OPTIMIZATION OF LOGICAL EXPRESSIONS IN COMPILERS

Jakub Martiško
Bachelor Degree Programme (4), FIT BUT
E-mail: xmarti52@stud.fit.vutbr.cz

Supervised by: Alexander Meduna
E-mail: meduna@fit.vutbr.cz

Abstract: The aim of this paper is introduction of methods designed for logical expressions optimization during program compilation. The described methods are specifically designed to utilize the properties of logical operations.

Keywords: compilers, optimization, logical expressions

1 ÚVOD

Jednou z částí překladu zdrojového kódu na spustitelný soubor je optimalizace výstupního programu. Literatura[1] uvadí mnoho optimalizačních metod, které jsou mimo jiné zaměřeny na eliminaci znovuvyhodnocování již jednou vypočtených výrazů, snížení počtu přístupů do paměti pomocí použití přímých hodnot namísto proměnných a zefektivňování cyklů. Optimalizace zaměřené specificky na logické operace jsou naproti tomu poměrně vzácné. Metody, které budou představeny v této práci, využívají vlastností booleovy algebry jako je idempotence, agresivita nuly a jedničky, neutralita nuly a jedničky a jiné.

Tato práce se zabývá dvěma přístupy k této problematice. První z nich se snaží transformovat jednotlivé výrazy na výrazy jednodušší na základě výše uvedených vlastností. Druhý se pak zabývá vztahem mezi vstupními hodnotami daného výrazu a jeho výslednou hodnotou. Na základě těchto vztahů pak zkoumá, není–li celý výraz, případně některý jeho podvýraz, například tautologií.

2 ANALÝZA VLASTNOSTÍ LOGICKÝCH OPERACÍ

Metody popsané v této sekci výchazí z vlastností logických operací a na jejich základě se snaží jednotlivé výrazy přepsat na výrazy ekvivalentní ale efektivnější. Jelikož mnoho z těchto vlastností pracuje s proměnnou a zároveň s její negací, je vhodné určit pro každý bod programu množinu, obsahující proměnné jež jsou negací jednotlivých proměnných. Obdobně je možné zavést druhou množinu, která bude obsahovat dvojité negace a následně ji využít k jejich nahrazení původní proměnnou.

Jedna z nejjednodušších optimalizací tohoto typu vychází z neutrality, případně agresivity pravdivostních hodnot. Známe–li pravdivostní hodnotu některého z operandů daného výrazu, můžeme v závislosti na této hodnotě a použité operaci nahradit celý výraz touto hodnotou případně druhým operandem.

Podobná analýza využívá komplementaritu a idempotenci pravdivostních hodnot. Narozdíl od předchozí metody není nutné znát hodnotu některého z operandů. Místo toho metoda hledá takové výrazy, kde jsou jednotlivé operandy shodné (jedná se o stejnou proměnnou), případně jeden z operandů je negací druhého. Tyto výrazy pak opět nahrazuje operandem (v případě shodnosti operandů) případně pravdivostní hodnotou (pokud je operand negací druhého) závislou na operaci použité v tomto výrazu.

Pro optimalizaci složitějších výrazů je nutné tyto metody modifikovat takovým způsobem, aby algoritmus dokázal zpracovat i příkazy, které vyhodnocují jednotlivé podvýrazy daného výrazu a vhodně

je přeskupit. Cílem tohoto přeskupení je prozkoumat příslušnou posloupnost instrukcí a na základě komutativity se pokusit najít páry proměnných a jejich negací. Pokud takovýto pár nalezne, může nahradit příslušný výraz přímou hodnotou. Podvýrazy, které původně vedly k výpočtu tohoto výrazu, se pak mohou stát mrtvým kódem.

Algoritmus zajišťující toto přeskupení prochází program ve směru zpracování kódu (forward–flow problém). Při prvním průchodu si pro každý bod programu a každý výraz sestaví množinu *variables[exp]*, která obsahuje názvy proměnných, se kterými tento výraz, případně výraz ze kterého zkoumaný výraz vychází, pracuje. Narazí–li optimalizátor na instrukci, která přiřazuje novou hodnotu proměnné, která se již nachazí v alespoň jedné množině *variables[exp]*, je nutné ji ze všech těchto množin odstranit. Má–li základní blok více přímých předchůdců, ve kterých se mohou vyskytovat některé z podvýrazů zkoumaného výrazu, pak je jako výsledná množina *variables[]* použit průnik příslušných množin v jednotlivých blocích. Následně prochází algoritmus program znovu a jakmile narazí na příkaz s logickou operací, zjistí jakého druhu jsou jeho operandy. Je–li operand výsledná hodnota nějakého předchozího logického výrazu, který využíva *stejnou* operaci (konjunkce případně disjunkce) jako momentálně zkoumaný výraz, pak algoritmus prozkoumá příslušnou množinu *variables[exp]*. Jakmile takto zpracuje oba své operandy, pokusí se najít dvojici X a Y kde $Y = \neg X$ a $X \in variables[operand1]$ a zároveň $Y \in variables[operand2]$. Nalezne–li algoritmus tuto dvojici, je pak možné nahradit celý tento výraz hodnotou $TRUE$ případně *FALSE* v závislosti na použité operaci.

Instrukce	$variables[T_1]$	$variables[T_2]$
$Y := \neg X$	$\emptyset$	$\emptyset$
$T_1 := Z \vee Y$	Z; Y	$\emptyset$
$T_2 := X \vee T_1$	Z; Y	Z; Y; X

Instrukce	$variables[T_1]$	$variables[T_2]$
$Y := \neg X$	$\emptyset$	$\emptyset$
$T_1 := Z \vee Y$	Z; Y	$\emptyset$
$T_2 := TRUE$	Z; Y	$\emptyset$

Tabulka 1: Optimalizace založená na komplementaritě

3 ANALÝZA VSTUPNÍCH HODNOT

Tato skupina metod se snaží nejprve sestavit tabulku s jednotlivými logickými výrazy pro základní blok. Následně zkoumá jakým způsobem je ovlivněna hodnota výrazu v závislosti na jednotlivých operandech. Na základě těchto optimalizací je možné určit, je–li výraz (případně jeho část) tautologií případně kontradikcí.

Tabulky používané těmito metodami se skládají ze dvou částí. První část obsahuje seznam všech proměnných pro některý z výrazů. Druhá část pak jednotlivé podvýrazy, včetně cílové proměnné pro uložení výsledku, názvů jednotlivých operandů a operaci použitou v tomto výrazu. Pro každou sérii podvýrazů, která celkově tvoří jeden složený výraz, je vytvořena samostatná tabulka. Po sestavení jsou jednotlivé proměnné v tabulkách nastaveny na všechny možné kombinace hodnot a na jejich základě jsou vypočteny hodnoty jednotlivých výrazů a ty jsou následně zkoumány. Například tabulka 2 reprezentuje rovnici 1.

$$T_3 = X \wedge ((Y \vee Z) \wedge X) \tag{1}$$

Algoritmus stojící za sestavením tabulky prochází kód programu proti směru jeho zpracování. Narazí–li na logický výraz, pokusí se jej najít v již existujících tabulkách. Pokud jej nenalezne pak vytvoří tabulku novou a do části pro proměnné uloží jeho operandy. Do části pro výrazy pak samotný výraz včetně jména proměnné pro uložení výsledku. Pokud tento výraz v některé z tabulek již je, pak, je–li v sekci pro proměnné, je přesunut do sekce pro výrazy a jeho operandy jsou uloženy do sekce pro proměnné. Je–li některý z jeho operandů již v sekci pro výrazy, je nutné všechny již existující výskyty

X	Y	Z	$T_1 = Y \vee Z$	$T_2 = T_1 \wedge X$	$T_3 = T_2 \wedge X$
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Tabulka 2: Tabulka reprezentující výraz popsaný rovnicí 1

této proměnné v tabulce přejmenovat (při použití SSA formy[1] tento problém zaniká) a do tabulky uložit tento nový výskyt jakožto novou proměnnou. Je–li v sekci pro výrazy, pak se jedná o znovuvýpočet stejné hodnoty (což lze eliminovat pomocí metod zaměřených na společné podvýrazy[1]), nebo došlo ke změně hodnoty některého z operandů a původní verze tohoto výrazu se již v tabulce nachází pod novým jménem. Nalezne–li algoritmus příkaz přiřazující konkrétní hodnotu nějaké proměnné, pak je tato hodnota nastavena jako konstantní pro všechny výskyty této proměnné a proměnná je přejmenována ve všech tabulkách (při použití SSA nutnost přejmenování opět odpadá). Pro sestavení tabulky v rámci více základních bloků je nutné vytvořit kopii tabulky pro každého přímého předchůdce aktuálního bloku a každou z těchto větví zpracovávat jakoby byla součástí aktuálně zkoumaného bloku. Následně je nutné porovnat jednotlivé kopie tabulek a jsou–li shodné, je možné je použít. V opačném případě je nutné sestavit tabulky jen pro jednotlivé bloky a výrazy, které jsou definovány v jiném bloku, brát dále jako proměnné.

Na základě takto sestavených tabulek je pak již snadné určit, není–li některý z výrazů, případně jedna či více z jeho částí tautologií nebo kontradikcí. Mimoto je také možné nalézt podvýrazy, které neovlivní celkový výsledek, jako například výraz T_3, v tabulce 2, který se oproti T_2 již dále nemění. Tato metoda může vést k některým optimalizacím, které by mohly být provedeny i pomocí metod uvedených v kapitole 2. Proto je vhodnější, v případě kombinování obou metod, využít nejdříve optimalizace z předchozí kapitoly, čímž může dojít ke zmenšení jednotlivých tabulek a tím i ke zrychlení optimalizace.

4 ZÁVĚR

Metody popsané v této práci jsou navrženy jako doplněk k obecným, často používaným metodám, které dokáží práci zde popsaných analýz zefektivnit. Narozdíl od těchto metod je jejich použití více závislé na konrétním zdrojovém kódu. V případě vhodného vstupního programu ovšem dokáží zjednodušit i poměrně složitější posloupnost instrukcí jen na několik výrazů.

REFERENCE

[1] AHO, Alfred V., Monica S. LAM, Ravi SETHI a Jeffrey D. ULLMAN. Compilers: principles, techniques and tools. 2nd ed. Boston: Addison Wesley, 2007, xxiv, 1009 s. ISBN 03-214-8681-1.

[2] MUCHNICK, Steven S. Advanced compiler design and implementation. San Francisco: Morgan Kaufmann, 1997, xxix, 856 s. ISBN 15-586-0320-4.

[1] viz kapitola 8.11 v[2]