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

Povzetek

Abstract

1 citati **1**

Literatura **7**

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

citati

Čeprav superoptimizator obstaja že 20 let, je njegov razvoj the tehnologije precej počasen. 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 [1].

Še ena bolj zanimiva verzija je uporaba kombinacije superoptimizatorja in podatkovne baze. Idea je, da program le 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 [2].

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, poleg tega pa ni fleksibilen, če se v originalni kodi pojavijo konstante, ki jih optimizator ne podpira pri iskanju [3].

Z3 intergrira moderno metodo, ki temelji na DPLL SAT reševalec, jederni teoretski reševalec, ki se ukvarja z enakovrednostmi in neinterpretiranimi funkcijami, satelitskimi reševalci (za aritmetiko, tabelo, etc.), in E-ujemanjem abstraktnih strojev (za kvantifikatorje) [4].

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 [5].

Algoritem v STOKE optimizatorju je posledično lahko reševal tudi druge probleme. Primer takega je avtomatično učenje ukazov. Ideja takega algoritma je, da je programu podan osnovni nabor ukazov, program pa bi moral iz osnovnih ukazov sestaviti zaporedje ukazov, ki je funkcijonalno enaka bolj kompleksnemu ukazu [6]

Pri iskanju ekvivalence, program ima implementiran podatkovno vodeno preverjanje ekvivalence (Data-Driven Equivalence Checking), 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.

$$\exists a, b \in \mathbb{N}, \forall \sigma, m_t = a * m_r + b \tag{1.1}$$

Za vse točke mora veljati izraz 1.1, 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 [7].

Čeprav superoptimizator obstaja že 20 let, je njegov razvoj the tehnologije precej počasen. Kjub temu, da imamo demostracije delovanja super optimizatorjev in imamo vednno bolj zmogljivejše računalnike, suptimizatorji imajo malo pozornosti pri razvijanju programske opreme [8].

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 povezavo 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 [9].

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 ukazov 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 [10].

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 spremenljivk v zankah [11].

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 predsta-

vitvi bolj kompleksnih funkcij, kot so množenje in deljenje, saj je posledica tega zelo velik graf in posledično tudi program potrebuje zelo veliko časa za dokazovanje ekvivalence [12].

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 [13, 14].

Še bolj se dejstvo zakomplicira, ko upoštevamo, da se med vsako operacijo število zaokrožuje. 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 številih povečuje, Relativna vrednost, pa se povečuje pri denomlaliziranih številih. Tak problem 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 [15].

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 [16].

Literatura

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

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

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

[4] 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.

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

[6] 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.

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

[8] 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].

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

[11] 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.

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

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

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

[15] 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.

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