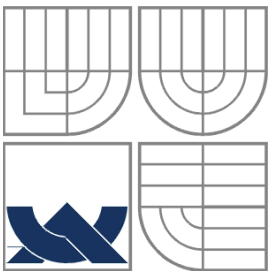
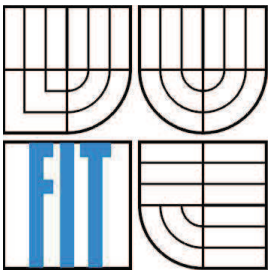


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATICKÉ VYHLEDÁVÁNÍ INSTRUKČNÍCH ROZŠÍŘENÍ APLIKAČNÍCH PROCESORŮ

AUTOMATIC SEARCHING OF INSTRUCTION EXTENSIONS FOR APPLICATION PROCESSORS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN ČEŠKA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. KAREL MASAŘÍK, Ph.D.

BRNO 2013

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2012/2013

Zadání bakalářské práce

Řešitel: **Češka Martin**
Obor: Informační technologie
Téma: **Automatické vyhledávání instrukčních rozšíření aplikačních procesorů**
Automatic Searching of Instruction Extensions for Application Processors
Kategorie: Překladače

Pokyny:

1. Seznamte se s vývojovým prostředím Codasip Framework, se způsobem generování kompilátoru C/C++, dále se způsobem profilování aplikací a vytváření instrukčních rozšíření aplikačních procesorů v tomto prostředí.
2. Seznamte se s kompilačním frameworkem LLVM.
3. Dle pokynů vedoucího navrhněte metodu pro nalézání instrukčních rozšíření aplikačních procesorů.
4. Zvolenou metodu implementujte a dostatečně otestujte.
5. Zhodnoťte svou práci z hlediska úspěšnosti nalezení instrukčních rozšíření na vhodné třídě aplikací.

Literatura:

- Manuál prostředí Codasip Framework a jazyka CodAL
- Manuál kompilačního frameworku LLVM
- Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997
- Dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1-3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Masařík Karel, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2012
Datum odevzdání: 15. května 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato práce se zabývá procesem automatického vyhledávání instrukčních rozšíření u aplikačně-specifických procesorů. K tomuto procesu je použito mírně upraveného algoritmu ISEGEN. Nejdříve jsou popsány veškeré důležité pojmy týkající se tohoto procesu včetně vybraného algoritmu ISEGEN. Následuje podrobný popis implementace celého procesu do jazyka C++. Nakonec je výsledný program zhodnocen na základě urychlení vykonávání vstupních programů na daném aplikačním procesoru při použití vyhledaných instrukčních rozšíření.

Abstract

This thesis deals with the process of automatic searching of instruction-set extensions for application-specific instruction-set processors. This process uses slightly edited ISEGEN algorithm. At first, all important terms including this algorithm are described. Then there is a detailed description of implementation of whole process in C++ programming language. At last, newly created program is considered as useful or useless based on speed-up of processor at performing of input program using found extensions.

Klíčová slova

ISE, rozšíření instrukční sady, ISEGEN, profilování, LLVM IR

Keywords

ISE, instruction set extension, ISEGEN, profiling, LLVM IR

Citace

Martin Češka: Automatické vyhledávání instrukčních rozšíření aplikačních procesorů, bakalářská práce, Brno, FIT VUT v Brně, 2013

Automatické vyhledávání instrukčních rozšíření aplikačních procesorů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Karla Masaříka, Ph.D. Další informace mi poskytl pan Ing. Adam Husár.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Češka
1.5.2013

Poděkování

Tímto bych rád poděkoval Ing. Karlu Masaříkovi Ph.D i Ing. Adamu Husárovi za jejich odbornou pomoc a cenné informace při tvorbě této práce.

Obsah

Obsah 1
1 Úvod 2
1.1 Cíl práce 2
1.2 Struktura dokumentu 2
2 Instrukční rozšíření 4
2.1 Procesory s aplikačně-specifickou instrukční sadou 4
2.2 LLVM IR 5
2.3 Profilování 6
2.3.1 Postup při vytváření profilu 6
2.4 Vytvoření grafu instrukcí 8
2.4.1 Rysy DFG 8
2.5 Algoritmus ISEGEN 9
2.5.1 Definice problému 9
2.5.2 Heuristika algoritmu 10
2.5.3 Zisková funkce 11
2.6 Generování instrukčního rozšíření 13
3 Implementace 14
3.1 Načtení dat 14
3.1.1 Počet vykonání základních bloků 14
3.2 Vytvoření grafu 14
3.2.1 Příklad vytvoření grafu 16
3.3 Algoritmus ISEGEN 20
3.3.1 IsegenAlg 20
3.3.2 Použité funkce 21
3.3.3 Příklad použití algoritmu 24
3.4 Generování ISE z řezu 26
3.4.1 Příklad vygenerovaní instrukčního rozšíření na základě řezu 28
4 Testování efektivity řešení 30
4.1 Množina testovaných programů 30
4.2 Výsledky testování 31
4.3 Vyhodnocení 33
5 Závěr 34
Literatura 35

1 Úvod

1.1 Cíl práce

Tato bakalářská práce pojednává o automatickém vyhledávání instrukčních rozšíření aplikačních procesorů. Aplikační procesory, mající schopnost rozšíření své instrukční sady novými instrukcemi podporovanými specifickými hardwarovými zdroji, si díky své flexibilitě získali velkou popularitu ve výrobě čipů. Výběr optimálních instrukčních rozšíření je kritickým prvkem, pro zvýšení výkonu aplikačních procesorů, a proto je mu v poslední době ve výzkumu přikládán vzhledem i k popularitě těchto procesorů stále větší význam.

Vyhledávání optimálních rozšíření je pro člověka relativně časově zdlouhavá záležitost, a proto bylo v posledních letech představeno několik technik napomáhajících k automatizaci tohoto procesu. Jednou z nich, algoritmem ISEGEN, se zabývá i tato práce. Tento algoritmus pracuje na principu algoritmu Kernighan-Lin, používaného pro řešení problému grafového dělení.

K vyhledávání instrukčních rozšíření dochází v rámci této práce uvnitř jednotlivých základních bloků vstupního programu. Pro každý takový základní blok pracuje algoritmus ISEGEN s jeho grafovou reprezentací. Jako typ grafu byl pro tento účel zvolen data-flow graph. Algoritmus se v grafu, znázorňujícím celý základní blok, snaží vyhledat řez splňující určitá omezení, který by měl být následně nahrazen instrukčním rozšířením.

Tato práce je zaměřena na vstupní programy napsané v jazyce C. Aby však bylo možné jednotlivé základní bloky těchto programů procházet, je potřeba jejich přeložení do vhodné formy. K tomuto účelu byl vybrán jazyk symbolických instrukcí LLVM IR.

Po nalezení optimálního řezu v grafu reprezentujícím základní blok následuje samotné vytvoření instrukčního rozšíření a jeho aplikace ve vstupním programu. Tato práce se zaměřuje pouze na druhý jmenovaný proces. V programu je vytvořena nová funkce, obsahující instrukce řezu, jejíž volání v příslušném základním bloku nahrazuje daný řez.

Samotným cílem této práce je aplikace právě popsaného procesu pro co nejefektivnější vyhledání instrukčních rozšíření. Výsledkem by tedy mělo být co nejvyšší zrychlení vykonávání vstupního programu na daném aplikačním procesoru.

1.2 Struktura dokumentu

V rámci této práce budou nejdříve představeny jednotlivé základní pojmy týkající se této problematiky, v rámci čehož bude také podrobně popsán vybraný algoritmus ISEGEN. V následující kapitole bude podrobně vysvětlena implementace celého procesu vyhledávání instrukčních rozšíření.

Nakonec bude u výsledného programu zhodnocena jeho účinnost, na základě urychlení vykonávání vstupních programů daným aplikačním procesorem.

2 Instrukční rozšíření

Hlavním cílem mé práce je urychlení vykonávání programu díky použití instrukčních rozšíření. Ta je možné generovat mnoha způsoby. Pro použití instrukčních rozšíření se v dnešní době používá aplikačně-specifických procesorů. Aby bylo možné instrukční rozšíření vytvořit, je potřeba zdrojový kód programu přeložit do jazyka symbolických instrukcí. K tomuto účelu byly zvoleny nástroje pro překlad do jazyka LLVM IR. Samotné vyhledávání instrukčních rozšíření pro aplikačně-specifické procesory je prováděno v programu přeloženém do tohoto jazyka.

2.1 Procesory s aplikačně-specifickou instrukční sadou

Vestavěné systémy jsou v dnešní době široce využívány v různých odvětvích [7]. Díky této skutečnosti je návrh moderních vestavěných systémů v nanometrových velikostech složitější než kdy dříve a problémy s tím spojené se stále zhoršují. Díky komplikovanosti a výzvám v elektrickém návrhu představovanými každou novou technologickou generací dochází k prohlubování těchto problémů, a to i přes přítomnost předražených vývojových nástrojů. Z tohoto důvodu vznikla potřeba použití programovatelných a rekonfigurovatelných řešení umožňujících větší flexibilitu pro specifikované změny a vyhýbání se návrhových chyb.

Aplikačně-specifické procesory si získali popularitu ve výrobě čipů stejně tak jako v jejich výzkumu. Nabízejí totiž funkční řešení pro kompromis mezi efektivitou a flexibilitou vestavěných systémů. Obecně platí, že aplikačně-specifické procesory (dále jen ASIP) mají schopnost rozšíření základní instrukční sady procesoru sadou upravených instrukcí podporovaných specifickými hardwarovými zdroji poskytnutými na ASIP. Hardware implementující specifické instrukce může být buď časově rekonfigurovatelnou funkční jednotkou, nebo předem syntetizovaným obvodem.

Výběr optimálních rozšíření instrukční sady je kritickým prvkem pro dosažení zvýšení výkonu ASIP. V rámci rozsáhlých programů se jedná o velmi náročný úkol, má-li být dosažen manuálním návrhem s přihlédnutím k různým návrhovým omezením (např. počet vstupních a výstupních operandů). Proto je zde potřeba, pro nejlepší možné využití výhody rozšiřitelnosti instrukční sady u ASIP, plně automatizovaného generátoru aplikačně-specifických instrukcí.

Několik technik napomáhajících k automatizaci návrhu již bylo v posledních letech představeno. Jednu z nich, popsanou níže, jsem si vybral pro uskutečnění mé bakalářské práce. Nejdříve je však potřeba popsat jazyk LLVM, v jehož reprezentaci je program analyzován, a na základě výsledků této analýzy jsou pak instrukční rozšíření generována.

2.2 LLVM IR

Low Level Virtual Machine Intermediate Representation je kódová reprezentace navržená pro použití ve třech rozdílných formách:

- jako vnitřní reprezentace vnitřně-paměťového překladače,
- jako na disku uložená reprezentace ve formátu bitcode vhodném pro rychlé načítání překladačem JIT (z ang. Just-In-Time),
- jako pro člověka čitelná reprezentace kódu ve formě jazyka symbolických instrukcí.

Tyto formy umožňují LLVM poskytovat mocnou vnitřní reprezentaci pro efektivní překladačové transformace a analýzy, při poskytování přirozených prostředků pro ladění a vizualizaci změn. Všechny tři formy jsou navzájem ekvivalentní [6]. V mé bakalářské práci budu pracovat převážně s reprezentací v jazyce symbolických instrukcí.

LLVM reprezentace je vytvořena tak, aby měla malou paměťovou stopu a pracovala na nízké, neboli instrukční úrovni, zatímco se jedná o jazyk expresivní, typovaný a zároveň rozšiřitelný. Díky svým vlastnostem bývá tento jazyk označován také jako „univerzální střední reprezentace“.

Jelikož jazyk LLVM poskytuje informace o typu, lze jej využít pro účely všemožných optimalizací.

2.3 Profilování

V informačních technologiích se profilováním nazývá proces zjišťování frekvence spuštění jednotlivých částí programu. Ten může být proveden instrumentováním kódu programu počitadly umístěnými do každé části programu. Při každém spuštění dané části pak dochází u příslušného počitadla k jeho inkrementaci. Z těchto dat lze poté velmi jednoduše spočítat přesnou frekvenci spuštění ke každé části. V rámci mé bakalářské práce využívám frekvence spuštění u základních bloků.

2.3.1 Postup při vytváření profilu

CLANG

Jedná se o přední část LLVM překladače jazyků C, C++ a Objective-C, který je zaměřený na poskytování velmi rychlých kompilací (přibližně 3 krát rychlejší než překladač GCC při kompilaci kódu v jazyce Objective-C v debugovacím módu) [6]. Poskytuje také velmi použitelné chybové zprávy a varování a rovněž slouží jako platforma pro sestavování nástrojů na zdrojové úrovni. Například *CLANG static analyzer* je nástrojem automaticky hledajícím chyby ve zdrojovém kódu a mimo to také skvělý příklad nástroje, který lze sestavit použitím překladače CLANG jako knihovny pro rozbor kódu v jazyce C nebo C++.

OPT

Nástroj `OPT` je modulární LLVM optimalizátor a analyzátor. Tento nástroj používá jako vstupní data LLVM zdrojové soubory, na kterých následně provádí specifikované optimalizace či analýzy a jako výstupní data vytváří optimalizovaný zdrojový soubor nebo výsledky analýzy. Tento program patří mezi základní stavební kameny mé práce. Při použití přepínačů `-block-freq` a `-insert-optimal-edge-profiling` totiž provádí instrumentaci vstupního programu pro základní bloky.

LLC

Program LLC překládá vstupní LLVM zdrojový soubor do jazyka symbolických instrukcí pro specifickou architekturu. Nízko-úrovňový výstup následně lze spustit assemblerem a sestavovacím programem a vytvořit tak spustitelný soubor.

LLVMC

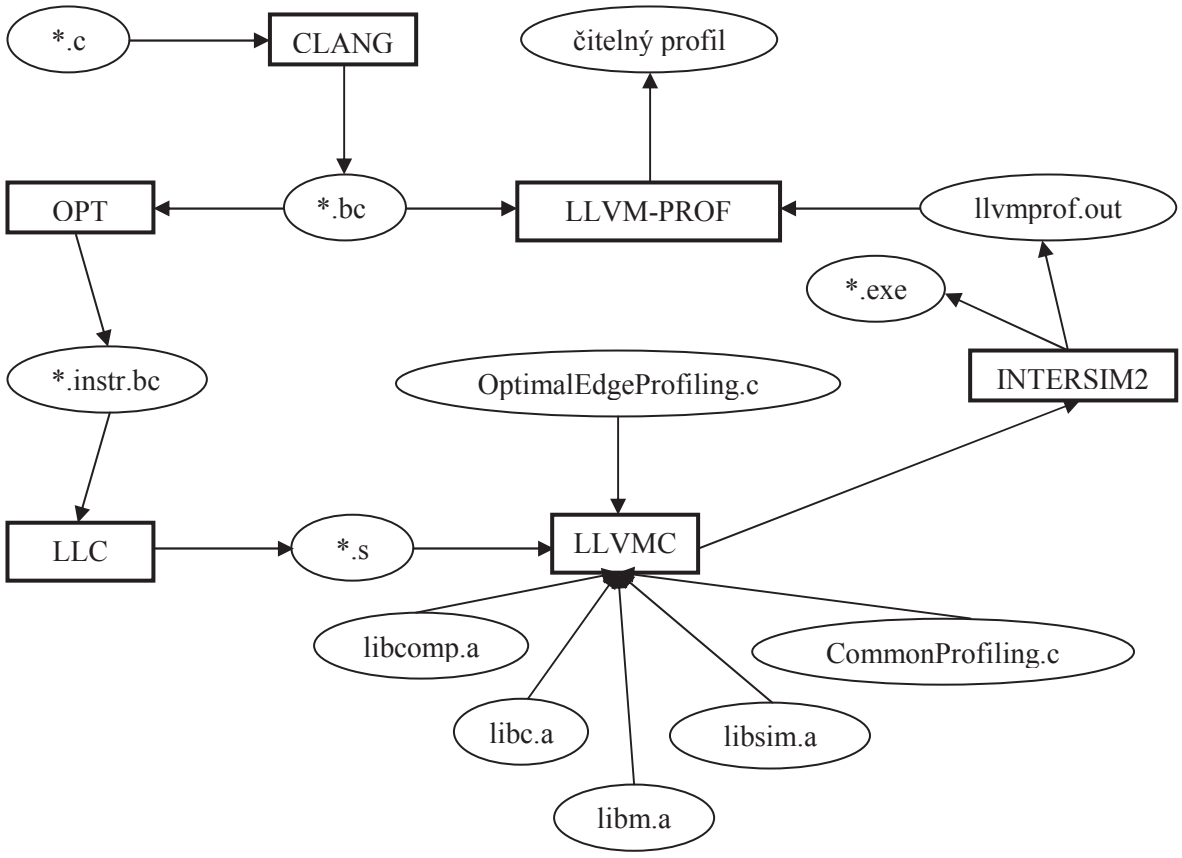
Tento nástroj slouží jako nastavitelný ovladač kompilátoru. Nejedná se přímo o kompilátor, optimalizátor, či sestavovací program. LLVMC sám pouze řídí software, který tyto úlohy provádí. LLVMC je velmi podobný nástroji GCC TOOL. V rámci mé bakalářské práce tento nástroj využívá mimo jiné také externí knihovny `CommonProfiling.c` a `OptimalEdgeProfiling.c`.

INTERSIM2

Jedná se o nástroj vykonávající samotnou simulaci. Na vstupu tento nástroj přijímá zdrojový program v jazyce symbolických instrukcí. Je-li tento program opatřen instrumentací, pak nástroj INTERSIM2 vytváří jako výstup soubor `llvmprof.out` obsahující data o frekvenci spuštění jednotlivých částí programu. Tento soubor však není v čitelné podobě, a proto je potřeba jej přeložit.

LLVM-PROF

Tento program čte na vstupu soubor `llvmprof.out` a zároveň soubor obsahující bitkódovou reprezentaci zdrojového souboru. Tu získáme jako výstup nástroje OPT. Program oba soubory zpracuje a vyprodukuje člověkem čitelnou zprávu obsahující informace získané profilací.



Obr. 2.3.1: Schéma postupu při vytváření profilu

2.4 Vytvoření grafu instrukcí

Pro hlavní část mého programu, nalezení vhodného rozšíření instrukční sady, je potřeba ze základního bloku předaného ve tvaru LLVM IR vytvořit vhodnou datovou strukturu. Vybraný algoritmus ISEGEN je vytvořen pro práci s grafy. S přihlédnutím ke skutečnosti, že pracujeme na úrovni instrukční zrnitosti („zrno“ reprezentuje instrukci), byl pro správnou funkci algoritmu zvolen typ grafu data-flow graph (dále jen DFG).

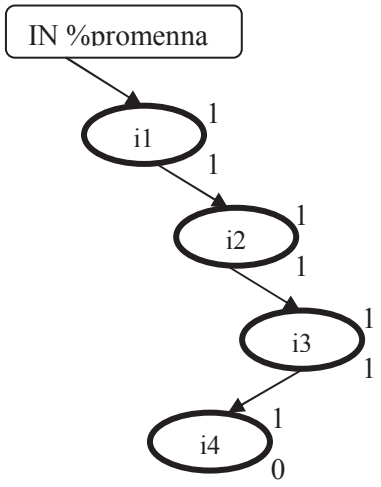
2.4.1 Rysy DFG

Struktura DFG odpovídá dvojici $G = (V, E)$, kde V jsou uzly reprezentující jednotlivé instrukce zpracovávaného základního bloku a E jsou hrany zachycující datové vztahy mezi nimi [1].

Každý uzel v grafu odpovídá jedné instrukci základního bloku. Hrany pak znázorňují datové toky mezi těmito instrukcemi. Vzhledem k našim potřebám (budou vysvětleny v sekci 2.5.2) musí každý uzel obsahovat kromě seznamů vstupních a výstupních hran také dvě číselné hodnoty. Ty reprezentují počet vstupních (I_{TOG}) a výstupních (O_{TOG}) hodnot k danému uzlu. Při vytváření grafu je hodnota I_{TOG} nastavena na počet vstupních hran do uzlu, zatímco hodnota O_{TOG} má většinou velikost inicializovanou na hodnotu 1 (výstup z běžných instrukcí je vždy právě jeden).

U uzlů je také potřeba určit, zdali mu není předávána hodnota z externího vstupu, či neukládá hodnotu na externí výstup. V takovém případě je uzel označen za tzv. bariéru (viz sekce 2.5.3). Stejně je uzel označen, týká-li se instrukce pracující s pamětí.

DFG je orientovaný (každá hrana má daný vstupní a výstupní uzel) a v naprosté většině případů se také jedná o graf acyklický (rozdílné vstupy a výstupy). Jednoduchý příklad DFG je znázorněn na následujícím schématu.



Obr. 2.4.12: Schéma postupu při vytváření profilu

2.5 Algoritmus ISEGEN

Tento algoritmus byl vytvořen výzkumným týmem ze Švýcarského federálního institutu v Laussane a Kalifornské univerzity [1].

Základním stavebním kamenem pro tvorbu ISEGENu se stala heuristika Kernighan-Lin. Kernighan-Lin min-cut algoritmus (dále jen K-L), aplikující tuto heuristiku, je velmi známý pro své použití při řešení problému grafového dělení. Původně byl však vytvořen pro dělení elektrických obvodů [2]. Pro použití K-L při rozšiřování instrukční sady bylo potřeba určitých úprav. Hlavním důvodem těchto úprav byla rozdílnost vstupního grafu obsahujícího instrukce vůči grafům, pro které byl K-L vytvářen.

2.5.1 Definice problému

Jak již bylo zmíněno v části 2.4, na úrovni instrukční zrnitosti (z Angl. instruction-level granularity) je hlavní datovou strukturou Graf datových toků. Pro připomenutí si uvedeme jeho základní rysy. Graf odpovídá dvojici $G = (V, E)$, kde V jsou uzly reprezentující jednotlivé instrukce zpracovávaného základního bloku a E jsou hrany zachycující datové vztahy mezi nimi [3]. V našem případě se v podstatě jedná o acyklický a orientovaný typ grafu.

Definujeme si řez C, pro který platí: $C \subseteq G$. C v našem případě reprezentuje možné rozšíření instrukční sady (dále jen ISE). Nechť $M(C)$ je funkce měřící přínos řezu C jako odhad zrychlení dosaženého při implementaci řezu C jako nového ISE. Samotný ISEGEN algoritmus nezávisí na definici měřící funkce.

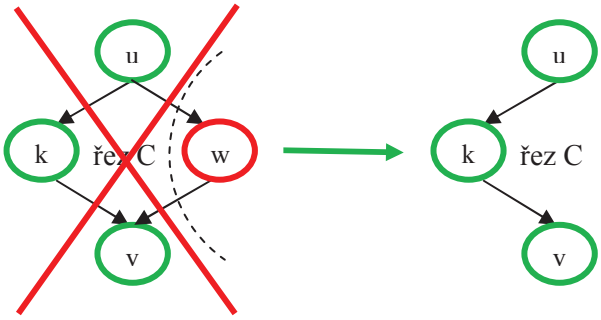
Nechť $IN(C)$ je funkce udávající počet uzlů grafu G, ze kterých do řezu vstupují hrany. Toto číslo odpovídá počtu vstupních hodnot do řezu C. Podobně $OUT(C)$ udává počet výstupních hodnot (uzly do nichž vedou hrany z C vystupující) z tohoto řezu. Tyto hodnoty jsou pak typicky následně použity v grafu G, či jiném základním bloku.

Dále je potřeba si vyjádřit konstanty reprezentující maximální počty vstupních a výstupních hodnot (typicky proměnných) ISE, které daná architektura podporuje. Tyto konstanty si označme jako N_{IN} (vstupní) a N_{OUT} (výstupní).

Problém generování ISE lze rozdělit do dvou následujících podproblémů:

1. Za předpokladu grafu datových toků $G = (V, E)$ vyjadřujícího daný základní blok, najdi řez $C \subseteq G$, který maximalizuje hodnotu $M(C)$ za následujících omezení:
    a. $IN(C) \leq N_{IN}$ a zároveň $OUT(C) \leq N_{OUT}$. Tímto omezením zajistíme validitu ISE s použitou architekturou.
    b. Řez C musí být konvexní. Tzn.: neexistuje cesta z uzlu $u \in C$ do jiného uzlu $v \in C$ přes uzel $w \notin C$. Jestliže řez konvexní není, pak nastává situace, kdy vstupní

operand ISE není dostupný v čase zavolání instrukce. Toto omezení zajišťuje architektonickou proveditelnost ISE. Nekonvexní řez může být přesto principiálně použit, avšak došlo by k příliš velkému nárůstu složitosti překladače. Problém je znázorněn na následujícím obrázku.



Obr. 2.5.13: Grafické vyjádření problému konvexnosti

2. Za předpokladu základního bloku v aplikaci a maximálního počtu ISE, najdi takové řezy, aby bylo docíleno co nejvyššího zrychlení aplikace.

2.5.2 Heuristika algoritmu

Algoritmus ISEGEN je, jak již bylo zmíněno, založen na principu K-L. Jedna z přebraných myšlenek je použita k tzv. řízení přepínání uzlů v grafu mezi softwarovou (dále jen SW) a hardwarovou (dále jen HW) částí založenému na ziskové (z angl. gain) funkci reprezentující návrhářovo počínání při ručním vytváření ISE. HW část reprezentuje řez C, zatímco SW část zbytek grafu G po odečtení tohoto řezu.

Efektivita K-L spočívá především ve schopnosti přecházení lokálních maxim bez jakýchkoliv zbytečných kroků. Označme si počet vstupů ISE v právě probíhajícím kroku algoritmu jako I_{RIS} a počet výstupů jako O_{ISE}. Jak jsem již uvedl v části 2.4, u každého uzlu jsou v grafu uvedeny 2 pomocné hodnoty: vstupní a výstupní změny na daném uzlu (I_{TOG} a O_{TOG}). Tedy při přepnutí uzlu z HW do SW dojde k přičtení těchto hodnot k aktuálním stavům vstupů a výstupů ISE (I_{ISE} a O_{ISE}).

Při inicializaci jsou hodnoty I_{ISE} a O_{ISE} rovny 0 a hodnoty I_{TOG} a O_{TOG} nastaveny na počet vstupů a výstupů korespondujícího uzlu.

Při přepnutí uzlu z HW do SW je velmi pravděpodobné ovlivnění hodnot I_{TOG} a O_{TOG} sousedních uzlů, tak aby tyto hodnoty odpovídali potřebné změně I_{ISE} a O_{ISE} při přepnutí daného sousedního uzlu.

Algoritmus funguje tak, že v iteracích prochází vstupní graf a pokaždé když dojde ke splnění následující podmínky: $N_{IN} = I_{ISE} \wedge N_{OUT} = O_{ISE}$, uloží se aktuální stav HW jako nejlepší možné ISE. Pokud v některém z kroků dojde k překročení maximálních počtů vstupních a výstupních hodnot ISE, algoritmus se nezastaví a pokračuje dále. K ukončení algoritmu dochází až v případě, že ani

po jedenácti iteracích od posledního splnění podmínky nedojde k jejímu opětovnému splnění. Počet iterací byl zvolen na základě experimentování.

Dalšími podrobnostmi o algoritmu včetně jeho implementace se budu zabývat v kapitole 3.2.

2.5.3 Zisková funkce

Při přepínání uzlu z SW do HW je potřeba zjistit, který z uzlů v SW je nejvýhodnější přepnout do HW, což zajišťuje zisková funkce. Tato funkce je navržena tak, aby co nejlépe vystihla, který z uzlů by se návrhář při ručním přidávání ISE snažil přepnout jako následující. Pro správnou funkčnost je potřeba splnit co nejlépe následujících 5 cílů:

1. maximalizovat zrychlení běhu programu použitím řezu,
2. uspokojit omezení pro počty vstupů a výstupů,
3. uspokojit omezení pro konvexnost řezu,
4. pokusit se o co největší řez,
5. povolit hledání nezávislých spojených komponent, pokud mají větší zrychlující potenciál.

Ke splnění těchto cílů se využívá následujících funkcí:

- **Měřící funkce**

  Nechť C′ je nový řez po přidání nebo odstranění uzlu n z řezu C přepnutím mezi HW a SW. Tato funkce zajišťuje dodržení 1. a 3. cíle.

$$merit = \begin{cases} M(C') & jestliže\ C'\ dodržuje\ omezení\ konvexivity \\ -\infty & jestliže\ C' omezení\ konvexivity\ porušuje \end{cases}$$

- **Porušení vstup-výstupních počtů**

  Vysoká hodnota je vrácena, pokud je maximální počet vstupních a výstupních hodnot překročen. Tato funkce zajišťuje dodržení 2. cíle.

$$io\_penlty = \left((I_{ISE}(C') - N_{IN}) + (O_{ISE}(C') - N_{OUT})\right)$$

- **Konvexní omezení**

  Přidání uzlu k řezu je ovlivněno umístěním jeho sousedů vůči řezu. Uzel přidávaný do uzlu je očekávatelný pro přidání, jsou-li jeho sousedi v řezu, zatímco uzel v řezu se hůře odstraňuje. Tato funkce zajišťuje dodržení 3. cíle.

$$conv = \begin{cases} +num\_nbrs(n, C) & jestliže\ je\ uzel\ n\ v\ SW \\ -num\_nbrs(n, C) & jestliže\ je\ uzel\ n\ v\ HW \end{cases}$$

- **Velký řez**

  Řez má povoleno růst v místech s vyšším potenciálem růstu. Externí vstupní a výstupní uzly se chovají jako bariéry, za které nelze růst. Od doby, kdy nelze přímo z ad-hoc funkční jednotky (bývalá část procesoru, která byla oddělena pro kritické výpočty, čímž došlo k větší efektivitě výpočetních operací), se paměťové operace staly barierou pro růst. Nechť d_to_bars_up(n) je minimální vzdálenost (v uzlech)

řezu od externích vstupních a paměťových bariér a d_to_bars_down(n) je minimální vzdálenost řezu od externích výstupních a paměťových bariér. Tato funkce zajišťuje dodržení 4. cíle.

$$cgp = \begin{cases} +|\text{d\_to\_bars\_up(n)} - \text{ d\_to\_bars\_down(n)}| & jestliže\ je\ uzel\ v\ SW \\ -|\text{d\_to\_bars\_up(n)} - \text{ d\_to\_bars\_down(n)}| & jestliže\ je\ uzel\ v\ HW \end{cases}$$

- **Nezávislé řezy**

  Existují případy, kdy spojenými nezávislými řezy dosáhneme větší efektivity. Přestože je tato možnost docela pravděpodobná u téměř každého většího základního bloku, nemusí být splněny ostatní náležitosti jako například vstup-výstupní omezení. Nechť CS(G) jsou spojené podgrafy v G vyjma podgrafu obsahujícího n. Tato funkce zajišťuje dodržení 5. cíle.

$$idc = \begin{cases} +max_{cs \in CS(G)} CP\_lat(cs) & jestliže\ je\ uzel\ n\ v\ HW \\ 0 & jestliže\ je\ uzel\ n\ v\ SW \end{cases}$$

  CP_lat(cs) vyjadřuje součet hardwarových zpoždění podél kritické cesty nezávisle spojených podgrafů cs. Použitím této komponenty je uzlům již umístěným v HW umožněno přepnutí zpět do SW za účelem růstu jiného podgrafu.

Celá zisková funkce se pak počítá následovně:

$$gain = \alpha_1 \cdot merit - \alpha_2 \cdot io\_penlty + \alpha_3 \cdot conv\_cons + \alpha_4 \cdot cgp + \alpha_5 \cdot idc$$

Vztahy mezi váhami $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5$ byly odvozeny experimentálně takto:

$$\alpha_1 = 4 \cdot \alpha_3$$

$$\alpha_2 = 2{,}5 \cdot \alpha_1$$

$$\alpha_3 = \alpha_4$$

$$\alpha_5 = 25 \cdot \alpha_1$$

2.6 Generování instrukčního rozšíření

Po nalezení ideálního řezu, který má být nahrazen speciální instrukcí, je potřeba vytvořit právě takové instrukční rozšíření, aby toto nahrazení mohlo být uskutečněno. Takový proces se skládá ze dvou hlavních částí. Je potřeba zároveň upravit samotný program tak, aby v něm byly původní instrukce nahrazeny novou speciální instrukcí, stejně tak jako upravit instrukční sadu procesoru. Nejprve bych rád popsal úpravu vstupního programu.

Úprava vstupního programu je stejně jako veškeré předchozí činnosti prováděna v LLVM IR reprezentaci. V ní je potřeba vytvoření nového základního bloku, který se naplní instrukcemi z řezu a bude reprezentovat základní blok pro nové instrukční rozšíření. To musí být nějakým způsobem v základním bloku, ve kterém jsme hledání prováděli, spouštěno. K tomuto účelu slouží nejlépe volání funkce. Proto je vytvořena nová funkce, obsahující pouze nově vytvořený základní blok, volaná z původního základního bloku. Vzhledem k heuristice algoritmu popsané v předchozí části, jsou funkci předávány parametry, pouze však v počtu nepřesahujícím maximum pro danou architekturu. Ty musí být správně provázány s instrukcemi přesunutými do nového základního bloku. Problém s provázaností je potřeba vyřešit i u návratových hodnot funkce. V rámci mé bakalářské práce však využívám pouze modelu procesoru, u kterého není žádné omezení v počtu vstupních a výstupních parametrů (teoretická neomezenost počtu registrů), a proto zůstává pouze starost s provázaností.

Druhou částí celého procesu je přidání nové instrukce do instrukční sady procesoru. Jak jsem již zmínil, moje bakalářská práce používá pouze model procesoru. Ten je popsaný v jazyce CodAL pro popis architektur. Pro přidání instrukčního rozšíření je využito tzv. inline assembleru.

3 Implementace

Cílem této bakalářské práce je vytvoření vyhledávače instrukčních rozšíření na základě zdrojového souboru. Pro účel implementace byl zvolen programovací jazyk C++. Program funguje jako tzv. průchod, k jehož spuštění dochází prostřednictvím nástroje *opt* (viz kapitola 2.3.1). Pro vytvoření správně fungujícího průchodu, pracujícího se vstupními daty ve formátu LLVM IR (viz 2.2) a také s profilovým souborem (viz 2.3), bylo použito mimo standardních knihoven jazyka C++ také několika externích knihoven vytvořených speciálně pro práci s daty v reprezentaci jazyka LLVM.

Program je vytvořen jako kombinace funkcionálního a objektově-orientovaného přístupu. Implementace probíhala v prostředí OS Fedora, a proto není zajištěna kompatibilita s OS Windows. Hlavní metodou, řídící průběh celého průchodu je metoda `runOnModule()`.

3.1 Načtení dat

Data profilového souboru jsou načtena do instance třídy `ProfileInfo`, zatímco data ze zdrojového souboru v reprezentaci LLVM do instance třídy `Module`. Oboje načtení je provedeno automaticky nástrojem `opt`. Veškerá následná práce s daty probíhá právě prostřednictvím těchto tříd.

3.1.1 Počet vykonání základních bloků

Jak již bylo zmíněno výše, data ze zdrojového souboru jsou načtena do instance třídy `Module`. Odtud postupně pomocí iterátoru dochází k průchodu seznamu funkcí zdrojového programu a v nich pak stejným způsobem i jednotlivých základních bloků.

U každého základního bloku pak v době, kdy je procházen, dochází ke zjištění počtu spuštění uloženého v již zmíněné instanci třídy `ProfileInfo` prostřednictvím metody `getExecutionCount()`. Není-li tato hodnota uvedena, je vrácen počet 0. Ke kontrole existence počtu spuštění slouží pomocná funkce `ignoreMissing()`. Počet spuštění je následně, spolu s ukazatelem na příslušný základní blok, uložen jako pár do předem definovaného vektoru.

Před dalším zpracování následuje seřazení těchto párů ve vektoru na základě počtu spuštění prostřednictvím funkce `sort()`. Při seřazení je použito struktury `PairSecondSortReverse`.

3.2 Vytvoření grafu

Jakmile je dokončeno seřazení jednotlivých párů ve vektoru, dochází k jejich postupnému průchodu. Pro každý základní blok se tak vytváří graf symbolizovaný množinou struktur typu `nodeISE`

reprezentujících jednotlivé instrukce v daném základním bloku. Naplnění těchto struktur provádí funkce `fillingTree()`.

Každá taková struktura obsahuje: ukazatel na příslušnou instrukci, několik důležitých příznaků potřebných pro správnou funkčnost mnou lehce upraveného algoritmu *ISEGEN*, počet vstupních a výstupních operandů, vzdálenosti od paměťových a terminátorových instrukcí směrem nahoru i dolů v grafu, hodnotu zisku a vektor instrukcí ve kterých je výsledek příslušné instrukce použit.

Struktura nodeISE:

- `Instruction* II;`
- `bool memoryInst;`
- `bool inCut;`
- `bool hasParentInCut;`
- `bool hasChildInCut`
- `bool cannotBeInCut;`
- `bool mark;`
- `int input;`
- `int output;`
- `unsigned memDistanceUp;`
- `unsigned memDistanceDn;`
- `double gain;`
- `std::vector<Instruction*> usedIn;`

Většina příznaků je inicializována na hodnotu 0, s výjimkou příznaku paměťových instrukcí. Pro zjištění počtu vstupních hodnot je použito metody `getOperand()`, která vrací ukazatel na operand. U každého z nich je pak pomocí funkce `isGlobalOrConstant()` zjištěno, zdali se nejedná o konstantu, nebo globální hodnotu. Pokud ne, je považován za vstup instrukce. Počet výstupů se zjišťuje jednoduše prostřednictvím metody `use_empty()`. Pokud má funkce nějaké použití, nastaví se počet výstupů na hodnotu 1.

Následně dochází k doplnění vztahů mezi uzly. U každého uzlu se naplní vektor `usedIn` veškerými instrukcemi, ve kterých je použit výsledek z instrukce náležící právě procházenému uzlu. Ty lze zjistit použitím metod `use_begin()` a `use_end()` třídy `Instruction` vracejících adresy začátku a konce jejich seznamu.

Po přidání vztahů mezi uzly je potřeba doplnit vzdálenosti uzlů od paměťových instrukcí, a to jak shora, tak i zdola (myšleno v souvislosti s grafem reprezentujícím celý základní blok). U uzlů reprezentujících instrukci typu terminátor, nebo instrukci pracující s pamětí jsou obě hodnoty vzdáleností nastaveny na 0. V rámci mého algoritmu je hodnota 0 nastavena také u některých speciálních instrukcí, jako jsou `PHI`, `ExtractValue` a jiné. V případě hodnoty globálního příznaku `USELOADSTORE` rovné 1 se vzdálenost u instrukcí `Load` a `Store` v tuto chvíli nenastavuje. Doplňování vzdáleností shora probíhá v rámci iterace všemi uzly v pořadí od prvního po poslední. U každého z nich je pak ke všem instrukcím z vektoru `UsedIn` nalezen uzel ke kterému tato

instrukce patří. K porovnání instrukcí slouží metoda `isIdenticalTo()`. Následně dochází k dědění hodnoty `memDistanceUp` inkrementované o 1 z rodiče na potomka (myšleno v rámci grafu jako rodičovský uzel a uzel potomka). Podobný, avšak lehce složitější princip slouží k doplnění hodnot vzdáleností zdola. K procházení jednotlivých uzlů dochází v opačném směru, čili od posledního. Ke každému uzlu se pak prohledává znovu celý vektor uzlů, dokud se nenajde takový uzel, u kterého vektor `UsedIn` obsahuje instrukci totožnou s instrukcí náležící k uzlu z opačně prohledávaného vektoru. V takovém případě se dědí z potomka na předka hodnota `memDistanceDn` rovněž inkrementovaná o 1. V případě, že je velikost vektoru `UsedIn` rovna 0, tedy uzel nemá potomka, se do `memDistanceDn` uloží hodnota 1.

Pokud by v mé bakalářské práci bylo použito algoritmu *ISEGEN* v podobě popsané v kapitole 2.5, bylo by potřeba ještě definování některých náležitostí, jako maximální počet vstupů a výstupů pro danou architekturu. V mém případě, díky neomezenosti počtu vstupů a výstupů pro nově vytvářené ISE, je tento algoritmus lehce pozměněn, a maximální hodnoty se v běžném případě nastavují na velikost `std::numeric_limits<unsigned>::max()`. Pouze při nastavení příznaku USED_MAX jsou maximální hodnoty překopírovány z konstant `MAX_INPUTS` a `MAX_OUTPUTS`.

Dalším krokem v iteraci jednotlivými základními bloky je spuštění mnou lehce upraveného algoritmu ISEGEN.

3.2.1 Příklad vytvoření grafu

K demonstraci vytvoření grafu jsem vybral jednoduchý program napsaný v jazyce C (Obr. 2.4.2-1), sloužící k výpočtu cyklického redundantního součtu (dále jen CRC). CRC je hašovací funkcí, používanou k detekci chyb během přenosu či ukládání dat [8]. Jedná se tedy o jeden ze způsobů tzv. kontrolního součtu. Vzhledem k četnosti použití takovéhoto programu je tedy důležité, aby vykonával svou funkčnost v co nejmenším čase, čímž se stává ideálním programem pro demonstraci mého algoritmu.

```
unsigned int crc32buf(unsigned int* buf, unsigned int len) {

     unsigned int oldcrc32;
     int ii, len_o;
     oldcrc32 = 0xFFFFFFFF;

     for (ii = 0; ii < 100000; ii++) {
            len_o = len;
            for ( ; len; --len, ++buf) {
```

```
                oldcrc32 = UPDC32(*buf, oldcrc32);
            }
            len = len_o;
            buf -= len;
    }
    return ~oldcrc32;
}

int main(int argc, char *argv[]) {
    unsigned int crc = { 0 };
          char res;
          crc = crc32buf(buf, 4);

    res = (crc >> 24 & 0xFF) ^ (crc >> 16 & 0xFF) ^ (crc >> 8 & 0xFF) ^
          (crc & 0xFF);
    return res;
}
```

CRC v jazyce C

Nejprve je potřeba přeložení zdrojového souboru do reprezentace v jazyce LLVM IR včetně následného vytvoření souboru s profilem (viz 2.2). Nástroje k těmto činnostem potřebné jsem již popsal v části 2.3.1. V rámci souboru s profilem zdrojového programu získáváme u každého ze základních bloků také číslo vyjadřující počet spuštění daného bloku při simulaci zdrojového programu. Jako výstup po profilaci získáváme také soubor *llvmprof.out*, jehož část obsahující právě seznam nejčastěji spuštěných základních bloků lze vidět na následujícím obrázku.

```
===-------------------------------------------------------------------------===
Top 20 most frequently executed basic blocks:

 ##      %%   Frequency
  1. 29.4117% 500000/1.7e+06  crc32buf() - for.cond1
  2. 23.5294% 400000/1.7e+06  crc32buf() - for.body2
  3. 23.5294% 400000/1.7e+06  crc32buf() - for.inc
  4. 5.8824% 100001/1.7e+06 crc32buf() - for.cond
  5. 5.88234% 100000/1.7e+06  crc32buf() - for.body
  6. 5.88234% 100000/1.7e+06  crc32buf() - for.end
  7. 5.88234% 100000/1.7e+06  crc32buf() - for.inc5
  8. 5.88234e-05%     1/1.7e+06 crc32buf() - entry
  9. 5.88234e-05%     1/1.7e+06 crc32buf() - for.end6
 10. 5.88234e-05%     1/1.7e+06 main() - entry

===-------------------------------------------------------------------------===
```

Obr. 3.1: Část programového profilu převedeného do čitelné formy

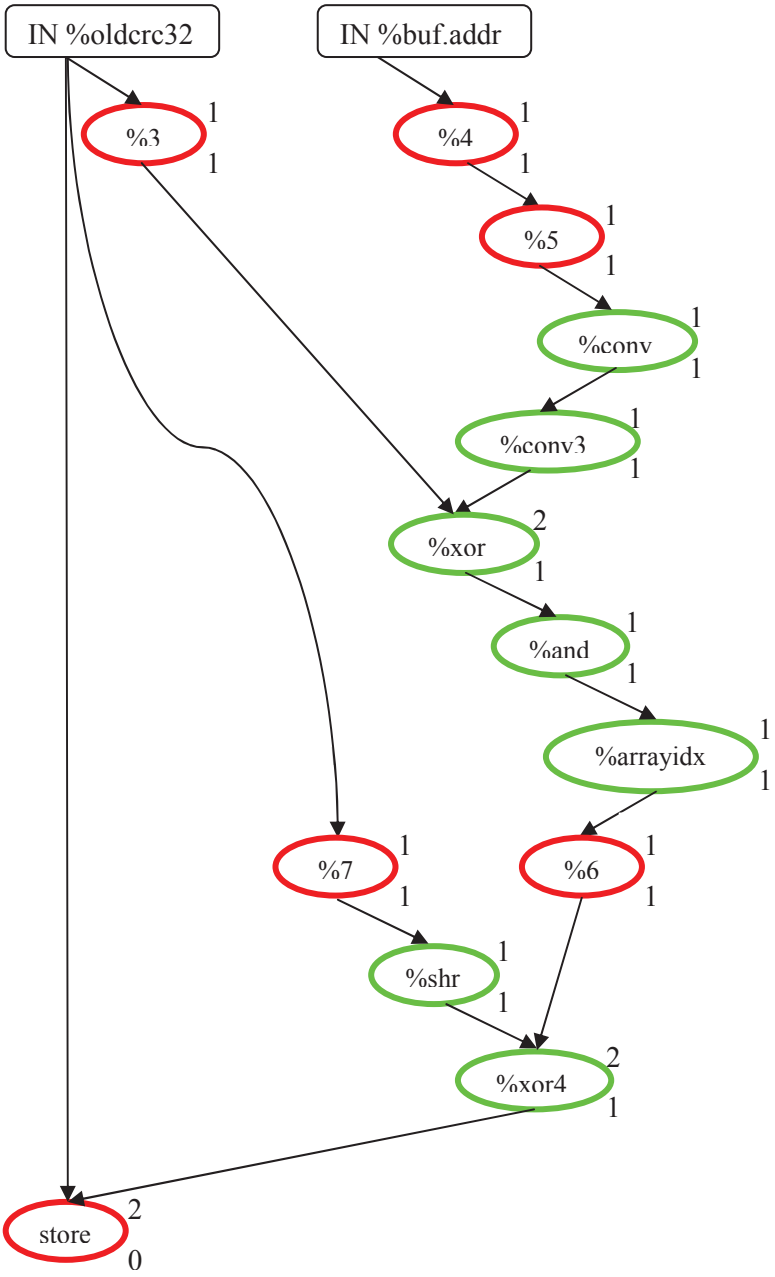
Tento seznam jsem uvedl pouze pro představu, jelikož, jak je možné si přečíst v kapitole 3.1, jej ve svém programu nijak nevyužívám. Vzhledem ke skutečnosti, že můj program prochází každý

základní blok, ve kterém hledá vhodné ISE, rozhodl jsem se uvést příklad pouze na jednom z nich. K tomuto účelu byl vybrán základní blok s druhým nejvyšším počtem spuštění, a jedním z nejvyšších počtu instrukcí. Z těchto důvodů je u tohoto základního bloku velká pravděpodobnost nalezení použitelného ISE.

```
for.body2:
    %3 = load i32* %oldcrc32, align 4
    %4 = load i32** %buf.addr, align 4
    %5 = load i32* %4
    %conv = trunc i32 %5 to i8
    %conv3 = sext i8 %conv to i32
    %xor = xor i32 %3, %conv3
    %and = and i32 %xor, 255
    %arrayidx = getelementptr inbounds [256 x i32]* @crc_32_tab,
          i32 0, i32 %and
    %6 = load i32* %arrayidx, align 4
    %7 = load i32* %oldcrc32, align 4
    %shr = lshr i32 %7, 8
    %xor4 = xor i32 %6, %shr
    store i32 %xor4, i32* %oldcrc32, align 4
    br label %for.inc
```

Druhý nejčastěji vykonávaný základní blok ze vstupního souboru v reprezentaci LLVM IR

Celý blok je zpracováván postupně, po jednotlivých instrukcích, přičemž dochází k vytváření datové struktury GDT po uzlech (viz 3.1). V rámci každého jednotlivého přidání uzlu musí být tedy struktura přidávaného uzlu naplněna informací o počtu jeho vstupů a výstupů, vzdáleností od paměťových instrukcí či terminátorů, a také nastaven příznak práce s pamětí společně s dalšími potřebnými příznaky. Kompletní struktura uzlu je znázorněna v předchozí kapitole. Výsledný graf, který vznikne po dokončení celého procesu, je znázorněn na obrázku na následující straně. Červeně jsou označeny paměťové instrukce, zatímco zeleně všechny ostatní. Vpravo od každé instrukce pak můžete najít informaci o počtu vstupů a výstupů z instrukce. Terminátor v tomto případě na grafu chybí, jelikož nemá ke grafu žádnou vazbu (nepoužívá výsledek ani jedné instrukce základního bloku).



Obr. 3.2: Druhý nejčastěji vykonávaný základní blok ze vstupního souboru v GDT reprezentaci

3.3 Algoritmus ISEGEN

Jak již bylo v této práci několikrát dříve zmíněno, jádro mé bakalářské práce, uskutečňující samotný výběr instrukčních rozšíření, je tvořeno algoritmem ISEGEN. Ten byl pro potřeby mého projektu lehce pozměněn. Narozdíl od originální verze, popsané v kapitole 2.5, je mnou použitý algoritmus v několika ohledech zjednodušen.

Jako jeden z důvodů tohoto zjednodušení lze uvést například vynechání nutnosti použití jakýchkoliv omezení pro počet vstupních operandů instrukčního rozšíření popsané v kapitole 2.6. Omezení existuje pouze pro počet výstupních operandů na pouhý jeden, jehož odůvodnění lze najít ve stejné kapitole.

V následující podkapitole popíši funkci `isegenAlg()`, vykonávající jádro tohoto algoritmu, a po ní v další podkapitole veškeré funkce v ní použité.

3.3.1 IsegenAlg

Funkce `isegenAlg()` byla vytvořena na základě algoritmu popsaném v pseudojazyce v [9]. Zásadním způsobem byla oproti algoritmu *ISEGEN* pozměněna především zisková funkce. Tím došlo k úpravě stěžejní části algoritmu, a proto bylo u algoritmu vynecháno šestinásobné opakování popsané v pseudojazyce. Ze stejného důvodu byla také odstraněná kontrola splnění podmínek po přidání uzlu do řezu. Funkce `isegenAlg()` je zobrazena zde:

```
static void isegenAlg (std::vector<nodeISE*> tree_nodes, treeInfo*
                       main_info) {
    while (isUnmarkedNode(tree_nodes)) {
        for (std::vector<nodeISE*>::iterator VS =
             tree_nodes.begin(), VE = tree_nodes.end(); VS != VE;
             ++VS) {
            (*VS)->gain = gainMerit(*VS, tree_nodes, *main_info);
        }
        nodeISE* best_node = bestGain(tree_nodes);
        if (best_node = NULL)
            break;
        if (best_node->gain < 0)
          break;
        toggleMark(best_node, main_info);
        toggleOther(best_node, tree_nodes);
        convexAll(tree_nodes);
    }
```

```
    return;
}
```

Jediný větší cyklus fungující v rámci algoritmu, který všech úpravách zůstal, je tedy zároveň hlavním cyklem algoritmu, který prochází jednotlivé uzly grafu, dokud nejsou všechny označeny. Na začátku každého cyklu dochází k výpočtu hodnoty ziskové funkce pro každý uzel v GDT. Tento výpočet je zajištěn funkcí `gainMerit()`. Po dokončení průchodu je spuštěna funkce `bestGain()` vracející ukazatel na uzel s nejvyšší hodnotou zisku. Je-li návratové hodnoty rovna NULL, pak již neexistuje v GDT žádný uzel, který by nepatřil do řezu a proto je hlavní cyklus přerušen a hledání instrukčního rozšíření ukončeno. V případě návratové hodnoty ukazující na uzel s hodnotou zisku menší než 0, dochází rovněž k přerušení hlavního cyklu, a tím i ukončeno hledání instrukčního rozšíření. V tomto případě již totiž neexistuje uzel, který by bylo možno přidat do řezu. Pokud je hodnota zisku u vráceného uzlu vyšší než 0, pak algoritmus pokračuje spuštěním funkce `toggleMark()`, která daný uzel označí jako převedený z HW do SW části (viz 2.5.2) a upraví jeho hodnoty vstupů a výstupů. Poté dochází ke spuštění funkce `toggleOther()`, která vypočte nové hodnoty pro počet vstupních a výstupních operandů pro možné změny při převádění do SW. Jako poslední je v rámci hlavního cyklu spouštěna funkce `convexAll()`. Ta upravuje příznaky u uzlů v grafu tak, aby se při výpočtu zisku jednotlivých uzlů nemohlo stát, že by se následně do SW části dostal uzel, který by narušil pravidlo konvexnosti řezu.

3.3.2 Použité funkce

isGlobalOrConstant

Tato funkce je na rozdíl od všech ostatních, uvedených v této podkapitole, použita při vytváření grafu, stojí však za zmínku. Funkce využívá operátoru `isa<>` zjišťujícího, zdali je operand instancí určené třídy. Díky tomu lze jednoduše zjistit, jestli je předávaný operand globální hodnotou, nebo konstantou. Pouze v takovém případě vrací tato funkce hodnotu `TRUE`.

isUnmarkedNode

Jedná se o velmi jednoduchou funkci procházející prostřednictvím iterátoru jednotlivé uzly v předávaném vektoru uzlů. V případě, že narazí na uzel s příznakem `mark` rovným nule, vrací hodnotu `TRUE`.

gainMerit

Funkce `gainMerit()` je pravděpodobně nejdůležitější a zároveň nejsložitější funkcí použitou v `isegenAlg()`. Tato funkce pracuje na základě kapitoly 2.5.3, z čehož vyplývá její rozdělení na 5 samostatných částí. Každá z nich produkuje výsledek typu `integer` a na konci celé funkce jsou jednotlivé výsledky sečteny a výsledná hodnota je vrácena, jako hodnota zisku uzlu, který je funkci předáván.

Na začátku funkce, ještě před vykonáváním jednotlivých částí, dochází ke zjištění, zdali uzel na základě příznaku neobsahuje paměťovou funkci, některou ze mnou pro ISE nepoužívaných instrukcí, nebo se nejedná o terminátor. V těchto případech je funkce okamžitě ukončena se zápornou návratovou hodnotou. Při příznaku `USELOADSTORE` nastaveném na hodnotu 1 se funkce pro paměťové instrukce `Load` a `Store` neukončuje.

Jako první z pěti částí přichází na řadu kontrola pravidla konvexnosti, při přiřazení uzlu do řezu. Ta se kontroluje na základě příznaku `cannotBeInCut` nastavovaného ve funkci `convexAll()` vždy v předchozí iteraci hlavního cyklu. V případě nekonvexnosti, je funkce ukončena se zápornou návratovou hodnotou.

Druhá část, kontrola vstupně-výstupních podmínek, je silně ovlivněna neexistujícím omezením pro maximální počet vstupních operandů, popsaným v kapitole 2.6. Oproti předloze sice nijak upravena nebyla, avšak v běžných případech jsou tyto podmínky naprosto nepotřebné. Pouze při nastaveném příznaku USED_MAX na hodnotu 1 lze očekávat vzhledem k nižším hodnotám maximální velikosti vstupních či výstupních operandů nově vytvářeného ISE pro danou architekturu, že v některých případech tyto podmínky nepustí všechny uzly, aby mohly být přidány do řezu. Takové případy jsou vyřešeny rovněž zápornou návratovou hodnotou.

Třetí část této funkce zjišťuje počet sousedů předávaného uzlu v řezu. Na začátku jsou inicializována hodnota počtu všech sousedů `cntAll` a sousedů v řezu `cntInCut` na velikost 0. V rámci této části dochází k postupnému průchodu jednotlivých uzlů v řezu. V rámci každého uzlu je procházen také vektor instrukcí `usedIn`, ve kterých je výsledek instrukce pařící k předávanému uzlu použit. Pokud právě procházený uzel patří k instrukci totožné s některou z instrukcí z vektoru `usedIn` - metodou `isIdenticalTo()`, pak se jedná o „potomka“ předávaného uzlu, a následuje inkrementace `cntAll`. V případě, že nalezený „potomek“ patří do řezu, inkrementuje se i hodnota `cntInCut`. Po tomto porovnání se pro každý z uzlů prochází jejich vektor `usedIn`. Je-li nalezena shoda mezi instrukcí z vektoru a instrukcí náležící předávanému uzlu, pak vektor náleží k uzlu „rodiče“ předávaného uzlu. V tomto případě dochází opět k inkrementaci hodnoty `cntAll`. Pokud „rodič“ patří do řezu, pak je inkrementována také hodnota `cntInCut`. Na konci je počet sousedů uložen do hodnoty `convCons`.

Čtvrtá část počítající vzdálenosti od paměťových instrukcí či terminátorů je velice primitivním uložením součtu vzdáleností `memDistanceUp` a `memDistanceDn` patřících k předávanému uzlu do proměnné `cgp`.

Jelikož jsem svůj algoritmus upravoval tak, aby hledal pouze 1 rozšíření pro každý základní blok, je v něm pátá část vynechána. Na konci je funkcí vrácen součet hodnoty `cgp` s hodnotou `convCons` vynásobenou konstantou 10. Ta byla určena tak, aby bylo vyhledávání co nejefektivnější, a tudíž se hledají nejprve sousedé uzlů již umístěných v řezu, a až teprve poté uzly, které mají největší potenciál růstu.

bestGain

Tato funkce slouží ke zjištění uzlu s nejvyšší hodnotou zisku. Na začátku se inicializuje hodnota nejvyššího zisku `best` na zápornou hodnotu `std::numeric_limits<double>::max()`. Kromě toho se také vytvoří prozatím prázdný ukazatel na uzel, který bude sloužit jako návratová hodnota. Následně se prochází veškeré uzly v GDT. Je-li aktuálně procházený uzel již součástí řezu, pak se automaticky přeskakuje. Pro každý uzel ležící mimo řez, s hodnotou `gain` vyšší, než aktuální hodnota uložená v `best`, je jeho hodnota `best` nahrazena jeho `gain`. Návratová hodnota se nastaví na ukazatel na tento uzel. Po ukončení průchod je vrácena návratová hodnota aktuálně ukazující na uzel s nejvyšším ziskem. Zůstane-li hodnota `best` i po skončení průchodu oproti počáteční hodnotě nezměněna, pak funkce vrací hodnotu `NULL`.

toggleMark

Tato funkce pouze nastavuje hodnotu příznaku `inCut` předávaného uzlu na 1 a mění hodnotu počtu jeho a řezových výstupních operandů. Tuto činnost uskutečňuje změnou znaménka u hodnoty `output` přidávaného uzlu a přičtením původní hodnoty k `Cout` hodnotě struktury obsahující informace o řezu.

toggleOther

Funkce `toggleOther()` má za úkol provést změny v hodnotách `output` u všech uzlů ovlivněných přidáním uzlu s nejvyšším ziskem do řezu. Ovlivněny jsou pouze uzly s přidávaným uzlem sousedící. Funkce tedy prochází s použitím iterátoru veškeré Instrukce uložené ve vektoru `usedIn` přidávaného uzlu. V rámci každé iterace jsou procházeny všechny uzly GDT. V případě, že instrukce náležící právě procházenému uzlu je shodná s instrukcí z vektoru `usedIn`, jedná se o „potomka“. Pokud tento uzel není v řezu, pak se od jeho hodnoty `output` odečte číslo 1. V rámci procházení každého uzlu GDT se rovněž prochází všechny instrukce v jeho vektoru `usedIn`. Pokud je některá z nich identická s instrukcí náležící přidávanému uzlu, pak se jedná o jeho „rodiče“. U něj, stejně jako u „potomka“, je od hodnoty `output` odečtena 1, ale rovněž pouze pokud daný uzel neleží v řezu.

convexAll

Tato funkce, spouštěná vždy po přidání uzlu do řezu, upravuje příznaky zajišťující konvexnost řezu u všech uzlů GDT. Upravovanými příznaky jsou `hasParentInCut`, `hasChildInCut` a `cannotBeInCut`. Již z názvu vyplývá, že první dva jmenované příznaky mají spíše informativní charakter, zatímco třetí je nejdůležitější a je později použit i ve funkci `gainMerit()`, kde v případě hodnoty rovné 1 zabraňuje přidání uzlu do řezu. Samotná funkce je rozdělena na tři základní části.

V první z nich se pouze pro každý uzel nastavuje hodnota `cannotBeInCut` na 0. Tato část zajišťuje bezchybné zajištění konvexnosti, které by jinak bylo narušeno změnou hodnoty `inCut` v přidávaném uzlu, což vyplyne z dalších dvou částí.

Ve druhé části dochází k zajištění konvexnosti v GDT ve směru dolů od uzlů v řezu. Funkce postupně v cyklu prochází veškeré uzly z GDT. U každého z nich dochází k průchodu instrukcí z vektoru `usedIn`. Pro každou instrukci je pak zvlášť procházen znovu každý uzel z GDT. Pokud se nalezne uzel patřící k instrukci z vektoru `usedIn`, zkontroluje se několik podmínek, na jejichž základě se stanoví hodnoty příznaků nalezeného uzlu s identickou instrukcí. Pokud je rodič uzlu v řezu, pak se nastaví hodnota `hasParentInCut` nalezeného uzlu na 1. Pokud uzel nemá rodiče v řezu, ale jeho rodič má hodnotu `hasParentInCut` nastavenou na 1, pak dochází, pro zachování konvexnosti řezu, k nastavení příznaku `cannotBeInCut` u nalezeného uzlu na hodnotu 1, stejně tak jako u příznaku `hasParentInCut`.

Třetí část funguje podobně jako část druhá, pouze s tím rozdílem, že k průchodu uzly GDT dochází v opačném pořadí, tedy zdola nahoru (myšleno v grafu). Pro každý uzel se pak prochází jeho jednotlivé operandy, zjištěné metodou `getOperand()`. U každého z nich pak je zkontrolováno, jedná-li se o operand obsahující hodnotu z jiného uzlu GDT pomocí funkce `isGlobalOrConstant()`. V případě návratové hodnoty rovné `TRUE` se operand přeskakuje. V rámci průchodu každého operandu je pak znovu procházen celý vektor obsahující uzly z GDT. Nalezne-li se na uzel, kterému náleží instrukce identická s operandem za podmínky nenáležitosti uzlu k řezu, pak se u něj, stejně jako ve druhé části funkce, kontroluje několik podmínek pro nastavování příznaků. Má-li nalezený uzel potomka patřícího do řezu, pak se u něj nastaví příznak `hasChildInCut` na hodnotu 1. V opačném případě se kontroluje, zdali má jeho potomek hodnotu `hasChildInCut` rovnu 1. V kladném případě jsou nastaveny příznaky `hasChildInCut` i `cannotBeInCut` na hodnotu 1.

3.3.3 Příklad použití algoritmu

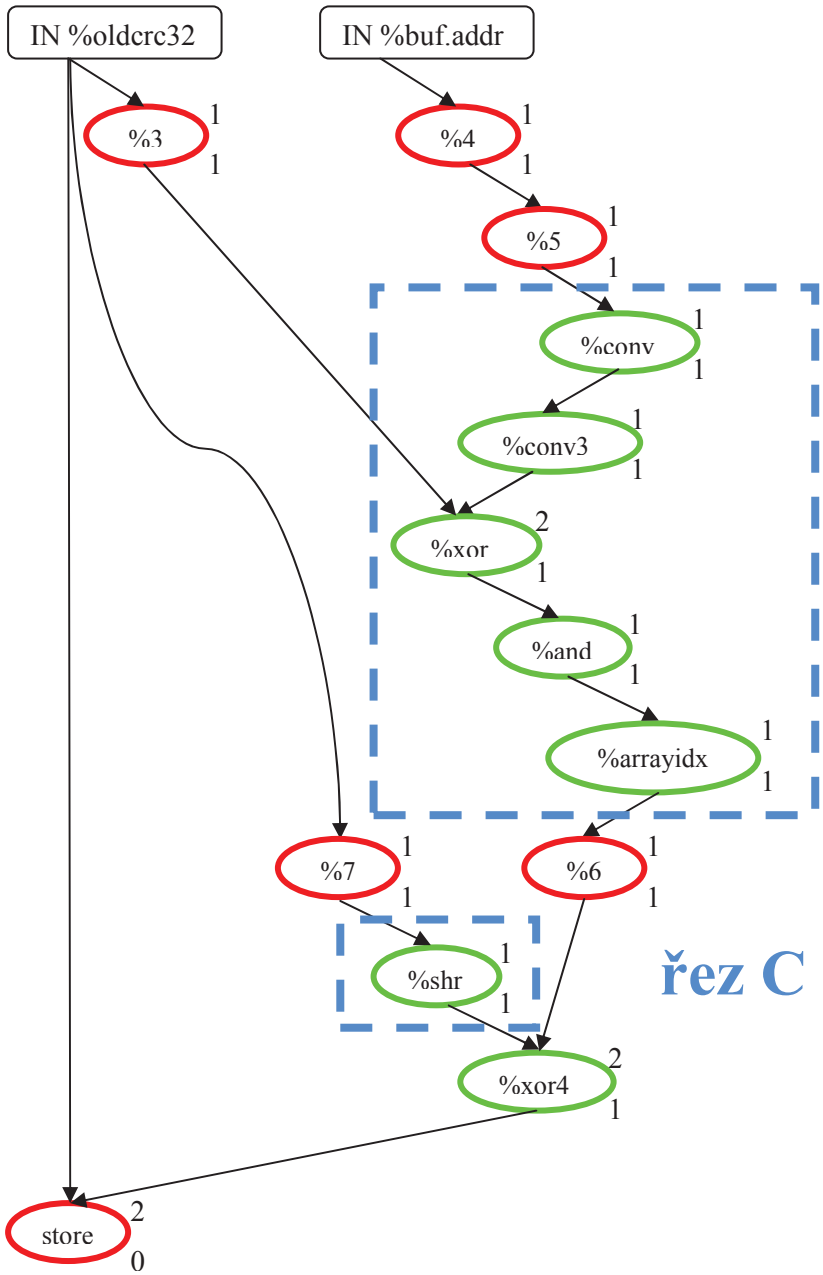
Pro znázornění funkčnosti mnou upraveného algoritmu bych rád pokračoval v příkladu uvedeném v kapitole 3.2.1. Pokud se podrobně podíváme na výstup po vytvoření grafu, lze již dopředu velmi jednoduše odhadnout podobu budoucího řezu.

Existují 2 možnosti, jak by mohl konečný řez vypadat. Rozdíl mezi nimi je převážně v použití příznaku `USELOADSTORE`, který, jak již bylo zmíněno výše, určuje použití instrukcí `Load` a `Store` v řezu. Pro názorný příklad bude hodnota tohoto příznaku nastavena na 0. Je tedy jasné, že maximální počet instrukcí v řezu bude 7.

Algoritmus funguje tak, že postupně přidává do řezu vždy jednu instrukci, na základě nejvyšší hodnoty zisku uložené v `gain` u každého uzlu. Jako první dojde k výběru instrukce označené v grafu jako `%arrayidx`.

Z výpočtu ve funkci `gainMerit()`, počítající nové hodnoty `gain` po každém přidání nového uzlu do řezu, je zřejmé, že následně se budou do řezu přidávat sousední uzly (pokud takové existují). Postupně, vzhledem k dodržení podmínek konvexnosti, tedy dojde k přidání postupnému

instrukcí `%and`, `%xor`, `%conv3` a `%conv`. Poslední přidanou instrukcí je instrukce s označením `%shr`. Jediná instrukce, s možností přidání do řezu, která nezůstane přiřazena, je instrukce označená `%xor`. Důvod jejího nepřidání je porušení pravidla konvexnosti řezu. Výsledný řez tedy bude vypadat následovně:



3.4 Generování ISE z řezu

Po nalezení řezu ve stromu nastává kontrola jeho velikosti. V případě, že uzlů v řezu nepřesahuje hodnotu 4, pak se nejedná o řez, na jehož základě by bylo jakkoliv efektivní vytvářet instrukční rozšíření. Pokud takový případ nastane, ke generování ISE nedochází, paměť obsahující data o uzlech celého stromu je uvolněna, a průchod aktuálním základním blokem je ukončen příkazem `continue`. Má-li však nalezený řez více než 4 uzly, pak algoritmus u aktuálně procházeného základního bloku zůstává a pokračuje voláním funkce `iseFunctionGenerator()`.

Na začátku této funkce je definováno několik potřebných vektorů: `inputTypes` pro typy operandů vstupující do nově vytvářené funkce, `inputVals` pro hodnoty těchto operandů, `outIns` pro instrukce jejichž výsledek je vracen funkcí jako návratová hodnota, `outVals` pro návratové hodnoty typu `Value`, `outTypes` pro typy těchto návratových hodnot, `outUses` pro instrukce využívající návratové hodnoty a `outIndexes` pro umístění na kterých jsou návratové hodnoty v instrukcích z vektoru `outUses` použity.

Aby bylo možno vytvořit instrukční rozšíření, je potřeba nejdříve vytvořit novou funkci, která ho bude reprezentovat. K tomu je však nutné znát veškeré její vstupní a výstupní operandy. Ty jsou zjišťovány postupnou iterací jednotlivými instrukcemi v řezu. Průchod každou instrukcí má dvě fáze.

V první z nich dochází k iteraci jednotlivými instrukcemi vektoru `usedIn`. Pro každou z nich je hledán příslušný uzel ve vektoru `tree_nodes`. V případě, že příslušný uzel neleží v řezu, pak se zcela jistě jedná o situaci, kdy výsledek instrukce z řezu slouží jako výstupní hodnota instrukčního rozšíření. Instrukce z řezu je vložena do vektoru `outIns` a její typ, zjisteny metodou `getType()`, do vektoru `outType`. Pro každou instrukci, u které byla nalezena shoda, se také procházejí všechny její operandy. Jakmile je nalezen operand, kterým tato instrukce přijímá hodnotu z instrukce v řezu, vloží se tento operand do vektoru `outVals`, instrukce, která ho přijímá do vektoru `outUses` a pořadí operandu mezi všemi operandy přijímající instrukce do vektoru `outIndexes`.

Ve druhé fázi se pro každou instrukci v řezu procházejí veškeré její operandy. Pokud je operandem globální hodnota, nebo konstanta, přeskakuje se. V ostatních případech se prochází pro každý z nich všechny uzly a hledá se takový, který neleží v řezu a je shodný s operandem. Pokud se takový nalezne, pak je uložen typ instrukce příslušné k nalezenému uzlu do vektoru `inputTypes` a procházený operand do vektoru `inputVals`. V případě, že žádný shodný uzel nalezen není, je zřejmé, že se jedná o hodnotu předávanou z vnějšku základního bloku a oba vektory se naplní příslušnými informacemi.

Po získání informací o vstupech a výstupech je zjištěn název nově vznikající funkce prostřednictvím funkce `getNameWithNumber()`. Ta vygeneruje název obsahující číslo na základě počtu již vytvořených instrukčních rozšíření při průchodu předchozích základních bloků. Další informace potřebná pro vytvoření funkce je její typ. Vzhledem k teoretickému neomezenému počtu

výstupních hodnot funkce je jako její návratová hodnota vytvářena struktura. Zůstal-li po zjišťování výstupních hodnot ISE vektor výstupních typů `outTypes` prázdný, pak se do něj vkládá typ `void`. Samotné vytvoření struktury je prováděno metodou `create()`. Následně je na základě této struktury a vektoru vstupních typů metodou `get()` vytvořen typ pro novou funkci. Nyní již přichází na řadu samotné vytvoření funkce metodou `Create()`.

Následně jsou procházeny veškeré vstupní argumenty této funkce. Předtím je ještě inicializován iterátor `IS` vektoru `inputVals` na jeho začátek. Pro každý ze vstupních argumentů se iteruje vektorem uzlů v řezu. V každém uzlu jsou procházeny vstupní operandy příslušné instrukce, dokud není nalezena instrukce shodná s hodnotou, na kterou ukazuje iterátor `IS`. V takovém případě je na místo tohoto operandu nastaven právě procházený vstupní operand instrukce. Poté následuje inkrementace iterátoru `IS`.

Konečně, aby nová funkce mohla správně fungovat, je potřeba k ní přiřadit základní blok. Ten se tedy vytváří metodou `Create()`. Následně je vytvořen prázdný seznam instrukcí `InstListiseX` patřící k tomuto základnímu bloku. Poté jsou veškeré instrukce patřící do řezu metodou `removeFromParent()` vymazány z originálního základního bloku a vloženy do seznamu instrukcí `InstListiseX`. Tím se zároveň vloží i do nového bloku.

Následuje vymazání všech instrukcí ve vektoru tree_nodes nepatřících do řezu z jejich základního bloku, čímž se vyprázdní originální základní blok. K tomu se vytvoří příslušný seznam instrukcí `instListMain`. Ihned poté se vloží do tohoto seznamu veškeré instrukce, které mají v originálním základním bloku instrukčnímu rozšíření předcházet, což se zjišťuje podle hodnoty příznaku `hasChildInCut`.

Pro dokončení funkce reprezentující instrukční rozšíření je také potřeba naplnit návratovou strukturu příslušnými hodnotami. Nejprve se vytvoří instance třídy `UndefValue` metodou `get()` s návratovým typem funkce jako parametrem. Poté je tato instance postupně plněna instrukcemi třídy `InsertValueInst`, vytvářenými metodou `Create()`, vkládajícími do ní jednotlivé hodnoty z vektoru `outVals`. Nakonec je prostřednictvím tzv. builderu vložena na konec seznamu `InstListiseX` instrukce návratová instrukce vracející naplněnou instanci.

Tím je nová funkce kompletně dokončena. Následuje extrahování hodnot z vrácené struktury v originálním základním bloku. Tento proces probíhá v rámci iterací hodnotami vektoru `outUses`. Mimo to se navíc iteruje vektorem `outIndexes`. V každé iteraci jsou pak postupně extrahovány jednotlivé hodnoty z návratové struktury. Hodnota je vždy následně po extrahování vložena do instrukce z vektoru `outUses` na místo dané hodnotou ve vektoru `outIndexes`.

Nakonec jsou do originálního základního bloku, na základě hodnoty `hasParentInCut`, vloženy instrukce, které za voláním nové funkce následují.

3.4.1 Příklad vygenerovaní instrukčního rozšíření na základě řezu

Ke znázornění vygenerování ISE využiju stejného ukázkového příkladu, jako v předchozích kapitolách. Původní podoba základního bloku, ve kterém se ISE vytváří je znázorněna v části 3.2.1.

Z tohoto základního bloku jsou veškeré instrukce patřící do řezu přesunuty do nově vytvořeného bloku. Ten je následně přiřazen nově vzniklé funkci reprezentující nové ISE. U instrukcí v řezu také kromě jejich přesunutí do jiného základního bloku dochází ke změně některých jejich operandů tak, aby jako operandy přijímali vstupní argumenty funkce. Ty jsou předávány při jejím volání, a to z instrukcí, ze kterých předtím jejich hodnoty instrukce v novém základním bloku přijímali přímo, ale nyní již nesou ve stejném bloku, a proto je potřeba je předávat skrz volání funkce.

V nově vzniklém bloku je pak potřeba ze stejného důvodu naplnit také návratovou hodnotu obsahující strukturu s hodnotami instrukcí náležících řezu použitými jako operandy instrukcí řezu nenáležícími.

Nakonec je potřeba tyto návratové hodnoty ze struktury extrahovat a nahradit jimi původní operandy v instrukcích v původním základním bloku.

Konečná podoba nově vzniklé funkce a původního základního bloku je znázorněna na následujícím výpisu.

```
for.body2:
    %4 = load i32* %oldcrc32, align 4
    %5 = load i32** %buf.addr, align 4
    %6 = load i32* %5, align 4
    %8 = load i32* %oldcrc32, align 4
    %ise = call %structureISE @ise1(i32* %6, i32* %4, i32* %8)
    %ex = extractvalue %structureISE %ise, 0
    %ex1= extractvalue %structureISE %ise, 1
    %7 = load i32* %ex, align 4
    %xor4 = xor i32 %7, %ex0
    store i32 %xor4, i32* %0, align 4
    br label %for.inc

define %structureISE @ise1(i32*, i32**, i32*, i32*) {
entryISE:
    %conv = trunc i32 %0 to i8
    %conv3 = sext i8 %conv to i32
    %xor = xor i32 %1, %conv3
```

```
    %and = and i32 %xor, 255
    %arrayidx = getelementptr inbounds [256 x i32]* @crc_32_tab,
                i32 0, i32 %and
    %shr = lshr i32 %2, 8
    %in = insertvalue %structureISE undef, i32 %arrayidx, 0
    %in1 = insertvalue %structureISE undef, i32 %shr, 0
    ret %structureISE undef
}
```

4 Testování efektivity řešení

Jak již bylo zmíněno v kapitole 2.1, proces vyhledávání instrukčních rozšíření vznikl kvůli urychlení běhu aplikací na aplikačních procesorech. Vzhledem k tomu, že se tato práce zabývá právě tímto vyhledáváním, je vhodné zjistit u vypracovaného programu pro toto vyhledávání jeho efektivitu.

Hlavním kritériem určujícím kvalitu a efektivitu vypracovaného programu je bezpochyby zrychleni běhu vstupního programu při aplikaci vyhledaných instrukčních rozšíření. Proto se při testování zjišťuje především počet taktů při vykonávání vstupního programu před, a po aplikaci instrukčních rozšíření.

Vhodným měřítkem pro zjištění kvality vypracovaného programu je také počet nalezených instrukčních rozšíření a jejich velikost určena počtem instrukcí, které dané rozšíření nahrazuje.
Kvalitnější program by tedy měl vyhledat v co největším počtu co největší rozšíření. Současně s růstem těchto hodnot také roste rychlost vykonávání vstupního programu.

Cílem této práce proto je nalezení takových instrukčních rozšířeni, které při použití dosáhnou co nejvyšších hodnot u zmíněných kritérií.

4.1 Množina testovaných programů

Bitcounter.c

Bitový čítač vytvořený tak, že postupně prochází prvky typu `integer` umístěné v globálně definované tabulce. Pro každý prvek pak na základě počtu prostřednictvím binární operace `and` zjišťuje počet jedničkových bitů. Pro každý z nich pak inkrementuje hodnotu celkového počtu. Výsledek je nakonec bitově posunut o 5 bitů doleva. Tento způsob je přibližně dvakrát rychlejší, než klasická používaná metoda posunu a testování.

Dijkstra.c

Tento program simuluje chování Dijkstrova vyhledávacího algoritmu sloužícího pro vyhledání nejkratší cesty v grafu. V rámci tohoto programu jsou vyhledávána čísla typu `integer` uvnitř matice z takovýchto čísel složené.

Crc.c

Tento program jsem již podrobně popsal v kapitole 3.2.1, kde sloužil jako příklad pro ukázku postupu vyhledávání instrukčního rozšíření vypracovaného v rámci této práce.

Quicksort.c

Tento program aplikuje stejnojmenný řadící algoritmus. `Quicksort` je velmi rychlým algoritmem, díky čemuž je hojně využíván, a je tak ideálním kandidátem na testovací vstupní program.

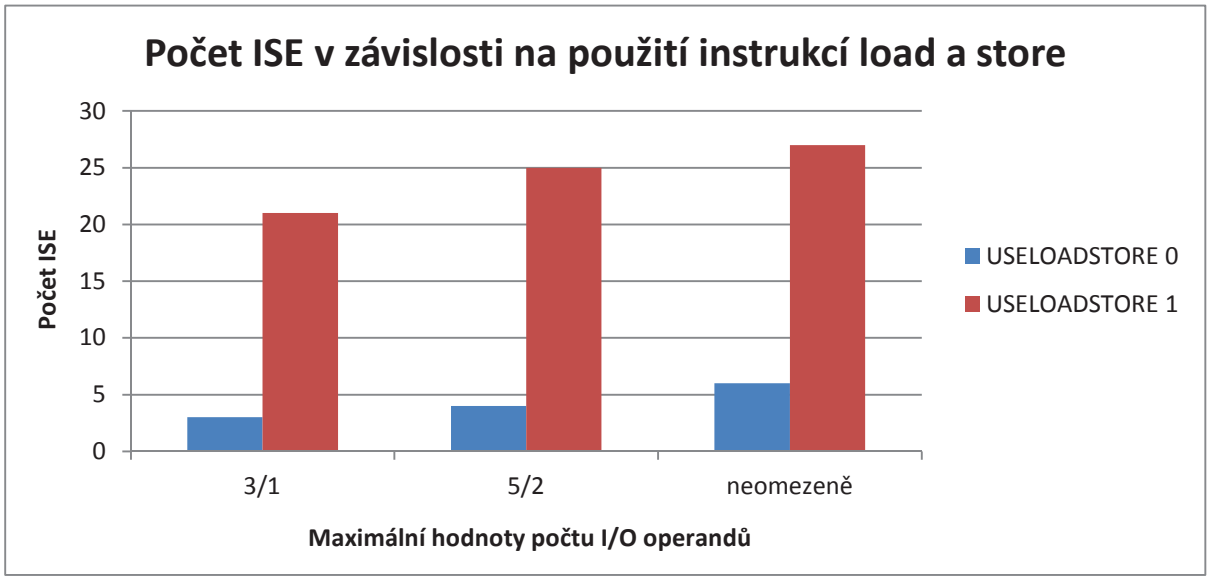
Isqrt.c

Program `isqrt` implementuje výpočet odmocniny vracející hodnotu v datovém typu `integer`.

4.2 Výsledky testování

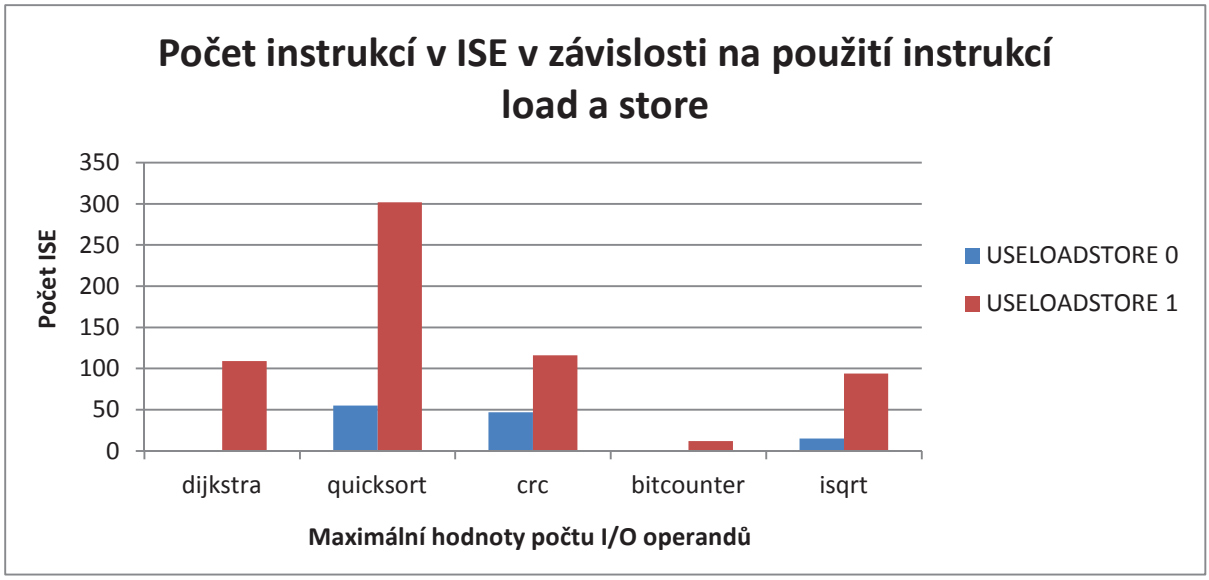
Nejprve probíhalo testování zjišťující počet nalezených instrukčních rozšíření pro dané programy. U každého programu probíhalo toto měření celkem šestkrát, pokaždé za jiných podmínek. Testování bylo rozděleno na dvě hlavní části. V první z nich se byla nastavena konstanta `USELOADSTORE` na hodnotu 0, ve druhé pak na hodnotu 1. V každé z těchto částí pak probíhalo měření ve třech kolech s měněnou hodnotou maximálního počtu vstupních a výstupních operandů pro funkce reprezentující instrukční rozšíření.

Následující graf znázorňuje, jak se měnili počty vyhledaných instrukčních rozšíření v závislosti na nastavených maximech, či použití instrukcí `Load` a `Store`.



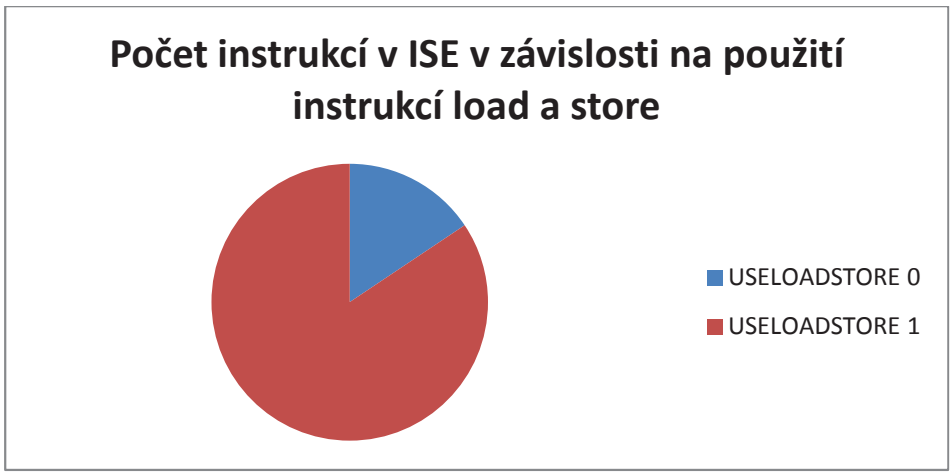
Obr. 4.1: Graf počtu nalezených instrukčních rozšíření v závislosti na konfiguraci vyhledávacího programu

Kromě počtu instrukčních rozšíření byl u jednotlivých vstupních programů při vyhledávání měřen počet instrukcí, které tyto instrukční rozšíření nahrazovaly. Počty instrukcí, které byly takto nahrazeny, jsou vyobrazeny na následujícím grafu.



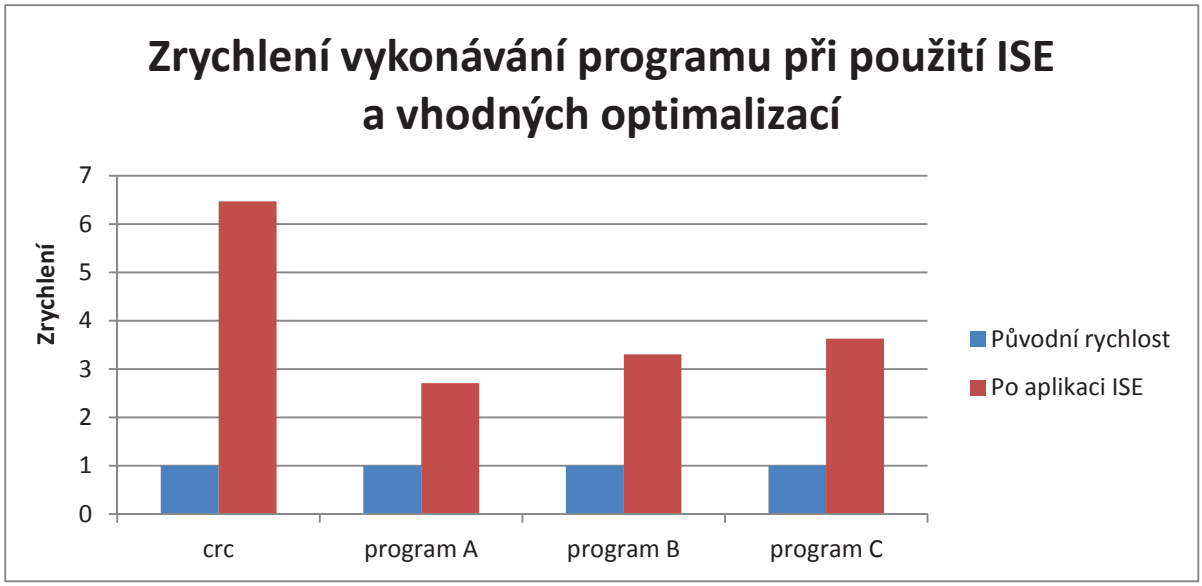
Obr. 4.2: Graf počtu instrukcí v nalezených ISE v závislosti na konfiguraci vyhledávacího programu

Pro lepší představu celkového rozdílu mezi počtem instrukcí nahrazeným instrukčními rozšířeními při používání instrukcí `Load` i `Store` v řezu, či nikoliv, je tento poměr zobrazen na následujícím grafu.



Obr. 4.3: Graf počtu instrukcích v ISE v závislosti na použití vybraných instrukcí v ISE

Současně s testováním počtu vytvořených ISE byl také zjišťován rozdíl v počtu taktů procesoru mezi běžným spuštěním vstupního programu a takovým spuštěním, kdy program prošel všemožnými optimalizacemi. Bohužel, vzhledem k omezením testovacích nástrojů, nebylo možné použití instrukcí `Load` a `Store` v ISE, a proto některé z programů uvedených v předchozí části i vzhledem k předchozím výsledkům testování, nebylo možné použít. U některých totiž bez těchto instrukcí nedošlo k nalezení možných instrukčních rozšíření. Proto se testovací skupina programů pro tuto část poněkud pozměnila i o speciální, pro toto testování vytvořené programy. Výsledky jsou zobrazeny v následujícím grafu.



Obr. 4.4: Graf znázorňující zrychlení vykonávání vstupních programů při použití vyhledaných ISE a optimalizací

4.3 Vyhodnocení

Testování ukázalo především obrovský rozdíl v použití instrukcí `Load` a `Store`. Pokud tyto instrukce lze v ISE použít, dochází k nalezení nalezeno mnohem většího počtu použitelných (s počtem instrukcí větším než 4) instrukčních rozšíření, čímž se vyhledávání stává efektivnějším.

Bohužel bylo testování výsledků této práce ovlivněno některými omezeními (maximální počet vstup-výstupních operandů roven 4, nebo nepoužití instrukcí `Load` a `Store`) a výsledky tak mohli být ještě o něco lepší. I tak si ale myslím, že zrychlení způsobené použitím mého postupu je výrazné, což dokazuje použitelnost vypracovaného programu v praxi.

5 Závěr

Tato bakalářská práce představuje proces automatického vyhledávání instrukčních rozšíření aplikačních procesorů. Hlavní část tohoto procesu tvoří algoritmus ISEGEN, upravený pro specifické potřeby vypracované aplikace. Tato práce se však již nezabývá aplikací těchto vyhledaných instrukčních rozšíření. Ty mají vzhledem k popularitě aplikačních procesorů v dnešní době stále větší význam, a proto bylo od této práce očekáváno co nejefektivnější vyhledávání instrukčních rozšíření, které by se v takovém procesoru daly použít.

Vypracovaný program byl vytvořen pro vyhledávání instrukčních rozšíření vstupních programů napsaných v jazyce C, přeložených do jazyka symbolických instrukcí LLVM IR. Použitý algoritmus takto přeložený program prochází po základních blocích, ke kterým vytváří grafovou reprezentaci, v níž hledá řez vhodný pro záměnu za instrukční rozšíření.

Na základě počtu instrukčních rozšíření, nalezených při testování, lze usoudit, že proces popsaný v této práci je účinný. Mimo to lze říci, že na základě zjištěného zrychlení u vstupních programů, při použití procesu popsaného v této práci, je tento algoritmus také velmi efektivní. Celkově lze tedy říci, že proces vyhledávání instrukčních rozšíření pro aplikační procesory, navržený v této práci, je efektivní a splňuje veškerá očekávání. I tak zde zůstává prostor pro určitá vylepšení do budoucna.

Literatura

[1] Biswas, P.; Banerjee, S.; Dutt, N.D.; Pozzi, L.; Ienne, P.; „ISEGEN: an iterative improvement-based ISE generation technique for fast customization of processors.“ In *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14 (7), 754-762, 2006.

[2] Biswas, P.; Banerjee, S.; Dutt, N.D.; Pozzi, L.; Ienne, P.; „ISEGEN: Adapting Kernighan-Lin Min-Cut Heuristic for Generation of Instruction Set Extensions.“ *UC Irvine, Technical Report CECS-TR-04-21.*

[3] Biswas, P.; Banerjee, S.; Dutt, N.D.; Pozzi, L.; Ienne, P.; „ISEGEN: Generation of high-quality instruction set extensions by iterative improvement.“ In *Design, Automation and Test in Europe,* 1246-1251, 2005.

[4] Kyu, Lim Sung: Kernighan and Lin Algorithm [online]. 2008-08-23 [cit. 2012-12-29]. Dostupné na URL: < http://users.ece.gatech.edu/limsk/book/slides.htm>

[5] Galuzzi , C.; Bertels , K.; „The instruction-set extension problem: A survey.“ *In Proc. 4th Int. workshop on Reconfigurable Computing, ARC ’08*, 209–220, 2008.

[6] The LLVM Compiler Infrastructure Project [online]. [cit. 2013-04-06]. Dostupné na URL: <http://llvm.org/>

[7] Cong, J.; Fan, Y.; Han, G.; Zhang, Z.; „Application-specific instruction generation for configurable processor architectures.” In *Proc. ACM International Symposium on Field-Programmable Gate Arrays*, 183-189, 2004.

[8] Borrelli, Chris. „IEEE 802.3 cyclic redundancy check.” *application note: Virtex Seriees and Virtex-II Family*, XAPP209 (v 1.0), 2001.

[9] Biswas, P.; Banerjee, S.; Dutt, N.; Pozzi, L.; Ienne, P.; „Fast automated generation of high-quality instruction set extensions for processor customization“. In *3rd Workshop on Application Specific Processors*, 2004.