Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Franko Jančič

Optimizacija strojne kode brez časovnih omejitev

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Peter Klepec
Somentor: izr. prof. dr. Martin Krpan

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Zaradi želje po hitrem prevajanju je optimizacija strojne kode med prevajanjem vedno časovno zelo omejena. Raziščite, ali bi z odpravo časovne omejitve pri optimizaciji strojne kode lahko pridobili bistveno učinkovitejšo strojno kodo glede na različne kriterije kot sta dolžina in hitrost kode. Preglejte ustrezno strokovno literaturo, podajte pregled in poskusite implementirati najobetavnejšo metodo, po možnosti tako, ki se jo da ustrezno paralelizirati.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
GCC	GNU Compiler Collection	GNU prevajalnikova knjižnjica
GNU	GNU's Not Unix	GNU-jev ne Unix
CPU	Computer Processing Unit	Centralno processna enota
SA	Simulated Annealing	Simulated Annealing
DDEC	Data-Driven Equivalence Checking	Podatkovno vodeno preverjanje ekvivalence
JIT	Just-in-time	ravno pravi čas
ULP	uncertainty in the last place	odstopanja v zadnjem mestu
GSO	Group search optimizer	skupinsko optimizaroe iskanja
EGCS	Experimental GNU Compiler System	Eksperimentalni GNU prevajalni sistem

Povzetek

Naslov: Optimizacija strojne kode brez časovnih omejitev

Avtor: Franko Jančič

V današnjem času nam je pisanje programov precej olajšano. Za pisanje lahko uporabimo visoko nivojske jezike (kot so Java, C, C++, Python itd) ali pa psevdo ukaze. Vendar moramo programe v večini primerov prevesti v strojno kodo, da jo lahko računalnik požene. Če pa bi si podrobno ogledali to strojno kodo, bi v nekaterih primerih opazili, da se prevedena koda ne ujema s izvorno kodo, klub temu, da je funkcijonalnost takega programa enaka kot v izvirni kodi. To je zato, ker ga prevajalnik optimizira. V nadaljevanju bomo spoznali koliko je možno program optimizirat v daljši časovni omejitvi in koliko se splača to delat.

Ključne besede: računalnik, superoptimizator, strojna koda,prevajalnik,optimizacija.

Abstract

Title: Optimization of the machine code without time constraint

Author: Franko Jančič

Nowadays we can write programs much easier. For writing the code we can used High-Level languages (such as Java, C, C ++, Python, etc.) or a pseudo-codes. However in most cases we have to compile into machine code that the computer can run. If you look more closely at the machine code, in some cases we notice that the translated code does not match the source code, however, the function of programmenwriten in machine code and source code are the same. This is because the compiler optimizes the program. Below, we will know how much it is possible to optimize the program with long time constraint and how much does it pays to work.

Keywords: computer,superoptimizator,machine code,asembler,optimization.

Poglavje 1

Uvod

V današnjem času nam je pisanje programov precej olajšano. Za pisanje lahko uporabimo visoko nivojske jezike (kot so Java, C, C++, Python itd) ali pa psevdo ukaze. Vendar moramo programe v večini primerov prevesti v strojno kodo, da jo lahko računalnik požene. Če pa bi si podrobno ogledali to strojno kodo, bi v nekaterih primerih opazili, da se prevedena koda ne ujema s izvorno kodo, klub temu, da je funkcijonalnost takega programa enaka kot v izvirni kodi. To je zato, ker prevajalnik v neki časovni meji optimizira kratke dele programa (ponavadi eno ali dve inštrukciji v skupini). Optimiziran program igra zelo pomebno vlogo saj lahko tak program izvede več stvari, nam omogoča dodajanje dodatnih funkcij in/ali zmanjšuje porabo omejenih virov.

1.1 Zgodovina

1.1.1 Prvi optimizator

Prvi primer poskusov se najde že leta 1987 [**?**]. Superoptimizer vzame program napisan v strojnem jeziku kot vhodni vir. Ugotovi najkrajši program, ki izračuna enako funkcijo kot izvorni program z izčrpnim iskanjem preko vseh možnih programov. Prostor iskanja je opredeljen z izbiro podskupinskega nabora strojnih ukazov, in funkcionalno enak vendar optimiziran nabor uka-

zov shranjenih v tabeli. Superoptimizator s pomočjo te tabele generira kodo, najprej dolžine 1, nato dolžine 2, in tako naprej. Vsak od teh ustvarjenih programov je testiran, in če je ugotovljeno, da se funkcija izvirnega programa ujema, superoptimizator natisne program in konča. Za skrajšanje časa za iskanje uporablja dve metodi. Prvi je hiter probabilističen test za določanje ekvivalence dveh programov. Druga je metoda obrezovanje preiskovalnega prostor, medtem pa ohranja jamstvo optimalnosti. Ker pa program ni mogel optimizirati kode daljše od 12 inštrukcij, je bila njegova uporabnost majhna.

1.1.2 GNU-Superoptimizer

Razvinjanje se je nadaljevalo v letu 1992 z kombiniranjem GNU C prevajalnikom in GSO superoptimizatorjem. GSO je približno 3000 vrstic kode C in je od gostitelja neodvisna. Za razliko od prvih suprotimizatorjev, superoptimizer GSO išče zaporedij navodil, izračuna eno več ciljnih funkcij, ki so bili prevedeni v GSO. Željeni cilj je izbran z možnostjo ukazne vrstice z imenon mnemonika. GSO generira zaporedja kode za veliko različnih ciljnih naprav in je zasnovan tako, da so lahko dodatne ciljne naprave enostavno dodane. Trenutno GSO podpira IBM RS / 6000, Sparc, Motorola 68K in 88k, AMD 29k in Intel 80386. Prenosljivost je dosežena z opredelitvijo splošnih inštrukcij, ki vključujejo združitev vseh navodili, ki jih GSO podpira pri vseh podprtih naprav [1].

1.1.3 STOKE

Leta 2013 so razvili stohastičen superoptimizator. Stohastičen optimizator uporablja brez zančni superoptimizacijsko nalogo kot stohastični problem iskanja. Konkurenčne omejitve od preoblikovanja pravilnosti in izboljševanja delovanja so kodirani kot izrazi v odvisnosti stroškov, ter Markov Chain Monte Carlo vzorčevanje se uporablja za hitro raziskovanje prostora vseh možnih programov, da najde tisti program, ki je optimizacija danega ciljnega programa. Čeprav njihova metoda žrtvuje popolnosti, obseg programov,

programe ki jih sposoben preučiti in kakovost programov, ki jih posledično proizvaja, daleč presegajo tiste proizvedene od obstoječih superoptimizatorjev. Že binarne datotek, prevedene z llvm -O0 izbranim za 64-bitne x86, njihova izvedba prototipa, STOKE, je sposobna proizvajati programe, ki bodisi so enako dobre ali prekašajo kodo, ki jih generira gcc -O3, icc -O3, in v nekaterih primerih tudi strokovno lastnoročno napisano kodo [2, 3].

1.2 Uporabjene metodologije

Pri tej diplomski nalogi bodo uporabljene naslednje metodologije:

- študija izvedljivosti:
  - možnost prenosa oz. uporabe starejših pristopov
  - možnost uporabe novejših pristopov na več arhitektur centralno procesne enote
- pregled področja:
  - starejši pristopi optimizacije
  - novejši pristopi otimizacije
- implementacija prilagoditev

1.3 Cilji in predvidevani rezultati diplomske naloge

Cilj diplomske naloge je razložitit in ponazoriti, kako do sedaj razviti optimizatorji delujejo in poiskati odgovore na naslednja vprašanja: Ali lahko uporabimo stari optimizator in če lahko ali je njegova uporabnost večja, ali lahko superoptimizator razširimo na več arhitektur centralno procesne enote in če lahko, kako, kako dobra bi bila optimizacija, če bi časovno omejitev izpustili in kakšni so rezultati pri uporabi takega superoptimizatorja. Dodaten

cilj je še narediti izvirno kodo z implementacijo superotimizatora. Ker pa so cilji lahko preveč visoki je možno, da nekaterih ciljev ne bomo dosegli in tudi rezultati ne bodo uspešni.

1.4 Pripravljeno okolje za testiranje

Ker so rezultati odvisni od programske opreme in strojne opreme, je pomembno omeniti, kje so bili primerki testirani, kakšen je bil način testiranja in zmogljivost naprave, ki je bila uporabljena pri testiranju. Raziskava je potekala na prenosnem računalniku z Intel Core i5-4200M centralno procesni enoti, z frenkvenco 2.5Ghz. Model centralno procesne enote najdemo v skupini Sandy Bridge. Na napravi je nameščen Windows 8.1 operacijski sistem. Optimizatorji, ki so bili uporabljeni za raziskavo, niso podprti za ta operacijski sistem. Za zagon je v potreben linux operacijski sistem Ubuntu 14.04. To oviro je bilo mogoče obiti s pomočjo programa VirtualBox. Virtual box je program, ki ustvari okolje za navidezno napravo, navidezna naprava pa deluje kot, pravi računalnik. V navidezni napravi je nameščen Ubuntu in vsa testiranja in raziskave so potekala v tem okolju. Za navidezno napravo je bilo dodeljeno 1024 MB prostorja na navideznem pomnilniku.

1.5 Pojasnilo kode v zbirniku

Jezik ki ga ti optimizatorji napišejo je strojni jezik. Ker pa je tak jezik težko berljiv, bo koda zapisana v zbirniku. Ker pa je lahko tak zapis za neizkušene zapleten, bo v tem poglavju pojasnjeno osnovno o takem zapisu. Zapis bo uporabljen za Intel arhitekturo centralno procesne enote. Ukazi lahko imajo več formatov, vendar običajno prevladuje en formata. Prvih nekaj bitov vedno predstavlja operacijsko kodo. Operacijska koda pove, kaj na naredi z vrednosti v registru in v pomnilniku. Ostalo pa so operandi, ki tipično opisujejo vrednost ali pa ime registra. Vsaka operacijska koda ima nič ali več operandov. Ponavadi sta dva. Format, ki prevladuje ima

dva operanda. Operand lahko vsebuje tudi takojšnje vrednost. Registri so ponavadi predznačeni z znakom °(nekateri registri imajo svoja imena), v Intel zbirniku pa so predznačeni s simbolom %. Poleg tega je dolžina ukaza lahko različna.

Poglavje 2

Splošen pregled optimiziranja

V tem poglavju bo omenjen splošen način optimiziranja. Pri tem bodo omenjena pravila, ki jih moramo upoštevati in na kaj moramo paziti pri določenih operandih. Poglavje je razdeljeno ne 3 dele: optimizacije brez skokov, s skoki in optimizacija klicev funkcij.

2.1 Optimizacija kode brez skokov

Optimizacija kode brez skokov je optimizacija programa ali dela programa, ki ne vsebuje skoke. S tem lahko predpostavimo, da se vrstni red izvajanje ukazov ne bo spreminjal. To nam omogoča, da lahko spreminjamo vrtni red ukazov, kjer je lahko eden izmed zadnjih ukazov med prvimi izveden in pri tem se funkcijonalnost ne spremeni. Položaj, od kje do kje lahko premikamo nek ukaz, je odvisno od kdaj smo nekemu registru, ki ga ukaz uporablja, priredili vrednost. Z drugimi besedami, vrstni red ukazov lahko spreminjamo dokler za vsak register velja, da je število branj registra med vsakim parom prirejanj istega registra enak izhodiščni kodi. Pri tem lahko vrstni red branj registra poljuben razen v primeru kjer v ukazu beremo in prirejamo iz istega registra. V tem primeru mora biti branje zadnje v vrstnem redu, saj vrednost registra najprej preberemo in nato priredimo. Pri tem lahko opazimo prvo mogočo optimizacijo. Če med dvema prirejanja istega registra ni branja iz

tega registra, potem je prvo prirejanje odveč in ga lahko odstranimo. Pri premikanju ukazu lahko dobimo zaporedne ukaze z enako operacijsko kodo. Pri tem je lahko možno, da ju združimo. Pri tem moramo pogledat kateri registri se pojavijo v operandu. Obstaja tudi optimizacija, kjer lahko namesto ene operacije uporabimo drugo operacijo, saj druga operacija porabi manj urinih ciklov kot prva operacija (Npr. moženje z 2, porabi več časa kot aritmetični premik v desno). Na koncu imamo še kombinacijo ukazov, ki lahko zamenjamo z drugo kombinacijo ukazov, ki je hitrejša od prve. Tako združevanje pa ni vedno mogoč. Prvo težavo predstavlja deljenje celih števil. Recimo, da imamo dva zaporedna ukaza deljenje in množenje s število 5 in registru imamo število 6. Če izvedemo ta dva ukaza, bi imeli v registru na koncu vrednost 5. Če pa bi naivno združili ta dva ukaza, bi na koncu ostala vrednost 6. Drugi večji problem predstavljajo števila s plavajočo vejico (decimalna števila). Decimalna števila so v registru predstavljena z eno bitnim predznakom, 11 bitno eksponento in 52 bitno mantiso. Tako število je pogosto po kakršnikoli operaciji predstavljeno z vrednostjo, ki je najbližji tej vrednosti. Z drugimi besedami je možno da se vrednost, ki jo dvakrat seštejmo manjšo vrednost z večjo, in vrednost, ki jo seštejemo z isto večjo vrednostjo z vsoto istimi manjšimi vrednosti, lahko razlikujeta. Zaradi tega se večina optimizatorjev ne ukvarja z optimizacijo takih števil [4]. Nazadnje moramo še paziti na vrednosti v posebnih registrih in zastavicah. Program lahko optimiziramo tudi tako, da uporabimo še dodatne registre za shranjevanje določenih vrednosti, ki jih v kodo večkrat uporabimo.

2.2 Optimizacija kode s skoki

Optimizacija kode s skoki je optimizacija, ki optimizira program ali del programa, ki ima v svoji kodi ukaz za skok. To pomeni, da vrstni red izvajanja ukazov ni enak vrstnemu redu ukazov zapisanega v programu. To predstavlja nov problem, saj moramo biti pozorni na žive registre pred skokom in žive registre na lokaciji kjer se izvede prvi ukaz po skoku. Težavo obidemo tako, da

kodo razdelimo na dele. Del kode lahko zavzema med skokoma, med skokom in lokacijo prvega ukaza po skoku in med dvema lokacijama prvega ukaza po skoku. Ukazi, ki so v določenem delu kode morajo ostati v tem delu. Optimizacijo lahko naredimo pri pogojnih skokih. Če lahko izračunamo kakšne vrednosti lahko nek pogoj zavzema, potem lahko izračunamo, če bo ta pogoj vedno izpolnjen ali neizpolnjen. Če lahko, potem lahko odstranimo nasprotno vejo, ker se ta veja ukazov ne bi nikoli izvajala. Problem je, da ta vrednost ni vedno izračunljiva. Kar še lahko naredimo je optimizacija zanke. Recimo, da imamo zanko kjer je pogoj da se na določenem registru nahaja število m, ki je manjše od n. To število m dobimo tako, da pomnožimo število j, ki ga povečujemo po vsaki opravljeni iteraciji za 1, s številom k. Namesto, da bi za vsak pogoj morali ponovno množiti, bi lahko število j povečali za k. Še en način optimizacije je izigibanjem pogojnih skokov z podvajanjem kode. Če ima kakšna zanka več pogojev, potem je lahko možno, da to zanko podvojimo tako, da v prvi zanki ostane en pogoj, v drugi pa drugi pogoj. S takimm načinom se lahko tudi izognemo izvajanu `if` stavkov, če pogoj ni odvisen od spremenljiv v zankah [5].

2.3 Optimizacija klicev funkcij

Pri optimizaciji funkcije veljajo pravila kot pri optimizaciji s skoki. Pri tem še obstaja pravilo, da vsakemu registru, ki ga uporabimo v klicani funkciji, moramo shraniti njegovo stanje. Izjeme so posebni registri, globalni registri (oz. registri za katere je dogovorjeno, da niso definirani ob klicu) in register za vračanje vrednosti. Pri optimizaciji klicev funkcije, skušamo klic obiti. Namesto klica funkcije bi lahko vstavili kodo. To bi prineslo možnost bolj optimizirane kode in izpust vhodnega in izhodnega dela funkcije. Rezultat take optimizacije, bi bila hitrejša koda za ceno prostorja na pomnilniku, kar bi bilo vredno, če je koda dovolj kratka. Če se taka optimizacija ne splača, potem imamo še eno vrsto optimizacije, ki skuša uporabiti, čim manj lokalnih registrov in čim več globalnih registrov za izvajanje funkcije.

Poglavje 3

GNU-Superoptimizer

Prvi superotimizator je bil razvit leta 1987, leta 1992 pa ga izboljšali in kombinirali s prevajalnikom C. Kot je že v uvodnem poglavju omenjeno GSO optimizator je dolg približno 3000 vrstic kode C in je od gostitelja neodvisna. Za razliko od prvih suprotimizatorjev, superoptimizer GSO išče zaporedij navodil, izračuna eno več ciljnih funkcij, ki so bili prevedeni v GSO. Željeni cilj je izbran z možnostjo ukazne vrstice z imenon mnemonika. GSO generira zaporedja kode za veliko različnih ciljnih naprav in je zasnovan tako, da so lahko dodatne ciljne naprave enostavno dodane. Trenutno GSO podpira IBM RS / 6000, Sparc, Motorola 68K in 88k, AMD 29k in Intel 80386. Prenosljivost je dosežena z opredelitvijo splošnih instrukcij, ki vključujejo združitev vseh navodili, ki jih GSO podpira pri vseh podprtih naprav [1].

3.1 Opis programa in načina delovanja

Kot je že v točki 3 omenjeno, optimizator je napisan v programskem jeziku C. Če želimo tak program uporabiti ga moremo najprej prevesti z C prevajalnikom, pri tem še moramo dodati za katero arhitekturo procesorja hočemo prevesti, saj se imena ukaznih operandov in registrov spreminjajo. Če smo podali ime, ki se nahaja na seznamu podprtih procesorjev, potem bo prevajalnik uspel prevesti kodo. Razlog zakaj moramo podati ime procesorja, se

nahaja v izvirni kodi.

Primer dela programske kode

```
#if SPARC
   "%i0", "%i1", "%i2", "%i3", "%i4", "%i5", "%i6", "%i7",
#elif POWER
   "r3", "r4", "r5", "r6", "r7", "r8", "r9", "r10",
#elif M88000
   "r2", "r3", "r4", "r5", "r6", "r7", "r8", "r9",
#elif AM29K
   "lr2", "lr3", "lr4", "lr5", "lr6", "lr7", "lr8", "lr9",
#elif M68000
   "d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7",
#elif I386
   "%eax", "%edx", "%ecx", "%ebx", "%esi", "%edi", "%noooo!", "%crash!!!",
#elif PYR
   "pr0", "pr1", "pr2", "pr3", "pr4", "pr5", "pr6", "pr7",
#elif ALPHA
   "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7",
#elif HPPA
   "%r26", "%r25", "%r24", "%r23", "%r22", "%r21", "%r20", "%r19",
#elif SH
   "r4", "r5", "r6", "r7", "r8", "r9", "r10", "r11",
#elif I960
   "g0", "g1", "g2", "g3", "g4", "g5", "g6", "g7",
#elif XCORE
   "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7",
#elif AVR
   "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7",
#else
        #error no register names for this instrution
#endif
```

(Primer kode se najde v datoteki superopt.c (vrstice 446-474))

Če pogledamo naslednji del izvirne kode, opazimo da so nekateri deli programa ločeni z `#if` stavki. Ti stavki se prevedejo le če so pogoji izpolnjeni. Vsak od teh stavkov definira vrednosti, potrebne za izvajanje programa in izpisa optimizirane kode v zbirnem jeziku. Preveden program nima grafičnega uporabniškega vmesnika in nima privzetih nastavitev za zagon na klik zato ni

prijazen do vsakdanih uporabnikov. Progam poženemo z ukazno vrstico. Pri tem moramo dodati cilj oz. funkcijo, ki jo hočemo optimizirati. Seznam ciljev se nahaja v datoteki goal.def, kje lahko dodajamo nove cilje. Po vsakem spreminjanju, moramo izvirno kodo optimizatorja obvezno ponovno prevesti. Optimizator predpostavlja, da za vsak ukaz porabimo 1 enoto časa oziroma 1 urino periodo.

Nekaj primerov definiranih ciljev

```
DEF_GOAL (EQ0,          1, "eq0",       { r = v0 == 0; })
DEF_GOAL (NE0,          1, "ne0",       { r = v0 != 0; })
DEF_GOAL (LES0,         1, "les0",      { r = (signed_word) v0 <= 0; })
DEF_GOAL (GES0,         1, "ges0",      { r = (signed_word) v0 >= 0; })
DEF_GOAL (LTS0,         1, "lts0",      { r = (signed_word) v0 < 0; })
DEF_GOAL (GTS0,         1, "gts0",      { r = (signed_word) v0 > 0; })
```

(Primer kode se najde v datoteki goal.def (vrstice 76-81). Prvi argument je notranje ime, drugi argument vsebuje število argumentov potrebnih za izračun ciljev, treji argument je zapisan cilj, ki se uporabi kot argument za zagon, četrti argument je koda v C jeziku)

Program mora imeti tudi definirane operacijske kode in opis manipulacije operandov pri posamezni operacijski kodi. Ogled, kako je vsak ukaz definiran, je malo bolj zapleten, saj je postopek inicijalizacije razdeljen v treh datotekah. Naprej moramo napisati macro funkcijo v datoteki superopt.h. Potem je potrebno dodati nov vnost v datoteko insn.def, kjer določimo tip in ime ukaza. Oba popravka moramo še dodati v synth.def, kjer določimo, katero vrsto rekurzije bo ukaz uporabljal tako, da podamo notranjo sistemsko ime. Na koncu še moramo posodobit run_program.def, da bo lahko program klical funkcijo.

Nekaj primerov definiranih ukazov

```
DEF_INSN (ADD,          'b', "add")
DEF_INSN (ADD_CI,       'b', "add_ci")
DEF_INSN (ADD_CO,       'b', "add_co")
DEF_INSN (ADD_CIO,      'b', "add_cio")
```

(Primer kode se najde v datoteki insn.def (vrstice 42-45). Prvi argument je notranje ime, drugi argument vsebuje tip ukaza, treji argument je zapis imena)

Primer kode:

```
PERFORM_ADD_CIO(v, co, r1, r2, ci);
CRECURSE_2OP(ADD_CIO, s1, s1, s2, 1, CY_JUST_SET);
```

(Primer kode se najde v datoteki synth.def (vrstice 2707-2708))

Primer kode:

```
#define PERFORM_ADD_CIO(d, co, r1, r2, ci)
  do { word __d = (r1) + (ci);
    word __cy = __d < (ci);
    (d) = __d + (r2);
    (co) = ((d) < __d) + __cy; } while (0)
```

(Primer kode se najde v datoteki superopt.def (vrstice 329-333))

3.1.1 Zagon

Kot je že omenjeno v 3.1, program moramo najprej prevesti preden lahko zaženemo program. Za to potrebujemo C prevajalnik (Linux že ima prevajalnik nameščen, za windows obstaja program Cygwin). Prevajanje zaženemo tako, da v ukazni vrstici zaženemo skripto datoteke Makefile in ji kot argument podamo ime procesorja, ki se nahaja na seznamu podprtih procesorjev. Ko je program preveden, ga zažene v ukazni vrstici z naslednjim formatom ukaza.

Format ukaza za zagon programa

```
superopt -f<goal-function> | -all  [-assembly] [-max-cost n]
[-shifts] [-extracts] [-no-carry-insns] [-extra-cost n]
```

V ukazni vrstici moramo vedno podati zastavico `-f` ali `-all`. Zastavica `-all` optimizira vse cilje ki smo jih prevedli iz datoteke goal.def. Pri zastavici `-f` moramo še podati cilj, ki ga mora optimizirat. Vse ostale zastavice so lahko nastavljene poljubno. Zastavica `-max-cost` nastavi največjo dolžino na

n, do katere program preiskuje preden konča (privzeta dolžina je 4). `-shifts` dovoljuje uporabo ukazov, ki premikajo bite levo ali desno. `-extracts` dovoljuje uporabo ukazov, ki delujejo na različno velikih podatkov. Zastavici `-shifts` in `-extracts` lahko močno zmanjšata učinkovitost optimizatorja. `-no-carry-insns` prepoveduje uporabo ukazov, ki uporabljajo prenosno zastavico v posebnem registru. Zastavica `-extra-cost` pove optimizatorju, da naj še izpiše rešitve, ki so za n večje od optimalne rešitve (Privzeta nastavitev je 0).

Primer ukaza:

```
superopt -fabs -shifts -extracts -max-cost 6
```

(ukaz kjer optimizator najde vse možne optimalne rešitve manjše za cilj abs, in pri tem vključuje še dodatne ukaze. Pri tem mora biti optimalna rešitev manjša od 6)

3.1.2 Način optimiziranja

V tem poglavju bo povzet način optimiziranja GSO optimizatorja omenjen v [1, 6, 7]. Program deluje na osnovi podanih ciljev in množici ukazov, programiranih v samem optimizatorju. Program skuša za te ciljev zgraditi zaporedje ukazov, ki je ekvivalenten tej kodi. Pri tem uporablja razširjeno iskanje, kar pomeni, da preizkuša vse možne kombinacije. Ker je lahko zaporedje različno dolgo, program začne z dolžino 1 in v primeru če ne najde rešitve, jo povečuje za 1. Program konča delovanje v primeru, če najde eno ali več rešitev, ali če preseže največjo dovoljeno dolžino, ki ga programer poda. Najdene rešitve potem izpiše v terminal. Po želji lahko program izpiše tudi v zbirnem jeziku. Program išče rešitev rekurzivno kar pomeni, da se iskanje nadaljuje v globino. Med generiranjem program uporablja operande oz. registre, ki so bili podani na vhodu ali generirani zaradi prejšnjih inštrukcij. Če prenosni bit ni nastavljen, potem instrukcije, ki uporabljajo ta bit, ne bodo generirani. Na začektu iskanja prenosni bit ni definiran in dolžina zaporedja ukazov je enaka nič. Pri vsaki rekurziji program pokliče funkcijo

`synth`, ki poišče list vseh definiranih ukazov in doda zaporedju en ukaz. Ker različne arhitekture zahtevajo različno število operandov, program mora imeti funkcije, ki bodo to dejstvo upoštevali. V tem primeru so težavo rešili z različnimi `synth` funkcijami, katerih lahko dostopajo le določene naprave. Pri tem je še treba poudariti, da se ukaz, ki preneša vrednost iz enega registra v drugi register, ne generira pri napravah, ki imajo 3-operandne ukaze, saj ukaz ni optimalen. Pri vsakem generiranem zaporedju program preveri njegovo delovanje na naključno izbranimi vrednosti, in če se rezultati ujemajo, potem nadaljuje z bolj obširnim preverjanjem. Če zaporedje ukazov opravi test, potem ga izpiše. Pri tem je zelo majhna verjetnost, da program izpiše tudi neekvivalentno kodo. Za pospešitev delovanja, optimizator uporablja manjhno število konstant v zaporedju (Npr. -1, 0, 1, $2^{31} - 1$ in 2^{31}). Posledica take omejitve, je povečana možnost, da optimizator zgreši optimizirano kodo. Kar je še treba poudariti, da kodo, ki jo program generira ne izvaja, na procesorju temveč jo z raznimi `PERFORM` metodami, dodeljen za vsak ukaz, simulira. Poleg osnovnega algoritma, obstjajo še nekatere predelane verzije, ki jih GSO v tem dokumentu nima implementiranih. Prva izmed idej je, da bi poleg uporabe testnih primerov, uporabili preverjanje z binarnim diagramov odločanja [8]. Idea takega iskanja ekvivalence je, da vsak bit v registru predstavlja boolean spremenljivko in vsaka operacija predstavlja odvisnost med biti pri izvedbi enega ukaza. Rezultat take predstavitve je graf, kjer vozlišče predstavlja boolean spremenljivko. Problem takega alogoritma se pojavi v predstavitvi bolj kompleksnih funkcij, kot so množenje in deljenje, saj je posledica tega zelo velik graf in posledično tudi program potrebije zelo veliko časa za dokazovanje ekvivalence. Druga ideje združuje poglavje o optimizaciji večvejitvenih poteh (Npr. `switch` stavki) in GSO. Dokument [9] opisuje kako naj bi optimizirali več vejitveno kodo z predelano verzijo superoptimizatorja. Superoptimizator naj bi vseboval predelano binarno drevo, ki prestavljalo vejitev skokov na določeno kodo. Vsaka točka v drevesu predstalja število ali operacijo in vsaka poveza bi predstavljal pot izpolnjenega ali neizponjega pogoja. Koren drevesa se vedno začne s številom in ima po-

vezavo do primerjalnega operatorja. Primerjalni operator potem to število uporabi za primerjanje vrednosti. V drevesu je možno, da sta na poti po drevesu navzdol zaporedno povezana dva primerjalna operatorja. V tem primeru bi algoritem vzel zadnjo prebrano vrednost na poti. Po končani optimizaciji bi optimizator nadaljeval z oprimizacijo vsake veje posebej. Še ena bolj zanimiva verzija je uporaba kombinacije superoptimizatorja in podatkovne baze [10]. Idea je, da program le generira avtomatično generira pravila, ter jih zapiše v podatkovno bazo, baza pa je kaseneje uporabjena za podporo prevajalniku pri optimizaciji kode. Optimizatorju je podan testni program, iz katerega vleče kodo, za optimizacijo. Vsaka izvlečena koda se izvede na testni napravi. Na koncu izvajanja, program zabeleži vse žive registre in stanje naprave. Za to stanje izračuna hash vrednost, ter jo s temu primernim zaporedjem ukazov v tabelo vstavi. Med izvajanjem tega algoritma, optimizator izvaja podoben algoritem kot pri GSO, ter generirano kodo izvede. Izvajanje poteka na dveh fiksnih stanji naprave. Za vsako generirano kodo, program poišče v tablei, če je v njej že shranjena izračunana hash vrednonst. V primeru najdene take vrednosti, optimizator preveri ekvivalenco in jo v primeru dokazane ekvivalence zapiše v podatkovno bazo. Če ne najde enake vrednosti, potem jo zavrže. Enako kot pri originalni različici, je tudi tu pri generiranju kode uporabljena omejena količina operacijskih kod. Poleg tega je potrebno vsako zaporedje posplošiti, saj bi bila količina primerov prevelika. Postopek posploševanja je relativno enostavno. Za vsak register in konstanto priredimo novo ime. Pri tem velja pravilo, da preimenujemo spremenljivke v zaporedju, kot jih koda izvaja. Pri tem algoritmu je še ena omejitev, vsaka izvlečena kone ne sme vsebovati točko vstopa v segment kode, kjer skok izvira iz segmenta. Poleg tega generirana kode ima le en skok izven segment. Pri je še dodatno omejena na največ 4 različne registre.

3.2 Primeri optimizacije

Tukaj je nekaj delov programa, ki jih je optimizator optimiziral in njihov pripadajoči optimiziran program. Vsi deli so majhni, saj optimizator porabi preprosto preveč časa z večje dele programa. Pri tem boste opazili, da je za primer podanih več funkcij ki pa imajo enako funkcijonalnost. Za prvi primer si lahko vzamemo del kode, ki izračuna absolutno vrednost podanega števila.

Primer dela kode:

```
return (v0 < 0 ? -v0 : v0);
```

Za tako kodo je progam našel 6 možnih rešitev, dolžine 4.

Ena izmed rešitev:

```
movl    %eax,%edx
sarl    $31,%edx
addl    %edx,%eax
xorl    %eax,%edx
```

Za naslednji test je optimizator iskal rešitev za enako funkcijonalnost, kot v prejšnjem testu, vendar koda je drugače napisana. Pri tej kodi manjka `else` stavek.

Primer Drugačne napisane kode:

```
if(x<0){x=-x;}
return x;
```

Enako kot za prejšnji primer, je optimizator našel 6 možnih rešitev, dolžine 4. Vse rešitve so enake kot v prejšnjem testu.

Ena izmed rešitev:

```
movl    %eax,%edx
sarl    $31,%edx
addl    %edx,%eax
xorl    %eax,%edx
```

Za naslednji test je bila podana funkcija, ki pove če je število manjše, večje ali enako številu 0. Pri tem je bila uporabljena naslednja koda.

Primer kode:

```
return (v0 > 0 ? 1 : (v0 < 0 ? -1 : 0));
```

Za takšen primer je optimizator poiskal 26 optimalnih kombinacij, dolžine 4.

Ena izmed rešitev:

```
addl    %eax,%eax
sbbl    %edx,%edx
subl    %eax,%edx
adcl    %eax,%edx
```

Za naslednji test je bila uporabljena funkcija z enako funkcijonalnostjo kot pri prejšnjem testu. Koda tokrat uporablja `else if` stavek in za vsak izpolnjen pogoj, takoj vrne rešitev.

Primer drugače zapisane kode:

```
if(x<0){return 1;}
else if(x>0){return -1;}
else{return 0;}
```

Pri tem primeru se najdejo razlike. Optimizator je našel optimalno rešitev šele pri dolžini 5. Pri tem je našel 245 možnih rešitev.

Ena izmed rešitev:

```
addl    %eax,%eax
movl    %eax,%edx
adcl    $-1,%edx
subl    %eax,%edx
sbbl    $-1,%edx
```

Naslednji podan primer nastavi 0, če je število večje ali enako 0, drugače 1.

Primer kode:

```
return v0 <= 0;
```

Pri tem primeru je optimizator našel 4 optimalne rešitve, dolžine 3.

Ena izmed rešitev:

```
cmpl    $1,%eax
sbbl    $0,%eax
shrl    $31,%eax
```

Kot pri ostalih poskusih, je tudi za ta test narejena še ena poskusna koda, ki pa tokrat vrača vrednosti, takoj ko je eden izmed pogojev izpolnjen.

Primer drugačne kode:

```
if(x<=0){return 1;}
else{return 0;}
```

Pri tem primeru je optimizator našel enake rešitve kot v prejšnjem primeru.

Ena izmed rešitev:

```
cmpl    $1,%eax
sbbl    $0,%eax
shrl    $31,%eax
```

Za zadnji primer je optimizator optimiziral zelo enostavno kodo. Koda vrne število, ki je za 4 večje od danega števila.

Primer kode:

```
return x+4;
```

Če pogledamo to kodo, bi ugotovili, da bi lahko tako kodo zapisali le z enim ukazom, vendar, če tako kodo damo optimizatorju, ugotovimo, da optimizator ne najde take rešitve. Najde pa rešitve, ki so dolžine 4.

Primer rešitve optimizatorja:

```
addl    $1,%eax
addl    $1,%eax
addl    $1,%eax
incl    %eax
```

Primer optimalne rešitve:

```
addl    $4,%eax
```

3.3 Prednosti in slabosti

Vsak optimizator ima svoje prednosti in slabosti in GNU-Superoptimizer ni nobena izjema. Toda pri tem optimizatorju je več slabosti kot prednosti. Največja prednost tega optimizatorja je, da vedno najde optimalen program, če upoštevamo nabor ukazov, ki jih lahko izpiše. Optimizator je tako osnovan, da lahko dodajamo nabor ukazov za druge vrste arhitekture. Program je tudi dovolj stabilen za uporabo in ga je enostavno namestiti. Prednost je tudi v tem, da najdeni programi ponavadi ne vsebujejo pogojne skoke, ki upočasnjujejo program. Tukaj pa se prednosti končajo. Največja slabost takega optimizatorja je časovna zahtevnost. Časovna zahtevnost takega programa je O($m * n^{2n}$), kjer m predstavlja število ukazov in n je najmanjša dolžina, ki predstavlja ciljno funkcijo. Optimizator zna uporabljat le podan nabor operacij [11], poleg tega pa ni fleksibilen, če se v originalni kodi pojavijo konstante, ki jih optimizator ne podpira pri iskanju. (Primer je omenjen v točki 5.2).

Poglavje 4

STOKE

Leta 2013 so razvili stohastičen superoptimizator. Stohastičen optimizator uporablja brez zančni superoptimizacijsko nalogo kot stohastični problem iskanja. Konkurenčne omejitve od preoblikovanja pravilnosti in izboljševanja delovanja so kodirani kot izrazi v odvisnosti stroškov, ter Markov Chain Monte Carlo vzorčevanje se uporablja za hitro raziskovanje prostora vseh možnih programov, da najde tisti program, ki je optimizacija danega ciljnega programa. Čeprav njihova metoda žrtvuje popolnosti, obseg programov, programe ki jih sposoben preučiti in kakovost programov, ki jih posledično proizvaja, daleč presegajo tiste proizvedene od obstoječih superoptimizatorjev. Že binarne datotek, prevedene z llvm -O0 izbranim za 64-bitne x86, njihova izvedba prototipa, STOKE, je sposobna proizvajati programe, ki bodisi so enako dobre ali prekašajo kodo, ki jih generira gcc -O3, icc -O3, in v nekaterih primerih tudi strokovno lastnoročno napisano kodo [2, 3].

4.1 Opis programa in načina delovanja

Program je napisan v več različnih jezikih (C,C++ in Python). Deluje na Haswell procesorji z AVX in AVX2 razširitvijo in Sandy Bridge sistemi z AVX razširitvijo. Trenutna različica deluje na operacijskem sistemu Ubuntu od verzije 12.04 do vključno z verzijo 15.10 (program jo možno pognati tudi

v drugih linux operacijskih sistemih). Tako kot v poglavju 3 moramo tudi v ta superoptimizator najprej prevesti preden ga lahko uporabljamo. Celoten program vsebuje še knjižnjice iz drugih repozitorijev (npr. knjižnica x64asm, cpputil, z3). Vse knjižnjice se povežejo z zagonom datoteke Makefile, ki jo lahko najdemo v glavni mapi stoke. Superoptimizator deluje preko ukazne vrstice. Najprej izvleče iz prevedenega programa vse funkcije, ki jih lahko najde. Potem naredi nekaj poskusov in rezultate zapiše v datoteko. Kasneje to datoteko uporabi za vodenje iskanja do optimalnega programa. Če najde program, ga potem izpiše v datoteko v zbirnem jeziku. Datoteka vsebuje tudi podatke, kje je bila najdena funkcija, kako dolgi so ukazi, s čim je ukaz predstavljen v optimizatoju in koliko je dolg ukaz. V datoteko lahko tudi sami vnesemo določene izjeme ki se upoštevajo pri optimizaciji (Npr. velikost podatka v operandu). Po optimizaciji ima uporabnik možnost, da optimizirano kodo vstavi nazaj v preveden program. Za vsak izmed procesov od branja funkcij do vstavljanje optimizirane kodo je potreben uporabnikov vhod podatkov.

4.1.1 Način optimiziranja

STOKE uporablja dva glavna načina optimiziranja programske kodo. Prvi način je sestavljanje programa. Pri sestavljanju programa optimizator poskuša najti ukaz ali skupino ukazov, ki so funkcijonalno enako kot ukazi v programu, vendar je zapis precej drugačen od prvotnega. Drugi način optimiziranja, je optimizacija že obstoječe kode. Prvi način ima prednost pri krajših programih, saj lahko hitro in z večjo verjetnostjo najde optimiziran program. Drugi program pa ima prednost pri večjih programih, saj že ima izhodiščni program, ki ga lahko optimizira. Pri iskanju optimalnega programa optimizator uporablja Metropolis-Hastings algoritem. Metropolis-Hastings algoritem je iskanje, ki uporablja verjetnost za iskanju globalnega optimuma. Metropolis-Hastings algoritem kot vhodni podatek potrebuje neko zaporedje podatkov (v tem primeru je to program, ki bi ga radi optimizirali), funkcijo vrednosti (št. urinih ciklov) in definicijo sosedov (zamenjava skupino ukazov,

ki so funcijonalno enaki z drugo skupino). Rešitev sprejme v dveh primerih, če je bolj ugodna od prejšnje rešitve ali pa če je naključno število manjše od izračunane verjetnosti iz izraza 4.1, kjer je rez' ocenjena vrednost tranformacije kode in rez ocenjena vrednost prvotne kode in β pdana konstanta s strani uporabnika [12, 2].

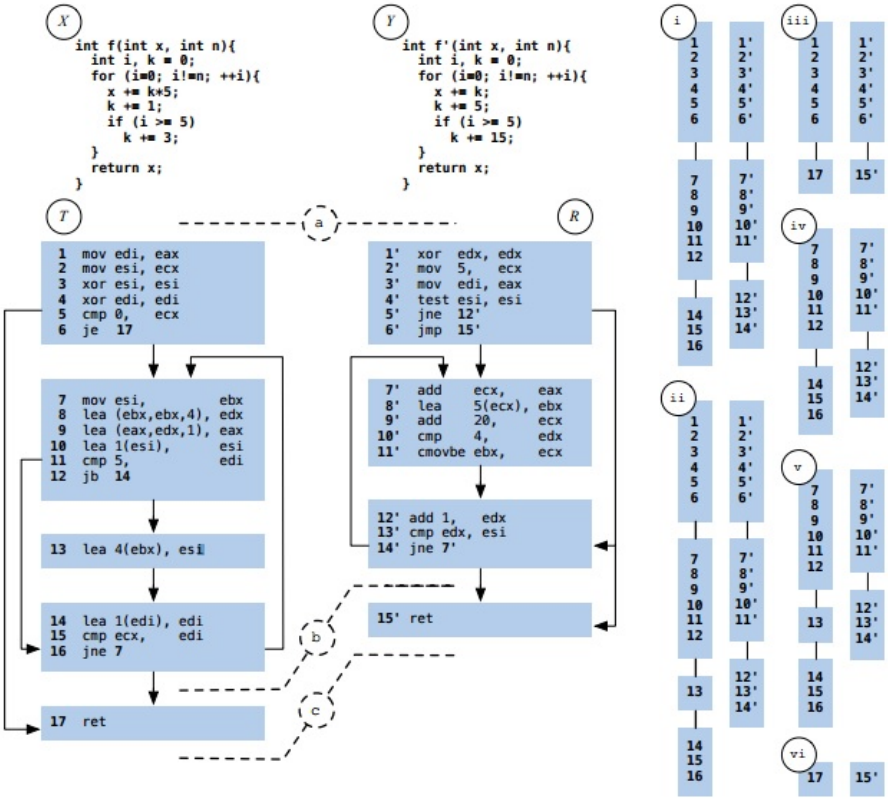
$$e^{-\beta * rez'/rez} \tag{4.1}$$

Tako iskanje zagotavlja, da se vedno približa enemu od globalnih maximumov. Pri iskanju ekvivalence, program ima implementiran podatkovno vodeno preverjanje ekvivalence (Data-Driven Equivalence Checking [13]), ki uporablja točke rezanja, ki kodo z zankami razdelijo v brezzančno kodo. Vsaka točka rezanja vsebuje par lokacij. Vsaka lokacija v paru je v svojem programu. Program generira točke rezanja z izčrpnim (brute force) iskanjem, kjer gleda razlike v stanju naprave.

$$\exists a, b \in \mathbb{N}, \forall \sigma, m_t = a * m_r + b \tag{4.2}$$

Za vse točke mora veljati izraz 4.2, kjer sta a in b konstanti, σ podanp stanje naprave, m_t ozančuje koliko krat gre izvajanje skozi točko v izvirnem programu in m_r ozančuje koliko krat gre izvajanje skozi točko v dobljenem programu. Program tudi zavrne točke, kje število lokacij v pomnilniku ni konstanten. Preostale točke izbira glede na razlike v stanju naprave, od najmanjše do največje. Izbiranje poteka, dokler ne dobimo brezzančne segmente kode. Nazadnje odstrani vse točke, kjer segmenti še vedno ostanejo brezzančni. Recimo da imamo originalen program T in program skuša preveriti ekvivalenco programa R. Program najprej naredi kočke rezanja `a`, `b` in `c`. Te toćke označujejo segmente od `a` do `b` (rez kode v zanki), od prvega ukaza v zanki, do `b` in od `b` do `c`. S pomočjo take razdelitve lahko optimizator induktivno dokaže, da je dobljen program ekvivalenten izvirnemu. Pri dokazovanju ekvivalence morajo veljati naslednja dejstva. Na žačetku kode oz. v točki `a` mora biti za obe programski kodi stanje naprave enako. Pri tem še mora veljati, da je končna vrednost v registru za vračanje vrednoti (V tem primeru `eax`) enaka. Program še mora določiti relacijo I med elementi med

stanji naprave. Ko je relacija določena, žačne preverjati. Najprej preveri če se tranzicija iz `a` do `c` z enakim stanjem naprave konča z enako vrnjeno verdnostjo. Potem preveri če pri tranziciji iz `a` do `b` obvelja relacija I. Kasneje preveri trazicijo od `b` do `b`, če relacija še vedno velja. Na koncu preveri če tranzicija med `b` in `c`, ki zadovoljuje relacijo I in vrne enako končno vrednost [14]. Optimizator opravi preverjanje z izvajanjem ukazov, ki so dodeljeni vsaki tranziciji.

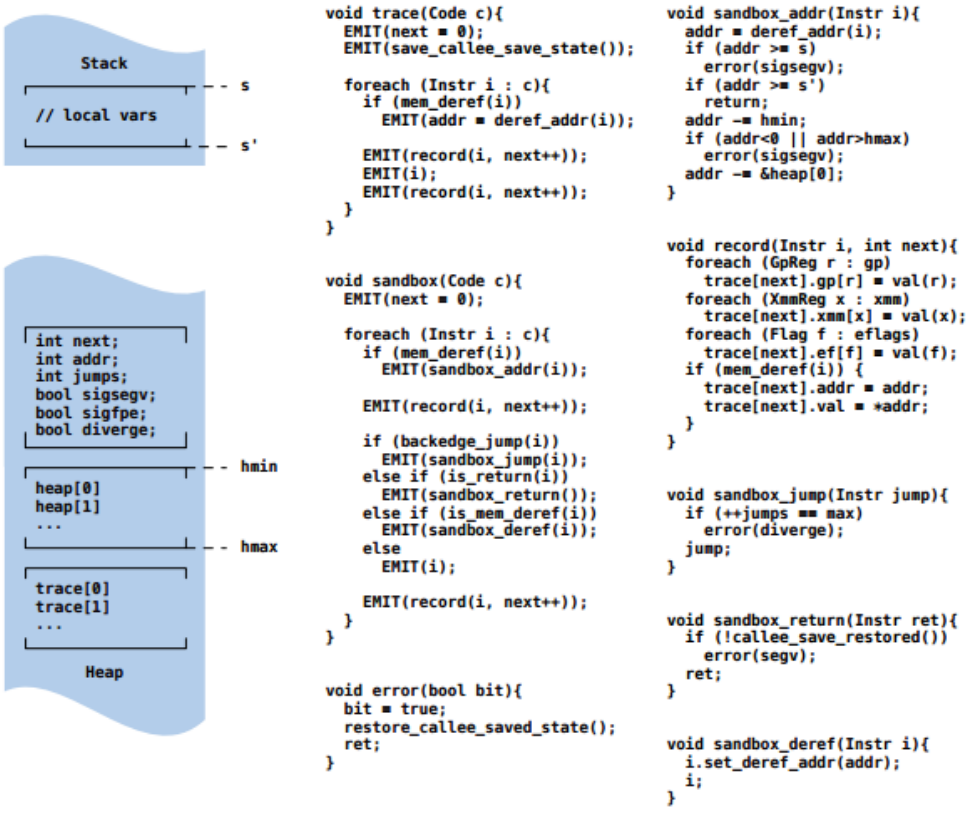


Slika 4.1: Preverjanje ekvivalence med izvirnim in optimiziranim programom. Točke `a`, `b`, `c` označujejo lokacije rezanja. `i-vi` označujejo sekvenco izvajanje kode za vsako izmed trazicij

V primeru programa na sliki 4.1 sekvenci `i` in `ii` pripadata tranziciji iz `a` v `b`, sekvenca `iii` pripada tranziciji iz `a` v `c`, sekvenci `iv` in `v` pripadata

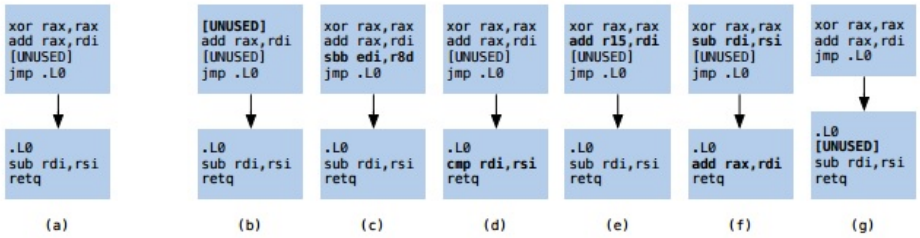
tranziciji iz `b` v `b` in sekvenca `vi` pripada tranziciji iz `b` v `c`. Problem, ki se pojavi med izvajanjem je, kakšne invarjante mora točka `b` imeti, da bi statistično dokazal, da izvajanje zaporedja R sledi izvajanju ustreznemu zaporedju programa T in da izvajanje obeh programov gre skozi enake koščke. Rešitev takega problema se najbe v analizi podatkov. Program indentificira pot z ujemanjem sledi obeh zank na enakem vhodu podatkov na množici točk rezanj. Za pogoje relacije pri točki `b` lahko program preveri vrednosti živih registrov dobljene pri izvajanju obeh programov. Pri tem pridobi matrico, v kateri prva vrstica vsebuje vrednosti registrov, ko prvič doseže točko `b` in druga vrstica vsebuje vrednosti, ko drugič doseže točko `b`. Iz tega lahko izvleče relacijo med vrednosti s pomočjo linearne algebre tako, da izračuna ničelni prostor matrice s pomočjo knjižnice Integer Matrix Library. Tak algoritem predpostavlja, da koda vsebuje le eno zanko, vendar je to možno trivialno razširiti na primere z več zank in z notranjimi zankami tako, da za vsak kos kode ponovno izvedemo rezanje. V algoritmu je tudo možno razširiti monžico invarjant, tako da določimo največjo stopnjo monoma in dodamo vse monome, ki so manjše od the stopnje. Ker lahko ne-linearen invarianta predstalja disjunkcijo linearnih enkakovrednosti, je lahko tudi v množici invarjant vsebovana tudi določeno število disjunktivnih izrazov enakovrednosti. Disjuktni izraze rešuje z že obstoječim algoritmom Z3 [15]. Pri računanju živih registrov DDEC upošteva, da na 64 bitnih x86 arhitekturah za nekatere registre obstaja množica imenov. Vsako ima naslavja en del registra (Npr. prvih 8, 16, 32 bitov). Kadar ukaz zahteva pisannje v en del reistra, program označi cel rigister kot živega. Da lahko program opazuje vrednosti živih registrov pri točkah rezanja, se zanaša na sposobnost opazovanja ciljne lokacje in dinamično spreminjanje vrednosti s pomočjo zbirnika JIT. Pred izvajanjem optimizacije, DDEC rezervira del pomnilnika. Pri izvajanju vsakega ukaza optimizator pošlje signal za klic funkcije `record()`, ki shrani stanje naprave in vrednosti, ki so bile prebrane ali napisane v pomnilniku. Ko se izvajanje ciljne funkcije konča, tabela sledov vsebuje vse zapise manipulacije vrednosti na registrih in pomnilniku. Problem lahko

nastane, da nekateri kandidati za ponovn zapis lahko kažejo na neveljano lokacijo v pomnilniku. Rešitev je se skriva v funkciji `sandbox()`. Pri izvajanju programa program pridobi največje velikost alociranih podatkov v pomnilniku in minimalen in maksimalen naslov v pomnilniku. Pridobljene vrednosti uporabi za rezervacijo drugega dela pomnilnika, da zavaruje izvajanje optimiziranega programa. Velikost podatkov rezervirenega pomnilnika ne presega velikosti rezerviranega izvirnega programa. Kazalci na pomnilniško mesto so zavarovano z funkcijo `sandbox_addr()`, ki poskrbi da dostop do pomnilnika znotraj meja in blokira vse vstale dostope in ob primeru blokade ustavi izvajanje. Če je potrebno, lahko delež rezerviranega pominilnika povečamo. Poleg tega se optimizator zavaruje pred nastalimi neskončnimi zankami s klicom `sandbox_jump`, ki prešteje kolikokrat je program skočil nazaj in prekine izvajanje, če preseže maksimalno število skokov nazaj. Program mora še poskrbeti, da se konča z ukazov za vrnitev na mesto izvajanje pred klicem, le če so invarjante izpolnjene. Za to poskrbi funkcija `sanbox_return()`. Zaradi strukture in načina delovanja, algoritem dopušča uporabo še za reševanje drugih težav (Npr. avtomatično učenje množice ukazov glede na podano osnovno množico [16]). STOKE trenutno vsebuje 8 transformacij. Prva transformacija se imenuje `add_nops`. Transformacija doda `nop` ukaz, ki ne naredi ničesar. Naslednja tranformacija je delete, ki odstrani en ukaz. Potem je transformacija `instruction`, ki zamenja instrukcijo z drugo naključno. Poleg zamenjave instrukcije obstaja transformacija `opcode` in `operand`, ki zamenjata le operacijsko kodo oz. operand. Naslednja se imenuje `rotate`, ki vzame en ukaz iz segmenta in ga doda drugemu segmentu, ne da bi spremenil zaporedje ostalih ukazov. Potem ob obstajata textttlocal_swap in textttglobal_swap, ki zamenjata dva ukaza med seboj. Rezlika je v tem, da sta ukaza, ki ju menjamo, v istem segmentu, v druigi pa nista. Transformacije dovoljujejo, da so lahko neuspešne. STOKE ima plementiran algoritem za optimizacijo računanja s števili s plavajočo vejico. Kot je v ploglavju 2.1 omenjeno, optimizacija števil s plavojočo vejico ni preprosta, saj je v registrih in pomnilnikih shranjena zaokrožena vrednost, ki je najbližji napisani vrednosti v izvirni kodi. Še bolj



Slika 4.2: Implementacija za sledenje izvajanje programov in izpisanih programov s pomočjo JIT zbirnika

se dejstvo zakomplicira, ko upoštevamo, da se med vsako operacijo število zaokrožuje [4]. Zato je v optimizatorju implementiran algoritem, ki preverja odstopanje med izvirnim in dobljenim programom. Pri računanju odstopanja pa se pojavi problem. Najbolj razširjena načina za računanje odstopa, sta relativna in absolutna razlika. Absolutno razliko dobimo tako, da iz dobljene vrenosti odštejemo prvotno vrednost. Če pa abolutno razliko delimo z prvotno vrednostjo, dobimo relativno vrednost. Oba načina imata problem pri računanju odstopanja. Absolutna razlika se v večjimi števili povečuje, Relativna vrednost, pa se povečuje pri denomlaliziranih številih. Tak pro-



Slika 4.3: Grafični pregled transformacij. Vsaka izmed primerov predstavlja tranformacijo: a) trenutni program, b) `delete`, c) `instruction`, d) `opcode`, e) `operand`, f) `global_swap`, g) `resize`. Unused lahko zamenjamo z ukazom `nop`

blem se še bolj povečuje z računanjem ne-sosednjih števil. Zato je za izračun odstopanja uporabljen algoritem negotovost v zadnjem mestu (uncertainty in the last place), ki izmeri razliko med realnim številom in najbližjim predstavljen številu s plavojoćčo vejico v registru. V tem primeru STOKE računa razliko med stare in nove vrednosti v registru v ciljnem programu. Prednost take funkcije je, da lahko razliko izračuna tudi, če je kot argment podan noskončno ali nedefinirano število. ULP v C-ju potrebuje dva argumenta tipa `double`. Funkcija najprej predefinira ragumente, tako da shranjeno vrednost v spremenljivki prebere kot celoštevilsko vrednost in vrne absolutno razliko med argumentoma. Pri tem še popravi porazdelitev števila s plavajočo vejico od minus neskončno do 0 tako da od celoštevilske vrednosti odšteje najmanjše predznačeno število, ki jo lahko celoštevilska vrednost zavzeme, saj prvotno bi najmanjša vrednost predstavljala negativno ničlo in 0 bi prestavljala negativno nedefinirano število. Skupno odstopanje med izvirnim in dobljenim programom izračuna tako, da vzeme maksimalno odstopanje enega podatka, saj s tem prepreči možnost preliva pri seštevanju takih števil. STOKE sprejme končen program v primeru, če ne odstopa za več kot n, ki je določena s strani uporabnika.

4.1.2 Zagon

Kot je že omenjeno v 4.1, za zagon potrebujemo linux sistem (najprimernejši Ubuntu 14.02). Zraven še moramo imeti nameščenih nekaj knjižnjic in programov. Vse potrebne programe lahko namestimo s pomočjo ukaza `apt-get` v ukazni vrstici. Pri tem še moramo preveriti, če je gcc prevajalnik (C prevajalnik) in g++ (C++ prevajalnik) verzije 4.9, sicer prevajalnika ne bomo mogli sestaviti. Najprej moramo s pomočjo `git` ukaza prenesti datoteke z repozitorija.

Git ukaz:

```
git clone https://github.com/StanfordPL/stoke
```

Ko je prenos končan, moramo nastaviti konfiguracijo za prevajanje optimizatorja v delujoč program. V glavni mapi stoke je skripta, ki nam to avtomatsko naredi, nato poženemo datoteko Makefile.

Zaporedje ukazov v ukazni vrstici:

```
./configure.sh
make
```

Datoteka Makefile bo poskrbela za vse konfiguracije in prevajanje programov. Da lahko program poženemo iz kateregakoli direktorija moramo nastaviti globalno spremenljivko `PATH` v operacijskem sistemu.

Ukaz v ukazni vrstici:

```
export PATH=$PATH:/<pot_do_stoke>/bin
```

(`<pot_do_stoke>` predstavlja direktorij, ki se nahaja v mapi stoke)

Pred zagonom potrebujemo že posebej preveden program. Optimizator poženemo z ukazno vrstico. Pri tem moramo podati argumente. Argumente lahko podamo na dva načina. Prvi način je, da podamo argumente v ukazni vrstici. Drugi način je, da podamo datoteko ki ima v njej zapisane argumente.

Primer ukaza v ukazni vrstici za prvi način:

```
stoke extract -i ./a.out -o bins
```

Primer ukaza v ukazni vrstici za drugi način:

```
stoke extract --config extract.conf
```

Primer datoteke extract.conf:

```
-i ./a.out  # Lokacija in ime datoteke, ki želimo optimizirati
-o bins     # Lokacija kjer shrani izvlečeno kodo
```

(Zank `#` označuje kdaj se komentar v vrstici začne.)

Primer kode, ki jo bomo obdelali:

```
#include <cstdlib>
#include <stddef.h>
#include <stdint.h>

using namespace std;

size_t popcnt(uint64_t x) {
        int res = 0;
        for ( ; x > 0; x >>= 1 ) {
                res += x & 0x1ull;
        }
        return res;
}

int main(int argc, char** argv) {
        const auto itr = atoi(argv[1]);

        auto ret = 0;
        for ( auto i = 0; i < itr; ++i ) {
                ret += popcnt(i);
        }

        return ret;
}
```

(Koda poišče vse bite, ki vsebujejo vrednost 1 od števila 0 do `itr` v bitni reprezentaciji, kjer `itr` podan kot prvi argument.)

Potem moramo generirati nekaj poskusnih primerov za določeno funkcijo. Ti primeri bodo uporabljeni za preverjanje pravilnosti kode. Funkcije najdene v programu imajo drugačna imena kot v izvirni kodi, zato je potrebno pogledati v mapi, kjer je bila izvlečena koda, kako se imenuje. Ponavadi je ime funkcije v izvirni kodi vsebovana v imenu datoteke.

Primer ukaza:

```
stoke testcase --config testcase.conf
```

Primer izgleda datoteke testcase.conf:

```
--bin ./a.out        # Lokacija in ima datoteke prevedenega programa
--args 10000000      # Argument, ki so podani za ./a.out
--functions bins     # Lokacija, kjer je shranjena izvlečena funkcija

-o popcnt.tc         # Pot do datoteke kjer naj shrani poskusne teste

--fxn _Z6popcntm     # Ime funkcije, za katerega naredi poskusne teste
--max_testcases 1024 # Maximalno število testov.
```

Primer datoteke _Z6popcntm:

```
.text
.globl _Z6popcntm
.type _Z6popcntm, @function #ime funkcije

#! file-offset 0x580
#! rip-offset  0x400580
#! capacity    48 bytes

# Text            #  Line  RIP       Bytes  Opcode
._Z6popcntm:      #        0x400580  0      OPC=<label>
testq %rdi, %rdi  #  1     0x400580  3      OPC=testq_r64_r64
je .L_40059f      #  2     0x400583  2      OPC=je_label
xorl %eax, %eax   #  3     0x400585  2      OPC=xorl_r32_r32
nop               #  4     0x400587  1      OPC=nop
nop               #  5     0x400588  1      OPC=nop
.L_400590:        #        0x400590  0      OPC=<label>
movl %edi, %edx   #  13    0x400590  2      OPC=movl_r32_r32
```

```
andl $0x1, %edx   #  14    0x400592  3      OPC=andl_r32_imm8
addl %edx, %eax   #  15    0x400595  2      OPC=addl_r32_r32
shrq $0x1, %rdi   #  16    0x400597  3      OPC=shrq_r64_one
jne .L_400590     #  17    0x40059a  2      OPC=jne_label
cltq              #  18    0x40059c  2      OPC=cltq
retq              #  19    0x40059e  1      OPC=retq
.L_40059f:        #        0x40059f  0      OPC=<label>
xorl %eax, %eax   #  20    0x40059f  2      OPC=xorl_r32_r32
retq              #  21    0x4005a1  1      OPC=retq
nop               #  22    0x4005a2  1      OPC=nop

.size _Z6popcntm, .-_Z6popcntm
```

(Zank # označuje kdaj se komentar v vrstici začne. Zgoraj navedena datoteka je skrajšana. Vsi manjkajoči ukazi so ukazi, ki ne naredijo ničesar)

V datoteki (v tem primeru popcnt.tc) je izpisano stanje programa pred vstopom v funkcijo. Prvih 60 vrstic označuje stanje vseh registrov. Naslednjih nekaj vrstic opisuje stanje v delu pomnilnika, kjer se nahajajo kazalci za delovanje programa.

Primer datoteke _Z6popcntm:

```
Testcase 0:

%rax     00 00 00 00 00 98 96 80
%rcx     00 00 00 00 00 00 00 00
%rdx     00 00 00 00 00 00 00 0a
%rbx     00 00 00 00 00 00 00 01
%rsp     00 00 7f ff 97 44 36 28
%rbp     00 00 00 00 00 00 00 00
%rsi     19 99 99 99 99 99 99 99
%rdi     00 00 00 00 00 00 00 00
%r8      00 00 2a c9 68 1a 50 40
%r9      00 00 7f ff 97 44 46 01
%r10     00 00 00 00 00 98 96 80
%r11     00 00 00 00 00 00 00 0a
%r12     00 00 00 00 00 98 96 80
%r13     00 00 7f ff 97 44 37 20
%r14     00 00 00 00 00 00 00 00
%r15     00 00 00 00 00 00 00 00
```

```
%ymm0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                  00 00 00 00 00 00 00 00 00 00 00 00 00 00 ff 00 00
%ymm1    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         2f 2f 2f 2f 2f 2f 2f 2f 2f 2f 2f 2f 2f 2f 2f 2f
%ymm2    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
             00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm3    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                     00 00 00 00 00 00 00 ff 00 00 00 00 00 00 00 ff
%ymm4    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm5    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm6    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm7    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm8    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm9    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm10   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm11   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm12   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm13   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm14   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
%ymm15   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

%cf      0
%1       1
%pf      1
%0       0
%af      0
%0       0
%zf      0
```

```
%sf      0
%tf      0
%if      1
%df      0
%of      0
%iopl[0] 0
%iopl[1] 0
%nt      0
%0       0
%rf      0
%vm      0
%ac      0
%vif     0
%vip     0
%id      0

 [ 00007fff 97443630 - 00007fff 97443620 ]
 [ 1 valid rows shown ]

 00007fff 97443628   d d d d d d d d   00 00 00 00 00 40 04 6c

 [ 00000000 00000000 - 00000000 00000000 ]
 [ 0 valid rows shown ]

 [ 00000000 00000000 - 00000000 00000000 ]
 [ 0 valid rows shown ]

 0 more segment(s)
```

(registri %ymm od 0 do 15 so v datoteki napisani vse v eni vrstici. Tukaj so napisani v dveh zaradi preglednosti.)

Šele zdaj lahko poženemo optimiziranje. Pri tem še moramo upošteviti, da različne kode lahko zahtevajo različno konfiguracijo. Paziti moramo na to, kateri registri so definirani pred vhodom v funkcijo in kateri registri so definirani ob izhodu iz funkcije. Pozorni moramo biti tudi na število iteracij za ponovni začetek in na maksimalno število iteracij. Večji programi potrebujejo večjo število iteracij za optimizacijo kot manjši programi.

Primer ukaza:

```
stoke synthesize --config synthesize.conf
```

Primer izgleda datoteke testcase.conf:

```
--out result.s               # Pot kjer naj napiše rezultat

--target bins/_Z6popcntm.s   # Pot do funkcije za optimizacijo

--def_in "{ %rax %rdi }"     # Registri ki so živi na začetku
--live_out "{ %rax }"        # Registri ki so živi na koncu

--testcases popcnt.tc        # Datoteka z poskusnimi testi
--training_set "{ 0 ... 7 }" # Testi za merjenje pravilnosti med iskanjem
--test_set "{ 8 ... 1023 }"  # Testi za preverjanje pravilnosti

--distance hamming           # Način za merejenje napak za izhodne ragistre
--misalign_penalty 1         # Kazen za razultate, ki se nahajajo
                             # na napačnem mestu
--reduction sum              # Metoda za izračun nepravilnosti
--sig_penalty 9999           # Točke za rezultat, ki povzročijo neničelne
                             # signale

--cost "correctness + latency" # Način izračuna izvedbe

--global_swap_mass 0         # Nastavitev cene
--instruction_mass 1         # Nastavitev cene
--local_swap_mass 1          # Nastavitev cene
--opcode_mass 1              # Nastavitev cene
--operand_mass 1             # Nastavitev cene
--rotate_mass 0              # Nastavitev cene

--beta 1                       # Konstanta za izračun sprejemljivosti
--initial_instruction_number 5 # Število nop operacij

--statistics_interval 100000  # Printanje statistike na vsakih n iteracij
--timeout_iterations 16000000 # Nastavitev število operacij preden konča
--cycle_timeout 1000000       # Ponovni začetek vsakih n iteracij

--strategy hold_out           # Določa strategijo preverjanje
```

Po končanem optimiziranju, optimizator napiše optimizirano kodo v da-

toteko, podano kot argument. Optimizator ima možnost, da optimizirano kodo vstavi nazaj v že preveden program.

Primer ukaza:

```
stoke replace --config replace.conf
```

Primer izgleda datoteke replace.conf:

```
-i ./a.out         # Pot do prevedenega programa
--rewrite result.s # Pot do optimizirane funkcije
```

4.2 Primeri optimizacije

Tukaj je nekaj delov programa, ki jih je optimizator optimiziral in njihov pripadajoči optimiziran program. Testiranje deluje v dveh delih. Prvi del je optimiziranje kratkih programov (enakih kot v 5.2), drugi del pa vsebuje poskuse na večjih programih. Taka delitev nam bo dala vpogled, kako dobro se je STOKE odrezal v primerjavi z GNU-Superoptimizer, pa čeprav pri GNU-Superoptimizer-ju nimamo rezultatov za večje programe.

4.2.1 Optimizacija krajših programov

Ko je že omenjeno v poglavju 4.2, v tem poglavju so rezultati optimizacij krajših programov. Vsi primeri kod so enaki kot v točki 5.2. Prvi primer optimizacije je funkcija, ki izračuna absolutno vrednost nekega števila.

Primer dela kode:

```
return (v0 < 0 ? -v0 : v0);
```

Optimizator je poskušal najti optimalno kodo. To je naslednji rezultat.

Predlagana rešitev:

```
.text
.globl _Z5abss2l
.type _Z5abss2l, @function
```

```
#! file-offset 0x690
#! rip-offset  0x400690
#! capacity    32 bytes

# Text            #  Line  RIP       Bytes  Opcode
._Z5abss2l:       #        0x400690  0      OPC=<label>
movq %rdi, %rdx   #  1     0x400690  3      OPC=movq_r64_r64
movq %rdi, %rax   #  2     0x400693  3      OPC=movq_r64_r64
sarq $0x3f, %rdx  #  3     0x400696  4      OPC=sarq_r64_imm8
xorq %rdx, %rax   #  4     0x40069a  3      OPC=xorq_r64_r64
subq %rdx, %rax   #  5     0x40069d  3      OPC=subq_r64_r64
retq              #  6     0x4006a0  1      OPC=retq
nop               #  7     0x4006a1  1      OPC=nop

.size _Z5abss2l, .-_Z5abss2l
```

Za naslednji test je optimizator iskal rešitev za enako funkcijonalnost, kot v prejšnjem testu, vendar koda je drugače napisana. Pri tej kodi manjka `else` stavek.

Primer Drugačne napisane kode:

```
if(x<0){x=-x;}
return x;
```

Za ta test je optimizator dobil enak rezultat kot pri prejšnjem testu.

Ena izmed rešitev:

```
.text
.globl _Z4abssl
.type _Z4abssl, @function

#! file-offset 0x640
#! rip-offset  0x400640
#! capacity    32 bytes

# Text            #  Line  RIP       Bytes  Opcode
._Z4abssl:        #        0x400640  0      OPC=<label>
movq %rdi, %rdx   #  1     0x400640  3      OPC=movq_r64_r64
movq %rdi, %rax   #  2     0x400643  3      OPC=movq_r64_r64
```

```
sarq $0x3f, %rdx  #  3     0x400646  4      OPC=sarq_r64_imm8
xorq %rdx, %rax   #  4     0x40064a  3      OPC=xorq_r64_r64
subq %rdx, %rax   #  5     0x40064d  3      OPC=subq_r64_r64
retq              #  6     0x400650  1      OPC=retq
nop               #  7     0x400651  1      OPC=nop

.size _Z4abssl, .-_Z4abssl
```

Za naslednji test je bila podana funkcija, ki pove če je število manjše, večje ali enako številu 0. Pri tem je bila uporabljena naslednja koda.

Primer kode:

```
return (v0 > 0 ? 1 : (v0 < 0 ? -1 : 0));
```

Za ta test lahko opazimo, da optimizator ne obide nekaterih pogojnih skokov, ki bi jih sicer lahko obšli.

Ena izmed rešitev:

```
.text
.globl _Z4sgn2l
.type _Z4sgn2l, @function

#! file-offset 0x6b0
#! rip-offset  0x4006b0
#! capacity    32 bytes

# Text            #  Line  RIP       Bytes  Opcode
._Z4sgn2l:        #        0x4006b0  0      OPC=<label>
cmpq $0x0, %rdi   #  1     0x4006b0  4      OPC=cmpq_r64_imm8
movl $0x1, %eax   #  2     0x4006b4  5      OPC=movl_r32_imm32
jle .L_4006c0     #  3     0x4006b9  2      OPC=jle_label
retq              #  4     0x4006bb  1      OPC=retq
nop               #  5     0x4006bc  1      OPC=nop
.L_4006c0:        #        0x4006c0  0      OPC=<label>
setne %al         #  9     0x4006c0  3      OPC=setne_r8
movzbl %al, %eax  #  10    0x4006c3  3      OPC=movzbl_r32_r8
negq %rax         #  11    0x4006c6  3      OPC=negq_r64
retq              #  12    0x4006c9  1      OPC=retq
nop               #  13    0x4006ca  1      OPC=nop

.size _Z4sgn2l, .-_Z4sgn2l
```

Za naslednji test je bila uporabljena funkcija z enako funkcijonalnostjo kot pri prejšnjem testu. Koda tokrat uporablja `else if` stavek in za vsak izpolnjen pogoj, takoj vrne rešitev.

Primer drugače zapisane kode:

```
if(x<0){return 1;}
else if(x>0){return -1;}
else{return 0;}
```

Ta poskus ni prinesel nobenih sprememb.

Ena izmed rešitev:

```
  .text
  .globl _Z3sgnl
  .type _Z3sgnl, @function

  #! file-offset 0x660
  #! rip-offset  0x400660
  #! capacity    32 bytes

  # Text            #  Line  RIP       Bytes  Opcode
  ._Z3sgnl:         #        0x400660  0      OPC=<label>
  cmpq $0x0, %rdi   #  1     0x400660  4      OPC=cmpq_r64_imm8
  movl $0x1, %eax   #  2     0x400664  5      OPC=movl_r32_imm32
  jle .L_400670     #  3     0x400669  2      OPC=jle_label
  retq              #  4     0x40066b  1      OPC=retq
  nop               #  5     0x40066c  1      OPC=nop
  .L_400670:        #        0x400670  0      OPC=<label>
  setne %al         #  9     0x400670  3      OPC=setne_r8
  movzbl %al, %eax  #  10    0x400673  3      OPC=movzbl_r32_r8
  negq %rax         #  11    0x400676  3      OPC=negq_r64
  retq              #  12    0x400679  1      OPC=retq
  nop               #  13    0x40067a  1      OPC=nop

  .size _Z3sgnl, .-_Z3sgnl
```

Naslednji podan primer nastavi 0, če je število večje ali enako 0, drugače 1.

Primer kode:

```
return v0 <= 0;
```

Optimizator najde optimalno kodo dolžine 3 inštrukcije, brez upoštevanja zadnjega `retq` ukaza, ki je ukaz za vrnitev na zadnji izveden ukaz na prejšnji funkciji.

Ena izmed rešitev:

```
.text
.globl _Z4lte2l
.type _Z4lte2l, @function

#! file-offset 0x6d0
#! rip-offset  0x4006d0
#! capacity    16 bytes

# Text            #  Line  RIP       Bytes  Opcode
._Z4lte2l:        #        0x4006d0  0      OPC=<label>
xorl %eax, %eax   #  1     0x4006d0  2      OPC=xorl_r32_r32
testq %rdi, %rdi  #  2     0x4006d2  3      OPC=testq_r64_r64
setle %al         #  3     0x4006d5  3      OPC=setle_r8
retq              #  4     0x4006d8  1      OPC=retq
nop               #  5     0x4006d9  1      OPC=nop

.size _Z4lte2l, .-_Z4lte2l
```

Kot pri ostalih poskusih, je tudi za ta test narejena še ena poskusna koda, ki pa tokrat vrača vrednosti, takoj ko je eden izmed pogojev izpolnjen.

Primer drugačne kode:

```
if(x<=0){return 1;}
else{return 0;}
```

Rezultat ne kaže na odvisnost, kako zapišemo kodo, zato je ostala enaka.

Ena izmed rešitev:

```
.text
.globl _Z3ltel
.type _Z3ltel, @function
```

```
#! file-offset 0x680
#! rip-offset  0x400680
#! capacity    16 bytes

# Text            #  Line  RIP       Bytes  Opcode
._Z3ltel:         #        0x400680  0      OPC=<label>
xorl %eax, %eax   #  1     0x400680  2      OPC=xorl_r32_r32
testq %rdi, %rdi  #  2     0x400682  3      OPC=testq_r64_r64
setle %al         #  3     0x400685  3      OPC=setle_r8
retq              #  4     0x400688  1      OPC=retq
nop               #  5     0x400689  1      OPC=nop

.size _Z3ltel, .-_Z3ltel
```

4.2.2 Optimizacija daljših programov

V tem poglavju so rezultati optimizacij daljših in bolj kompleksnih programov. Ti programi so bili optimizirani večkrat z različnimi konfiguracijami. Razlog za takšen način testiranja je, da ugotovimo koliko časa se splača optimizirati nek program. Načrt je bil, da naprej izmerimo čas optimizacije programa, potem pa preverimo pravilnost optimiziranega programa in hitrost njegovega delovanja. Kljub temu pa načrt ni bil uspešen, saj so se pojavile nepravilnosti v optimiziranem programu. Razlog za nepravilnosti je lahko več. Prvi razlog je napačna konfiguracija optimizatorja med procesom optimiziranja. Pri vsaki funkciji moramo podati drugačno konfigurirajo. Kljub velikemu času posvečenem iskanju, je bilo neuspešno. Drugi problem je, da je sam optimizator nestabilen. Optimizator poskuša uporabljati ukaze za katere ni podprto validacijo. Zaradi tega ostaja možnost, da je iskanje povsem neuspešno.

4.3 Prednosti in slabosti

Kot vsi optimizatorji ima tudi ta optimizator nekaj prednosti in slabosti. Prva prednost takega optimizatorja je, da lahko uporablja bolj kompleksne

ukaze (npr. `popcnt`). Druga prednost je način iskanja optimizacija nekega programa. Optimizator ima veliko različnih konfiguracij in načinov iskanja. Uporabnik lahko izbira med gradnjo kode ali optimizacijo prejšnje kode, strategijo načina iskanja. Dodatna vrednost je še v tem, da ima optimizator ločene dele programov, ki lahko spreminjamo in dodajamo ne da bi ogrozili ostale module. Slaba stvar takega optimizatorja je, da uporabnik so mora spoznati na arhitekturo procesorja in na zbirno kodo, ki ga ta procesor uporablja. Pri tem še mora vedeti, kako nastaviti konfiguracijo za iskanje, kar je za preprostega uporabnika preprosto preveč. Poleg tega je še potrben človeški vnos skozi cel postopek optimiziranja. Najšlabše dejstvo pa je, da ta optimizator ni stabilen, saj potrebuje točno določen prevajalnik, točno določen operacijski sistem in ima še vedno težave pri prevajanju in zagonu optimizatorja. Superoptimizator je bolj uporaben za optimizacijo manjših funkcij v programu, saj lahko za celo vsebino funkcije zamenja z enim samim ukazom. Po vsaki optimizaciji moramo preveriti, če program deluje pravilno.

Poglavje 5

Lastna implementacija

Poleg obstoječih programov, je bil še narejena lastnen poskus implementacije superoptimizarotja. Ta optimizator se razlikuje od prejšnjih optimizatorjev, omenjenih V poglavju 3 in 4. Kot prvo, program je napisan v programskem jeziku Java. Kot drugo, program uporablja drugačen zbirni jezik. Omenjana optimizatorja sta uporabljala zbirni jezik za Intelov procesor, lastna implementacija pa uporablja MMIX zbirni jezik. Ta zbirni jezik je uporabljen, ker je bolj enostaven jezik in za ta jezik lažje implementirati kot Intelov. Programski jezik Java je bil uporabljen, ker podpira objektno programiranje in dedovanje. Poleg tega je bil še uporabjen pomožni program NetBeans.

5.1 Opis programa in načina delovanja

Kot je že v točki 5 omenjeno, optimizator je napisan v programskem jeziku Java. Če želimo tak program uporabiti ga moramo najprej prevesti z Java prevajalnikom. Ker je bil poleg prevajalnika uporabljen še pomožni program, ga je lažje prevesti. Programu je podan kot vhod ime besedilne datoteke. V Datoteki so napisani ukazi v MMIX zbirnem jeziku. Program ga potem prebere in shrani zaporedje ukazov v spremenljivko razreda oz. `class Code`. Vsak ukaz je shranjen v spremenljivki razreda `Instruction`. Razred Instrucktions vesebuje spremenljivke za shranjevanje operandov v ukazov. Ti

operandi so razdeljeni v tobelo vhodnih operandov in tabelo izhodnih operandov. Poleg tega je so še definirane tabela vhodnih in tabela izhodnih posebnih registrov, s katerimmi določena operacijska koda manipulira (Npr. `MULU` uporablja kot dodaten izhod posebni register `rH`). Pri branju programa optimizator naprej določi operacijsko kodo, potem določi kakšnega tipa so operandi in potem poskuša najti ujemanje v podani množici naborov ukazov v `if` stavku. Za dodajanje novega tipa ukaza je potrebno spremeniti v `if` ali `else if` pogoj, ali stavku dodati nov `else if` stavek za že obsteječo operacijsko kodo ali pa dodati nov `case`, če dodajamo ukaz z novo operacijsko kodo.

`if` primer:

```
if(opCode.equals(Constants.opCodes.ADD.toString())){
    if(operands.length == 3){
            if(operands[0] instanceof Register &&
            operands[1] instanceof Register &&
            (operands[2] instanceof Register ||
             operands[2] instanceof OpByte)){
            Operand[] in = {operands[1],operands[2]};
            Operand[] out = {operands[0]};
            Operand[] sIn = {};
            Operand[] sOut = {getOperand("rA")};
            list.add(
             new Instruction(label,Constants.opCodes.ADD,in,out,sIn,sOut)
            );
        }
        else{
            System.out.println("Wrong operands in: "+line);
            System.exit(1);
        }
    }
    else{
        System.out.println("Wrong operands in: "+line);
        System.exit(1);
    }
}
else if(opCode.equals(Constants.opCodes.SUB.toString())){
```

```
    if(operands.length == 3){
        if(operands[0] instanceof Register &&
         operands[1] instanceof Register &&
         (operands[2] instanceof Register ||
          operands[2] instanceof OpByte)){
            Operand[] in = {operands[1],operands[2]};
            Operand[] out = {operands[0]};
            Operand[] sIn = {};
            Operand[] sOut = {getOperand("rA")};
            list.add(
             new Instruction(label,Constants.opCodes.SUB,in,out,sIn,sOut)
            );
        }
        else{
            System.out.println("Wrong operands in: "+line);
            System.exit(1);
        }
    }
    else{
        System.out.println("Wrong operands in: "+line);
        System.exit(1);
    }
}
```

(Primer kode se najde v datoteki MMIXOptimizator.java (vrstice 234-263). V vsakem pogojnem stavku se preveri, če je ukaz pravilne oblike. Naprej se preveri če operacijska koda obstaja, potem če je dovolj operandov za ta ukaz in potem se preveri če je vsak opreand pravilnega tipa.)

Če dodajamo novo operacijsko kodo, je potem še potrebmo dodati nov znak, ki predstavlja novo operacijsko kodo, v `enum` spremenljivko `opCodes`. Oprimizator vsebuje skupno metodo `change` za spreminjanje kode.V tej metodi je več kot 70 različnih metod za spreminjanje kode. Pri dodajanju novega vzorca je potrebno napisat novo metodo, To metodo je ptem potrebno vstaviti v metodo `change`.

Primer skupne metode:

```
private static void change(TreeSet<Code> codeList){
```

```
        Code org = new Code(codeList.first().list);
        found.add(org);
        refreshLive(org);
        codeList.pollFirst();

        //metode spreminjanja kode
        deleteDead(org);
        changeMulToShift(org);
        changeShiftToMul(org);
        doublePlusU(org);
        doubleMinus(org);
        doubleMinusU(org);
        plusMinusU(org);
}
```

(Primer kode se najde v datoteki MMIXOptimizator.java. Metoda se nahaja v 2917. vrstici in je skrajšana za ta primer.)

5.1.1 Zagon

Kot je že omenjeno v poglavju 5.1, program je potrebno prevesti, ter ga potem pognati. Prevajalnik se zažene s pomočjo ukazne vrstice. Ukaz je enostaven in ne potrebuje nobenih zastavic. Preveden program potem poženemo z ukazno vrstico. Pri tem moramo podati kot argument imena datotek, ki jih želimo optimizirati.

Format ukaza za prevoda programa

```
javac <datoteka>
```

Format ukaza za zagon programa

```
java <datoteka_brez_končnice> <prva_datoteka> <druga_datoteka> <tretja_datoteka...>
```

Primer ukaza za prevoda programa

```
javac MMIXOptimizator.java
```

Format ukaza za zagon programa

```
java MMIXOptimizator test1.txt test2.txt
```

Ker je optimizator narejen s pomočjo NetBeans programa, lahko NetBeans uporabimo kot alternitivo za zagon programa. Pomožni program poskrbi za avtomatsko prevajanje in zogon, in je uporabniku bolj prijazen.

5.1.2 Način optimiziranja

MMIX superoptimizator optimizira glede na podan nabor vzorcev. Ti vzorci so opisani v obliki metode. Vsaka metoda je potem uporabljena v skupni metodi `change`. To omogoči, da je vsaka transformacija funkcijonalno enaka. Pri iskanju po prostoru se uporablja mešanica iskanja po širini in globini. Program najprej išče najbolj optimalno optimizacijo po širini in shrani vsako najdeno kombinacijo v tabelo. Potem iz tabele vzame najkrajšo najdeno kodo in na njej pononvo naredi vse transformaije, ter vse transformacije shrani nazaj v tabelo. Program konča ko nima več nobene kode za tranformirat. Tukaj se lahko pojavi problem. Recimo da imamo transformacijo, ki spremeni ukaz tako, da uporabi na mesto tega ukaza nek funkcionalno ekvivalentni ukaz z drugo operacijsko kodo (Npr iz `MUL` v `SR`). Potem pa še dodamo dodatno transformacijo, ki gre v obratno smer od prejšnje tranformacija. Če imamo taki transformaciji, se potem program zacikla saj, ko išečmo v širimo, v tabelo vstavimo kodo, ki se potem nazaj preslika v prenšnjo kodo. Ta problem se rešimo tako, da imamo še eno tabelo, kjer hranimmo vse transformacije kode.

5.2 Primeri optimizacije

Tukaj je nekaj delov programa, ki jih je optimizator optimiziral in njihov pripadajoči optimiziran program. Vsi deli so majhni, saj optimizator porabi preprosto preveč časa z večje dele programa. Pri tem boste opazili, da je za primer podanih več funkcij ki pa imajo enako funkcijonalnost. Za prvi primer si lahko vzamemo del kode, ki izračuna absolutno vrednost podanega števila.

Primer dela kode:

```
return (v0 < 0 ? -v0 : v0);
```

Za tako kodo je progam našel 6 možnih rešitev, dolžine 4.

Ena izmed rešitev:

```
movl    %eax,%edx
sarl    $31,%edx
addl    %edx,%eax
xorl    %eax,%edx
```

Za naslednji test je optimizator iskal rešitev za enako funkcijonalnost, kot v prejšnjem testu, vendar koda je drugače napisana. Pri tej kodi manjka `else` stavek.

Primer Drugačne napisane kode:

```
if(x<0){x=-x;}
return x;
```

Enako kot za prejšnji primer, je optimizator našel 6 možnih rešitev, dolžine 4. Vse rešitve so enake kot v prejšnjem testu.

Ena izmed rešitev:

```
movl    %eax,%edx
sarl    $31,%edx
addl    %edx,%eax
xorl    %eax,%edx
```

Za naslednji test je bila podana funkcija, ki pove če je število manjše, večje ali enako številu 0. Pri tem je bila uporabljena naslednja koda.

Primer kode:

```
return (v0 > 0 ? 1 : (v0 < 0 ? -1 : 0));
```

Za takšen primer je optimizator poiskal 26 optimalnih kombinacij, dolžine 4.

Ena izmed rešitev:

```
addl    %eax,%eax
sbbl    %edx,%edx
subl    %eax,%edx
adcl    %eax,%edx
```

Za naslednji test je bila uporabljena funkcija z enako funkcijonalnostjo kot pri prejšnjem testu. Koda tokrat uporablja `else if` stavek in za vsak izpolnjen pogoj, takoj vrne rešitev.

Primer drugače zapisane kode:

```
if(x<0){return 1;}
else if(x>0){return -1;}
else{return 0;}
```

Pri tem primeru se najdejo razlike. Optimizator je našel optimalno rešitev šele pri dolžini 5. Pri tem je našel 245 možnih rešitev.

Ena izmed rešitev:

```
addl    %eax,%eax
movl    %eax,%edx
adcl    $-1,%edx
subl    %eax,%edx
sbbl    $-1,%edx
```

Naslednji podan primer nastavi 0, če je število večje ali enako 0, drugače 1.

Primer kode:

```
return v0 <= 0;
```

Pri tem primeru je optimizator našel 4 optimalne rešitve, dolžine 3.

Ena izmed rešitev:

```
cmpl    $1,%eax
sbbl    $0,%eax
shrl    $31,%eax
```

Kot pri ostalih poskusih, je tudi za ta test narejena še ena poskusna koda, ki pa tokrat vrača vrednosti, takoj ko je eden izmed pogojev izpolnjen.

Primer drugačne kode:

```
if(x<=0){return 1;}
else{return 0;}
```

Pri tem primeru je optimizator našel enake rešitve kot v prejšnjem primeru.

Ena izmed rešitev:

```
cmpl    $1,%eax
sbbl    $0,%eax
shrl    $31,%eax
```

Za zadnji primer je optimizator optimiziral zelo enostavno kodo. Koda vrne število, ki je za 4 večje od danega števila.

Primer kode:

```
return x+4;
```

Če pogledamo to kodo, bi ugotovili, da bi lahko tako kodo zapisali le z enim ukazom, vendar, če tako kodo damo optimizatorju, ugotovimo, da optimizator ne najde take rešitve. Najde pa rešitve, ki so dolžine 4.

Primer rešitve optimizatorja:

```
addl    $1,%eax
addl    $1,%eax
addl    $1,%eax
incl    %eax
```

Primer optimalne rešitve:

```
addl    $4,%eax
```

5.3 Prednosti in slabosti

Vsak optimizator ima svoje prednosti in slabosti in GNU-Superoptimizer ni nobena izjema. Toda pri tem optimizatorju je več slabosti kot prednosti. Največja prednost tega optimizatorja je, da vedno najde optimalen program, če upoštevamo nabor ukazov, ki jih lahko izpiše. Optimizator je tako osnovan, da lahko dodajamo nabor ukazov za druge vrste arhitekture. Program je tudi dovolj stabilen za uporabo in ga je enostavno namestiti. Prednost je tudi v tem, da najdeni programi ponavadi ne vsebujejo pogojne skoke, ki upočasnjujejo program. Tukaj pa se prednosti končajo. Največja slabost takega optimizatorja je časovna zahtevnost. Časovna zahtevnost takega programa je O($m * n^{2n}$), kjer m predstavlja število ukazov in n je najmanjša dolžina, ki predstavlja ciljno funkcijo. Optimizator zna uporabljat le podan nabor operacij [11], poleg tega pa ni fleksibilen, če se v originalni kodi pojavijo konstante, ki jih optimizator ne podpira pri iskanju. (Primer je omenjen v točki 5.2).

Poglavje 6

Paralelizacija algoritma optimiziranja

V prejšnjih pogljavih je od vseh časovnih meritvah najbolj pomembna `user` in `sys` meritev, saj nam poveta koliko časa je procesor zaseden z izvajanjem določenega programa. V nekaterih primerih, je lahko vsota teh dveh časov večja od real časovne meritve. To je množno pri večjedernih procesorjev. To dejstvo bi lahko iskoristili, tako da bi delo poradelili na več procesov, ki bi se lahko hkrati izvajali. Pri tem bi program več časa zasedal jedro procesorja, vedar bi zmanjšali čakalni čas uporabnika. Postopek kjer delo porazdelimo na več procesov oz. niti imenujemo pralelizacija programa. S tem postopkom bi lahko zmanjšali čakanje pri optimizaciji nekega programa. Pri tem se lahko pojavi problem branja podatkov na pomnilniku. Če ima nek program skupno pomnilniško mesto, lahko pride do desinhronizacije podatkov. Recimo, da imamo funkcijo, ki prebere iz polmnilnika vrednost, jo poveča za ena, jo izpiše na standardni vhod in jo potem shrani nazaj na isto mesto. In recimo, da to funcijo izvajamo na dveh različnih nitih. Če bi tak proces izvedli, bi se lahko zgodili 3 scenariji. Prvi scenarij je, da prvi proces izpiše število manjše kot izpis drugega procesa, v drugem scenarijo prvi proces izpiše manjše kot izpis drugega procesa, v tretjem pa bi oba procesa izpisala enako število. Razlog za tak odnos programa se skriva v dodeljevanju časovne

rezine določenemu procesu. Operacijski sistem skrbi za "hkratno"izvajanje programov. Pri tem uporablja prioritetni sistem, ki narekuje kako pogosto naj določenemu procesu dodelimo časovno rezino. Ko proces porabi časovno rezino, potem operacijski sistem dodeli rezino drugemu procesu. Kar iz tega sledi, da ne moremo zagotoviti pravilno zaporedje izvajanje procesov in celotno izvajanje kritičnega dela kode brez semaforjer za sinhronizacijo. Problem lahko rešimo tudi z omejitvijo, kjer vsak proces lahko obdela le določen del podatkov in meja ne seka z ostalimi omejitvami ostalih procesov. Čeprav programi navedeni v dokumentu nimajo paraleliziranih metod oz. funkcij, lahko iz zagona skript za namestitev ali prevajanje programov lahko že nekaj sklepamo. Pri prevjanju izvirne kode STOKE optimizatorja, lahko opazimo, da skripta, ki jo moramo zagnati, naredi več procesov oz. niti, ki prevajajo različne programe, potrebnih za izvajanje optimizatorja. Čeprav prevedena koda, ki se kasneje uporabi za sestavo optimizatorja, še ni binarna koda, lahko že ima nekare optimizacije narejene. Iz tega lahko sklepamo, da bi lahko istočasno optimizacijo delali na več različnih programih. Vprašanje pa je, koliko bi morali zmanjšati velikost segmenta kode, da bo pralelizacija dela programa nemogoča in kakšni morajo biti pogoji za to? Del odgovora se skriva že v načinu delovanja STOKE optimizatorja. Kot je bilo že v poglavju 4.1.2 omenjeno, STOKE izvaja optimizacijo nad posmezno funkcijo posebej in optimizirano funkcijo vnese nazaj v binirano datoteko prevedenega programa. To bi lahko uporabili za paralelizacijo, kjer bi naredili tak porgram, ki optimizira vse funkije tako, da bi za vsako funkcijo uporabili eno nit. Pri tem lahko nastane problem, kjer bi morali ugotoviti, kateri registri so živi ob klicu in kateri registri so živi ob koncu klicane funkcije. Na srečo je problem že rešen z dogovori iz nekaj let nazaj, kako mora nek program klicati funkcijo in kako se mora klicana funkcija končati. Program bi še lahko mogoče dalo paralelno optimizirati še na majnše kose (Npr. if stavki ali zanke). Za tako paralelizacijo bi morali obvezno poračunati kateri registri so živi na začetku `if` ali zanke in kateri so živi na koncu `if` stavka ali zanke. Paralelizacija pa ni priporočljiva, saj nam onemogoča dinamično spreminjanje živih regi-

srov, ki bi lahko kodo precej bolj optimiziralo (Npr. dodaten register, ki bi hranih naslov objekta, iz katerega bi pogosto brali). Kar je še pomembno omeniti, je to, da ni vedno vredno uporabiti paralelni algoritem optimizacije kode. To še posebej velja za segmente kode, ki zelo majhni, saj bi za ustvarjanje nove niti porabili več časa kot za samo optimizacijo. Rešitev, kako bi lahko določili, kdaj bi bilo vredno uporabiti paralelni algoritem, se skriva v hevristični oceni. Kot primer bi lahko vzeli program, ki bi ocenil dolžino segmenta kode in glede na to dolžino, bi se lahko odločil, če bi bilo vrednod izvesti paralelno.

Poglavje 7

Zaključek

Če prav superoptimizator obstaja že 20 let, je njegov razvoj the tehnologije precej počasen [6]. Razlogi za počasen razvoj so počasnos programa (za razvijanje kode dolžine 12 instrukcij potrebuje nekaj ur), programi s kazalci na pomnilniško mesto vzemejo veliko časa, saj lahko kazalci kažejo na katerokoli pomnilniško mesto. Za nedvisno od naprav verzijo optimizatorja je omejen le na zelo kratke naprave, lahko funkcijonalno deluje na generiranih primerih, nevemo pa če deluje pravilno na vseh primerih in je pogosto razvit za podporo pri razvijanju prevajalnika [17]. Po testih različnih optimizatorjev (tako iz preteklosti kot v sedanjosti) lahko pridemo do nekaterih zaključkov. Prvi zaključek je, da so superoptimizatorji skoraj neuporabni. GNU-Superoptimizator je preveč počasen in uporablja le podane ukaze. STOKE optimizator je preveč nestabile in neprijazen za navadnega uporabnika. Optimizator tudi ne deluje pravilno pri večjih programih, saj optimizira glede na testne primere, ki pa pogosto nepokrivajo vse možne rezultate, ki jih program potrebuje. Najbolje je uporabiti GNU Compiler Collection, saj polega prevajalnika vsebuje tudi optimizator. Sam optimizator je lahko precej unčikovit, če mu damo dovolj časa. Kateri nivo je najboljši za določen program je odvisen od velikosti programa. Čeprav prvi nivo optimizacije že prinese občutno hitrejši program, je bolj primeren za prostorsko kot hitrostno optimizacijo. Po rezultatih sodeč v poglvju **??** je tudi dovolj

dober za krajše programe oz. za kratko delujoče programe. Če pa imamo dovolj časa in imamo zelo dolg program, potem se splača uporabiti tretji nivo optimizacije. [8]

Literatura

[1] T. Granlund, R. Kenner, "Eliminating branches using a superoptimizer and the GNU C compiler," *SIGPLAN Not.*, št. 7, zv. 27, str. 341–352, 1992.

[2] E. Schkufza, R. Sharma,, A. Aiken, "Stochastic superoptimization," *SIGPLAN Not.*, št. 4, zv. 48, str. 305–316, 2013.

[3] E. Schkufza, R. Sharma,, A. Aiken, "Stochastic program optimization," *Commun. ACM*, št. 2, zv. 59, str. 114–122, 2016.

[4] E. Schkufza, R. Sharma,, A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," *SIGPLAN Not.*, št. 6, zv. 49, str. 53–64, 2014.

[5] F. Mueller, D. B. Whalley, "Avoiding conditional branches by code replication," v zborniku *In Proceedings of the ACM SIGPLAN'95 Conference onProgramming Language Design and Implementation (PLDI)*, str. 56–66, 1995.

[6] T. Hume, "Literature review: Superoptimizers." Dosegljivo: `http://www.sis.uta.fi/~pt/TIEA5_Thesis_Course/Session_09_2013_02_15/Example%20Literature%20Reviews/Example%20Literature%20review%20Chronological.pdf`, 2011. [Dostopano: 27. marec 2017].

[7] H. Massalin, "Superoptimizer: A look at the smallest program," *SIGPLAN Not.*, št. 10, zv. 22, str. 122–126, 1987.

[8] J. Särnesjö, *Using Binary Decision Diagrams to Determine Program Equivalence in a Superoptimizer.* PhD thesis, KTH, School of Computer Science and Communication (CSC), 2011.

[9] A. J. Hutton, C. C. Ross, B. Elliston, J. Johnson, M. Mitchell, T. Morita, D. Novillo, G. Pfeifer, I. Lance, C. C. Ross,, R. A. Sayle, "A superoptimizer analysis of multiway branch code generation," v zborniku *Proceedings of the GCC Developers' Summit*, str. 103–117, 2008.

[10] S. Bansal, A. Aiken, "Automatic generation of peephole superoptimizers," *SIGOPS Oper. Syst. Rev.*, št. 5, zv. 40, str. 394–403, 2006.

[11] T. Crick, *Superoptimisation: provably optimal code generation using answer set programming.* PhD thesis, University of Bath, 2009.

[12] W. K. Hastings, "Monte carlo sampling methods using Markov chains and their applications," *Biometrika*, št. 1, zv. 57, str. 97–109, 1970.

[13] R. Sharma, E. Schkufza, B. Churchill,, A. Aiken, "Data-driven equivalence checking," *SIGPLAN Not.*, št. 10, zv. 48, str. 391–406, 2013.

[14] R. Sharma, E. Schkufza, B. Churchill,, A. Aiken, "Conditionally correct superoptimization," *SIGPLAN Not.*, št. 10, zv. 50, str. 147–162, 2015.

[15] L. De Moura, N. Bjørner, "Z3: An efficient SMT solver," v zborniku *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Berlin, Heidelberg, str. 337–340, Springer-Verlag, 2008.

[16] S. Heule, E. Schkufza, R. Sharma,, A. Aiken, "Stratified synthesis: Automatically learning the x86-64 instruction set," *SIGPLAN Not.*, št. 6, zv. 51, str. 237–250, 2016.

[17] "A generic superoptimizer." Dosegljivo: https://anthonythecoder.wordpress.com/2016/03/04/a-generic-superoptimizer/. [Dostopano: 27. marec 2017].