RECONSTRUCTION OF INSTRUCTION IDIOMS IN THE RETARGETABLE DECOMPILER

Fridolín Pokorný
Bachelor Degree Programme (3), FIT BUT
E-mail: xpokor32@stud.fit.vutbr.cz

Supervised by: Jakub Křoustek
E-mail: ikroustek@fit.vutbr.cz

Abstract: The goal of this work is to detect and transform instruction idioms used in modern compilers. These instruction idioms are used to optimize code and produce faster or even smaller executable files. On the other hand, they can confuse an user of a decompiler. Reconstructing instruction optimizations leads to more understandable source code when decompiling an executable.

This work is a part of the Lissom project.

Keywords: instruction idioms, compiler optimizations, reverse engeneering, decompiler, Lissom

1 ÚVOD

Pri preklade zdrojového kódu dochádza k optimalizáciám, ktorých účelom je produkovať čo najefektívnejšie riešenie. Takéto riešenie často vyžaduje podrobnú znalosť cieľovej architektúry a inštrukčnej sady, ktorou disponuje. Optimalizácie pomocou inštrukčných idiomov nie sú bežne známe ani u skúsených programátorov vo vyšších programovacích jazykoch, ktorí od cieľovej platformy abstrahujú na úroveň používaného vyššieho programovacieho jazyka. Pri prípadnom spätnom preklade tak dochádza k nežiadúcemu zatemneniu spätne preloženého zdrojového kódu a jeho pochopenie sa stáva náročným problémom.

Hlavným dôvodom zavedenia inštrukčných idiomov je nahradenie pomalého riešenia za rýchlejšie. U vstavaných systémoch (angl. *embedded systems*) sa stáva veľmi zaujimavým problémom nahradenie postupnosti inštrukcií za inú postupnosť inštrukcií, pričom dbáme na veľkosť cieľového spustiteľného súboru. Veľmi často disponujeme obmedzenou veľkosťou dostupnej pamäti (napr. *firmware*). Rôznorodosť a pestrosť inštrukčných sád umožňuje problém riešiť hneď niekoľkými spôsobmi. Ako základ pre riešenie problému slúži Booleova algebra, výber najoptimálnejšieho riešenia je však špecifické pre danú architektúru[1].

Táto práca vzniká vrámci projektu Lissom vedenom na FIT VUT v Brne.

2 SPÄTNÝ PREKLADAČ PROJEKTU LISSOM A INŠTRUKČNÉ IDIOMY

Spätný prekladač vyvýjaný pod záštitou projektu Lissom má za cieľ byť rekonfigurovateľný. Už dnes podporuje preklad spustiteľných súborov `ELF` či `PE` pre architektúry `ARM`, `MIPS` a `x86`. Na hľadanie inštrukčných idiomov sú tak kladené nároky univerzálnosti použitia bez ohľadu na cieľovú architektúru, či formát spustiteľného súboru.

Pri hľadaní inštrukčných idiomov bola vybraná vzorka prekladačov, u ktorých bol hľadaný výskyt jednotlivých idiomov. Medzi dnes najpoužívanejšie prekladače nepochybne patrí prekladač `GCC`[2], dostupný pod slobodnou licenciou. Významnejšie používaným prekladačom je i prekladač obsiahnutý a štandardne dodávaný vo vývojovom prostredí `Visual Studio` od firmy `Microsoft`. Ďalším,

a z pohľadu optimalizácií veľmi zaujímavým prekladačom, je prekladač od firmy `Intel`. Skúmané boli i pomerne staršie prekladače `Open Watcom`[3] a prekladač od firmy `Borland`.

U prekladačov `GCC` a `Open Watcom` boli dostupné zdrojové kódy, čo umožnilo priamo vyhľadať inštrukčné idiomy a získať podrobný popis idiomu až na implementačnej úrovni. U ostatných skúmaných prekladačoch však licencia, pod ktorou sú jednotlivé prekladače distribuované, neumožnila nahliadnuť do zdrojových kódov a vyhľadanie inštrukčných idiomov tak bolo náročnejšie.

Veľmi často prekladače využívajú niektoré inštrukčné idiomy spoločne. Pri implementácii rekonštrukcie inštrukčných idiomov tak bolo nutné získať univerzálne riešenie, ktoré by bolo použiteľné bez ohľadu na použitý prekladač a architektúru.

Prekladače disponujú rôznymi úrovňami optimalizácie. Čím je pôvodný zdrojový kód viac optimalizovaný, tým je ťažšie spätne rekonštruovaný do jeho pôvodného tvaru. Naviac pri preklade dochádza k odstráneniu niektorých informácií akými sú názvy premenných a komentáre, čo zhoršuje samotný spätný preklad ako aj celkovú čitateľnosť spätne preloženého zdrojového kódu. Prekladače disponujú naviac možnosťou plánovať inštrukcie. Tým je možné efektívnejšie pracovať s poskytovanými zdrojmi pri výpočte. Príkladom môžu byť časovo relatívne náročné prístupy do pamäte RAM. Rozpestrením prístupov do pamäti medzi ostatné produktívne inštrukcie tak získavame možnosť paralelne vykonávať program bez nutného čakania na vybavenie požiadavku na sprístupnenie dát v pamäti RAM. Na druhú stranu pri rekonštrukcií idiomov tak vzniká ďalší požiadavok, a to počítať s možným planovaní inštrukcií a to nie len plánovanie prístupu do pamäti RAM.

3 INŠTRUKČNÉ IDIOMY

Spätný preklad väčších spustiteľných súborov je časovo a výpočetne pomerne náročná operácia. Maximálna optimalizácia a eliminácia zbytočného vyhľadávania idiomu, ktorý použitý prekladač nepoužíva je preto vhodná možnosť ako dobu spätného prekladu skrátiť. Pri rekonštrukcii inštrukčných idiomov je tak možné vybrať detekciu a následnú transformáciu len tých inštrukčných idiomov, ktoré použitý prekladač na danej architektúre používa. Pokiaľ nie je prekladač korektne zistený, prípadne si užívateľ želá vyhľadať všetky idiomy, aktuálna implementácia to umožňuje.

Príkladom inštrukčného idiomu môže byť delenie, ktoré je obecne veľmi náročná operácia z pohľadu taktov procesoru nutných pre získanie výsledku. Zamenenie delenia za bitový posun vpravo v prípade delenia mocninou dvojky je tak veľmi výrazná optimalizácia. Moderné a vysoko optimalizované prekladače ako napríklad `GCC`, prekladač obsiahnutý vo vývojovom prostredi `Visual Studio`, či prekladač od firmy `Intel`, inštrukciu delenia nikdy nepoužijú. Namiesto toho sa používa násobenie *magickou konštantou*[5] s dodatočným ustanovením výsledku pomocou posunov. Pri rekonštrukcii je tak nutné tento idiom korektne detekovať a správne odvodiť konštantu, čo je mnohokrát netriviálny problém.

```
1 int main(void) {
2   int a;
3   // ...
4   a = a / 10;
5
6   // ...
7 }
```

```
1 int main(void) {
2   int a;
3   // ...
4   a = (lshr(a * 1717986919, 32)
5         >> 2) - (a >> 31);
6   // ...
7 }
```

```
1 int main(void) {
2   int a;
3   // ...
4   a = a / 10;
5
6   // ...
7 }
```

Program, ktorý realizuje delenie číslom 10 (vľavo) je spätne preložený bez rekonštrukcie inštrukčných idiomov do podoby uvedenom v zdrojovom kóde uprostred ukážky. Na prvý pohľad je veľmi obtiažne usúdiť, že ide vo svojej podstate o delenie. I so znalosťou použitia idiomu je však dopočítanie konštanty náročné a zbytočne zaťažuje užívateľa spätného prekladača. Po rekonštrukcii inštruk-

čného idiomu, čo je predmetom tejto práce, je výsledok späného prekladu uvedený vpravo. Operácia použitá v programe sa tak stáva porozumiteľná a hneď pochopiteľná.

Samotná rekonštrukcia inštrukčných idiomov je realizovaná nad platformovo nezávislou reprezentáciou založenou na LLVM inštrukciách[4]. Je tak možné abstrahovať od konkrétnej inštrukčnej sady a inštrukčné idiomy tak hľadať nezávisle na cielovej platforme, pre ktorú bol spustiteľný súbor vytvorený.

```
1  %v1 = load i32* @regs1
2  %v2 = add i32 7, 0
3  %v3 = and i32 %v1, %v2
4  store i32 %v3 i32* @regs1
```

Príklad uvedený vyššie prezentuje platformovo nezávislý idiom. Počítanie operácie modulo mocninou dvojky je obecne náročnejšia operácia z pohľadu taktov na všetkých platformách. Preto prekladače medzi svoje platformovo nezávislé optimalizácie často zaraďujú i transformáciu modula mocninou dvojky na jednoduché vymaskovanie bitov – v prípade modula 8 je to vymaskovanie spodných troch bitov.

Najmä u prekladača `GCC` boli nájdené optimalizácie i niektorých knižničných funkcií, funkcií zo štandardnej knižnice jazyka C. Zámenou volania funkcie za niekoľko elementárnych inštrukcií bez nutnosti predávať riadenie je pomerne výrazná optimalizácia. Na druhú stranu tieto optimalizácie sú na prvý pohľad ťažko čitateľné v prípade spätného prekladu, preto ich zámena na pôvodné volanie je rovnako ako v prípade optimalizácií aritmetických veľmi vhodná.

4 ZÁVER

Výsledkom tejto práce je detekcia a následná zámena inštrukčných idiomov v rekonfigurovateľnom spätnom prekladači projektu Lissom. Výsledný spätne preložený kód je tak čitateľnejší a viac odpovedá pôvodne zapísanému zdrojovému kódu, ktorý bol pôvodne preložený. Najmä u prekladačov, ktoré sú distribuované pod uzavretou licenciou, je veľmi obtiažne nájsť všetky idiomy, ktoré používajú. Naviac je v niektorých prípadoch detekcia inštrukčných idiomov veľmi náročná. Napriek popísaným problémom je v súčasnosti detekovaných a následne transformovaných viac ako 20 inštrukčných idiomov dnes populárne používaných prekladačov na architektúrach `x86`, `ARM` a `MIPS`.

POĎAKOVANIE

Tento príspevok vznikol za podpory výskumného zámeru MSM0021630528 podporeným grantom TAČR TA01010667.

REFERENCE

[1] Warren, H. S.: Hacker's Delight, Addison-Wesley, 2002, ISBN-13 978-0-201-91465-8

[2] Stallman M. R. and *GCC* Developer Community: GNU Compiler Collection Internals, Free Software Foundation, Inc., 1998 – 2013.
URL `http://gcc.gnu.org/onlinedocs/gccint.pdf`

[3] Open Watcom Contributors: Open Watcom – Programmer's Guide, Sybase, Inc. and its subsidiaries, 1984 – 2002.
URL `http://www.openwatcom.org/ftp/manuals/current/pguide.pdf`

[4] Popis platformy *LLVM* [online], 2013 [cit. 2013-03-01]. Dostupný z WWW: `www.llvm.org`

[5] Granlund, T., Montgomery P. L., Division by Invariant Integers using Multiplication
URL `http://gmplib.org/~tege/divcnst-pldi94.pdf`