieiga per internetą: http://software.intel.com/en-us/articles/recognizing-efficient-use-of-caches-in-code-for-the-itaniumr-processor-family/

[BT08b] Joe Bissell, Walt Triebel ”Scaling Itanium Architecture for Higher Performance”, 2008. Prieiga per internetą: http://software.intel.com/en-us/articles/scaling-itaniumr-architecture-for-higher-performance/

[GSG08] Gcc steering group, “A Gnu Manual”, 2008. Prieiga per internetą: http://gcc.gnu.org/onlinedocs/gcc-4.4.0/gcc/

[HMR00+] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, Rumi Zahir „Introducing the IA-64 architecture“ IEEE 2000

[HR00] Jerry Huck, Rumi Zahir “The IA-64 System Architecture: Tutorial For Hardware, OS, & Application Developers”, 2000

[Int00] Intel, “IA-64 Assembly Language Reference Guide”, 2000

[Int01] Intel, “Itanium Software Conventions and Runtime Architecture Guide”, 2002

[Int04] Intel. „Intel Itanium 2 Processor reference manual for software developement and optimization“, www.intel.com, 2004

[Int06a] Intel. „Intel Itanium architecture Software Developer‘s manual – application architecture“, www.intel.com, 2006

[Int06b] Intel „Intel Itanium architecture Software Developer‘s manual – system architecture“, www.intel.com 2006

[Int06c] Intel „Intel Itanium architecture Software Developer‘s manual – instruction set reference“, www.intel.com 2006

[Int06d] Intel, “Intel Itanium Architecture. Software Developer’s Manual - Volume 3: Instruction Set Reference”, 2006

[ISN08a] Intel Software Network. „How to Perform Code Timing and Profiling for Linux on 64-Bit Intel Architecture“, 2008. Prieiga per internetą: http://software.intel.com/en-us/articles/how-to-perform-code-timing-and-profiling-for-linux-on-64-bit-intel-architecture/

[ISN08b] Intel Software Network. “How to Handle Streaming Data Optimally on 64-Bit Intel Architecture”, 2008. Prieiga per internetą: http://software.intel.com/en-us/articles/how-to-handle-streaming-data-optimally-on-64-bit-intel-architecture/

[ISN08c] Intel Software Network. “How to Optimize Instruction Latencies in Assembly Code for 64-Bit Intel Architecture”, 2008. Prieiga per internetą: http://software.intel.com/en-us/articles/how-to-optimize-instruction-latencies-in-assembly-code-for-64-bit-intel-architecture/

[JAR02] Sverre Jarp “A Methodology for using the Itanium 2 Performance Counters for Bottleneck Analysis ”, 2002.

[KIF99] Allan Knies, Wei Li, Jesse Fang “IA64 architecture and Compilers”, 1999

[KKL00+] Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery „An advanced optimizer for the IA-64 architecture“ IEEE 2000

[LBT06] Jiangjiang Liu, Brian Bell, Tan Truong „Analysis and characterization on Intel Itanium instruction bundles for improving VLIW processor performance“ IEEE 2006

[LCC03+] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew „Speculative register promotion using advanced load address table (ALAT)“ IEEE 2003

[Lev08] David Levinthal, “Black Belt Itanium Processor Performance: Visual Inspection of Compiler-Generated Assembly (Part 5 of 5)”, 2008. Preiga per internetą: http://software.intel.com/en-us/articles/black-belt-itaniumr-processor-performance-visual-inspection-of-compiler-generated-assembly-part-5-of-5/

[RDG04+] Hongbo Rong, Alban Douillet, R. Govindarajan, Guang R Gao „Code generation for single-dimension software pipelining of multi-dimensional loops“ IEEE 2004

[SA00] Harsh Sharangpani, Ken Arora „Itanium processor microarchitecture“ IEEE 2000

[YLW05] C. Yang, C. Li, F. Wang “Performance Improvements for GCC Using Architecture Features on IA-64”, National University of Defence Technology, 2005