VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Kompiliatorių optimizavimas IA-64 architektūroje

Compiler optimizations on IA-64 architecture

Magistro baigiamasis darbas

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

Vilnius – 2009

Santrauka lietuvių kalba

Šiame darbe buvo išnagrinėtos Intel Itanium (IA-64) architektūros savybės, įtakojančios kompiliatoriaus generuojamą kodą, ir išanalizuotos kompiliatoriaus optimizacijos, kurios buvo pritaikytos IA-64 architektūrai. Buvo prieita prie išvados, kad tokias optimizacijas galima susiskirstyti į kelis tipus. Pirmiausia nuo architektūros priklausomos optimizacijos, kurių efektyvumą galima padidinti išnaudojant predikaciją ir prognozavimo savybes ar kitas IA-64 specifines savybes. Antra, nuo architektūros nepriklausomos tradicinės optimizacijos, kurių pertvarkomo kodo efektyvumą galima padidinti parenkant kitokius šias optimizacijas valdančius kompiliavimo parametrus. Tyrime buvo išnagrinėtos ciklų optimizacijos, kurių kodą galimą būtų pakeisti valdomais parametrais. Tyrimas parodė, kad iš tiesų įmanoma sugeneruoti efektyvesnį kodą Intel Itanium architektūroje, keičiant šių parametrų reikšmes nuo numatytųjų reikšmių.

Raktiniai žodžiai: Itanium, ciklų optimizacijos, kompiliatorius, kompiliatoriaus optimizacijos, GCC

Summary

This thesis deeply explored Intel Itanium architecture features that improve a code generated by compiler. Compiler optimizations which are tuned to this architecture are also described. Accomplished research showed that there were several types of optimizations which can be improved on IA-64 architecture. Firstly, optimizations which are dependent on architecture can be optimized using predication and speculation or other unique IA-64 features. Secondly, optimizations that are undependable from traditional architecture can be improved using more aggressive compilation controllable parameters than they are by default. Loop optimizations were chosen for final research. Research proved that changing values of these parameters from default can improve program performance.

Keywords: Itanium, loop optimizations, compiler, compiler optimizations, GCC

Turinys

Įvadas

„IA-64“ (Intel Architecture-64) – yra 64 bitų procesorių architektūra, sukurta Intel ir Hewlett-Packard kompanijų ir realizuota Itanium bei Itanium 2 procesoriuose. Praėjusio dešimtmečio viduryje HP ir Intel bendromis pradėjo kurti naują procesorių architektūrą, kuri turėjo du pagrindinius tikslus:

1. Suteikti galimybę papildomai išlygiagretinti programos kodą, panaudojant gilesnę kodo analizę.

2. Supaprastinti procesoriaus dizainą ir sumažinti energijos naudojimą, eliminuojant funkcinius elementus, kurie atsakingi už vykdomo kodo planavimą.

Tokiu būdu atsirado nauja 64 bitų Intel Itanium (IA-64) procesorių architektūra, kuri evoliucionavo iš VLIW (very long instruction word) modelio. Nauja paradigma buvo pavadinta EPIC (Explicitly Parallel Instruction Computing). Itanium procesoriai labai remiasi kompiliatoriaus atliekamu darbu ir jo gebėjimu, pasinaudojus naujomis architektūrinėmis savybėmis, pertvarkyti kodą taip, kad jis būtų vykdomas išlygiagretintai. Dėl šios priežasties yra labai svarbu, kad kompiliatoriaus naudojamos optimizacijos sugebėtų efektyviai išnaudoti esamus procesoriaus resursus [BCC+00]. Pagrindinės architektūrinės savybės, kurios leidžia kompiliatoriui atrasti kodą, kurį būti galima išlygiagretinti:

1. dideli bendros paskirties ir specializuotų registrų resursai;

2. predikacija;

3. kontrolės ir duomenų prognozavimas;

4. rotuojantys registrai;

5. lygiagretumo semantika.

Šie architektūriniai sprendimai leidžia sukurti naujas optimizacijas ir patobulinti esamas tradicines optimizacijas arba jas efektyviau išnaudoti nei tradicinėse architektūrose. Pavyzdžiui, predikatai padeda išspręsti sunkiai atspėjamus sąlygos sakinius, eliminuojant teisingos atšakos spėjimo būtinybę, kadangi abi sąlygos sakinio atšakos įvykdomos vienu metu. Prognozavimas perkelia sąlyginai lėtesnes užkrovimo instrukcijas į ankstesnę programos stadiją, net jei ir yra užkrovimui reikalingų duomenų trūkumo pavojus.

Norint atitikti šiuolaikinių architektūrų reikalavimus, kompiliatorių optimizavimas reikalauja vis daugiau sudėtingų transformacijos algoritmų. Priklausymas nuo kompiliatorių

ypatingai pasireiškia aiškiai lygiagrečiose, nevienalyčių resursų platformose, tarp kurių yra ir Intel Itanium arba Philips TriMedia architektūros [TVV+03]. Šiose ir kitose architektūrose kompiliatorius nebegali pasikliauti tik paprastomis metrikomis kaip instrukcijų skaičius. Kompiliatorius turėtų atidžiai subalansuoti vykdymo resursų panaudojimą, registrų naudojimą ir priklausomybės lygį, kai reikia minimizuoti nereikalingus uždelsimus dėl įvairių dinaminių veiksnių, pavyzdžiui, spartinančiosios atminties trūkumo arba neteisingų atšakos spėjimų.

Darbo tikslas

Darbo tikslas yra nustatyti sąryšius tarp IA-64 architektūros savybių ir kompiliatoriaus optimizuojančių pertvarkymų bei atrasti ciklo optimizacijas valdančių parametrų rinkinį ir jų reikšmes, kurios leistų generuoti efektyviau vykdomą kodą.

Uždaviniai

Siekiant įgyvendinti darbo tikslą yra keliami šie pagrindiniai uždaviniai:

- Išnagrinėti IA-64 architektūros savybes, nuo kurių priklauso kompiliatoriaus generuojamas kodas.
- Išanalizuoti IA-64 ir tradicinių architektūrų kompiliatorių optimizuojančius pertvarkymus ir nustatyti sąryšius su IA-64 architektūros savybėmis.
- Atrasti ciklo optimizacijas valdančių parametrų rinkinį ir jų reikšmes, kurios generuotų efektyvesnį kodą.
- Verifikuoti parametro reikšmių rinkinį testavimo rezultatais.

1. IA-64 architektūros savybių apžvalga

Prieš susipažįstant su IA-64 kompiliatorių optimizuojančiais pertvarkymais, pradžioje turėtume susipažinti su architektūros savybėmis, nuo kurių priklauso kompiliatoriaus generuojamas kodas, ir kokius privalumus jos suteikia. IA-64 architektūra išsiskiria tokiomis savybėmis:

- Dideli registrų resursai (328 bendros paskirties ir specialūs registrai).
- Aiškus lygiagrečiai vykdomų instrukcijų išskyrimas kode (EPIC paradigma). Priklausomybės tarp komandų paiešką vykdo kompiliatorius, o ne procesorius.
- Predikacija (Prediction). Sąlygos sakinių atšakos, pažymėtos predikatais, yra vykdomos lygiagrečiai.
- Užkrovimas prognozuojant (Speculative loading). Sąlyginai lėtų užkrovimo iš atminties instrukcijų perkėlimas į ankstesnę programos stadiją.

Itanium turi šešis konvejerius (kartu su trimis perėjimų prognozavimo blokais), du FPU konvejerius, SIMD instrukcijų modulį, suderinamą su SSE2, du užkrovimo blokus, trys ciklų analizės blokus ir du saugojimo blokus. Procesorius gali atlikti iki šešių instrukcijų per vieną procesoriaus taktą. Procesorius turi didelę trečio lygio spartinančiąją atmintį.

Itanium rodo vienus geriausių rezultatų su slankiojo kablelio ir sveikaisiais skaičiais, todėl šie procesoriai yra stiprūs moksliniuose skaičiavimuose bei vykdant CAD tipo programas.

Pav. 1 Skirtumai tarp IA-64 ir x86 architektūrų [Fos05]

1.1 Registrų resursai

IA-64 architektūroje yra daug bendros paskirties ir specialiųjų registrų. Tai sumažina kreipimosi į atmintį dažnumą duomenų užkrovimui ir tarpinių duomenų išrinkimui.

Architektūroje yra 128 64 bitų bendrosios paskirties registrų, kiekvienas iš jų turi dar vieną papildomą bitą NaT (Not a Thing), kuris naudojamas pažymėti instrukcijas, priklausančias nuo prognozavimo savybės. Jei prognozavimas buvo neteisingas, čia bus pažymima klaida.

Be šių registrų taip pat IA-64 architektūroje dar yra šie registrai:

- 128 82 bitų slenkančio kablelio registrai;
- aštuoni 64 bitų atšakos (branch) registrai funkcijų iškvietimo sujungimui ir grąžinimui;
- 64 vieno bito predikatų registrai, kurie saugo sąlygos reiškinių rezultatą (true/false).

Kadangi bendras architektūrinių registrų skaičius yra lygus 328, labai svarbus yra registrų failo greitas darbas. Akivaizdu, kad kuo jis didesnis, tuo sunkiau jam įgyvendinti šią užduotį, todėl buvo įgyvendinta rotuojančių registrų savybė. Registrų failo reikšmės yra „apsukamos“ pagal tam tikrą periodą, o jei registrų pritrūksta, registrų reikšmių užpildymu ir atlaisvinimu automatiškai rūpinasi RSE (register stack engine).

1.2 Lygiagretumo semantika

IA-64 asemblerio kode, kuris paprastai generuojamas kompiliatoriaus, komandų grupės, kurios yra skirtos lygiagrečiam vykdymui, turi būti aiškiai išskirtos. Aiškus paralelizmas yra kertinė architektūros savybė. Šios grupės asemblerio kode žymimos stop žyme ir visos vienos grupės komandos, priklausomai nuo esamų resursų, gali būti vykdomos lygiagrečiai be papildomo patikrinimo. Intel tai vadina kontroliuojamo srauto lygiagretumu ir jų nuomone padeda sumažinti nuoseklių atšakų vykdymo skaičių [HMR+00].

Iliustruojant šią savybę pavyzdžiu, panagrinėkime šį C kalba parašytą sąlygos sakinį:

```
if ( (a==0) || (b<=5) ||
(c!=d) || (f & 0x2) )
{
  r3 = 8;
}
```

Asemblerio kalba jis atrodys taip:

```
cmp.ne p1 = r0,r0
add t = -5, b;;
```

```
cmp.eq.or p1 = 0,a
cmp.ge.or p1 = 0,t
cmp.ne.or p1 = c,d
tbit.or p1 = 1,f,1 ;;
(p1) mov r3=8
```

Asemblerio kodas yra suskirstytas į trys komandų grupes, atskirtas „;;“ simboliu, kurių kiekvienoje komandos gali būti saugiai vykdomos lygiagrečiai be papildomų patikrinimų. Pirmoje grupėje yra sukuriamas p1 operandas, kurio pradinė reikšmė yra *false*. Antroje grupėje kiekvienas sąlygos sakinio „or“ reiškinys yra vykdomas lygiagrečiai vienas nuo kito. Jei nors vienas jų teisingas, p1 operandas įgauna *true* reikšmę, priešingu atveju išlieka *false*. Paskutinėje grupėje priklausomai nuo predikato reikšmės r3 įgaus reikšmę arba instrukcija bus atmesta.

Komandos yra sugrupuojamos į 128 bitų ilgio junginį. Junginys susideda iš trijų komandų ir šablono, kuriame nurodoma priklausomybė tarp komandų (ar jos gali būti paleidžiamos lygiagrečiai) ir kitų sąryšių (pvz.: ar galima paraleliai paleisti k1 komandą iš junginio s1 ir komandą k4 iš junginio s2). Junginiai neturi jokios įtakos komandų grupėms ir jų riboms.

Pav. 2 IA-64 architektūros instrukcijų formatas [Fos05]

1.3 Predikacija

Intel Itanium architektūra naudoja pilną predikacijos modelį, kuriame kompiliatorius gali pridėti predikato požymį prie visų instrukcijų. Tam architektūroje yra realizuoti 64 1 bito predikatų registrai, kurie gali įgyti dvi reikšmes – tiesa/netiesa (true/false) [ШПа00]. Predikatai yra paprasčiausia žymė, kuri įgalina sąlyginį programos vykdymą, priklausantį nuo predikato reikšmės, kurio reikšmė tuo tarpu priklauso nuo sąlygos. Instrukcija, kurios predikato reikšmė yra „true“, vykdoma normaliai. Tuo tarpu, jei predikato reikšmė yra „false“, prie jos pririšta instrukcija, nors ir įvykdoma, jos rezultatas nėra įrašomas į atmintį ar registrus. Predikacija yra

efektyvi panaikinant atšakas ir sumažinant nuostolius dėl atšakų neteisingo spėjimo. Predikatų veikimą galima iliustruoti paprastu sąlygos sakinio pavyzdžiu:

```
IF a[i].ptr != 0
  b[i] = a[i].l;
ELSE
   b[i] = a[i].r;
i = i+1;
```

Tradicinėse architektūrose programa užkrauna duomenis iš atminties ir palygina a(i).ptr reikšmę su nuliu ir, priklausomai nuo rezultato, atlieka vieną iš sąlygos sakinio atšakų [Dul98]. Dėl sąlygos sakinio tradicinis kompiliatorius sukuria keturių blokų struktūrą:

```
load a[i].ptr
p1,p2 = cmp a[i].ptr != 0
jump if p2
load r8= a[i].l
store b[i] = r8
jump
load r9 = a[i].r
store b[i] = r9
i = i+1;
```

Procesorius privalo vykdyti kiekvieną iš keturių blokų paeiliui. Instrukcijos atšakose blokuoja jų išlygiagretinimą. Predikacija naudojama pašalinant sunkiai atspėjamas atšakas į vieną bazinį bloką.

Itanium architektūroje yra sugeneruojami du predikatai:

```
load a[i].ptr
p1,p2 = cmp a[i].ptr != 0
<p1> load r8= a[i].l      <p2> load r9 = a[i].r
<p1> store b[i] = r8      <p2> store b[i] = r9
i = i+1;
```

Predikato reikšmė yra priskiriama 1, jei palyginimo rezultatas yra „true“, bei predikato reikšmė yra priskiriama 0, jei palyginimo rezultatas yra „false“. Galutinis kodas neturi atšakų, nes „then“ ir „else“ dalys yra vykdomos lygiagrečiai. Kadangi p1 ir p2 vienu metu gali turėti tik priešingas reikšmes, dvi saugojimo instrukcijos gali būti vykdomos lygiagrečiai, netgi kai jos rezultatus saugo į vieną adresą. Predikatas leidžia tik vienai iš šių reikšmių patekti į atmintį. Kita reikšmė yra tiesiog išmetama. Taip yra pasiekiamas instrukcijos lygio lygiagretumas ir panaikinimas neteisingų spėjimų nuostolis.

IA-64 komandų sistema suteikia kiekvienai komandai 6 bitų erdvę šio predikato saugojimui [ШПа00]. Tokiu būdu vienu metu gali būti panaudoti 64 skirtingi predikatai. Po to,

kai komandos yra pažymėtos, kompiliatorius nustato, kurie iš jų gali būti vykdomi lygiagrečiai. Kompiliatorius taip pat privalo nustatyti duomenų priklausomybę (dvi komandos, iš kurių viena naudoja prieš tai esančios komandos rezultatus, negali būti vykdomos lygiagrečiai). Kadangi kiekviena atšakos dalis yra bent dalinai nepriklausoma nuo kitų dalių, visada bus randama kodo dalis, kurią galima išlygiagretinti. Ne visais atvejais galima komandas atžymėti predikatais, nes kartais technologijos panaudojimas gali privesti prie to, kad bus sunaudota daugiau taktų nei sutaupoma.

Vykdymo metu, kai procesorius aptinka pažymėtas atšakas, vietoj to, kad bandytų atspėti, kurią atšaką vykdyti, jis pradeda lygiagrečiai vykdyti blokus, kurie atitinka visas sąlygos sakinio atšakas. Tokiu būdu mašininiu požiūriu atšakų nėra. Kažkokiu laiko momentu procesorius išsiaiškins sąlygos rezultatą. Pavyzdžiui, sąlygos reikšmė yra TRUE ir teisingas kelias pažymimas predikate P1. Tokiu atveju P1 registre bus įrašytas 1, kituose nulis.

Iki šio momento procesorius jau tikriausiai įvykdė dalį komandų, kurios atitinka abi sąlygos sakinio atšakas, tačiau iki šiol neišsaugojo rezultatus. Prieš tai darydamas, procesorius patikrina komandą atitinkantį predikato registrą. Jei jame 1 – tada komanda baigiama vykdyti ir rezultatai įrašomi į atmintį. Jeigu 0 – komanda atmetama.

1.4 Prognozavimas (speculation)

Dar viena Itanium architektūros kertinė savybė yra duomenų užkrovimas prognozuojant. Prastovos kraunant duomenis iš atminties, net jei tai yra spartinančioji atmintis, yra gana didelės. Viena iš IA-64 architektūros savybių yra išankstinis duomenų užkrovimas, perkeliant užkrovimo instrukcijas į ankstesnę programos stadiją. Pagrindinis išankstinio užkrovimo tikslas – atskirti užkrovimo ir duomenų naudojimo etapus. Kaip ir komandų atžymėjimo predikatais technologijoje, čia irgi kooperuojasi optimizavimas kompiliavimo ir vykdymo metu.

Kompiliatorius iš pradžių peržiūri programos kodą, atpažindamas komandas, kurios naudoja duomenis iš atminties. Visur, kur tai įmanoma, pridedama išankstinio užkrovimo komanda, kuri patalpinama gerokai ankstesnėje stadijoje nei pati komanda, naudojanti tuos duomenis.

Vykdymo metu procesorius iš pradžių atranda išankstinio užkrovimo komandą ir mėgina užkrauti duomenis iš atminties. Kai kada šis mėginimas būna nesėkmingas, pavyzdžiui, jeigu komanda, reikalaujanti duomenų dar yra atšakoje, kurios sąlygos iki šiol nėra apskaičiuotos. „Įprastinėse“ architektūroje tuoj pat generuojama klaida, tuo tarpu IA-64 architektūroje klaidos generavimas atidedamas iki sutinkama duomenų reikalaujanti komanda.

Yra išskiriami du prognozavimo tipai: valdymo ir duomenų. Valdymo prognozavimas leidžia perkelti duomenų užkrovimus virš sąlygos sakinio atšakos. Tam yra naudojami specialūs NaT požymiai. Šiuo požymiu komanda pažymi klaidą, jei duomenų užkrovimas buvo nesėkmingas, ir klaidos valdymas atidedamas tol, kol nesutinkama check.s tikrinimo komanda. Tuo tarpu duomenų prognozavimas leidžia išvengti galimo konflikto kreipiantis į atmintį. Tai naudinga programuojant kalbomis, kurios turi rodyklinius duomenų tipus, pavyzdžiui, C. Paprastai kompiliatorius tokiu atveju negali nustatyti, kuri atminties vieta bus naudojama. Duomenų prognozavimas leidžia kompiliatoriui suplanuoti užkrovimą, net jei kompiliatorius nėra įsitikinęs, ar adresai neuždengs vienas kito. Kompiliatorius veikia analogiškai kaip ir valdymo prognozavimo atveju, naudodamas išankstinio užkrovimo komandą (load.a) ir vėliau prideda užkrovimo tikrinimo komandą (check.a). Tai padeda minimizuoti delsimą dėl kreipimosi į atmintį.

1.4.1 Valdymo prognozavimas

Ši savybė leidžia perkelti duomenų užkrovimą virš sąlygos sakinių, kai jų nepavyksta pašalinti predikacijos metodu.

Kadangi užkrovimo operacijos turi ilgesnį gaišties laiką nei dauguma skaičiavimo instrukcijų ir jie linkę pradėti ilgai atliekamų instrukcijų grandinę, kiekvienas apribojimas, siekiant juos perkelti į ankstesnę programos stadiją, gali sumažinti galimybę išlygiagretinti programos kodą. Vienas iš tokių apribojimų susijęs su taisyklingu išimčių (exceptions) valdymu. Pavyzdžiui, užkrovimo instrukcija gali bandyti pasiekti duomenis, kurių programa neturi teisės pasiekti. Kai programa atlieka tokį neteisėtą veiksmą, ji paprastai privalo būti nutraukta. Be to visos išimtys taip pat privalo būti pateiktos tokia seka, lyg jos buvo vykdomos pagal programuotojo aprašytą eiliškumą. Kadangi užkrovimo perkėlimas virš sąlygos sakinio atšakos pakeičia priėjimo prie atminties eiliškumą ir jos atitikimą programos valdymo sekai, ne EPIC architektūros draudžia tokį kodo perkėlimą.

IA-64 architektūroje realizuota nauja klasė užkrovimo instrukcijų, kurios gali būti saugiai suplanuotos prieš vieną ar kelias atšakas. Bloke, kuriame programuotojas pradžioje užrašė užkrovimą, kompiliatorius pažymi prognozuojamą patikrinimą (chk.s instrukciją). IA-64 šis procesas yra vadinamas valdymo prognozavimu [HMR+00].

Jeigu vykdymo metu prognozuojamas užkrovimas užsibaigia klaida, išimtinė situacija yra atidedama ir atidėjimo (NaT) žymė yra įrašoma į registrą. Vėliau buvusio užkrovimo instrukcijos vietoje patalpinta chk.s instrukcija patikrina šio registro NaT žymę ir, jei ji

egzistuoja, atsišakoja į specialų „taisymo“ kodą (kurį taip pat sugeneruoja kompiliatorius). Jei reikia, šis kodas iš naujo užkrauna duomenis ir vėl grįžta į pradinį programos kūną.

Kadangi beveik visos instrukcijos vykdymo metu Itanium architektūroje teiks pirmenybę NaT (vietoje klaidos išmetimo), visos grandinės gali būti suplanuotos prognozavimo metodu. Pavyzdžiui, kai vieno registro operandas sudėties instrukcijoje turi NaT, sudėtis neišmeta klaidos. Vietoje to, jis įrašo NaT į sudėties rezultato registrą, taip sukurdamas atidėtą išimtį. Jei vienas ar du iš prognozavimo užkrovimo rezultatų galiausiai yra naudojami bendruose skaičiavimuose, perduodamos NaT reikšmės leidžia kompiliatoriui įdėti tik vieną chk.s instrukciją, siekiant patikrinti kelis prognozuojamus skaičiavimus.

Jeigu chk.s tikrinimo metu randama atidėta išimtis viename iš skaičiavimo grandinės rezultatų, pataisymo kodas paprasčiausiai iš naujo įvykdo visą grandinę, išspręsdamas surastas klaidas. Šis paprastas mechanizmas leidžia kompiliatoriui išlygiagretinti nemažą dalį vykdomo kodo ir pagerinti programos vykdymo laiką.

1.4.2 Duomenų prognozavimas

Populiarios programavimo kalbos, tokios kaip C, leidžia naudoti rodyklinius duomenų tipus, kuriais galima pasiekti atmintį. Tačiau rodyklės dažnai neleidžia kompiliatoriui nustatyti, į kurią atminties vietą jos rodo. Dar tiksliau, tokios nuorodos gali sutrukdyti kompiliatoriui sužinoti, ar saugojimo operacija ir sekanti po jos užkrovimo operacija nenurodo į tą pačią atminties sritį. Tai trukdo kompiliatoriui pertvarkyti instrukcijas.

Intel Itanium išsprendžia šią problemą instrukcijomis, kurios leidžia kompiliatoriui suplanuoti užkrovimą prieš vieną ar daugiau išsaugojimų komandų (store), netgi tada, kai kompiliatorius nežino, ar jos nenurodo į tą pačią atminties sritį. Tai vadinama duomenų prognozavimu ir jos veikimo principas panašus į valdymo prognozavimo veikimą [HMR+00].

Kai kompiliatorius nori suplanuoti užkrovimą anksčiau ankstesnio saugojimo, jis naudoja išankstinio užkrovimo ld.a instrukciją. Tada patalpina užkrovimo patikrinimo instrukciją chk.a po visų įsiterpusių saugojimo operacijų. Pavyzdys:

```
Tradicinė instrukcijų seka:
instrA
instrB
...
store
ld8 r1=[r2]

use r1
```

IA-64 instrukcijų seka
ld8 r1=[r2]

use r1
instrA
instrB
...
store
chk.a

Išankstinis užkrovimas vyksta taip pat kaip ir tradicinis užkrovimas, tačiau vykdymo metu jis išsaugo informaciją apie taikomą registrą, naudojamą atminties adresą ir naudojamos atminties ilgį į išankstinio užkrovimo adresų lentelę (ALAT). ALAT yra spartinančiosios atminties tipo aparatinė struktūra su turinio adresuojama atmintimi.

Kai vykdoma saugojimo operacija, aparatinė įranga patikrina saugojimo adresą su ALAT įrašais ir ištrina iš ALAT adresus, kurie persidengia su saugojimo adresu. Vėliau, kai yra įvykdoma chk.a instrukcija, aparatinė įranga patikrina ALAT įrašą su išankstinį užkrovimą atitinkančiu įrašu. Jei įrašas randamas, išankstinis užkrovimas buvo sėkmingas ir chk.a instrukcija nieko nedaro. Priešingu atveju galėjo būti adresų kolizija ir tikrinimo instrukcija nukeliauja į pataisymo bloką ir iš naujo įvykdo kodą (kaip ir valdymo prognozavimo metu).

Dėl šio mechanizmo kompiliatorius gali prognozuoti ne tik vieną užkrovimą, bet ir visą grandinę skaičiavimų prieš visą eilę konfliktuojančių saugojimo operacijų.

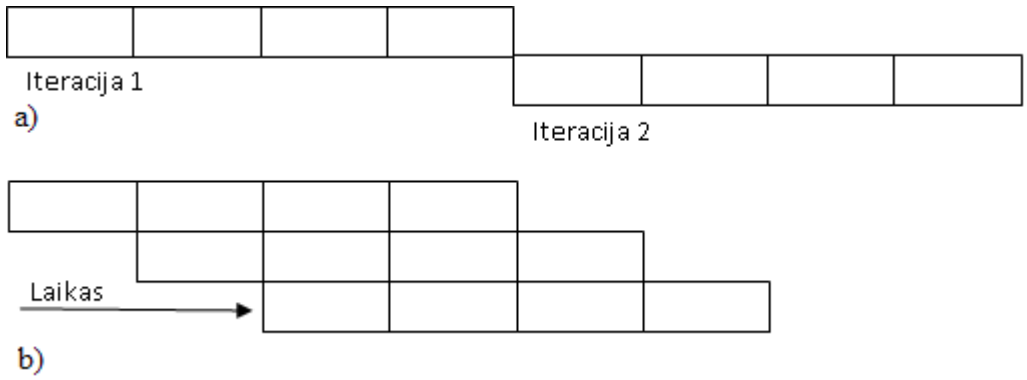
Palyginus kitas struktūras, tokias kaip spartinančioji atmintis, ALAT realizacija yra paprastesnė ir mažesnė procesoriaus integrinėje schemoje negu ekvivalenčios struktūros, reikalingos „out-of-order“ tipo procesoriuose. Taip pat ši savybė leidžia efektyviau pertvarkyti kompiliuojamą kodą jo vykdymo išlygiagretinimui.

1.5 Programinis konvejeris (Software pipelining)

Programinis konvejeris – tai optimizacija, skirta optimizuoti ciklus, panaudojant aparatinio konvejerio metodiką. Ji pagerina ciklų našumą, iš dalies uždengdama kelių iteracijų vykdymą, ir efektyviau išnaudoja aparatinius resursus, padidindama instrukcijų lygio lygiagretumą [DKK+99].

Kompiuteriai paprastai gerai atlieka pakartotines operacijas. Dėl šios priežasties dauguma programų naudoja ciklines struktūras, kurios daug kartų pakartoja tas pačias operacijas. Kadangi šie ciklai paprastai sudaro nemažą programos vykdymo laiko dalį, yra svarbu juose atrasti ir išnaudoti kuo daugiau kodo, kurį būtų galima vykdyti lygiagrečiai.

Nors instrukcijos cikle yra vykdomos dažnai, jos gali būti nepakankamai išlygiagretintos ir dėl šios priežasties procesoriaus vykdymo resursai nebus pilnai išnaudojami [HMR+00]. Ciklo iteracijų perdengimas (Pav. 3 b)) dažnai gali padėti padidinti lygiagrečiai atliekamo kodo lygį. Tokiu atveju, kada duotoji ciklo iteracija pradedama tuo metu, kai dar prieš tai buvusi iteracija nesibaigė, tada šis procesas yra vadinamas programiniu konvejeriu. Analogiškai veikia ir aparatinis konvejeris su instrukcijomis.



Pav. 3 Programinio konvejerio veikimo principas [HMR+00]

Nors šis požiūris gali atrodyti paprastas, tačiau be pakankamo aparatinės įrangos palaikymo įvairios problemos apriboja programinio konvejerio efektyvumą. Šios technikos naudojimas reikalauja papildomų veiksmų:

- ciklo skaitliuko valdymas;
- registrų pervadinimo valdymas konvejeryje;
- baigimas einamo darbo, kai ciklas pasibaigia (epilogo fazė);
- konvejerio sukūrimas, kai įeinama į ciklą (prologo fazė);
- ciklo išvyniojimas, siekiant išsiaiškinti tarp iteracinį lygiagretumą.

Kai kuriais atvejais šių priemonių įgyvendinimas gali padidinti pradinį ciklo kodą net iki 10 kartų. Dėl šios priežasties programinis konvejeris paprastai naudojamas tik specialiose programose su techniniais skaičiavimais, kur ciklo iteracijų skaičius yra labai didelis ir kodo didėjimo problema amortizuojasi.

IA-64 architektūroje dauguma iš šių problemų yra eliminuotos. Specialūs programos registrai, palaikantys ciklo skaitliuką ir konvejerio ilgį jo ištuštinimui (epilogo skaitliukas), padeda sumažinti ciklo iteracijų skaičiavimo ir ciklo sustabdymo problemą.

Kartu su rotuojančiais registrais specialios ciklais paremtos atšakos atlieka keletą veiksmų:

- po kiekvienos iteracijos automatiškai sumažina ciklo skaitliuką;
- tikrina ciklo skaitliuko reikšmę ir, jei reikia, sustabdo ciklo vykdymą;
- po kiekvienos stadijos automatiškai pervadina bendro naudojimo, slankiojo kablelio ir predikatų registrus.

Po kiekvienos rotacijos atrodo, kad kiekvienas rotuojantis registras peržengia viena pozicija į priekį, o paskutinis rotuojantis registras apsisuka atgal į apačią. Kiekviena rotacija efektyviai peržengia konvejerį per vieną stadiją į priekį.

Bendrų rotuojančių registrų rinkinį galima nustatyti naudojant alloc instrukciją. Rinkinys predikatų ir slankiojo kablelio registrų, kurie rotuoja, yra fiksuotas. Instrukcijos br.ctop ir br.exit suteikia skaitliuko palaikymą.

Rotuojantys predikatų registrai taip pat yra svarbūs, nes jie atlieka konvejerio stadijų validumo tikrinimo funkciją. Tai leidžia automatiškai ištuštinti programinį konvejerį, išjungiant ar įjungiant instrukcijas, priklausomai nuo to, ar konvejeris prasideda, yra vykdomas, ar baigiasi.

Kombinacija šių ciklo savybių ir predikatai leidžia kompiliatoriui sugeneruoti kompaktišką kodą, kuris pagrindinį darbą atlieka labai išlygiagretintai. Ir visai tai gali būti padaroma su tuo pačiu kodo kiekiu kaip ir be programinio konvejerio. Dėl šios priežasties IA-64 architektūra leidžia naudoti programinį konvejerį daug dažniau ir platesniam programų ratui nei kitose architektūrose.

1.6 Registrų modelis

IA-64 architektūra iš viso turi 328 registrus: 128 bendros paskirties registrus, 128 slankiojo kablelio procesoriaus registrus ir 64 1 bito predikatų registrus bei 8 atšakų registrus [Upg04]. Bendros paskirties registrai yra suskirstyti į 32 statinius (r0 - r31) ir 96 stekinius registrus, kurie gali būti pervadinti programiniu būdu. Pirmieji 32 registrai yra valdomi taip pat kaip ir RISC architektūroje. Stekiniai registrai sudaro IA-64 registrų steką. Įeinant į procedūrą šis mechanizmas automatiškai pateikia kompiliatoriui 96 naujų registrų rinkinį (r32 – r127). Nors

registrų stekas suteikia kompiliatoriui neribotos registrų erdvės iliuziją tarp procedūrų kvietimų, aparatinė įranga iš tikrųjų išsaugo ir atstato procesoriaus registrus iš atminties.

Aiškiai valdydamas registrus, naudodamas alloc funkciją, kompiliatorius kontroliuoja fizinio registro naudojimą. Stekinių registrai procedūrose visada prasideda r32.

Jei procedūros reikalauja daugiau registrų nei yra laisvų duotuoju momentu, registrų steko variklis (RSE) automatiškai išsaugo ankstesnių registrų reikšmes atmintyje lygiagrečiai su iškviestos procedūros vykdymu. Analogiškai RSE atstato kviečiamos procedūros registrus iš atminties. Tai naujas dalykas palyginus su kitomis architektūromis ir jo dėka nereikia nei programos įsikišimo nei procesoriaus resursų, valdant duomenų pašalinimą ar įdėjimą registruose [HK00].

1.7 Išlygiagretinimo savybių reziumė

Šios architektūros savybės, kurios leidžia išlygiagretinti kodą, veikia sąveikaudamos viena su kita. Pavyzdžiui, programos ciklai gali turėti užkrovimus ir saugojimus per rodykles. Duomenų prognozavimas leidžia kompiliatoriui naudoti programinio konvejerio mechanizmą pilnai perdengiamam vykdymui, net jeigu programa naudoja rodykles, kurios gali rodyti į tą pačią atminties sritį. Taip pat, planuojant užkrovimus, dažnai tenka jį suplanuoti anksčiau ankstesnėje programos stadijoje vykdomų saugojimo operacijų. Abu kontrolės ir duomenų prognozavimo mechanizmai gali būti naudojami vienu metu. Šis padidintas instrukcijų lygio lygiagretumas leidžia geriau išnaudoti procesoriaus funkcinius elementus ir žymiai sumažinti programos vykdymo laiką.

2. Kompiliatorių optimizacijos

Kompiliatorių kūrimas tapo nauju iššūkiu nuo pat skaitmeninių mašinų atsiradimo 1940-ųjų pabaigoje 1950-ųjų pradžioje. Tuo metu automatinis perkodavimas iš matematikams daug geriau suprantamų reiškinių į mašininį kodą buvo sudėtingas uždavinys. Žmogui reikėjo surasti, kaip pakeisti matematinius reiškinius į instrukcijas, kaip išsaugoti duomenis atmintyje, kaip pasirinkti instrukcijas procedūrų ir funkcijų kūrimui. Šeštame ir septintame dešimtmečiuose šie uždaviniai buvo automatizuoti iki tokio lygio, kad paprastus kompiliatorius galėjo sukurti dauguma informatikos specialistų [Mor97].

Pagal Bob Morgan kompiliatorių optimizacijos tikslas yra efektyviai išnaudoti esamus kompiuterio resursus. Kompiliatorius konvertuoja programinį kodą į mašinines instrukcijas, naudodamas skirtingus skaičiavimo elementus. Idealus atvejis yra tada, kai kiekvienas skaičiavimo elementas yra naudingai išnaudojamas kiekvieno instrukcijos vykdymo ciklo metu.

Deja, tai yra beveik neįmanomas uždavinys dėl įvairių priežasčių, pavyzdžiui, dėl nesubalansuoto poreikio skirtingiems skaičiavimo tipams (su slankiuoju kableliu ir sveikaisiais skaičiais). D. Gries programos optimizaciją vadina pertvarkymu, surištu su kompiliuojamos programos operacijų pergrupavimu ir pakeitimu, siekiant gauti efektyviau veikiančią objektinę programą [Gri71]. Geriausi kompiliatoriai, transformuojantys aukšto lygio programavimo kalbas į binarinį kodą, gali suformuoti neprastesnę programą efektyvumo, kokybės požiūriu negu kvalifikuotojo programuotojo asemblerio kalba parašyta programa, tuo pačiu sutaupant brangų laiką ir resursus.

Kompiliatorius privalo kompensuoti nesubalansuotos sistemos trūkumus. Idealiu atveju atminties ir išvedimo/įvedimo sistemos greitis turėtų sutapti su procesoriaus greičiu. Šiuolaikinėse sistemose taip nebūna, procesorius yra greitesnis už atmintį, todėl kompiliatorius turi generuoti kodą, kuris turėtų sumažinti atminties sistemos naudojimą, išlaikydamas reikalingas reikšmes registruose arba išlaikydamas duomenis spartinančiojoje atmintyje.

Kompiliatorių optimizavimas, vykdant programą, stengiasi išnaudoti visus procesoriaus ir atminties resursus kiek įmanoma efektyviau. Kompiliatorius privalo transformuoti programą taip, kad pavyktų atgauti subalansuotą skaičiavimo elementų ir atminties naudojimą. Taip pat instrukcijų pasirinkimas privalo būti teisingas ir, siekiant to balanso, panaudoti jų kiek įmanoma mažiau. Be abejo, visa tai nėra 100% įmanoma, bet kompiliatorius privalo stengtis padaryti tai, kas įmanoma.

2.1 Tradicinės kompiliatorių optimizacijos

Tradicinės kompiliatorių optimizacijos vienokia ar kitokia forma yra naudojamos Itanium ir kitų architektūrų procesoriuose, tačiau gali būti tobulinamos, panaudojant vieną ar daugiau architektūros savybių, arba efektyviau išnaudojamos, valdant optimizacijų generuojamą kodą. Dėl šios priežasties yra svarbu jas žinoti ir suprasti. Galima išskirti šiuos pagrindinius optimizacijų metodus: bendrų segmentų eliminavimą, kopijų platinimą, nereikalingo kodo pašalinimą, ciklų optimizaciją, kodo perkėlimą ir kt.

[ACУ03] knygoje nurodomi trys pagrindiniai kriterijai, kurių reikėtų laikytis gerinant kodą:

- Pirmiausia pakeitimai neturi pakeisti pradinius programos rezultatus arba priešingu atveju išvesti klaidą.
- Pakeitimai turi pagreitinti programą (nebūtinai sutrumpinti kodą).

- Įdėtos pastangos pakeitimams pasiekti turi atitikti naudą. Pavyzdžiui, jeigu programa bus panaudota tik keletą kartų, nėra prasmės gaišti laiką ir intelektualinius resursus jos optimizavimui.

[Gri71] knygoje išskiriamos dvi optimizavimo pertvarkymų kategorijos: pirmoji pertvarko programą jos vidinėje formoje, kuri nepriklauso nuo programavimo kalbos, antroji – objektinės programos lygyje. Pirmojoje kategorijoje galima išskirti šiuos keturis pagrindinius metodus:

- Kompresija ir konstantų dubliavimo metodas, t. y. operacijų, kurių operandai yra žinomi kompiliacijos metu, įvykdymas.
- Perteklinių operacijų pašalinimas.
- Invariantinių operacijų iš ciklo iškėlimas, t.y. operacijų, kurių operandai ciklo metu nesikeičia.
- Gana sudėtingų operacijų pakeitimais paprastesniais cikluose.

Kompiliatoriaus optimizacijos lygis priklauso nuo daugelio faktorių. Ne visi kompiliatoriai turi vykdyti maksimaliai pilną optimizaciją. Pavyzdžiui, reikėtų vengti sudėtingų algoritmų pritaikymą kompiliatoriuose, kurie intensyviai naudojami klaidų paieškai ir taisymui, nes optimizacijos stipriai pakeičia operacijų tvarką, taip sunkindami klaidų paiešką.

2.1.1 Kompresija

Kompresija – tai tokių operacijų įvykdymas kompiliacijos metu, kurių operandų reikšmės jau yra žinomos, todėl nėra prasmės juos vykdyti programos vykdymo metu. Vienas iš pavyzdžių galėtų būti skaičiavimų sutraukimas. Kompresija dažnai pritaikoma aritmetikos operatoriams (vietoj reiškinio 2 * 3.14 rašome 6.28.), nes jie dažnai sutinkami kode, operatorių pertvarkymams (pvz., CVIR iš Integer į Real).

Operatorių su statiniais operandais kompresija yra vykdoma paprastai ir suprantamai. Tuo tarpu kompresija operatorių, kurių reikšmės gali būti surandamos po vienokios ar kitokios analizės yra sudėtingesnė. Kompresija paprastai vykdoma nuoseklios dalies viduje. Kompresijos proceso metu lentelė T saugo poras (A, K) visiems paprastiems A kintamiesiems, kuriems žinoma einamoji K reikšmė. Be to, kiekviena suspausta triada pakeičiama nauja (C, K, 0), kur C (konstanta) – naujas operatorius, dėl kurio nereikia generuoti komandų, o K – galutinė sukompresuotos triados reikšmė. Kompresijos algoritmas nuosekliai peržiūri programos dalies triadas ir kiekvienai triadai atlieka šiuos veiksmus:

- Jeigu operandas yra kintamasis, kuris yra saugomas lentelėje T, tai operando reikšmė pakeičiama į atitinkamą K reikšmę.
- Jeigu operandas yra nuoroda į (C, K, 0) tipo triadą, tai operandas keičiamas konstanta K.
- Jeigu operandas yra konstanta ir operacija gali būti sukompresuota, tai toji triada įvykdoma ir vietoje jos pastatoma (C, K, 0) triada, kur K – gauto rezultato reikšmė.
- Jei triada yra priskyrimas A:=B be A indekso tai:
    a. Jeigu B yra konstanta, tai A su B reikšme įdedama į T lentelę (senesnė A reikšmė, jei tokia buvo, panaikinama);
    b. Jeigu B ne konstanta, tai A su savo reikšme panaikinama iš T lentelės (jei joje egzistavo).

Sudėtingų operacijų keitimas gali būti atliktas su algebrinių tapatybių metodu, pavyzdžiui:

x + 0 =0 + x = x
x – 0 = x
x*1 = 1*x = x
x / 1 = x

2.1.2 Perteklinių operacijų pašalinimas

i-toji nuoseklios programos dalies operacija laikoma pertekline, jeigu egzistuoja ankstesnė identiška j-toji operacija ir joks kintamasis, nuo kurios priklauso ši operacija, nėra pakeičiamas trečia operacija tarp j ir i. Pvz.: D:= D+C*B, A:=D+C*B, C:=D+C*B. C*B visur vienoda, todėl jų vykdymą antroje ir trečioje operacijoje galima pakeisti nuoroda į pirmąją. Tačiau, nors visur D sudedama su C*B, mes jos negalime pakeisti antrojoje operacijoje nuoroda, nes pirmoje ji pakeičia reikšmę.

Programuotojai paprastai moka programuoti be perteklinių operacijų, dėl ko, atrodytų, šios optimizacijos efektyvumas būtų abejotinas. Tačiau paprastai programavimas be operacijų ekonomijos būna lengvesnis ir padaro programą lengviau skaitomą. Be to, kai kurios perteklinės operacijos atsiranda kintamųjų indeksacijos metu ir nebėra sukontroliuojamos programuotojo.

Perteklinių operacijų pašalinimo algoritmas peržiūri operacijas pagal jų atsiradimo eilę. Jei i-toji triada identiška jau buvusiai j-jai, tai ji pakeičiama triada (SAME, j, 0), kur SAME nieko nedaro ir nereikalauja jokių operacijų generacijos metu. Kad pavyktų sekti vidinę priklausomybę tarp triadų ir kintamųjų, mes jiems atitinkamai parenkame taip vadinamus priklausomybės skaičius („dependency numbers“) pagal šias taisykles:

- Iš pradžių kintamajam A priklausomybės skaičius dep(A) lygus 0, dėl to, kad jos pradinė reikšmė nepriklauso nė nuo vienos triados.
- Po i-tosios triados pertvarkymo, kurioje kintamajam A priskiriama kažkokia reikšmė, dep(A) keičiama į i, dėl to, kad jos reikšmė priklauso nuo i-tosios triados.
- Pertvarkant i-tąją triadą jos priklausomybės skaičius dep(i) lygus 1+ (maksimalus jos operandų priklausomybės skaičius).

i-toji triada (i>j) laikoma perteklinė, jei ji identiška j-tajai triadai ir dep(i)=dep(j).

Algoritmas veikia šiuo principu:

- Jei operandas rodo į triadą (SAME, j, 0), tai ji keičiama (j).
- Išsiaiškinama dep(i)= i-tosios triados priklausomybės skaičius, kuris lygus (maksimalus jos operandų priklausomybės skaičius).
- Jeigu egzistuoja identiška j-toji triada (j<i) ir dep(i)=dep(j), tai i-toji triada perteklinė ir yra keičiama (SAME, j, 0) (Šis tikrinimas daromas su i-1, i-2, ..., 1 triadomis).
- Jeigu i-toji triada priskiria reikšmę B masyvo elementui arba paprastam B kintamajam, tai dep(B) įgyja reikšmę, lygią i.

Pateikti aukščiau algoritmai nėra optimalūs. Juos galima patobulinti, pasinaudojus kai kurių operacijų komutatyvumo dėsniu.

2.1.3 Ciklų optimizacijos

Ciklų optimizacijos yra labai svarbios, nes tai gali gerokai pagreitinti programos veikimą. Optimizuojant ciklus yra svarbos trys pagrindinės technologijos: kodo perkėlimas (code motion), kuris perkelia cikle esantį kodą iš jo, indukcinio kintamojo pašalinimas (induction-variable elimination) ir kainos mažinimas (reduction in strength), kurios daugiau kainuojančias operacijas pakeičia paprastesnėmis, pavyzdžiui, daugybą pakeičia sudėtimi.

Operacija cikle vadinsis invariantine, jeigu nė vienas operandas, nuo kurių ji priklauso, nesikeičia ciklo metu. Viena iš svarbiausių optimizacijų susideda iš to, kad pavyktų iškelti invariantines operacijas iš ciklo vidaus. Pavyzdžiui, jei 1000 kartų kartojamame cikle esančią daugybą, iškelsime iš ciklo, sutaupysime 999 operacijas.

Kainos mažinimo ir indukcijos kintamojo pašalinimo metodai susideda iš operacijų pakeitimų į greitesnes operacijas ir indukcinių kintamųjų pašalinimo iš ciklo. Pavyzdžiui, šių optimizacijų metu žemiau pateiktame cikle svarbiausias tikslas pakeisti būtų pakeisti daugybą I*K į sudėtį, kur I yra indukcinis ciklo kintamasis:

```
FOR I:=A STEP B UNTIL C DO
BEGIN
... T1 := I*K...
END
```

K yra invariantinė. Pradinė I reikšmė lygi A. Ciklo viduje I keičiasi rekursiškai ir kiekvieną kartą pridedama ar atimama žingsnio B reikšme. Kiekvieną kartą, kai I reikšmė pasikeičia, reikšmė I*K padidėja ar sumažėja B*K reikšme. Tai yra (I+B)*K = I*K+B*K. Taigi, jeigu T1 niekur daugiau cikle nesikeičia, mes galime pakeisti sudėtingą I*K operaciją, tokiu būdu pakeičiant ciklo užduotis:

- Prieš ciklą įstatomos operacijos T1 := A*K; T2 :=B*K, kur T2 – naujas laikinas kintamasis. Tai pradinė T1 reikšmė ir apskaičiuojamas padidėjimas B*K.
- Iš ciklo išmetamas T1 := I*K.
- Ciklo pabaigoje pridedamas T1 := T1+T2.

Rezultate gaunamas toks ciklas:

```
T1:= A*K; T2:=B*K;
FOR I := A STEP B UNTIL C DO
BEGIN
... ... T1 := T1 + T2;
END
```

Sudėtingų operacijų pakeitimas paprastai visada sukuria efektyvesnę programą. Tai nepritaikoma tik ciklams, susidedantiems iš keleto dalių, tarp kurių tik nedaugelis įvykdomi kiekvieno ciklo žingsnio metu.

2.2 IA-64 kompiliatorių optimizacijos

Dėl to, kad IA-64 architektūra netapo tokia populiari, kokia tikėtasi, nėra labai daug resursų, aprašančių optimizacijas šiai architektūrai, tačiau darbai buvo vykdomi ir dalį jų pabandžiau apžvelgti šioje analizėje.

Pagrindiniai IA-64 architektūros kompiliatoriaus tikslai yra padidinti atminties naudojimo efektyvumą, minimizuoti atšakų skaičių ir padidinti instrukcijų lygio lygiagretumą [DKK+99]. Optimizacijos IA-64 kompiliatoriuje gali būti sugrupuotos į aukšto lygio optimizacijas (į kurias įeina atminties optimizacijos, vektorizacija, paralelizacija), skaliarines optimizacijas ir planavimą bei kodo generavimą, kurie visi kartu pasieka anksčiau paminėtus tikslus.

Aukšto lygio optimizacijos, transformuodamos duomenis, pagerina atminties pasiekiamumą, atskleidžia blogai išlygiagretintas programos vietas, vektorizuoja ir išnaudoja aukštesnio lygio instrukcijų lygiagretumą.

Skaliarinės optimizacijos sumažina skaičiavimų ir nuorodų į atmintį skaičių. Šiam tikslui naudojamas patobulintas PRE (partial redundancy elimination) algoritmas, kuris minimizuoja reiškinių įvertinimo skaičių. IA-64 patobulinimai apima ne tik skaičiavimus, bet ir tos pačios reikšmės užkrovimų eliminavimą, taip pat išnaudoja duomenų ir valdymo prognozavimą.

Planavimas ir kodo generavimas efektyviau išnaudoja predikacijos, prognozavimo ir registrų rotacijos savybes (sąlygos sakinių pertvarkymo, globalaus kodo planavimo, programinio konvejerio ir rotacinių registrų alokacijos mechanizmais).

2.2.1 Profilio informacija (PROFILE-GUIDED) valdomos optimizacijos

Profilio informacija susideda iš kiekvieno bazinio bloko dažnumo ir atšakų programoje tikimybės. Su šia informacija kompiliatorius gali geriau išnaudoti IA-64 architektūros galimybes. Profilio informacija yra surenkama dviem būdais: statiniu ir dinaminiu. Statiniu būdu kompiliatorius naudoja euristikas, bandydamas įvertinti dažnumą ir tikimybę. Ši informacija yra apytikslė. Dinaminiu būdu, programa yra sukompiliuojama, tada paleidžiama vieną ar kelis kartus informacijos surinkimui ir vėl sukompiliuojama. Profilio informacija yra naudojama siekiant integruoti dažniausiai paleidžiamas procedūras, taip pat surikiuoti procedūras ir blokus procedūrose taip, kad sumažintų instrukcijų spartinančiosios atminties klaidas ir išnaudotų prognozavimo bei planavimo savybes.

2.2.2 Tarpprocedūrinė analizė ir optimizavimas

Kadangi IA-64 architektūra sugeba apdoroti daug instrukcijų per vieną procesoriaus taktą, šioje architektūroje tapo svarbu išnaudoti tarpprocedūrinę analizę ir optimizaciją. Tradiciniai kompiliatoriai vienu momentu operuoja viena programos procedūra, tuo tarpu IA-64 stengiasi plačiai išnaudoti tarprocedūrinę analizę ir optimizavimą. Pagrindiniai uždaviniai yra rodyklių (points-to), mod/ref analizė, pašalinio poveikio ir konstantų paieška. Rodyklių analizės tikslas tiksliai nustatyti, kurios atminties vietos gali būti nurodomos didelėje kodo srityje (pavyzdžiui, jei į R33 jau buvo vieną kartą įkrauta reikšmė, kurios prireiks kitoje kodo dalyje, tai galima išsaugoti vieną pakrovimą į registrą iš atminties). Be to, dėl IA-64 duomenų prognozavimo savybės, yra galimybė eliminuoti pakrovimą, jeigu kreipimasis į registrą nedažnai konfliktuoja. Panašiai, atminties nuorodų perkėlimas reikalauja žinojimo, kas yra pakeista ar nurodoma funkcijos iškvietimu. Tai suteikia mod/ref analizė.

2.2.3 Ciklų transformacijos

Vienos iš aukšto lygio optimizacijų IA-64 architektūroje yra įvairios ciklų optimizacijos. Šie optimizuojantys pertvarkymai leidžia pagerinti spartinančiosios atminties lokalumą ir geriau

išlygiagretinti kodą. Tiesinės ciklų transformacijos, ciklų suliejimas, ciklo dengimas (tiling), ciklų išskirstymas gali pagerinti masyvo nuorodų spartinančiojoje atmintyje lokalumą. Ciklo išvyniojimas ir sutraukimas (unroll and jam) ir ciklo išskleidimas išnaudoja didelį registrų failą, eliminuojant perteklines nuorodas į masyvo elementus ir pateikia labiau išlygiagretintą kodą planuotojui ir kodo generatoriui [DKK+99]. Šios optimizacijos nėra tiesiogiai priklausomos nuo architektūros, tačiau Intel Itanium sistemose gali būti vykdomos efektyviau arba be savo pagrindinių funkcijų atstovauja kaip pagalbinės optimizacijos kai kurioms nuo architektūros priklausančioms optimizacijoms.

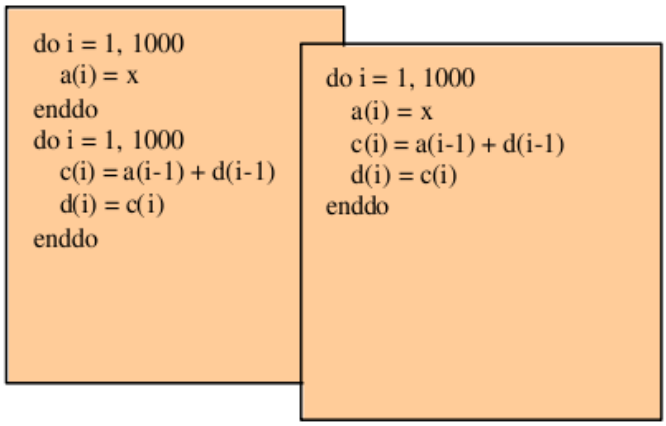
2.2.3.1 Tiesinės ciklų transformacijos (Linear loop transformation)

Tiesinės ciklų transformacijos susideda iš rinkinio kitų transformacijų: ciklų apvertimas (reversal), ciklų apsikeitimas (interchange), ciklų fazinis postūmis (skew) ir ciklų suspaudimas (scaling). Ciklų apvertimas perstato ciklo iteracijų vykdymo eiliškumą, tuo tarpu ciklų apsikeitimas apsikeičia ciklo lygio eiliškumu cikluose cikle. Ciklo fazinis postūmis pakeičia ciklo iteracijų vietos formą pagal kompiliatoriaus apibrėžtą faktorių. Ciklo išvyniojimas modifikuoja ciklą taip, kad jis turėtų ne vienetinius žingsnius. Bendra šių tiesinių ciklo transformacijų kombinacija gali gerokai pagerinti atminties pasiekiamumo lokalumą. Jie taip pat gali padidinti kitų optimizacijų, pavyzdžiui, skaliarinio apkeitimo, pastovaus kodo cikluose šalinimo, programinio konvejerio efektyvumą.

2.2.3.2 Ciklų suliejimas (Loop fusion)

Ciklų suliejimas sudeda gretimus atitinkančius ciklus į vieną naują ciklą [DKK+99]. Ciklų suliejimas yra efektyvus, siekiant pagerinti spartinančiosios atminties veikimą, nes jis sujungia skirtingų ciklų spartinančiosios atminties kontekstus į vieną naują ciklą. Tokiu būdu, duomenų pakartotinis panaudojimas tarp ciklų yra vykdomas viename cikle. Tai taip pat padidina galimybę sumažinti papildomą darbą masyvo nuorodoms, jas pakeičiant kompiliatoriaus sugeneruotais skaliariniais kintamaisiais. Ciklų suliejimas taip pat pagerina duomenų išankstinio užkrovimo (data prefetching) efektyvumą. Ši optimizacija Intel IA-64 kompiliatoriuje yra efektyviau išnaudojama nei IA-32 ar RISC kompiliatoriuose, pavyzdžiui, IA-64 architektūroje ciklų suliejimas išnaudoja didelį IA-64 architektūros registrų skaičių.

Pav. 4 paveikslėlio dešinėje pusėje matyti, kad spartinančiosios atminties lokalumas yra pagerintas, nes masyvas a yra du kartus naudojamas viename cikle. Taip pat tai įgalina kompiliatorių pakeisti nuorodas į a ir d masyvus į kompiliatoriaus sugeneruotas skaliarines nuorodas.



Pav. 4 [Dul98]

2.2.3.3 Ciklų blokavimas-išvyniojimas-suspaudimas (Loop block-unroll-jam)

Ciklų išvyniojimas ir suspaudimas išvynioja išorinius ciklus ir suklijuoja juos kartu. Po šių pakeitimų keletas išorinio ciklo iteracijų yra sujungiamos į vieną iteraciją naujame cikle. Pavyzdyje žemiau matyti, kad dviejų lygių cikle esantis išorinis ciklas yra išvyniojamas faktoriumi lygiu dviem (šis faktorius nurodo, kiek kartų ciklas buvo išvyniojamas). Tada du ciklai yra sujungiami į vieną naują vidinį ciklą.

```
do i = 1, 2*n
  do j = 1, 2*n
    b(j, i) = a(j, i-1) + a(j, i) + a(j, i+1)
  enddo
enddo

do i = 1, 2*n,2
  do j = 1, 2*n
    b(j, i) = a(j, i-1) + a(j, i) + a(j, i+1)
    b(j, i+1) = a(j, i) + a(j, i+1) + a(j, i+2)
  enddo
enddo
```

Kai visi cikle esantys vidiniai ciklai yra blokuoti, ciklų blokavimas ir dengimas (tiling) transformuoja išorinį n-laipsnio ciklą į 2-n laipsnio ciklą, kuriame vidiniai n-ciklai praeina iteracijas pradinio bloko arba perdengtos iteracijos erdvėje. Ciklo blokavimas yra pagrindinis žingsnis optimizuojant spartinančiosios atminties darbą su programomis ar bibliotekomis, kurios operuoja didelėmis duomenų matricomis.

Intel IA-64 kompiliatorius, priešingai nuo tradicinių kompiliatorių, ciklų blokavimo, išvyniojimo ir suspaudimo bei vidinio ciklo išvyniojimo optimizacijas sujungia į vieną bendrą

optimizaciją. Tradiciniai kompiliatoriai realizuoja šias optimizacijas atskirai. Tokiu atveju tokie kompiliatoriai turi naudoti daugiau nei vieną kodo generavimo mechanizmą ir optimizacijos kainos įvertinimo modelį. Iš tikrųjų šios optimizacijos viena su kita susijusios. Visų jų tikslas sukelti kuo daugiau susijusių kreipimųsi į masyvą ir skaičiavimus į vidinius ciklus.

Taip pat ciklo išvyniojimo optimizacija IA-64 architektūroje yra efektyvesnė, jei yra labiau išnaudojama nei tradicinėse architektūrose. Taip buvo įrodyta keičiant šios optimizacijos valdomą GCC kompiliatoriaus parametrą MAX_UNROLLED_INSNS [SXC05], nurodantį kiek maksimaliai instrukcijų gali būti išvyniotame cikle. Ciklo išvyniojimas GCC kompiliatoriuje veikia tokiu principu: tarkime, kad LOOP_CNT yra iteracijų skaičius cikle. NUM_INSNS yra instrukcijų skaičius cikle. UNROLL_FACTOR – ciklo išvyniojimo kiekis. UNROLL_FACTOR yra pasirenkamas taip, kad visada atitiktų šią sąlygą: NUM_INSNS x UNROLL_FACTOR < MAX_UNROLLED_INSNS. Tokiu atveju, kai gali būti paskaičiuotas tikslus LOOP_CNT skaičius (pvz.: kompiliavimo metu), ciklas bus pilnai išvyniotas su šia sąlyga: NUM_INSNS x LOOP_CNT < MAX_UNROLLED_INSNS. Kitu atveju UNROLL_FACTOR nustatomas kaip didžiausias dalomas LOOP_CNT faktorius, taip, kad pirmoji nelygybė liktų teisinga.

Padidinus MAX_UNROLLED_INSNS parametrą nuo standartinės reikšmės lygios 200 iki 600, ši optimizacija pasidarė efektyvesnė. Išvyniojant daugiau iteracijų, cikle padaugėja lygiagrečiai vykdomų instrukcijų. Dėl šios priežasties sumažėjo stovinčių taktų skaičius bei skaičius taktų, kurie atliekami slankaus kablelio procesoriuje. Tačiau tokiu atveju išauga registrų spaudimas ir kodo kiekis. Kadangi IA-64 architektūra turi daugiau registrų resursų, ji sugeba susitvarkyti su didesniu registrų spaudimu geriau nei tradicinės architektūros.

2.2.3.4 Ciklų padalijimas (Loop distribution)

Ciklų padalijimo rezultatas yra priešingas ciklo suliejimui. Šis metodas atskiria vieną ciklą į kelis naujus ciklus, kurie turi panašią struktūrą. Skaičiavimai ir masyvo pasiekiamumas pradiniame cikle yra paskirstomi tarp naujai susiformavusių ciklų. Įgalindamas kitas optimizacijas, ciklų išskirstymas išplatina potencialiai didelį spartinančiosios atminties kontekstą pradiniame cikle į skirtingus naujus ciklus taip, kad nauji ciklai turėtų lengviau valdomus spartinančiosios atminties kontekstus ir dažnesnius spartinančiosios atminties pasiekiamumo lygius ir lokalizaciją. Taip pat tai leidžia sukurti ciklus su mažesnėmis duomenų priklausomybėmis.

2.2.4 Įkrovimo ir laikymo eliminavimas

Kadangi IA-64 turi gerokai daugiau registrų nei tradicinės architektūros, tai galima išnaudoti eliminuojant įkrovimus ir saugojimą. Aprašysiu dvi technikas: skaliarinis pakeitimas ir registrų blokavimas.

Skaliarinis pakeitimas – optimizacija, kuri pakeičia nuorodas į atmintį kompiliatoriaus sugeneruotais laikinais skaliariniais kintamaisiais, kurie yra patalpinami į registrus. Dauguma back-end optimizacijos technikų patalpina masyvo nuorodas į registrus tada, kai duomenys yra nepriklausomi nuo ciklų iteracijų. Tačiau back-end optimizacijos neturi tikslios informacijos apie priklausomybes, kurios reikia siekiant pakeisti nuorodas į atmintį, priklausomas nuo ciklo nešamos informacijos, skaliariniais kintamaisiais. Tuo tarpu skaliarinis pakeitimas Intel IA-64 kompiliatoriuje taip pat pakeičia ir nuo ciklo nepriklausomas nuorodas į atmintį skaliariniais kintamaisiais atitinkamuose ciklo lygiuose.

Skaliarinis atminties nuorodų apkeitimo pavyzdžiu imkime pirmąjį ciklą. Transformuojame cikle visos skaitomos nuorodos į masyvą a yra pakeistos kompiliatoriaus įdėtais laikinais skaliariniais kintamaisiais.

```
do i=2,n
a(i)=a(i-1)*...
...=a(i)-a(i-1)
enddo


t(1)=a(1)
do i=2,n
t2=t1*...
a(i)=t2
=t2-t1
t1=t2
enddo
```

IA-64 architektūra suteikia rotuojančius registrus, kurie yra kiekvieną kartą pakeičiami viena registro pozicija į priekį, kai yra įvykdoma speciali ciklo atšakos instrukcija. Ši architektūrinė savybė leidžia kompiliatoriui priskirti kompiliatoriaus įterptus skaliarus tiesiogiai į rotuojančius registrus.

Atminties nuorodų skaliarinis pakeitimas naudoja priklausomus duomenų grafus ir rodyklinius vektorius, siekiant surasti atminties nuorodas, kurias galima pakeisti į skaliarus. Kompiliatorius egzaminuoja kiekvieno ciklo iteracijos duomenų grafą, priklausomai nuo to ar tai išvesties, esami ar įvesties duomenys. Atminties nuorodos yra išdėstomos pagal

priklausomybių atstumą ir topologiją. Pirmiausia apdorojamos nuo ciklo nepriklausančios atminties nuorodos ir ciklo nešamos esamos informacijos priklausomybės, vėliau ciklo nešamos išvesties informacijos priklausomybės.

Registrų blokavimas paverčia nuo ciklų priklausomus duomenis į nuo ciklų nepriklausomus duomenis. Registrų blokavimas transformuoja ciklą į naują ciklą, kur kūnas susideda iš kelių gretimų ciklų iteracijų.

2.2.5 RSE srauto sumažinimas

Kiekviena procedūra Itanium architektūroje gali turėti savo varijuojančio dydžio iki 96 registrų steko erdvę. Kiekvienas toks registras procedūroje yra vadinamas architektūriniu registru. Aparatinė įranga susieja juos su fiziniais architektūros registrais nuo r32 iki r128. Su alloc instrukcija, kodo generatorius aiškiai apibrėžia procedūros registro erdvę: 1) įeinančių parametrų skaičius, 2) skaičius procedūros viduje naudojamų lokalių registrų ir 3) skaičius išeinančių parametrų. Šių skaičių suma vienai procedūrai gali būti lygi 96. Ši registrų erdvė yra valdoma procesoriaus registrų steko varikliu (RSE) [HKS+04].

Itanium architektūra leidžia optimizacijas, kurios sumažina registrų steką prieš funkcijos iškvietimą ir atstato steką į pradinę būseną po jo. Tai gali sumažinti bendrą registrų naudojimą ir RSE srautą. Gyvybingumo analizatorius (liviness analysis) nustato, kurie iš rezervuotų registrų yra nebenaudojami. Jei nebenaudojamų priskirtų registrų skaičius viršija norimų rezervuoti registrų skaičių, registrų stekas yra sumažinamas šių nenaudojamų registrų skaičiumi.

Dvi funkcijos, iš kurių viena kviečiama iš pirmosios, bandys rezervuot 140 registrų, tai viršys registrų steką 44 registrais, kas sąlygos papildomą RSE srautą, nes jis turės perkrauti dalį registrų.

Pateikiamas būdas to išvengti yra papildoma registrų gyvybingumo analizė prieš antros funkcijos kvietimą (t.y. papildomos alloc komandos pridėjimas). Jei pirmoji funkcija tuo metu turės nenaudojamų registrų, tai leis registro steko varikliui sumažinti registro steko erdvę ir ją atlaisvinti kviečiamai antrai funkcijai. Pasibaigus antros funkcijos vykdymui, registro stekas vėl bus atstatomas iki pradinio dydžio, kuris buvo rezervuotas pirmajai funkcijai. Duotame kodo pavyzdyje:

```
foo():
alloc rx=0,90,0
1:
        bar():
call bar()              alloc ry=0,50,0
2:
…
```

```
alloc rz=0,90,0 _return
```

Funkcija *foo* rezervuoja 90 stekinių registrų, o joje esanti funkcija *bar* vėliau rezervuoja dar 50. Galutiniam variante abi funkcijos naudoja 140 registrų, o tai 44 registrais viršija registrų steką. Dėl to bus reikalingas papildomas RSE darbas.

```
foo():
alloc rx=0,90,0
1:
…
alloc rz=0,30,0
        2: bar():
call bar() _            alloc ry=0,50,0
        3:…
alloc rz=0,90,0         return
```

Papildoma alloc instrukcija prieš bar() funkciją bando sumažinti registro erdvę iki 30. Jei gyvybingumo analizatorius nustato, kad tuo metu yra 60 nebenaudojamų registrų, RSE tai ir padaro. Tada naujai funkcijai rezervavus dar 50 registrų, registrų stekas naudoja iš viso tik 80. Galiausiai, grįžus iš bar funkcijos, pakeičiamos steko ribos atgal iki 90 registrų. Kombinacija abiejų funkcijų iš viso panaudos tik 90 registrų, kaip ir viena *foo* funkcija.

2.2.6 Skaliarinės optimizacijos

Pagrindiniai skaliarinių optimizacijų uždaviniai yra sumažinti skaičiavimų ir nuorodų į atmintį sumažinimas. Dalinė dubliavimų eliminacija (PRE – partial redundancy elimination) yra gerai žinoma skaliarinė optimizacija [Dul98].

2.2.6.1 Tradicinis PRE

Reiškinys programos taške p bus pilnai besidubliuojantis, jeigu programos valdymo tėkmės grafike (control flow graphic) toks reiškinys jau egzistuoja. Reiškinys e egzistuoja taške p, jeigu kiekviename tėkmės kelyje nuo programos pirminio taško iki to taško p egzistuoja reiškinys e, kurio reikšmė nepasikeičia iš naujo apibrėžiant jo operandų reikšmes. Dubliavimasis yra panaikinamas patalpinant reiškinio rezultatą į laikiną kintamąjį ir vėliau panaudojant jį vietoje reiškinio perskaičiavimo.

Reiškinys e yra dalinai egzistuojantis taške p, jeigu jis egzistuoja tik kai kuriose tėkmės keliuose nuo programos pirminio taško iki p. Dalinis dubliavimasis panaikinamas įtraukiant reiškinio kopiją tuose valdymo tėkmės keliuose, kur jo nebuvo. Tokiu būdu reiškinys tampa pilnai besidubliuojančiu.

PRE optimizacija leidžia perkelti nuo ciklo nepriklausomą kodą virš ciklo. Visgi optimizuojantis kompiliatorius turėtų būti atsargus, prieš įdėdamas reiškinio kopiją ten, kur jis

nebuvo apskaičiuotas pradiniame kode. Reiškinio e įkėlimas į programos tašką p yra saugus, jeigu nukopijuotas reiškinys kiekviename tėkmės kelyje nuo taško p iki programos pabaigos turės tokią pačią reikšmę kaip ir pradinis reiškinys e. Nesaugaus kodo įdėjimas gali sukelti dvi problemas: 1) programa bus vykdoma klaidingai 2) viename programos kelyje instrukcijų skaičius bus sumažintas, tačiau kitame jų padaugės.

2.2.6.2 Praplėstas IA-64 PRE

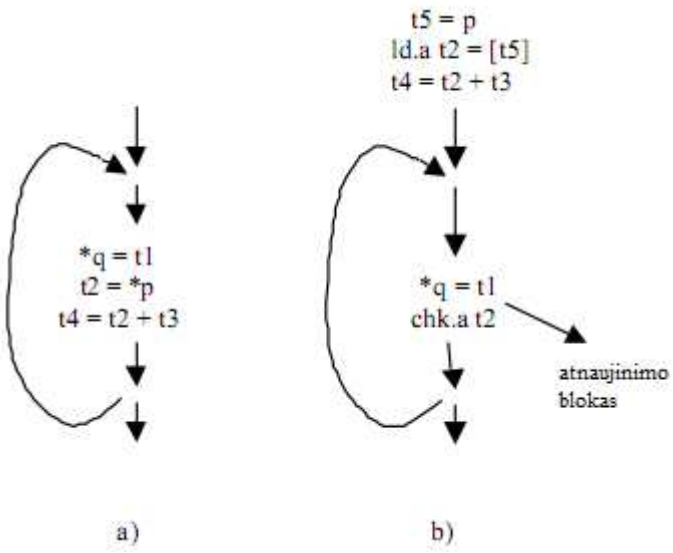
Įprastas PRE algoritmas pašalina perteklines operacijas tik tada, kai jas šalinti yra saugu. IA-64 architektūroje PRE algoritmas buvo praplėstas, panaudojant valdymo prognozavimą [Dul98]. Tokiu atveju galima panaikinti besidubliuojančias dalis viename valdymo tėkmės kelyje, galbūt kito, mažiau svarbaus kelio kaina. Jei galima pašalinti besidubliuojančią dalį viename iš kelių, išvengdami nekorektiškai veikiančios programos, bendras kodo vykdymo našumas gali būti pagerintas, net jei bus vykdoma papildoma instrukcija kitame tėkmės kelyje.

Tose vietose, kur yra nesaugu įterpti besidubliuojanti užkrovimą, panaudojama valdymo prognozavimo savybė (1.4.1), o vietoje pradinio užkrovimo įdedama chk.s patikrinimo instrukcija. Tikrinimo instrukcijos vykdymas yra efektyvesnis negu besidubliuojantis užkrovimas, nes tikrinimas nereikalauja atminties resursų ir užkrovimo gaišties laikas yra paslepiamas, vykdant užkrovimą anksčiau. Taip pat, besidubliuojančio užkrovimo pašalinimas gali pagelbėti eliminuojant kitas besidubliuojančias instrukcijas, kurios yra priklausomos nuo šio užkrovimo.

Perteklinių užkrovimų eliminavimui kartais gali sutrukdyti įsiterpiančios išsaugojimo operacijos. Nuo ciklo nepriklausantis užkrovimas *p negalės būti pašalintas tol, kol kompiliatorius negalės įrodyti, kad *q saugojimo operacija nebando pasiekti tos pačios atminties vietos. Atpažinimo procesas, kuriuo bandoma nustatyti ar dvi nuorodos į atmintį nerodo į tą pačią atminties sritį, vadinamas atminties dviprasmiškumo pašalinimas.

Jei kompiliatorius gali rasti, kad yra maža, nors iki galo nežinoma tikimybė, jog *p ir *q bando pasiekti tą pačią atminties sritį, nuo ciklo nepriklausantis užkrovimas ir nuo jo priklausanti sudėties operacija, gali būti pašalinta, naudojant IA-64 duomenų prognozavimo savybę (Pav. 5). *p operacija yra įterpiama prieš ciklą, naudojant išankstinį užkrovimą, o vietoje pradinio besidubliuojančio užkrovimo, yra įdedama duomenų prognozavimo patikrinimo instrukcija chk.a. Jei *q bandys pasiekti tą pačią atminties sritį kaip ir *p, bus vykdomas atstatymo bloko kodas. Atstatymo blokas susideda iš *p perkrovimo ir iš naujo įvykdytos

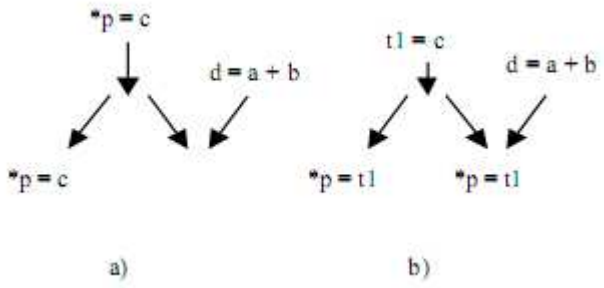
sudėties operacijos. Priešingu atveju, jei *q ir *p rodys į skirtingas atminties sritis, tai bus įvykdyta tik patikrinimo operacija vietoje besidubliuojančios užkrovimo ir sudėties operacijos.



Pav. 5 [Dul98]

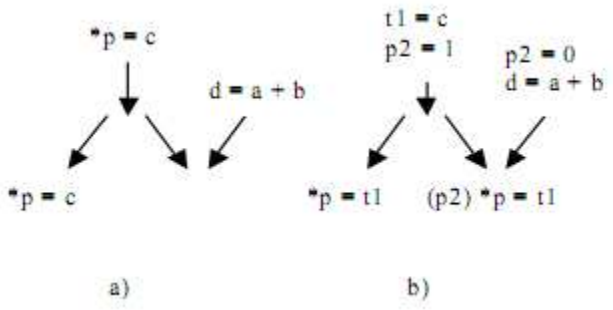
2.2.6.3 Dalinė nepasiekiamo saugojimo eliminacija (Partial dead store elimination)

PDSE optimizacija nuo PRE skiriasi tuo, kad vietoje perteklinių užkrovimų pašalina perteklinius saugojimus. Dalinis dubliavimasis yra pašalinamas, įdedant dalinai nepasiekiamą saugojimą į tą programos šaką, kur jo nebuvo. Kaip ir su PRE, taip ir su šia optimizacija, kompiliatorius privalo būti atsargus tam, kad neįtrauktų saugojimo operacijos ten, kur ji negali būti įvykdyta. Pav. 6 matyti, kad joks *p saugojimas nėra vykdomas, kai yra vykdoma d=a+b operacija.



Pav. 6 [Dul98]

Šios problemos sprendimui yra naudojamas predikatas. Instrukcijos turi nustatyti predikato registrą 1, kai saugojimo operaciją reikia įvykdyti, ir 0, kai nereikia. Ši realizacija matoma septintame paveikslėlyje (Pav. 7). Daroma prielaida, kad kairinis programos kelias vykdomas daug dažniau nei dešininis. Predikato reikšmės tikrinimo operacijos vykdymas yra efektyvesnis negu saugojimo operacija, nes jis nenaudoja atminties resursų. Kadangi dešinė kelio dalis vykdoma rečiau, papildoma predikato priskyrimo nuliui instrukcija beveik nesumažina programos našumo.



Pav. 7 [Dul98]

2.2.7 Planavimas ir kodo generavimas

Planuojant ir generuojant kodą yra naudojamos globalaus kodo planavimo, programinio konvejerio ir rotacinių registrų alokacijos technikos, kuriuose išnaudojama predikacija, prognozavimas ir rotaciniai registrai.

2.2.7.1 Predikacija

Predikacija yra naudojama atšakų efektyvumui padidinti. Kaip jau minėta architektūrinėje dalyje (1.3), IA-64 predikacijai naudoja predikatų registrus sąlygos sakinių pašalinimui. Siekiant geriau įvertinti atšakos kainą, pavyzdžiui, jos neteisingo spėjimo galimybę, kompiliatorius gali pasinaudoti dinamine profilio informacija (2.2.1). Priešingu atveju, jis fokusuojasi tik prie procesoriaus resursų išnaudojimo ir kritinių vietų suderinamumo.

2.2.7.2 Globalaus kodo planavimas

Globalaus kodo planuotojas planuoja kodą necikliniuose kodo srauto regionuose. Lokalaus kodo planuotojas pertvarko kodą baziniuose blokuose ir yra paleidžiamas po registrų alokacijos išbarstyto kodo suplanavimui.

Daugumai planavimo algoritmų yra sudėtinga surasti teisingus sprendimus generuojant ir planuojant kodą. IA-64 kompiliatoriuje naudojami *wavefront* planavimas ir atidėtas

kompensavimas (*deferred compensation*). *Wavefront* planavimas padalina kodą į nepriklausomų blokų rinkinį, kuriuose instrukcijos jau yra suplanuotos. *Wavefront* gali būti įsivaizduojamas kaip riba tarp suplanuoto ir nesuplanuoto kodo. Jeigu planavimas eina iš viršaus į apačią, mazgai virš *wavefront* yra jau suplanuoti ir jų suplanavimas jau nebus pakeistas. Mazgai žemiau neturi suplanuoto kodo. Kandidatai suplanavimui į bloką susideda iš nesuplanuotų instrukcijų su gatavais duomenimis, esančių tame pačiame ar kituose blokuose virš ar po *wavefront* riba. Kai kodo suplanavimas į bloką yra pabaigiamas, planuotojas paskelbia jį uždarytu ir perkelia ribą iki bloko pabaigos. Visos nesuplanuotos instrukcijos yra nuleidžiamos žemiau šios ribos.

2.2.8 Space Exploration

Vienas iš būdų IA-64 architektūros kompiliatorių optimizavimui yra *Space Exploration* technika [TVV+03].

Kompiliatorius, realizuojantis Space Exploration optimizavimo (OSE) techniką, optimizuoja kiekvieną kodo segmentą su įvairiomis optimizacijų konfigūracijomis ir tada egzaminuoją gautą kodą, pasirinkdamas geriausią variantą.

OSE kompiliatorius kiekviename kodo segmente vienu metu vykdo keletą transformacijos šakų. Kiekviena versija yra optimizuojama naudojant skirtingas optimizavimo konfigūracijas. Kompiliatorius išskiria labiausiai tinkamą versiją, nustatydamas ją efektyvumo vertintoju.

Idealiu atveju, kompiliatorius turėtų peržiūrėti visus įmanomus konfigūracijų variantus kiekvienam kodo segmentui ir pasirinkti labiausiai tinkančią programą išvedimui. Deja, tokio varianto neįmanoma realizuoti. Šios metodikos tikslas greitai pasirinkti kurias optimizacijų konfigūracijas pasirinkti kompiliacijos metu, greitai išrinkti geriausią optimizuotą kodą ir nustatyti kaip kuriems kodo segmentams pritaikyti OSE.

Konfigūracijų apribojimas pirmiausia yra vykdomas nuo tų konfigūracijų atmetimo, kurios greičiausiai nepadės optimizuojamam kodui, kurios veikia blogiau nei nustatymai pagal nutylėjimą ir kurios labai viena į kitą panašios. Toliau mažinamas konfigūracijų skaičius kiekvienam kodo segmentui.

Atrinkus konfigūracijas ir atlikus optimizacijas, toliau turi būti atliekami efektyvūs gautų optimizuotų kodų palyginimai. Straipsnyje aprašoma, kad idealiu atveju kompiliatorius sukompiliuotų visą programą visais įmanomais būdais ir paleistų kiekvieną programos versiją. Tačiau buvo siekiama išlaikyti kompiliavimo laiką priimtinu, todėl OSE kompiliatorius turi rasti geriausią kodą kodo segmente, ignoruodamas kodo ir segmento tarpusavio sąveiką. OSE

kompiliatorius tai atlieka įvertindamas kodo segmento efektyvumą, naudojant mašininį modelį ir profilio duomenis.

Testai parodė, kad ši metodika sugebėjo žymiai pagerinti programų efektyvumą išlaikant priimtiną kompiliavimo laiką.

3. Analizės rezultatai ir darbo tyrimas

Intel Itanium architektūra išsiskiria savo savybėmis nuo daugumos kitų architektūrų tuo, kad didžiąją optimizuojančių pertvarkymų dalį prieš kodo vykdymą turi atlikti kompiliatorius. Architektūrinės savybės tik suteikia kompiliatoriui platesnes galimybes tą kodą pertvarkyti. Pirmiausia Itanium optimizacijos fokusuojasi prie kuo didesnio vykdymo išlygiagretinimo, aiškiai tai pažymint procesoriui. Antra optimizacijų sritis yra kuo efektyvesnis didelio registrų kiekio ir kitokios aparatinės įrangos išnaudojimas: spartinančiosios atminties optimizacijos, registrų valdymas. IA-64 architektūroje gali būti efektyviau išnaudojamos tradicinės optimizacijos, kurios nepriklauso nuo architektūros, pavyzdžiui ciklų optimizacijos. Taip pat gali būti patobulintos ir nuo architektūros priklausomos optimizacijos, panaudojant Itanium architektūros kompiliatoriui teikiamas aparatines galimybes. Tai liečia programinio konvejerio, kodo planavimo optimizacijas, kurias galima patobulinti, panaudojant prognozuojamą užkrovimą, predikatus ar rotuojančius bei specializuotus registrus.

Iki šiol buvo detaliai išanalizuotas Intel Itanium architektūros savybių rinkinys, nuo kurių priklauso kompiliatoriaus generuojamas kodas, ir optimizacijos, kurios priklauso nuo architektūros ar gali būti efektyviau išnaudojamos Itanium sistemoje. IA-64 architektūra daugiausia remiasi kompiliatoriaus gebėjimu optimizuoti kodą, o ne savo galimybėmis pertvarkyti instrukcijas vykdymo metu. Dėl šios priežasties buvo įgyvendintos savybės, kurios kompiliatoriui leidžia pertvarkyti kodą taip, kad jis būtų vykdomas lygiagrečiai ar išnaudotų didelius architektūros resursus. Pagrindinės tam skirtos architektūros savybės yra predikacija, duomenų bei kontrolės prognozavimas ir registrų modelis.

Predikatų registrai svarbūs naikinant sąlygos atšakas ir programiniame konvejeryje valdant epilogo fazę. Tam architektūroje realizuoti 64 1 bito predikatų registrai, o kompiliatorius prie kiekvienos instrukcijos turi 6 bitų erdvę predikato žymei pažymėti. Valdymo prognozavimas leidžia perkelti užkrovimo instrukcijas virš tų sąlygos sakinių, kurias nepavyko panaikinti predikatais. Tuo tarpu duomenų prognozavimas perkelia užkrovimo instrukcijas virš saugojimo instrukcijų, net jeigu nežinoma, ar jie nerodys į tą pačią atminties sritį. Šios dvi funkcijos realizuojamos kompiliatoriui perkėlus užkrovimą į ankstesnę programos stadiją, o pradinėj jos

vietoj pažymėjus patikrinimo instrukcijas, kurios įvykdomos daug greičiau. Jeigu išankstinis užkrovimas buvo nesėkmingas, vykdomas kompiliatoriaus sugeneruotas atstatymo blokas.

Tuo tarpu optimizacijas galima susiskirstyti į tris tipus:

- Specifinės Itanium arcihtektūrai nuo jos savybių priklausomos optimizacijos (sąlygos sakinių eliminavimas predikatais).
- Patobulintos tradicinės optimizacijos, kurios buvo modifikuotos taip, kad išnaudotų vieną ar daugiau iš Intel Itanium unikalių architektūrinių savybių (praplėstos perteklinių užkrovimų eliminavimo bei dalinės nepasiekiamo saugojimo eliminacijos realizacijos).
- Optimizacijos, kurių efektyvumą galima pagerinti palyginus su tradicinėmis architektūromis, parinkus kitokius tą optimizaciją valdančius kompiliavimo parametrus. Paprastai šie parametrai leidžia sugeneruoti daugiau lygiagrečiai vykdomo kodo ir dėl didesnių Itanium resursų jis yra vykdomas efektyviau nei su įprastai naudojamais parametrais tradicinėse architektūrose (ciklų suliejimas, ciklų išvyniojimas).

Pirmo tipo optimizacijos yra tiesiogiai priklausomos nuo predikacijos, duomenų ir kontrolės prognozavimo bei Itanium registrų modelio. Pagrindinis šių optimizacijų tobulinimo metodas yra kompiliatoriaus kodo planavimo, registrų priskyrimo ir predikatų išnaudojimo sąlyginėse situacijose efektyvumo didinimas.

Antro tipo optimizacijų pagerinimo metodai susiję su tuo, kad būtų atrandamos galimybės, kaip praplėsti tradicines optimizacijas, išnaudojant unikalias architektūros savybes. Pavyzdžiui, jei optimizacijos naudoja ar pertvarko užkrovimo instrukcijas, reikia įvertinti galimybę naudoti vieną iš prognozavimo savybių, kaip tai buvo atlikta dalinių dubliavimų eliminacijos optimizacijoje (2.2.6.2 skyrius).

Tuo tarpu trečiojo tipo optimizacijos, pavyzdžiui, ciklų suliejimas ar ciklų išvyniojimas, pertvarko kodą panašiai kaip ir tradicinėse architektūrose, tačiau, pakeitus kai kurių juos valdančių parametrų reikšmes, kompiliatorius sugeneruoja kodą, kuris efektyviau išnaudoja IA-64 architektūrinius resursus, tokius kaip didelis registrų skaičius, ilgas instrukcijų žodis ir rotuojantys registrai. Šie parametrai ir jų reikšmės nėra trivialūs ir reikia atlikti analizę, kurie iš jų labiausiai optimizuoja kodo vykdymą.

Magistrinio darbo tyrime buvo analizuojamos ciklo optimizacijos, kurių generuojamą kodą galima valdyti kompiliatoriaus parametrais. Ciklo optimizacijos yra vienos lengviausiai pasiduodančių kodo išlygiagretinimui ir gerina darbą su atmintimi. Atviro kodo GCC kompiliatoriuje ciklų optimizacijos nėra iki galo patobulintos ir neprilygsta rezultatams, kuriuos gauna komercinis Intel kompiliatorius [SXC05]. Vienas iš jų gerinimo metodų – pakeisti šias optimizacijas valdančių parametrų reikšmes, efektyviau išnaudojant didelius Intel Itanium procesoriaus resursus.

Siekiant atrasti optimaliausią parametrų rinkinį bei jų reikšmes, kuris sugeneruotų efektyviau vykdomą kodą, turėtų būti įgyvendinti šie žingsniai:

1. Atrasti ir išanalizuoti parametrus, kurie galėtų labiau išlygiagretinti instrukcijas.
2. Atlikti testavimą su įvairiomis parametrų reikšmėmis, siekiant parinkti optimaliausias reikšmes.
3. Palyginti rezultatus su tais, kurie gaunami, naudojant standartines parametrų reikšmes.
4. Pateikti rekomendacijas, kokie parametrai ir jų reikšmės turi būti perduodamos kompiliatoriui, pagrindžiant juos gautais testavimo rezultatais.

GCC kompiliatoriaus 4.1.2 turi 14 valdomų kompiliatoriaus parametrų ciklo transformacijų generuojamam kodui valdyti. Šių parametrų numatytosios reikšmės nėra optimizuotos konkrečiai architektūrai, jos yra universaliai parinktos visoms GCC kompiliatoriaus palaikomoms architektūroms. Šiame tyrime bus išnagrinėta, kurie iš jų gali įtakoti kodą Itanium sistemoje ir kiek galima pakeisti šių parametrų reikšmes, siekiant sugeneruoti efektyviau vykdomą kodą. Šie parametrai keičia ciklų išvyniojimo (loop unrolling), ciklų pjaustymo (loop peeling), ciklų atjungimo (loop unswitch) bei kitas ciklų optimizacijas. Šios optimizacijos nėra nuo architektūros priklausomos, tačiau jų efektyvumą turėtų būti galima padidinti keičiant kai kuriuos iš šias optimizacijas valdančių kompiliatoriaus parametrų.

3.1 Darbo priemonės

Tyrimui buvo naudojamos dvi sistemos: Intel Itanium sistema su Itanium 9020 1.4GHz procesoriumi ir GCC 4.1.2 20070115 versijos kompiliatoriumi, taip pat Core i7 920 2.66GHz sistema su ski 1.3.2 itanium2 procesoriaus emuliatoriumi ir gcc 4.1.2 versijos (cross) kompiliatoriumi, kuris generuoja itanium kodą. Cross kompiliatorius buvo sukompiliuotas naudojantis vedliu iš Gelato@UNSW tinklalapio [Net03]. Testavimui taip pat buvo naudojama pfmon programa. Tai stebėjimo programa, kuri leidžia surinkti informaciją apie procesoriaus

skaitliukus, tokius kaip taktų skaičius, užkrovimų ir saugojimo į atmintį skaičius, programos vykdymo metu. Šiuos procesoriaus skaitliukus programai pateikia Itanium 2 vykdymo stebėjimo įrenginys (Performance Monitor Unit).

3.2 Pradiniai duomenys

GCC kompiliatorius susideda iš nuo kalbos priklausančios programinės įrangos sąsajos (front-end), nuo kalbos nepriklausančios vidinės dalies (back-end) ir iš architektūros specifinių mašinos aprašymų (architecture-specific machine descriptions). Programavimo kalbos sąsaja tos kalbos kodą paverčia į abstraktų sintaksinį medį GIMPLE. Aukšto lygio optimizacijos, pavyzdžiui, dalinė dubliavimų eliminacija (partial redundancy elimination) atliekama su GIMPLE. Kai atliekamos visos tokios optimizacijos, sintaksinis medis yra paverčiamas į tarpinę RTL (Register Transfer Language) reprezentaciją. Daug klasikinių optimizacijų yra atliekami RTL lygyje, pavyzdžiui, ciklų išvyniojimas, instrukcijų planavimas, registrų alokacija ir nuo architektūros priklausomos optimizacijos. Palyginus su medžio lygio optimizacijomis (kurios atliekamos GIMPLE lygyje), RTL į RTL transformacijos yra daug sudėtingesnės ir labiau efektyvios, siekiant pagerinti programos veikimą [SXC05]. Galiausiai RTL reprezentacija yra paverčiama į asemblerio kodą.

GCC kompiliatorius kai kuriose optimizacijose naudoja įvairias konstantas optimizacijos vykdymui valdyti. Kai kurias iš šių konstantų galima nustatyti naudojant --param *name = value* pasirinkimą, kur name yra parametro vardas, o value – jo reikšmė. Visi valdomi parametrai ir jų numatytosios reikšmės yra aprašomos GCC kompiliatoriaus programinio kodo params.def faile.

Tyrimui buvo parašyta nedidelė programa C++ programavimo kalba, kuri atlieka įvairius veiksmus su dvejomis 30x30 dydžio matricomis ir taip pat atlieka apie ~10 įvairių ciklų su matematiniais skaičiavimais (sudėtis, atimtis, dalyba ir daugyba). Tokio kodo efektyvumas priklauso nuo ciklų optimizacijų, nes naudojama daug operacijų su masyvais ir aritmetiniais skaičiavimais (priedas 1).

Vykdymo laikas buvo skaičiuojamas naudojant C standarto bibliotekoje apibrėžtą C Time biblioteką (time.h). Clock() funkcija grąžina procesoriaus taktų skaičių nuo tada, kai startavo programa. Tuo tarpu CLOCK_PER_SEC makro konstanta nurodo ryšį tarp taktų ir sekundžių (taktų skaičius per sekundę). Programos pradžioje ir pabaigoje buvo paimama clock reikšmė ir jų skirtumas paverstas į milisekundes, naudojant CLOCK_PER_SEC makro konstantą. Taip pat programos efektyvumas tikrinimas ir paleidžiant programą su pfmon įrankiu. Buvo stebimi šie skaitliukai: bendras ir sustabdytų taktų skaičius (CPU_OP_CYCLES_ALL, CPU_OP_CYCLES_HALTED), įvykdytų instrukcijų skaičius (IA64_INST_RETIRED_THIS),

tuščių NOP instrukcijų skaičius (NOPS_RETIRED), visų lygių spartinančiosios atminties praleidimo skaitliukai (L1D_READ_MISSES_ALL, L2D_MISSES, L3_MISSES), įvairūs darbo su slankaus kablelio procesoriumi skaitliukai, atšakų (jump) instrukcijų skaitliukas (BR_MISPRED_DETAIL_ALL_ALL_PRED). Papildomai informacijai gauti kartais buvo naudojami ir kiti skaitliukai.

3.3 Tyrimo eiga

3.3.1 Ciklų išvyniojimas

Apie ciklo išvyniojimo optimizaciją plačiau jau buvo aprašyta 2.2.3.3 skyriuje. Pagrindinis jos tikslas yra paspartinti programos veikimą, sumažinus ciklo iteracijų skaičių (padidinant iteracijų žingsnių dydį) ar eliminuojant ciklą iš viso.

Ciklų išvyniojimo optimizacija GCC yra įjungiama -funroll-loops parametru. Ji yra taip pat valdoma trimis parametrais: MAX_UNROLLED_INSNS, MAX_AVERAGE_UNROLLED_INSNS ir MAX_UNROLL_TIMES. MAX_UNROLLED_INSNS nusako, koks gali būti maksimalus instrukcijų skaičius išvyniotame cikle. Numatytoji reikšmė lygi 200. MAX_AVERAGE_UNROLLED_INSNS nurodo maksimalų instrukcijų skaičių priklausomai nuo jų vykdymo tikimybės, kuriuos ciklas turėtų turėti, jeigu ciklas yra išvyniojamas ir, jeigu ciklas yra išvyniojamas, jis nurodo, kiek kartų ši operacija atliekama su ciklo kodu. Numatytoji reikšmė yra lygi 80. MAX_UNROLL_TIMES nurodo maksimalų vieno ciklo išvyniojimų skaičių. Reikšmė pagal nutylėjimą yra trys.

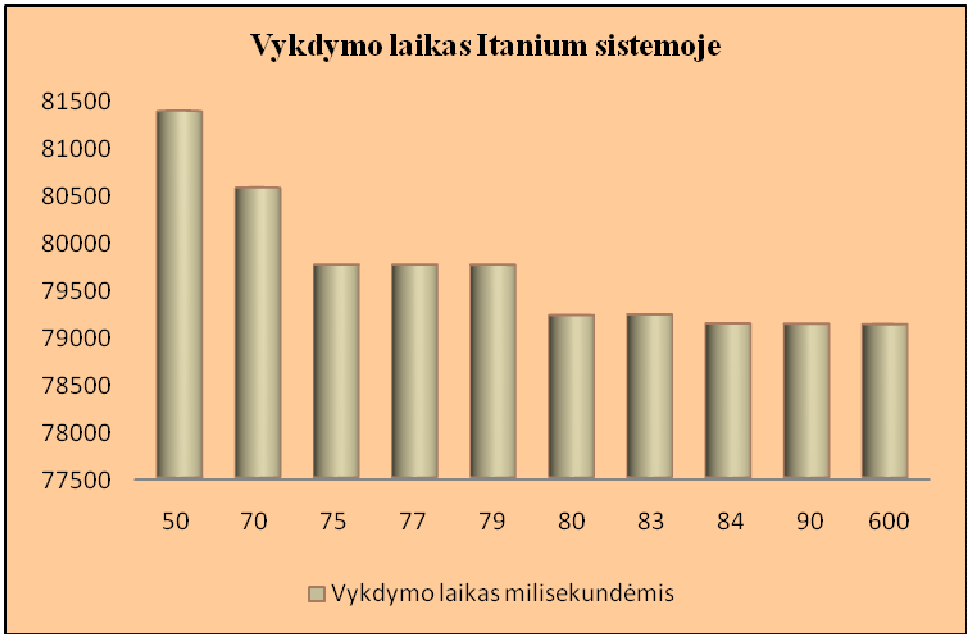
Ankstesni tyrimai parodė, kad reikėtų pakeisti MAX_UNROLLED_INSNS reikšmę iki 600 [YLW+05], [SXC05]. Dėl šios priežasties pradžioje programa buvo kompiliuojama su tokiais parametrais: g++ -O1 -funroll-loops --param max-unrolled-insns=600. -O1 įjungia kodo optimizavimo režimą, -funroll-loops skirtas įgalinti ciklų išvyniojimo optimizaciją ir galiausiai max-unrolled-insns=600 yra valdoma MAX_UNROLLED_INSNS parametro reikšmė. Programa praleidžiama 5 kartus ir lyginamas vykdymo laikas.

Pradinis testavimas parodė, kad parametro keitimas į didesnį nei numatytoji reikšmė neturi jokio efekto ir programos efektyvumas visiškai nesikeitė. Tą rodė ir asemblerio kodo generavimas (kompiliuojant programą su tais pačiais parametrais, tačiau pridedant –S parametrą, kad generuotų asemblerio kodą vietoje vykdomosios programos). Generuojamas asemblerio kodas sutapdavo nepriklausomai nuo to, kokia MAX_UNROLLED_INSNS reikšmė buvo pasirenkama. Siekiant išsiaiškinti priežastis ir aš iš viso programa tikrai priklauso nuo šio parametro, reikšmės buvo sumažintos iki mažesnių nei numatytoji.

Viena iš klaidų paaiškėjo iš karto. Įjungus bent vieną iš gcc kompiliatoriaus kodo optimizavimų parametrų Ox (kur, x nuo 1), kompiliatorius iš karto ieško kaip sumažinti kodo dydį ir vykdymo laiką. Viena iš tokių optimizacijų yra kodo pašalinimas, kuris neįtakoja tolimesnio programos veikimo. Pavyzdžiui, jei atliekami matematiniai skaičiavimai, kurie po to nėra panaudojami kituose skaičiavimuose ar jų rezultatas nėra išvedamas į ekraną, tai tokie skaičiavimai yra pašalinami. Automatiškai beveik visa mano kodo dalis su ciklais buvo tiesiog išmesta ir ciklo išvyniojimo optimizacija net nebuvo taikoma.

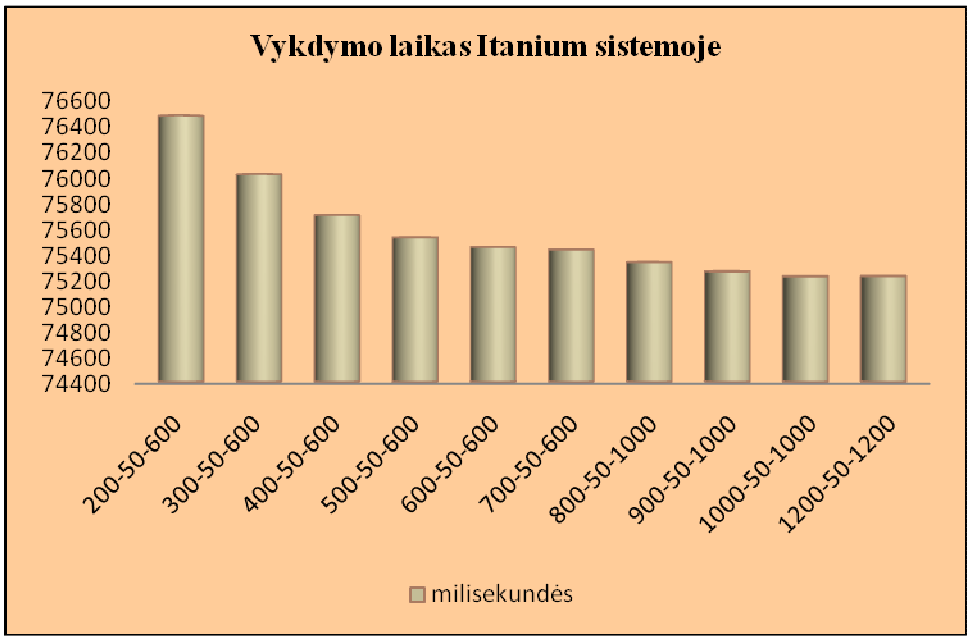
Įvedus galutinių rezultatų išvedimą į ekraną, kodas nebebuvo išmetamas ir visi numatyti skaičiavimai atliekami. Po šio pakeitimo MAX_UNROLLED_INSNS parametro didinimas virš numatytosios reikšmės taip pat nedavė jokių rezultatų. Tačiau parametro reikšmės sumažinimas iki 10 pradėjo generuoti kitokį kodą. Palyginus taip sukompiliuotos programos vykdymo laiką su numatytosios reikšmės programos vykdymo laiku, jis iš tiesų pradėjo skirtis. Taigi, buvo galima daryti išvadą, kad -funroll-loops ir -max-unrolled-insns parametrai iš tiesų veikia ir daro įtaką mano parašytam kodui.

Pirmas mano tikslas buvo išsiaiškinti, iki kurio parametro reikšmės kodas keičiasi. Tam padariau seriją testų, kurie parodė, kad kodo generavimas kinta iki MAX_UNROLLED_INSNS pasieka 84. Asemblerio kodo analizėje matyti, kad, didinant parametrą nuo 20 iki 84, kodas didėja, tačiau daugėja ir lygiagrečiam vykdymui atskirtų instrukcijų blokų. Jeigu pirmu atveju tokių atskyrimų buvo 586, tai antru atveju 1336. Toliau kodas nebesikeičia, kaip ir vykdymo efektyvumas (Pav. 8). Ši taisyklė pasitvirtino abejose sistemose, tiek kompiliuojant kodą itanium sistemoje, tiek su cross kompiliatoriumi ir vykdant jį per ski procesoriaus emuliatorių. Iš testavimo matyti, kad šio parametro didinimas leisdavo programai įsivykdyti sparčiau, tačiau tolimesnis jo didinimas yra kažkuo ribojamas. Specialiai tam aš keičiau iteracijų cikluose skaičių, bet tai nieko nekeitė, todėl nusprendžiau išsiaiškinti, ar neįtakoja šios optimizacijos ir kiti du parametrai ir ar jie nebus tas ribojantis faktorius, dėl kurio MAX_UNROLLED_INSNS reikšmės padidinimas nebeduodavo jokio efekto.



Pav. 8

Testavimas buvo atliktas iš naujo, pridėjus kitus du parametrus, nuo kurių priklauso ciklo išvyniojimo optimizacija: MAX_AVERAGE_UNROLLED_INSNS ir MAX_UNROLL_TIMES (g++ -O1 -funroll-loops --param max-unrolled-insns=<skaičius> max-unrolled-times=<skaicius2> max-average-unroled-insns=<skaicius3>). Rezultatai patvirtino prognozes. Naudojant visus tris parametrus kartu, vykdymo efektyvumą buvo galima padidinti naudojant didesnes parametrų reikšmes negu jų numatytoji reikšmė. Intel Itanium sistemose mano kodo vykdymo laikas spartėjo iki instrukcijų kiekio skaičius max_average_unrolled_insns ir max_unrolled_insns buvo didinimas iki 1000, o išvyniojimo kiekis buvo apribotas iki 50 (max-unroll-times). Dar didesnis instrukcijų kiekio išvyniotame cikle didinimas jau turėjo neigiamą efektą ir kodo vykdymas nebegreitėjo. Padidinus parametrą maksimalų ir vidutinį instrukcijų kiekį iki 1200, asembelerio kodas dar keitėsi nuo ankstesnių parametrų, tačiau vykdymo efektyvumas netgi šiek tiek sulėtėjo (Pav. 9). Taip pat su labai dideliais parametrais tampa juntamas kompiliavimo laiko sulėtėjimas. Reikėtų įvertinti ir tai, kad šių parametrų didinimas didina ir kodo kiekį. Didinant išvyniojimų maksimalų skaičių (MAX_UNROLL_TIMES) dešimčia, kodas gali padidėti 15-20%. Keičiant maksimalų instrukcijų skaičių, bet nekeičiant išvyniojimo kiekio, kodas nedidėjo linijine progresija. Su parametrais max-unrolled-insns=200 ir max-avg-unrolled-insns=200 bei max-unroll-times=50 lygiagrečiam vykdymui atskirtų grupių buvo 2151, 400 - 4325, 600 – 5433, 900 - 5092, 1200 - 5684. Iš to matyti, kad instrukcijų kiekis tarp parametrų 600 ir 900 netgi sumažėjo. Tad aš nusprendžiau apsistoti ties parametrais 900 ir 50 kaip labiausiai efektyviais.



Pav. 9 Vykdymo laikas priklausomai nuo parametrų reikšmių

Panagrinėjus rezultatus su pfmon įrankiu, matyti, kad be vykdymo laiko gana juntamai sumažėja ir atšakos (arba jump) instrukcijų skaičius. Padidinus max-unrolled-insns ir max-average-unrolled-insns nuo 200 iki 900 bei turint max-unroll-times=30, atšakos instrukcijų (bendras jų skaičius kartu su teisingais ir neteisingais spėjimais) sumažėja apie 40%. Taip pat sumažėjo trečio lygio spartinančiosios atminties praleidimų (cache misses) apie 30%. Pirmo ir antro lygio praleidimų beveik nepasikeičia. Max-unroll-times padidinus iki 50 atšakos instrukcijų skaičius sumažėja dar apie 35% procentus.

3.3.2 Ciklų pjaustymas (Loop peeling)

Ciklų pjaustymas yra viena iš ciklo padalijimo į dalis versijų. Jo paskirtis yra atskirti ir iškelti dalį iteracijų virš ciklo kūno. Pavyzdyje matyti, kad p=10 lygi tik pirmai iteracijai, toliau visose iteracijose p=i-1. Dėl šios priežasties ciklo pjaustymo optimizacija iškelia vieną iteraciją virš ciklo kūno:

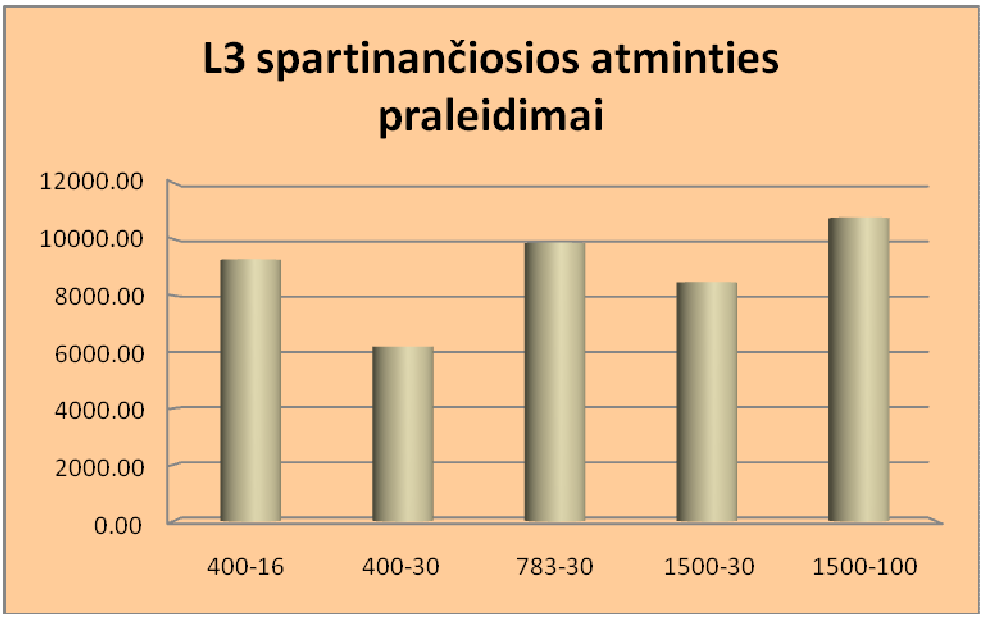
```
int p = 10;
for (int i=0; i<10; ++i)
{
  y[i] = x[i] + x[p];
  p = i;
}
-------------------------------------------------
y[0] = x[0] + x[10];
for (int i=1; i<10; ++i)
{
  y[i] = x[i] + x[i-1];
}
```

Ciklų pjaustymas GCC kompiliatoriuje įjungiamas -fpeel-loops parametru. Jis taip pat valdomas keturiais papildomais parametrais: MAX_PEELED_INSNS – maksimalus instrukcijų skaičius optimizuotame cikle ir nurodo kiek kartų ciklas yra „supjaustomas“. Numatytoji reikšmė yra 400. MAX_PEEL_TIMES nurodo maksimalų vieno ciklo „pjaustymų“ skaičių. Numatytoji reikšmė yra 16. MAX_COMPLETELY_PEELED_INSNS – maksimalus instrukcijų skaičius visiškai „supjaustytam“ cikle. Numatytoji reikšmė yra 400. MAX_COMPLETELY_PEEL_TIMES nurodo maksimalų iteracijų skaičių cikle, kurie būtų tinkami atlikti šią optimizaciją. Numatytoji reikšmė yra 16.

Iš pradžių pabandžiau sukompiliuoti kodą su standartinėmis valdomų parametrų reikšmėmis. Deja, su standartinėmis reikšmėmis generuojamas asemblerio kodas visiškai nesikeitė nuo to, kuris sugeneruojamas neįjungus šios optimizacijos. Tai yra ši optimizacija su standartiniais kompiliatoriaus parametrais neturėjo jokio efekto generuojamam kodui.

Didindamas kiekvieną iš keturių parametrų atskirai galiausiai pavyko gauti kitokį kodą. Padidinus maksimalų iteracijų skaičių cikle, kurie būtų tinkami atlikti optimizacijai, (MAX_COMPLETELY_PEEL_TIMES) iki 30, asemblerio kodo eilučių skaičius išaugo 41%. Lygiagrečiam vykdymui atskirtų blokų padidėjo 50%. Padidinus šį parametrą, pradėjau didinti ir kitus. Paaiškėjo, kad pakeitus šį parametrą, galima didinti ir MAX_COMPLETELY_PEELED_INSNS, tai yra maksimalų iteracijų skaičių, kuriuos būtų galima optimizuoti cikle. Padidinus šį parametrą nuo 400 iki 783 asemblerio kodas taip pat pasikeičia. Tačiau šis parametras, skirtingai nuo „pjaustymų“ skaičiaus keitimo, nepadidina kodo kiekio. Palyginus kodą su parametru lygiu 400, jis sumažėja viena eilute, lygiagrečiam vykdymui atskirtų blokų vienu padaugėja. Įvykdžius gilesnę asembelerio kodo analizę, galima matyti, kad pakeitimų yra ir daugiau. Nors pradžioje instrukcijos sudėliojamos daugiau ar mažiau vienodai, bet kažkur nuo vidurio kodo jos ir pertvarkomos visiškai skirtingai. Unwind_Resume kvietimas asembelerio kode su 783 parametro reikšme yra daug vėlesnėje stadijoje nei su 400 reikšmės parametru. Taigi nors kodo dydžiu tai ir nesimato, bet instrukcijų pertvarkymas parodo, kad optimizacija yra įvykdoma kiek kitaip ir greičiausiai yra daugiau instrukcijų iškeliama virš ciklo kodo. Panaši tendencija išlieka didinant šiuos parametrus dar labiau. „Pjaustymų“ skaičiaus didinimas gana smarkiai didina ir kodą, tuo tarpu iteracijų skaičiaus keitimas kiek kitaip pertvarko kodą. Keičiant tuos pačius parametrus tam pačiam kodui x86 architektūroje, kodas su jais nesikeitė, tad panašu, kad Itanium architektūros specifika leidžia pertvarkyti šį kodą savaip.

Vykdant vykdymo laiko testavimą šių parametrų keitimas nelabai įtakojo rezultatus. Nepaisant to, kiek bedidintum parametrus ir kiek bepasikeistų asembelerio kodas, programos vykdymo laikas išlikdavo beveik toks pats. Tada kodo vykdymą ištyriau su pfmon programa. Ji patvirtino, kad taktų skaičius programos vykdymo metu beveik nekinta (jis kinta mažiau nei 3% ribose). Taip pat nesikeičia ir blogų (sustabdytų ir NOP) instrukcijų skaičius. Tačiau padidinus MAX_COMPLETELY_PEEL_TIMES parametrą iki 30 ir nekeičiant MAX_COMPLETELY_PEELED_INSNS (t.y. ji paliekant lygią numatytajai reikšmei), galima pastebėti pagerėjusį darbą su spartinančiąja atmintimi. 80% sumažėja užstabdytų FPU instrukcijų iš pirmo lygio spartinančiosios atminties, 35% trečio lygio spartinančiosios atminties praleidimai (misses) ir 10% antrojo lygio atminties praleidimai. MAX_COMPLETELY_PEELED_INSNS padidinimas iki 783 ar didesnių reikšmių didelių pakeitimų tuose pačiuose parametruose nepadaro, dažniausiai net vėl suprastėja darbas su spartinančiąja atmintimi apie 10%. Tikrinant max-completely-peeled-insns=1500 ir varijuojant max-completely-peel-times tarp 30 ir 100, matyti, kad geresnis darbas su spartinančiąja atmintimi vyksta, kai „pjaustymų“ skaičius yra lygus 30. Pavyzdžiui, trečio lygio spartinančioji atminties turi apie 20% mažiau praleidimų. Tačiau palyginus šiuos rezultatus su 400 ir 30, jie yra gerokai prastesni ir beveik sutampa su rezultatais be šios optimizacijos (Pav. 10). Taigi, įvertinus šiuos rezultatus, geriausia yra padidinti tik vieną max-completely-peel-times parametrą iki 30. Galima spręsti, kad šis parametras geriausiai išlygiagretina kodą bei lokalizuoja registrų naudojimą taip, kad Itanium efektyviau skirstytų duomenis iš spartinančiosios atminties.



Pav. 10 L3 spartinančios atminties praleidimai ciklų pjaustymo optimizacijoje

3.3.3 Ciklų atjungimas (Loop unswitching)

Dar viena ciklų pertvarkymo optimizacija, kurios generuojamą kodą galima valdyti su papildomais parametrais yra ciklų atjungimas (loop unswitching). Ji perkelia sąlygos sakinį iš ciklo kūno, nukopijuodama to ciklo kūną ir patalpindama šio ciklo versiją kiekviename iš sąlygos atšakų. Tai gali padėti išlygiagretinti ciklą. Pavyzdys:

```
for i to 1000 do
   x[i] = x[i] + y[i];
   if (w) then y[i] = 0;
 end_for;
---------------------------------
if (w) then
   for i to 1000 do
     x[i] = x[i] + y[i];
     y[i] = 0;
   end_for;
 else
   for i to 1000 do
     x[i] = x[i] + y[i];
   end_for
 end_if;
```

Ši optimizacija įjungiama su -funswitch-loops parametru ir turi du valdomus parametrus: max-unswitch-insns ir max-unswitch-level. Pirmasis iš jų nusako maksimalų instrukcijų skaičių optimizuotame cikle. Antrasis nurodo maksimalų atšakų skaičių, kuriuos būtų galima pertvarkyti viename cikle. MAX_UNSWITCH_INSNS numatytoji reikšmė yra 50, o MAX_UNSWITCH_LEVEL – 3.

Įjungus šią optimizaciją su standartinėmis reikšmėmis (-O1 -funswitch-loops --param max-unswitch-insns=50 --param max-unswitch-level=3) generuojamas asemblerio kodas beveik nesiskiria nuo to, kuris generuojamas be šios optimizacijos. Skiriasi tik keletą eilučių su procedūrų kvietimais. Vykdymo laikas ar kiti parametrai taip beveik nesiskiria.

Padėtis pasikeičia, kai maksimalų instrukcijų skaičių padidiname iki 560. Asemblerio kodas išauga apie 40%. Predikatų naudojimas kode ir lygiagrečiam vykdymui atskirtų instrukcijų blokų išauga apie 36%. Vykdymo laikas keičiant šiuos parametrus keičiasi tik keliais procentais. Taip pat nesikeičia naudingų instrukcijų santykis. Tačiau šio parametro didinimas įtakoja darbą su pirmo lygio spartinančiąja atmintimi ir TLB (translation lookaside buffer) buferiu. Pirmo lygio spartinančiosios atminties praleidimų (L1 cache misses) sumažėjo ~10%, TLB praleidimų ~20%. Taigi galima daryti išvadą, kad max-unswitch-insns padidinimas leidžia pagerinti vykdomų instrukcijų lokalumą spartinančiojoje atmintyje ir geriau išnaudoti spartinančiosios atminties resursus.

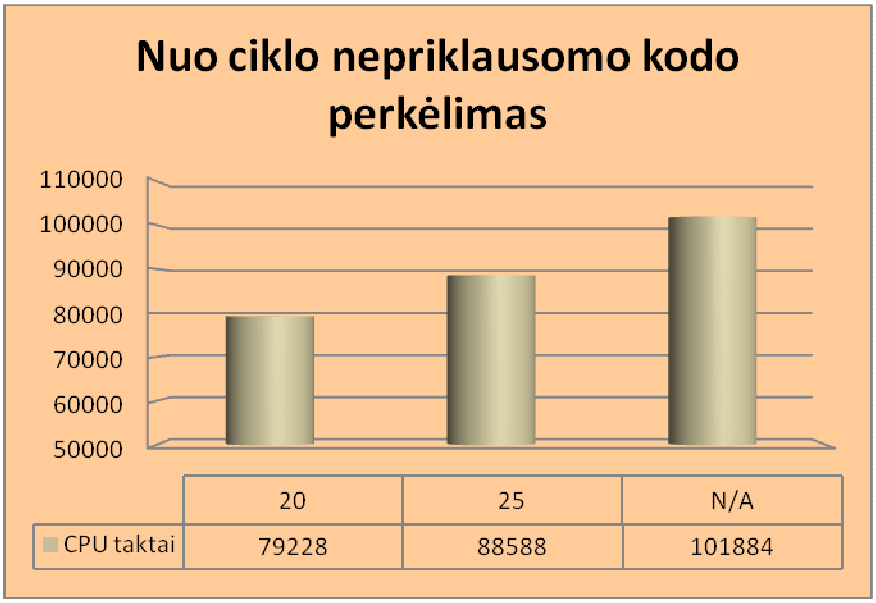
Padidinus max-unswitch-insns iki 560, kartu su juo galima didinti ir max-unswitch-level parametrą. Tokiu būdu įmanoma dar keliais procentais padidinti spartinančiosios atminties lokalizaciją. Visgi, didinant šį parametrą, labai jaučiasi didelis kompiliavimo laiko sulėtėjimas. Pavyzdžiui, padidinus parametrą iki 7, to paties kodo kompiliavimo laikas išaugo daugiau nei dvigubai. Asemblerio kodo kiekis išaugo 12%, tuo tarpu TLB ir L1 spartinančiosios atminties praleidimo parametras sumažėjo tik apie 5%.

3.3.4 Nuo ciklo nepriklausomo kodo perkėlimas

Nuo ciklo nepriklausomo kodo perkėlimas (loop-invariant code motion) yra viena žinomesnių tradicinių optimizacijų, kuri perkelia iš ciklo kūno kodą, nuo kurio nepasikeičia kodo semantika. Ši optimizacija GCC kompiliatoriuje įjungiama su --fmove-loop-invariants. Turi vieną valdomą parametrą lim-expensive, kuris nusako, kokia turi būti minimali perkeliamos instrukcijos kaina. Jo numatytoji reikšmė lygi 20. Pagal GCC dokumentaciją buvo paminėta, kad ši optimizacija įjungiama automatiškai naudojant -O1 optimizavimo parametrą.

Sugeneravus asemblerio kodą rankiniu būdu pridedant minėtuosius parametrus (--fmove-loop-invariants ir lim-expensive) paaiškėjo, kad kodas yra generuojamas kitoks nei kompiliuojant programą tik -O1 parametru. Kodo dydis skiriasi tik keliomis eilutėmis, tačiau nagrinėjant kodą, matyti, kad dalis skaičiavimų yra perkelta į ankstesnę failo dalį ir kodas yra pertvarkytas gana skirtingai nei be šio parametro.

Programos vykdymas po šios optimizacijos įjungimo rankomis pagreitėjo net apie 20% (Pav. 11), TLB praleidimų sumažėjo 95%. Lim-expensive parametro sumažinimas nebekeitė kodo, o didinimas turėjo neigiamą efektą. Kai lim-expensive buvo lygus 25, programa sulėtėjo 10%, TLB buferio praleidimų išaugo apie 80%. Galima daryti išvadą, kad numatytasis parametras yra tinkamas Itanium architektūrai, atliekant skaičiavimus su masyvais, tačiau GCC dokumentacijoje paminėtas optimizacijos įjungimas su -O1 parametru neatitiko tiesos. Šią optimizaciją reikia įjungti rankomis.



Pav. 11 CPU taktų skaičius, priklausomai nuo lim_expensive reikšmės

3.3.5 Kiti ciklų optimizacijų valdomi parametrai

Likę 4 ciklų optimizacijų valdomi parametrai yra susiję su GIMPLE lygio pertvarkymais sintaksiniame medyje (sudėtingų skaičiavimų keitimas paprastesniais, spėjamų iteracijų skaičius). Šie parametrai praktiškai nepriklauso nuo architektūros ir yra įjungiami kartu su -O1 bei aukštesniu optimizavimo parametru automatiškai, tad jų keitimas neduodavo jokio efekto mano programai, ji generuodavo tokį patį kodą nepriklausomai nuo reikšmių. Galima spręsti, kad numatytosios reikšmės yra tinkamos ir pakankamai optimizuotos toms ciklų optimizacijoms, kurias jos valdo, bei jų didinimas tiesiog neįtakoja taip kodo, kaip tai vyksta su RTL lygio pertvarkymais.

Rezultatai

IA-64 architektūra daugiausia remiasi kompiliatoriaus gebėjimu optimizuoti kodą, o ne savo galimybėmis pertvarkyti instrukcijas vykdymo metu. Dėl šios priežasties buvo įgyvendintos savybės, kurios kompiliatoriui leidžia pertvarkyti kodą taip, kad jis būtų vykdomas lygiagrečiai ar išnaudotų didelius architektūros resursus. Pagrindinės tam skirtos architektūros savybės yra predikacija, duomenų bei kontrolės prognozavimas ir dideli registrų resursai.

Magistriniam darbui buvo pasirinktos nuo architektūros nepriklausomos ciklų pertvarkymo optimizacijos, kurios yra vienos iš lengviausiai pasiduodančių, siekiant išlygiagretinti kodą ir pagerinti darbą su atmintimi. Buvo nagrinėjamos tos optimizacijos, kurių generuojamą kodą galima valdyti, keičiant valdomus kompiliavimo parametrus GCC kompiliatoriuje. Šių valdomų parametrų numatytosios reikšmės kompiliatoriuje yra vienodos visose architektūrose, todėl jos gali būti nepakankamai optimizuotos Itanium sistemai. Ankstesniuose tyrimuose [YLW+05] buvo įrodyta, kad šių parametrų pakeitimas leidžia sugeneruoti efektyviau vykdomą kodą Itanium architektūrai.

Savo magistrinio darbo tyrime ištyriau ciklo optimizacijų valdomų parametrų daromą įtaką programos vykdymo efektyvumui. Valdomų parametrų keitimas turėjo įtaką ciklų išvyniojimo (loop unrolling), ciklų pjaustymo (loop peeling), ciklų išjungimo (loop unswitching) ir nuo ciklo nepriklausomo kodo perkėlimo (loop-invariant code motion) optimizacijoms.

Tyrimo rezultatai parodė, kad ciklo išvyniojimo optimizacijai reikia naudoti visų trijų valdomų parametrų kombinaciją max-unrolled-insns, max-average-unrolled-insns, max-unroll-times ir juos didinti vienu metu. Pirmu dviejų parametrų reikšmes galima didinti iki 900, o trečiojo iki 50. Ciklo pjaustymo optimizacijai verta padidinti maksimalų „pjaustymo“ optimizacijos vykdymo kartus (max-completely-peel-times) iki 30. Tai pagerina programos darbą su pirmo lygio spartinančiąja atmintimi ir TLB buferiu. Ciklo atjungimo optimizacijoje taip pat galima pagerinti darbą su šiomis atmintimis, max-unswitch-insns parametrą padidinus nuo standartinių 50 iki 560. Nuo ciklo nepriklausančio kodo iškėlimo virš ciklo optimizacijoje lim-expensive numatytoji reikšmė parodė geriausius rezultatus, paspartinant programą 20% ir pagerinus darbą su TLB ir L1 spartinančiąja atmintimi. Jo didinimas rezultatus tik blogino. Tad lim-expensive optimaliausia reikšmė pasirodė lygi 20.

Išvados

Viena iš būdų pagerinti tam tikrų programų grupių efektyvumą Itanium sistemoje yra agresyvesnis optimizacijų, kurios įtakoja kodo išlygiagretinimą, išnaudojimas. Kadangi Itanium sistema gali palaikyti didesnį registrų spaudimą bei Itanium kompiliatorius sugeba atrasti ir aiškiai išskirti daugiau lygiagrečiai vykdomų instrukcijų, tai leidžia Itanium architektūrai efektyviau išnaudoti optimizacijas ir sugeneruoti labiau išlygiagretintą kodą. Dėl šios priežasties šių optimizacijų generuojamą kodą valdantys parametrai GCC kompiliatoriuje, kurių numatytosios reikšmės yra vienodos visoms architektūroms, ne visais atvejais yra tinkamos ir Itanium procesoriui. Tie parametrai, kurių reikšmės leidžia atlikti gilesnę kodo analizę ar leidžia įtalpinti daugiau instrukcijų optimizuotame kode, dažnai leidžia pagerinti vykdymo greitį arba darbą su spartinančiąja atmintimi ir TLB buferiu.

Vienos iš tokių optimizacijų yra ciklo optimizacijos. Itanium architektūroje rekomenduotina keisti ciklų išvyniojimo, ciklų atjungimo, ciklų pjaustymo ir nepriklausomo nuo ciklo kodo perkėlimo optimizacijas valdančių parametrų reikšmes. Šias optimizacijas valdančių parametrų reikšmių keitimas iš tiesų leidžia generuoti efektyviau vykdomą kodą.

Dažniausiai šių parametrų reikšmių didinimas sugeneruoja ir didesnį programos kodą bei sulėtina kompiliavimo laiką (išskyrūs loop-invariant code motion optimizacijoje), todėl reikėtų įvertinti, ar parametrų didinimas yra priimtinas naudojamoje srityje.

GCC valdomų parametrų rinkinys gali skirtis keičiantis versijoms ir jų naudojimas gali pareikalauti papildomų sąlygų. Pavyzdžiui, ciklo išvyniojimo optimizacijoje anksčiau nebuvo max-average-unrolled-insns parametro ir optimizacija priklausė tik nuo max-unrolled-insns. Šiuo metu tai yra vienas kitą ribojantys parametrai ir, siekiant sugeneruoti efektyvesnį kodą, reikia keisti juos abu vienu metu. Visi šie parametrai yra aprašyti GCC kompiliatoriaus kodo params.def faile, todėl tolimesniuose darbuose galima būtų nagrinėti naujai atsirandančių parametrų įtaką optimizacijos generuojamam kodui. Taip pat ateityje būtų galima įvertinti procesoriaus registrų spaudimą ir atminties apkrovą. Be to, galima būti įvertinti ir kitų Intel Itanium kompiliatorių analogiškas optimizacijas, pavyzdžiui, Open64, openIMPACT ar ORC.

Šaltinių sąrašas

[АСУ03] Альфред Ахо, Рави Сети, Джеффри Ульман. Компиляторы: принципы, технологии, инструменты. М. Издательский дом "Вильямс" 2003г.

[BCC+00] J. Bharadwaj, W. Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, J. Pierce. The Intel IA-64 compiler code generator. Intel, 2000.

[DKK+99] Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, David Sehr. An Overview of the Intel® IA-64 Compiler. Intel Technology Journal, 1999.

[Dul98] Carole Dulong. The IA-64 Architecture at Work, „Computer“ journal July 1998. [žiūrėta 2009-05-21]. Prieiga per internetą: <http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/mags/co/&toc=comp/mags/co/1998/07/r7toc.xml&DOI=10.1109/2.689674>

[Fos05] Реферат "Разработка и создание страхового фонда документации", раздел “ Архитектура IA- 64”. [žiūrėta 2009-05-21]. Prieiga per internetą: <http://www.fos.ru/technic/14215.html>

[Gri71] David Gries. Compiler Construction for Digital Computers. John Wiley & Sons, Inc, 1971.

[HK00] Hazelwood, Kim Michelle. Dynamic Optimization Infrastructure and Algorithms for IA-64, 2000.

[HKS+04] Gerolf Hoflehner, Knud Kirkegaard, Rod Skinner, Daniel Lavery, Yong-fong Lee, Wei Li. Compiler Optimizations for Transaction Processing Workloads on Itanium Linux Systems. Intel Compiler Lab, 2004.

[HMR+00] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, Rumi Zahir. Introduction to the IA-64 architecture. IEEE, 2000.

[Мед01] Александр Медведев**.** Intel и AMD: понемногу об архитектурных новшествах, 2001. [žiūrėta 2009-05-21]. Prieiga per internetą: <http://www.ixbt.com/editorial/a-vs-i-part2.shtml>

[Mor97] Building an Optimizing Compile - Bob Morgan. Elsevier Science, 1998.

[Net03] Marco Aurélio Stelmar Netto. Installation of a Cross-Compiler (IA32 -> Itanium), 2003. [žiūrėta 2009-05-21]. Prieiga per internetą: <http://www.gelato.unsw.edu.au/IA64wiki/AlternateCrossCompilation2>

[SXC05] Thambipillai Srikanthan, Jingling Xue, Chip-Hong Chang. Advances in Computer Systems Architecture – 10th Asia-Pacific Conference, ACSAC 2005, Singapore, October 24-26, 2005 : Proceedings. [žiūrėta 2009-05-21]. Prieiga per internetą: <http://books.google.com/books?id=Zo0KRa-22ggC&printsec=frontcover>

[Шпа00] Шпаковский Г.И. Параллельные микропроцессоры для цифровой обработки сигналов и медиа данных. – Мн.: БГУ, 2000. – 196 с. ISBN985-445-306-5.

[TVV+03] S. Triantafyllis, M. Vachharajani, N. Vachharajani, D. I. August. Compiler Optimization-Space Exploration. IEEE, 2003.

[Upg04] Современные серверные процессоры, часть 2. Intel Itanium, HP PA8700, Alpha, žurnalas “Upgrade” Nr.1(16) 2004.
[žiūrėta 2009-05-21]. Prieiga per internetą:
<www.npk.ru/articles/article.html?id=47_1&pv=1>

[YLW+05] Canqun Yang, Chunjiang Li, and Feng Wang. Proceedings of the GCC Developers’ Summit. Performance Improvements for GCC Using Architecture Features on IA-64. 2005, pp. 199-210.
[žiūrėta 2009-05-21]. Prieiga per internetą:
<http://www.gccsummit.org/2005/2005-GCC-Summit-Proceedings.pdf>

Priedai

1 priedas. Testavimo programos kodas

```
#include <iostream>
#include <time.h>
#include <cstdlib>
#define MAX_RAND (1000)

using namespace std;

int main()
{

  time_t time_start;
  time_t time_end;
  time(&time_start);
  clock_t time_mili;
  int array_size = 100;
  double mili_t, middle, last=0, last2=0, last3=0;
  int i,j,z;
  double matrix1[array_size][array_size];
  double matrix2[array_size][array_size];
  double a[30], b[30], c[30];

  double random_integer;

  srand((unsigned)time(0));
  time_mili = clock();
  printf("%s", ctime(&time_start));
// uzpildau matricas
  for (i=0;i<array_size;i++)
  {
    for (j=0;j<array_size;j++)
    {
        random_integer = rand()/10000;
        while(random_integer>1100)
                random_integer/=10;

        matrix1[i][j] = random_integer;
        random_integer = rand();
        while(random_integer>1)
                random_integer/=10;
        matrix1[i][j]+= random_integer;
    }
  }

  for (i=0;i<array_size;i++)
  {
    for (j=0;j<array_size;j++)
    {
```

```
        while(random_integer>1100)
                random_integer/=10;

        random_integer = rand()/11257;
        matrix2[i][j] = random_integer;
        random_integer = rand();
        while(random_integer>1)
                random_integer/=10;
        matrix2[i][j] += random_integer;
    }
  }
// matricu daugyba
  for (i=0;i<array_size;i++)
  {
    for (j=0;j<array_size;j++)
    {
        middle = 0;
        for (z=0;z<array_size;z++)
                middle += matrix1[i][z]*matrix2[z][j];
        matrix2[i][j] = middle;
    }
  }
// matricu transponavimas
  for (i=0;i<array_size;i++)
  {
    for (j=0;j<array_size;j++)
    {
        middle = matrix1[j][i];
        matrix1[j][i] = matrix1[i][j];
        matrix1[i][j] = middle;

        middle = matrix2[j][i];
        matrix2[j][i] = matrix1[i][j];
        matrix2[i][j] = middle;

    }
  }


  for(i=0;i<30;i++)
  {
    a[i]=rand()/100000;
    b[i]=rand()/100000;
    c[i]=rand()/100000;
  }

  for(j=0;j<30;j++)
  for (i=0;i<10000000;i++)
  {
   b[j] = b[j] + c[j];
   c[j]--;
   a[j] = 12 + b[j];
```

```
}

for(j=0;j<30;j++)
for (i=0;i<10000000;i++)
{
  c[j] = (c[j]/10 * a[j]/10) + 10 + b[j];
  c[j] = c[j] + 8;
}

for(j=0;j<30;j++)
for (i=0;i<20000000;i++)
{
  c[j] = a[j] + 10 / c[j] + 12;
}

for(j=0;j<30;j++)
for (i=0;i<20000000;i++)
{
  c[j] = a[j] + 10 / c[j] + 12;
}

for(j=0;j<30;j++)
for (i=0;i<20000000;i++)
{
  c[j] = a[j] + 10 / c[j] + 12;
}

for(j=0;j<30;j++)
for (i=0;i<20000000;i++)
{
  a[j] = (a[j] * 10) / (c[j] * 12);
  b[j] = a[j] + 20+c[j];
  b[j] = b[j] -25*100/7;
}


for(j=0;j<30;j++)
for (i=0;i<40000000;i++)
{
  a[j] = a[j] + 14 / c[j] - 10;
  b[j] = c[j] + 23;
}

for(j=0;j<30;j++)
for (i=0;i<15000;i++)
{
  c[j]--;
  a[j] = c[j] + 12;
  c[j] = c[j] - b[j] + 20;
}

for(j=0;j<30;j++)
```

```
 for (i=0;i<10000;i++)
 {
  a[j] = a[j]/2*3;
  c[j] = b[j] + a[j];
 }

 for(i=0;i<30;i++)
 {
   a[i]=rand()/100000;;
   b[i]=rand()/100000;
   c[i]=rand()/100000;
 }

 for(j=0;j<30;j++)
 for (i=0;i<15000000;i++)
 {
  a[j] = a[j] + 10 / c[j] + 12;
  b[j] = a[j] + 20;
 }

 for(j=0;j<30;j++)
 for (i=0;i<1200000;i++)
 {
  a[j] = a[j] + 10 / c[j] + 12;
  b[j] = a[j] + 20;
 }

 for(j=0;j<30;j++)
 {
       last += a[j]/10000 + b[j]/10000;
 }

 for (i=0;i<array_size;i++)
 {
   for (j=0;j<array_size;j++)
   {
       last2 += matrix1[i][j]/100000;
       last3 += matrix2[i][j]/100000;
   }
 }
 mili_t =(clock() - time_mili)/(double)CLOCKS_PER_SEC*1000;
 time(&time_end);
 printf("baigta %s", ctime(&time_end));
 cout<<"last"<<last<<" "<<last2<<" "<<last3<<"\n";
 cout<<"milisekundes"<<" "<<mili_t<<"\n";

 return 0;
}
```