VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

ROZVOJ INSTRUMENTACE PROGRAMU PŘI PŘEKLADU

DEVELOPMENT OF INSTRUMENTATION DURING COMPILATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. VÁCLAV ŠEVČÍK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA, Ph.D.
SUPERVISOR

BRNO 2020

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání diplomové práce

21397

Student: **Ševčík Václav, Bc.**
Program: Informační technologie Obor: Bezpečnost informačních technologií
Název: **Rozvoj instrumentace programu při překladu**
Development of Instrumentation during Compilation
Kategorie: Analýza a testování softwaru
Zadání:

1. Nastudujte testování a dynamickou analýzu programů v jazyce C. Nastudujte infrastrukturu překladače LLVM/Clang. Nastudujte instrukční sadu LLVM IR.
2. Navrhněte nástroj pro jednoduchou konfiguraci instrumentace testovaných programů. Nástroj musí umožňovat instrumentaci volaných funkcí, přístup k parametrům, získání návratových hodnot funkcí a přístupů do paměti.
3. Implementujte prototyp nástroje jako modul do platformy Testos.
4. Vytvořte testovací případy pro ověření hlavní funkcionality.

Literatura:

- R. Tschüter, J. Ziegenbalg, B. Wesarg, M. Weber, C. Herold, S. Döbel, R. Brendel. An LLVM Instrumentation Plug-in for Score-P. 2017. In Proceedings of LLVM-HPC'17. Denver, CO, USA. doi:10.1145/3148173.3148187
- Dokumentace k překladači LLVM/Clang. Dostupné na URL http://llvm.org/docs/
- A. Vitch, D. Berris, E. Anderson, E. Heintze, N. Wang. XRay: A Function Call Tracing System. 2016-04-05. Dostupné na URL: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45287.pdf

Při obhajobě semestrální části projektu je požadováno:

- Studium a návrh rámce

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 31. října 2019

Abstrakt

Diplomová práce se věnuje vytvoření nástroje pro instrumentaci programu během překladu v LLVM překladači. Nástroj umožňuje instrumentovat přístupy do paměti a funkce. Instrumentace byla realizována pomocí přidání průchodu v optimalizační fázi překladače LLVM. Informace o proměnných jsou spravovány vytvořeným frameworkem, který se připojí k programu během sestavování. Doba běhu programu se zavedenou instrumentací zvýší časovou režii programu při vypnuté nepřímé adresaci průměrně o 14 % a při zapnuté nepřímé adresaci o 23 %. Hlavním přínosem práce je poskytnutí snadné instrumentace programu, která dokáže sledovat i operace nad lokálními proměnnými (nepřímou adresaci) a umožňuje instrumentovat i vícevláknové programy. Nástroj je také začleněn do sady nástrojů Testos, kde poskytuje automatickou instrumentaci pro nástroj Spectra.

Abstract

The focus of this master's thesis is on the topic of instrumentation during the compilation process in the LLVM compiler. The tool enables to instrument memory accesses and functions. The instrumentation is realized through adding a novel pass to the LLVM's optimalization phase. Information about variables are managed by the created framework. The framework is linked with the program. The overhead of the instrumentation increases duration of the program by about 14 % in the case of switched off indirect addressing and 23 % in the case of switched on indirect addressing. The main benefit of the work is the possibility of easy instrumentation of the program which can even monitor operation of local variables through indirect addressing) and support multithread programs. The framework is part of Testos's tools where it provides automatic instrumentation in the Spectra tool.

Klíčová slova

LLVM, LLVM IR, instrumentace během překladu, Testos, LLVM průchod, instrumentační knihovna, instrumentační framework, instrumentace funkce, instrumentace přístupu do paměti, modul LLVM, vícevláknovost, nepřímá adresace

Keywords

LLVM, LLVM IR, source code instrumentation, Testos, LLVM pass, instrumentation library, instrumentation framework, instrumentation function, instrumentation memory access, LLVM modul, multithreading, indirect addressing

Citace

ŠEVČÍK, Václav. *Rozvoj instrumentace programu při překladu.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Rozvoj instrumentace programu při překladu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doktora Aleše Smrčky. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . . . .
Václav Ševčík
3. června 2020

Poděkování

Děkuji vedoucímu práce Ing. Aleši Smrčkovi Ph.D., který mě doprovázel při tvorbě této práce, dával mi inspiraci k dalším krokům v pokračování mé práce a byl neocenitelným rádcem během její tvorby. Hlavní můj dík patří Bohu, který mě provází po celou dobu studia. Děkuji také kolegovi Jiřímu Pavelovi za pomoc při korektuře práce. Poděkování náleží v neposlední řadě také mé rodině, která mi poskytovala psychickou a materiální podporu.

Obsah

1 Úvod **4**

2 Použité technologie pro instrumentaci **6**
2.1 Testování softwaru a dynamická analýza 6
2.1.1 Dynamická analýza 6
2.1.2 Testování softwaru 8
2.1.3 Instrumentace programů 10
2.1.4 Testování softwaru a dynamická analýza v jazyku C 12
2.2 Infrastruktura a překlad LLVM / Clang 13
2.2.1 Struktura překladače LLVM 13
2.2.2 Optimalizační průchody v LLVM 16
2.3 Instrukční sada LLVM 16
2.3.1 Jazyk LLVM IR 16
2.3.2 Metadata LLVM IR 19

3 Další instrumentační nástroje **20**
3.1 XRay 20
3.2 Score-P 21
3.3 Instrumentace pomocí přepínače finstrument-functions 22
3.4 SBT-instrumentation 24
3.5 The CSI Framework for Compiler-Inserted Program Instrumentation 25
3.6 Shrnutí existujících řešení 26

4 Analýza způsobů řešení instrumentace **29**
4.1 Specifikace požadavků 29
4.2 Analýza způsobu realizace frameworku 30
4.2.1 Výběr přístupu k instrumentaci, volby překladače a způsobu tvorby nástroje 30
4.2.2 Analýza možností instrumentace v modulu 31

5 Návrh instrumentačního frameworku **34**
5.1 Struktura frameworku a kooperace programů 34
5.2 Struktura průchodu 36
5.3 Konfigurační soubor Tforc 37
5.4 Specifikační jazyk Tforc 38
5.4.1 Dekorování jména funkce 38
5.4.2 Komentáře a mezery ve specifikačním jazyce 39
5.4.3 Instrumentace funkce 39

5.4.4 Instrumentace proměnné 40
5.5 Způsob použití frameworku 41

6 Implementace instrumentačního frameworku 44
6.1 Správa záznamů pro instrumentaci 44
6.2 Průchod a filtrování instrukcí 45
6.3 Instrumentace funkcí a proměnných 46
6.4 Komunikace mezi průchodem a Tforc runtime engine 48
6.5 Správa adres v Tforc 49
6.5.1 Hašovací tabulka s proměnnými 50
6.5.2 Zásobník adres 50
6.5.3 LUT tabulka 51
6.6 Implementační závislosti a omezení 52

7 Evaluace instrumentačního frameworku 54
7.1 Závislosti vývojového prostředí 54
7.2 Použití frameworku Tforc 55
7.3 Validace 56
7.3.1 Funkcionalita testů 56
7.3.2 Struktura testovacích souborů 57
7.3.3 Důvody omezení funkčnosti 57
7.4 Experimenty 58
7.5 Testos 59
7.6 Integrace s nástrojem Spectra 60

8 Závěr 62

Literatura 63

A Gramatika k specifikačnímu jazyku Tforc 68

B Prototypy funkcí poskytující uživateli informace z TRE 70

C Zdrojový kód hledání extrémů 71

D Konfigurace specifikací experimentů 72

E Obsah přiloženého média 73

Seznam obrázků

2.1 Schéma analyzátoru dynamické analýzy 7
2.2 Testování softwaru na základě vstupů a výstupů SUT 8
2.3 Vennův diagram - grafické znázornění pokrytí u testování 9
2.4 Instrumentace během překladu 11
2.5 Binární instrumentace 12
2.6 Třífázová architektura překladače LLVM 14
2.7 Reprezentace struktury programu v LLVM překladači 15

3.1 Princip instrumentace pomocí přepínače finstrument 23
3.2 Schéma konfigurace nástroje SBT 24

4.1 Možnosti instrumentace funkce 32
4.2 Instrumentace operací `load` a `store` 33

5.1 Kooperace mezi soubory a nástroji v instrumentaci frameworku Tforc . . . 35
5.2 Schéma vnitřní komunikace frameworku Tforc 36
5.3 Příklad konfiguračního souboru 37
5.4 Příklad dekorovaného jména funkce 38
5.5 Příklad specifikace instrumentace funkce 40
5.6 Příklad specifikace instrumentace přístupu do paměti 41
5.7 Diagram kolaborace pro použití frameworku Tforc 42

6.1 Typy záznamů instrumentace 45
6.2 Volání funkce pro nepřímou adresaci 48
6.3 Mapování adres na záznamy s informacemi o proměnných 50
6.4 Zásobník TRE obsahující aktuálně používané adresy 51

7.1 Schéma infrastruktury projektu Testos 59

Kapitola 1

Úvod

„Existují dva způsoby, jak psát bezchybné programy; funguje pouze třetí.“
– Alan Jay Perlis

Tvorba programu jde ruku v ruce s faktem, že do programu bude zavlečena chyba. Hledání chyb může být velmi zdlouhavá práce pro vývojáře, která ne vždy musí vést k nalezení hledané chyby. Pro usnadnění práce programátorů v hledání chyb existuje celá řada přístupů. U softwaru to mohou být například testování a dynamická analýza — sledující chování programu při spuštění. Pro sledování vnitřního stavu programu je zapotřebí tzv. instrumentačního frameworku. Instrumentace se dnes provádí na úrovni binárního kódu (u kompilovaných jazyků), na úrovni interpretu (u interpretovaných jazyků), v době překladu a v době psaní zdrojových kódů. Cílem této práce je vytvoření instrumentačního nástroje pro instrumentaci během překladu.

Nástroj lze využít samostatně nebo jako doplněk k nástroji Spectra[1] (v sadě nástrojů Testos), kde může doplňovat chybějící funkcionalitu automatické instrumentace. Nástroj, vyvinutý v této diplomové práci, umožňuje instrumentaci kódu během překladu, jež umožňuje sledovat běh programu (a nebo ho vkládáním obsluh ovlivňovat) u přístupů do paměti a u funkcí. U přístupů do paměti dovoluje nástroj instrumentovat jak lokální, tak globální proměnné s přístupem jak přímým, tak i nepřímým. Z přístupu do paměti zpřístupňuje informace o proměnných uložených na daném místě (adresu, hodnotu, programovou lokaci). U funkcí je možné pracovat s argumenty instrumentované funkce a její návratovou hodnotou. S ohledem na rostoucí snahu provádět výpočty paralelně, nástroj podporuje instrumentaci také vícevláknových programů.

Nástroj je vyvinutý tak, aby minimalizoval množství úkonů, které musí uživatel pro instrumentaci vykonat. Kvůli tomu byl taktéž vyvinut vlastní specifikační jazyk, který umožňuje snadněji uživateli zadat své požadavky na instrumentaci.

Kapitola 2 se věnuje vysvětlení technologií, které byly použity v této diplomové práci. Jedná se v první části o dynamickou analýzu, testování a instrumentaci a v dalších částech se zabývá překladačem LLVM a jeho architekturou. Poslední část se zabývá nízkoúrovňovým jazykem LLVM IR. V kapitole 3 jsou přiblíženy již existující nástroje, které se zabývají — stejně jako vyvíjeny nástroj — instrumentací během překladu. Nástroje jsou krátce představeny a jsou u nich ukázány jejich silné a slabé stránky. Další kapitola 4 pojednává o analýze možností realizace nástroje a jsou zde představeny varianty řešení. Poté jsou tyto varianty zváženy a je vybráno řešení, které bylo zvoleno pro realizaci. Analýza začíná obecně výběrem způsobu implementace a končí u analýz konkrétních možností instrumentace v rámci reali-

[1] https://pajda.fit.vutbr.cz/testos/spectra

zace. Kapitola 5 představuje návrh, jak by měl nástroj komunikovat s vnějšími programy, a kooperaci uvnitř samotného nástroje. Poté jsou zde navrženy formáty obsahů jednotlivých souborů. Jedná se jak o konfigurační soubor, tak o specifikační jazyk. Poslední část se zabývá návrhem způsobu použití frameworku. Samotná implementace nástroje je popsána v kapitole 6. Je zde popsána implementace od načtení konfigurace, přes instrumentaci, až po modul, který se přidává k programu během sestavování. Taktéž jsou zde popsány datové struktury, které se využívají pro práci s informacemi o proměnných. Na konci kapitoly jsou uvedeny implementační omezení a závislosti vzniklého nástroje. Poslední kapitola 7 je věnována vyhodnocení vzniklého nástroje Tforc. Prvně jsou vyjmenovány závislosti vývojového prostředí, následuje použití frameworku Tforc. Důležitou součástí vyhodnocení je též validace vzniklého programu pomocí testů, jimž je věnována další část. Pro vyhodnocení vzniklé režie jsou zde popsány experimenty na rozličných příkladech. Poslední část se věnuje sadě nástrojů Testos a integraci Tforc s nástrojem Spectra.

Kapitola 2

Použité technologie pro instrumentaci

Kapitola podává informace a souvislosti o technologiích, které byly využity k tvorbě této diplomové práce.

Na úvod kapitoly je nastíněn význam pojmů testování a dynamické analýzy a souvislost mezi nimi. Je zde též představen způsob získávání dat z programu pomocí instrumentace (viz podkapitola 2.1). Druhá podkapitola popisuje principy překladače LLVM, jeho infrastrukturu a využití v překladači Clang (více v podkapitole 2.2). Třetí a poslední podkapitola představuje instrukční sadu LLVM IR (viz podkapitola 2.3), kterou překladač LLVM využívá ve svém mezikódu během překladu. V mezikódu probíhají optimalizace a též se získávají informace o datech.

2.1 Testování softwaru a dynamická analýza

Podkapitola se zabývá principy dynamické analýzy a podrobněji se zaměřuje na testování, pro které je v této práci navržen a vytvořen nový framework[1]). Vytvořený framework má pomáhat programátorům v jejich práci během testování programů. Jak napsal Boris Beizer:„Testování je mentální disciplína, která pomáhá IT profesionálům vyvíjet kvalitnější produkt“. [17]

Začátek podkapitoly je věnován vysvětlení základních principů dynamické analýzy (viz oddíl 2.1.1), jenž umožňuje získat komplexní informace o chování programu za běhu. Poté se podkapitola zaměřuje na testování, což je jedna z nejvíce rozšířených a používaných forem dynamické analýzy (více v oddílu 2.1.2). V posledním oddílu se zabývá získáním informací z programu formou instrumentace (viz oddíl 2.1.3).

2.1.1 Dynamická analýza

Verifikace (proces ověření správnosti dle specifikace) softwaru se dá rozdělit na dva disjunktní přístupy (analýzy[2]) ke zkoumání programu, a to na statickou analýzu, která zkoumá vlastnosti softwaru bez jeho provádění, a na dynamickou analýzu, která naopak zkoumá vlastnosti na základě provádění kódu (o této analýze pojednává tento oddíl).

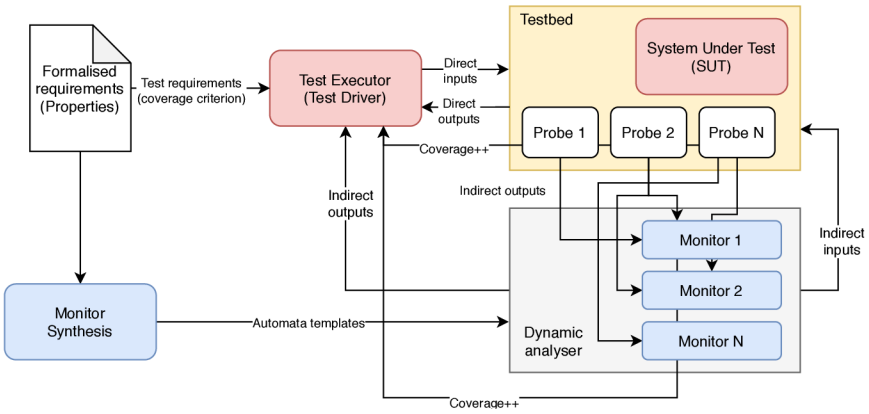
[1] Framework - softwarová struktura, která slouží jako podpora při programování, vývoji a organizaci jiných softwarových projektů

[2] Analýza slouží k získání komplexních informací o zkoumaném programu.

Informace o dynamické analýze (rovněž známá jako např. runtime verification, runtime monitoring nebo též trace analysis) byly čerpány z článku [14], který popisuje koncept této analýzy, dále z článku [16], který vysvětluje jednotlivé části analýzy a z přednášky [50], která se zabývá problematikou dynamické analýzy.

Dynamická analýza nevyžaduje tolik prostředků jako jiné typy analýz (metoda dokazování a ověřování modelů), a přitom poskytuje praktičtější přístup. Cenou za to je, že neposkytuje tak velké pokrytí jako jiné analýzy. Získávání dat dynamickou analýzou obvykle bývá zajišťováno pomocí instrumentace kódu [14], o které pojednává oddíl 2.1.3. Dynamická analýza umožňuje jak nalézt různé typy chyb, tak poskytovat další užitečné informace o chování programu.

Analyzátor je program, který provádí dynamickou analýzu nad zvoleným programem. Analyzátor obsahuje části, jež jsou znázorněny v obrázku 2.1 a popsány níže. Čerpáno z přednášky [50] o dynamické analýze.



Obrázek 2.1: Schéma analyzátoru zachycuje jednotlivé části analyzátoru a komunikaci mezi nimi. Konkrétně po načtení a vytvoření monitorů z formálních požadavků (*Formalised requirements*) převezme iniciativu kontrolér (*Test Executor*). Kontrolér posílá vstupy (*Direct inputs*) do prostředí (*Testbed*), a při splnění specifické podmínky jsou pomocí sond (*Probe*) odeslána data ke zpracování do monitoru. V monitoru jsou tato data agregována a poslána do kontroléru (*Test Executor*), ze kterého se posléze vyšle příkaz k ovlivnění běhu programu zpět do prostředí (případně do sondy). Obrázek převzatý z přednášek k předmětu [51].

Části analyzátoru:

- Sonda (*probe*) — sleduje vybraný artefakt (stav zásobníku, hodnotu proměnných v paměti, volání funkcí, signály, systémové volání, průchod daným úsekem kódu, atd.).
- Monitor (*monitor, tracer*) — poskytuje kontroléru předzpracovaný a agregovaný záznam dat poskytnutých ze sond.
- Kontrolér (*test executor, controller, analyzer, driver*) — rozhoduje, zda a jak bude dále pokračovat SUT[3] pomocí pravidel, která jsou v něm předem definovaná (konfiguračním souborem) a instrumentuje SUT předpřipraveným kódem. Kontrolér někdy bývá uváděn jako část monitoru.
- Prostředí (*testbed, platform*) — prostředí, ve kterém běží SUT. Jako prostředí mohou sloužit knihovny, operační systémy či virtualizace.

[3] SUT - system under test - testovaný systém

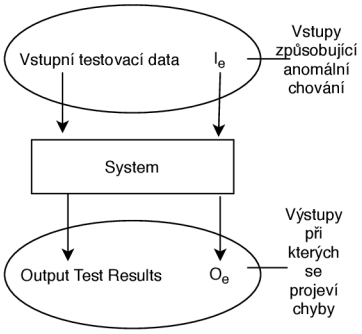
Vzorkování Dynamická analýza může zpomalit běh programu až o několik řádů, což je pro některé uživatele neúnosné. Kvůli tomuto zpomalení existuje možnost vzorkování v průběhu analýzy. Vzorky, které se budou či nebudou sledovat, jsou zvoleny podle určité heuristiky (výběr vhodných vzorků). Nevýhodou vzorkování je možnost přehlédnutí částečné chyby (particular error) v rámci všech vzorků. Chybu sice analýza nenalezne, ale na rozdíl od dlouhého nevzorkovaného běhu proběhne mnohem rychleji (v relativně krátké době). Díky vhodnému výběru vzorků nalezne analýza v rámci některého z běhů nejzávažnější chyby [28].

Dynamická analýza se využívá k simulacím a taktéž při testování softwaru, jemuž je věnován další oddíl kapitoly (viz oddíl 2.1.2).

2.1.2 Testování softwaru

Oddíl nastiňuje a vysvětluje některé základní principy testování. Informace čerpá z knih [31, 41, 52], které popisují základní principy a pojmy testování, a vychází též z textových materiálů k předmětu Testování a dynamická analýza [50] přednášenému na Fakultě informačních technologií VUT.

Napsat bezchybný kód je velmi obtížné — v případě komplexního kódu pak téměř nemožné — a chyby v kódu jsou tak běžnou součástí vývoje jakéhokoliv software. Kvůli tomu se za účelem určité formy ověření funkčnosti a kvality vyvíjeného programu využívá testování. Princip testování spočívá ve transformaci vstupních dat programem do výstupních dat, která jsou následně porovnána s očekávanými výsledky (viz obrázek 2.2). Kontrolou vybraných průchodů kódu (pokrytí) se předchází nechtěnému chování programu jako je nedefinované chování systému, pád systému, ovlivnění jiného programu, zneplatnění dat a nepřesným výpočtům.



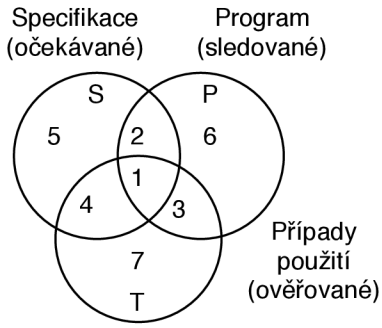
Obrázek 2.2: Testování části softwaru na základě vstupů a výstupů z SUT. Proměnná I_e znázorňuje ve schématu vstupy, které způsobí v SUT nepředvídatelné chování, nebo formou výsledku propagaci chyby do výstupů. Výstupy znázorňuje proměnná O_e, ve které neočekávané výsledky mohou odhalit přítomnost chyby v kódu. Myšlenka obrázku převzata z knihy [52].

Testování softwaru (jedna z možných definic): Testování softwaru je proces nebo skupina procesů navržených tak, aby zjistil/zjistily, jestli program vykonává to, na co byl navržen a nevykonává nic nechtěného. Software by měl být předvídatelný a konzistentní, neměl by uživatele překvapit neočekávaným chováním. [41]

Techniky testování. Existují dvě základní techniky testování, jmenovitě:

1. Testování bez znalostí vnitřní implementace (black-box testing) — nezná vnitřní strukturu kódu a zabývá se vstupy a výstupy z objektů.
2. Testování se znalostí vnitřní implementace (white-box testing) — zná vnitřní strukturu kódu a vychází z logických konstrukcí v implementaci.

Pokrytí je míra udávající, jak důkladně daná testovací sada kontroluje SUT. Jako model pokrytí testovaného programu mohou posloužit 3 množiny znázorňující vztahy mezi specifikací, programem a testovací sadou. Popis těchto vztahů je znázorněn na obrázku 2.3.



Obrázek 2.3: Vennův diagram znázorňující tři množiny (Specifikace, programu a testovacích případů), které ukazují pokrytí chování programu při testování. Konkrétně oblasti 2 a 5 vyjadřují specifikaci chování programu, která není testována. Oblasti 1 a 4 naopak popisují specifikaci chování, která testovaná je. Dále oblasti 3 a 7, které znázorňují testovací případy zkoumající chování mimo specifikaci. Chování, které není testované, je vyjádřeno oblastmi 2 a 6. Naopak testované chování programu znázorňují oblasti 1 a 3. Poslední kombinací jsou oblasti 4 a 7 představující chování, které nebylo implementováno, ale je testováno. Diagram je přejatý z knihy [31].

Důslednost testování se zkoumá daným kritériem pokrytí kódu a procentuální úspěšností jeho testů. Existuje více druhů kritérií pokrytí, podle kterých se může testování klasifikovat. Zde jsou vyjmenována některá testovací pokrytí [41] pro obě techniky testování:

Se znalostí vnitřní implementace:

- Pokrytí příkazů/řádků (Statement / Line coverage)
- Pokrytí podmínek (Condition coverage)
- Pokrytí funkcí / metod (Function / Method coverage)
- Pokrytí uzlů / hran / cest (Node / Edge / Path coverage)

Bez znalosti vnitřní implementace:

- Analýza hraničních hodnot
- Graf příčiny a důsledku
- Testování podle tříd ekvivalence
- Předpokládání chyb

Kritérium pokrytí zahrnuje jiné kritérium pokrytí, když každá testovací sada splňující první kritérium (pokrývajícího) splňuje také druhé kritérium (např. pokrytí příkazů zahrnuje pokrytí funkcí).

Některá kritéria jsou neporovnatelná, proto jedno kritérium může chybu nalézt a druhé ji minout (popřípadě naopak). Čím přísnější kritérium pokrytí (zahrnující jiné) se zvolí, tím větší pravděpodobnost, že bude chyba v kódu odhalena. Při volbě správného kritéria je potřeba vyvážit množství dostupných zdrojů, které je možné investovat do implementace kritéria pokrytí, a schopnosti tohoto kritéria najít chybu ve zkoumaném programu.

Testování v oblasti vývoje počítačových systémů probíhá podle stupně abstrakce testovaných částí (jako jsou systémy, funkce, atd.) a jejich testování v časových fázích vývoje softwaru (kdy testy probíhají). Druhy testování těchto částí jsou následující (čerpáno též z diplomové práce [32]):

- Jednotkové testování — většinou mají na starosti vývojáři. Běžně přitom využívají frameworků pro ulehčení práce. Testování se zavádí špatně na již běžících projektech.
- Testování modulů — je vyšší abstrakce jednotkového testování.
- Integrační testování — kontroluje komunikaci mezi komponentami v rámci programu.
- Systémové testování — ověřuje program jako celek.
- Akceptační testování — provádí testeři od zákazníka, kteří monitorují, jestli program splňuje požadovanou specifikaci.
- Regresní testování — sleduje zpětnou kompatibilitu s předešlými verzemi.

2.1.3 Instrumentace programů

Termín instrumentace programů odkazuje na mechanismus používající sondy k extrakci signálů, cest, událostí a dalších požadovaných informací ze sledovaného programu / zařízení během jeho běhu.

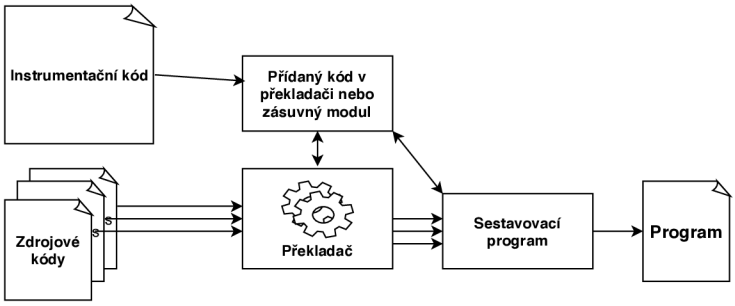
Instrumentace je využívána jak na software, tak hardware úrovni. U hardware většinou probíhá instrumentace pomocí fyzického připojení drátu a vysláním smíšeného analogového signálu. Na rozdíl od hardware se u software instrumentace provádí pomocí vkládání instrukcí, které je silně spojeno s nízkoúrovňovým programovacím jazykem, jako jsou např. bytecode, LLVM IR, assembler aj. Pod pojmem instrumentace v tomto oddíle bude myšlena softwarová instrumentace. Informace k tomuto oddílu pochází ze článků [16, 46], které objasňují rozdíl mezi jednotlivými druhy instrumentace a též je detailněji popisují.

Instrumentace je hojně používanou metodou v mnoha nástrojích dynamické analýzy. Poskytuje přidání uživatelského kódu za účelem sledování průběhu softwarových komponent a podává výstup sledujícímu monitoru. K instrumentaci je možno přistupovat dvěma způsoby — buď provádět instrumentaci už během překladu (viz odstavec 2.1.3), nebo instrumentovat až binární soubor po jeho sestavení (více v odstavci 2.1.3).
Instrumentace může být též dělena podle toho, kdy se daná instrumentace provádí:

- Statická — instrumentace probíhá během překladu nebo sestavování programu.
- Dynamická — instrumentace se provádí v rámci běhu programu.

Instrumentace během překladu (Source code level instrumentation).

Programy instrumentující program během překladu využívají překladače pro vložení vlastního kódu do SUT. Samotné fungování instrumentace je schématicky znázorněno na obrázku 2.4, kde je zachycena posloupnost kroků při překladu. Tato technika je velmi oblíbená u programátorů díky schopnosti zlepšit přehled o dění v SUT.



Obrázek 2.4: Schéma zobrazuje instrumentaci během překladu, kde zdrojové kódy jsou na vstupu překladače. Instrumentace je prováděna překladačem samotným (případně za pomoci zásuvného modulu) již v rámci překladu nebo později během sestavení programu. Kód instrumentace se nachází v samostatném souboru. Po dokončení instrumentace se program sestaví a vznikne instrumentovaný spustitelný soubor.

Takto vyvinutých nástrojů je v současnosti nepřeberné množství (jako jsou například nástroje pro detekci souběhu [24, 25, 40], kontrolu paměti [15, 47], simulaci cache [23, 54], generátoru grafu volání funkcí [27, 30], analýzu pokrytí kódu [56] a výkonnostní a škálovatelné profilování [29]).

Tyto nástroje umožňují programátorovi zasáhnout do nízkoúrovňové reprezentace překladače a tím provést instrumentaci pouze požadované části kódu (funkce, události, atd.). Nástroje běží na pozadí spuštěné aplikace a shromažďují požadované informace. Tyto nástroje mohou následně ovlivňovat běh programu na základě shromážděných informací nebo zaznamenávat jeho aktivitu. Tento typ instrumentace umožňuje využít optimalizace poskytované překladačem, díky tomu je možné vytvořit program s velmi nízkou přidanou režií.

Hlavní nevýhodou tohoto typu instrumentace je potřeba modifikovat fázi překladu. To vyžaduje odborné znalosti problematiky daného překladače, které často programátoři vyvíjející nástroj nemají. Snadnou orientaci ve zdrojovém kódu stěžují komplexní zdrojové kódy dnešních nejvyužívanějších překladačů. Některé nejvyužívanější překladače umožňují zásah do překladu taktéž pomocí zásuvných modulů. Tato vlastnost ulehčuje programátorům vytvářet nástroje, ale stále vyžaduje expertní znalost některých vnitřních struktur překladače.

Pokud programátor nevyužije možnost zásuvných modulů (nutnost úprav pouze při změně rozhraní), má na výběr dvě možnosti:

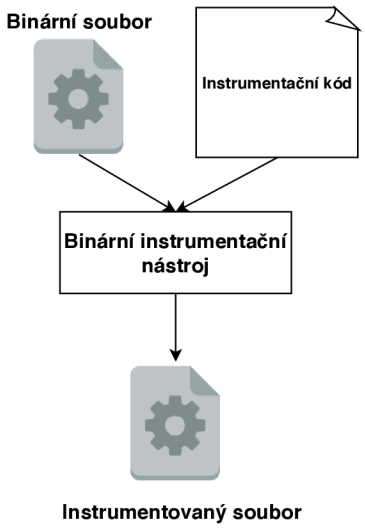
- Vytvořit si vlastní modifikaci překladače založenou na aktuální verzi. Údržba takto vyvinutého nástroje je však náročná a nákladná. Pro uživatele je též důležitá důvěryhodnost vyvíjeného překladače a jeho úprav.

- Vytvořit a začlenit tento nástroj do samotného překladače. Náklady vynaložené na začlenění nástroje výrazně přesahují jeho samotný vývoj.

Využití tohoto typu instrumentace sice umožňuje rychlý vývoj a testování nových inovativních nástrojů, ale značně znesnadňuje jejich údržbu (změna verze překladače).

Instrumentace binárních souborů (Binary level instrumentation).

Druhým typem instrumentace je instrumentace binárních souborů, jež umožňuje přímo instrumentovat binární spustitelný soubor (viz obrázek 2.5). Tuto možnost instrumentace využívají například nástroje [22, 36, 53].



Obrázek 2.5: Schéma zobrazuje instrumentaci binárních souborů, kde na vstupu nástroje je binární soubor a soubor s kódem k instrumentaci. Nástroj vloží na vhodná místa (určené instrumentačním frameworkem) binární instrukci volání funkce ze souboru. Výsledkem nástroje je spustitelný binární soubor.

Největší výhodou této instrumentace je schopnost instrumentovat program bez znalosti zdrojových kódů, ze kterých byl program sestaven. Toto umožňuje vývojářům třetích stran upravovat kód knihoven či aplikací. Na rozdíl od instrumentace během překladu, u binární instrumentace je problematické instrumentování některých oblastí kódu (např. *inline* funkce, makra).

K samotné instrumentaci jsou často vývojáři využívány binární instrumentační frameworky jako jsou Pin [39], DynamoRIO [20], Valgrind [42] nebo DynInst [18], které poskytují snadnější přístup k jednotlivým částem kódu v binárním souboru. Oproti ruční binární instrumentaci, která režii výrazně navyšuje, frameworky snižují přidanou režii pomocí optimalizací, které poskytují. Napříč této snaze frameworků jsou programy časově náročnější, protože optimalizace nemají takové množství informací jako optimalizace probíhající během překladu.

2.1.4 Testování softwaru a dynamická analýza v jazyku C

Principy, jež jsou nastíněny v rámci této podkapitoly platí taktéž pro jazyk C. Odlišnosti v testování a dynamické analýze jazyka C mohou být ve schopnosti získat podrobné informace o dění v programu díky jeho nízké úrovni abstrakce (např. práce s alokací paměti a ukazateli).

Pro Jazyk C vzniklo velké množství analyzátorů. Je to způsobeno tím, že patří mezi nejpoužívanější jazyky a též jeho častým nasazením v kritických systémech, ve kterých se dbá na minimalizaci rizika chyby. Jazyk C je taktéž často používán pro psaní interpretů[4] ostatních jazyků (CPython). Analyzátory jazyka C jsou často rozšiřovány o funkcionalitu jazyka C++. Využívají rychlosti tohoto jazyka, která dovoluje rychlejší provádění jak dynamické, tak statické analýzy.

2.2 Infrastruktura a překlad LLVM / Clang

LLVM není jeden monolitický překladač, jako spíš rozsáhlá překladačová infrastruktura (pro zjednodušení se bude dále používat překladač LLVM), která obsahuje sadu nástrojů a technologií, které společně poskytují samotný překlad. O struktuře pojednává oddíl 2.2.1, jenž poskytuje informace o překladači LLVM a též o reprezentaci programu v něm. Druhý oddíl 2.2.2 informuje o optimalizačních průchodech v rámci LLVM a možnostech vytvářet nové průchody, a tím zasáhnout do průběhu LLVM.

2.2.1 Struktura překladače LLVM

Oddíl čerpá z knihy [38], která se zabývá překladačem LLVM. Zdrojem informací jsou též dokumentace překladače [5, 3], které vysvětlují význam jednotlivých částí.

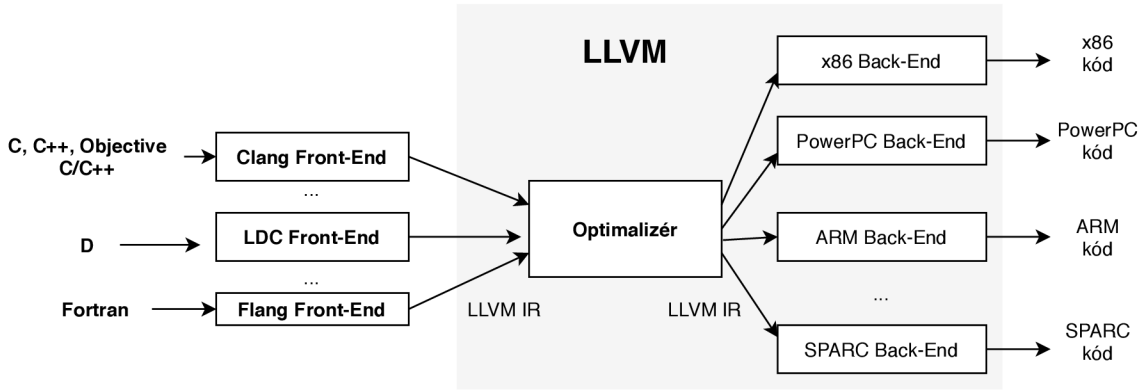
Základy LLVM infrastruktury byly položeny v článku [37], který představuje důvody, proč byl LLVM překladač vytvořen a také jeho hlavní výhody. Samotné fungování fáze překladu LLVM přibližuje kniha [44], kde jsou ukázány příklady fungování jednotlivých fází překladu v prostředí LLVM.

LLVM infrastrukturu lze rozdělit do tří modulárních částí, a to na vstupní fázi (frontend), optimalizační fázi (optimzer) a výstupní fázi (backend) — proto se tento překlad nazývá třífázový. Tyto části jsou zachyceny ve schématu překladače na obrázku 2.6. Na obrázku je patrné, že v rámci samotné LLVM infrastruktury jsou zahrnuty pouze optimalizační fáze a koncová fáze. Vstupní fáze je z infrastruktury vyňata, aby měli vývojáři třetích stran možnost ji implementovat pro libovolný programovací jazyk. Některé programovací jazyky jsou přímo podporovány komunitou vyvíjející LLVM. Mezi tyto jazyky patří jazyk C (popř. C++) a jeho vstupní fáze Clang (popř. Clang++).
Třífázový překlad probíhá v následujících krocích:

- Vstupní fáze — v této fázi probíhá převod vstupního jazyka do LLVM IR. Při něm je zároveň kontrolována syntaktická a sémantická správnost vstupního kódu.
- Optimalizační fáze — provádí optimalizace a další zvolené analýzy nad LLVM IR kódem.
- Výstupní fáze — také známá jako generátor kódu — vytváří strojový kód z LLVM IR.

Standardní překladače jako GCC, které jsou vyvíjeny jako monolitický nástroj, jsou silně limitovány jejich flexibilitou při překladu a možnostech přidání optimalizací. Tyto překladače přijímají pouze jeden jazyk a generují jiný, bez možnosti rozšířit množství jazyků. Rozšiřitelnost těchto nástrojů je proto velmi omezená. Jediným možným způsobem, jak zasáhnout do procesu překladu těchto překladačů, je vytvořit zásuvný modul. Avšak tento způsob podporují pouze některé z překladačů.

[4] Interpret - Program pro vyhodnocování zdrojového kódu v reálném čase.



Obrázek 2.6: Schéma zobrazuje třífázovou architekturu překladače LLVM. První, vstupní, fáze znázorňuje libovolný počet nástrojů, které provedou překlad ze vstupního jazyka do mezikódu LLVM IR. Druhá, optimalizační, fáze je pro všechny vstupní a výstupní fáze stejná. Umožňuje optimalizace a dodatečné analýzy nad kódem. Výstupní fáze se znovu může skládat z několika možných výstupních nástrojů, které generují strojový kód. Obrázek je přejat ze stránky prezentující architekturu LLVM [3].

Další nevýhodou monolitického překladače je, že překladač nelze využít jako knihovnu. Je to způsobené nekontrolovaným používáním velkého množství globálních proměnných, zvětšujícím se množstvím zdrojových kódů překladače a použitím maker, která zabraňují překladu s podporou více než jednoho vstupního a výstupního jazyka najednou. Největším problémem těchto překladačů je architektura, která je spoutána závislostmi mezi vrstvami, kvůli nimž nejde překladač rozdělit do více modulů. Myšlenka architektury překladačů v době jejich návrhů se lišila od architektur používaných v současnosti. Hlavním představitelem popsaných monolitických překladačů je GCC.

Třífázový překlad se vyznačuje podporou libovolného množství vstupních a výstupních jazyků bez nutnosti měnit samotné jádro překladače. Jádro překladu je zachované díky striktnímu oddělení fází pomocí mezikódu LLVM IR (viz podkapitola 2.3).

Porovnání výkonnosti zástupců (GCC a LLVM) dvou zmíněných architektur ve všech měřeních neukázalo jasnou převahu jednoho z nich. [6] Velkou výhodou překladače LLVM je svobodnější licence pro přejímání a využívání kódu formou open source. LLVM (od verze 9) poskytuje licenci Apache 2 na rozdíl od GCC, který má licenci GNU.

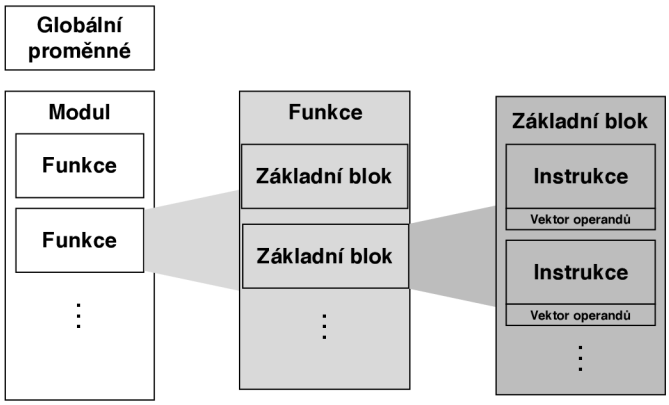
Kromě možnosti využití nového překladače lze využít i stávajících překladačů sloužících pro překlad jiných jazyků. K jejich využití je potřeba vytvoření překladače (transpileru[5]), který převede námi požadovaný jazyk do jazyku známého již existujícím překladačem. Díky tomu je možné použít optimalizace a generátor kódu stávajícího překladače (syntaktická a sémantická analýza je v transpileru), dále je zajištěna dobrá flexibilita a udržitelnost transpileru. Bohužel má toto řešení také své negativní stránky — neefektivní implementace zachytávání výjimek, horší analýza a ladění programu, zpomalení překladu a chybějící překlad specifických programových konstrukcí, které nemají ekvivalent v jazyce mezipřekladu.

[5] Transpiler je typ překladače, který přeloží kód z jednoho programovacího jazyku do druhého

Infrastruktura LLVM poskytuje velké množství nástrojů pro překlad, které lze využívat samostatně. Mezi přední zástupce těchto nástrojů patří:

- opt — reprezentuje optimalizační fázi v modulárním LLVM překladu. Nástroj slouží pro analýzu a optimalizace nad vstupním souborem v LLVM IR. Výstupem nástroje jsou optimalizovaný soubor a výsledky analýz.
- llvm-link — nástroj zajišťující sestavení více jednotlivých LLVM IR souborů do jediného souboru.
- llc — je statický překladač, který převede vstupní soubor v LLVM IR do výstupního souboru ve strojovém kódu dané architektury.
- lldb — slouží jako knihovna pro ladění programu, jež je rychlejší a paměťově úspornější než běžný GDB u GCC.
- lli — umožňuje JIT interpretaci přímo z LLVM IR kódu. Nástroj není pouhým emulátorem, ale je plnohodnotným interpretem LLVM IR kódu.

V překladači LLVM je samotný program definovaný vnitřní hierarchickou strukturou. Na obrázku 2.7 je vidět zanoření těchto prvků.[7]



Obrázek 2.7: Obrázek zachycuje reprezentaci struktury programu v překladači LLVM. Program se dělí na globální proměnné a modul, který obsahuje jednotlivé funkce. Funkce jsou složené ze základních bloků. Bloky tvoří instrukce, které mají k sobě vázaný vektor operandů. Inspirováno obrázkem 3 v článku [57].

Struktura programu v LLVM se skládá z prvků:

- Modul (module) je základní jednotka pro překlad, analýzu a optimalizaci.
- Funkce (function) přibližně odpovídá funkci v jazyku C.
- Základní blok (basic block) seskupuje instrukce do struktury, pro kterou platí, že bude prováděna sekvenčně v pořadí dané její posloupností [50].
- Instrukce (instruction) je základním prvkem LLVM, který obsahuje vektor operandů (s jejich typem) a návratový typ instrukce (pokud instrukce vrací hodnotu).

2.2.2 Optimalizační průchody v LLVM

Modul zajišťující optimalizační fázi překladu v rámci překladače LLVM umožňuje vložit vlastní optimalizační průchod kódem. Tento průchod slouží uživateli k provedení analýzy či optimalizace daného LLVM IR kódu.

Samotný optimalizační modul poskytuje některé předpřipravené průchody. Tyto průchody se zabývají získáváním informací, laděním a vizualizačním znázorněním statistik. Průchody se dělí podle zaměření na analytické, transformační a pomocné. [10]

Během optimalizací lze kromě využití předpřipravených průchodů použít též průchody vytvořené uživatelem. Tyto průchody lze vytvořit pomocí dědění z podtříd třídy `Pass`, které jsou pro vytvoření uživatelského průchodu určeny. Pro využití tohoto průchodu je nutné ho ještě registrovat u správce registrací (PassManager). Pro implementaci funkcionality průchodu se využívá funkce `runOn<objekt>`. Obsah této funkce bývá uživatelem přepsán (overwrite). [11] Většina průchodů umožňuje přístup k určité úrovni hierarchické struktury (viz obrázek 2.7), či poskytuje průchod grafem.
V rámci LLVM jsou k dispozici například tyto průchody:

- ModulePass — poskytuje průchod celým modulem
- FunctionPass — poskytuje průchod jednotlivými funkcemi modulu
- LoopPass — prochází cykly vyskytující se v modulu
- BasicBlockPass — prochází všemi základními bloky modulu
- CallGraphSCCPass — je průchod grafu volání zdola nahoru

2.3 Instrukční sada LLVM

Po představení překladače LLVM v minulé podkapitole 2.2 je v této podkapitole přiblížen nízkoúrovňový jazyk LLVM IR, který slouží jako mezikód v překladu LLVM překladače. V prvním oddílu 2.3.1 jsou popsány vlastnosti a struktura jazyka LLVM IR. Druhá část (viz oddíl 2.3.2) pojednává o metadatech — jejich formě, provázání a umístění.

Prvotní myšlenka a důvody pro vytvoření instrukční sady byly představeny v článku [12]. Dále byly rozvinuty v článku [37], kde byl také poprvé představen překladač LLVM. Informace k LLVM IR čerpá tato podkapitola z knihy [38] a z prezentace [21].

LLVM IR je páteří celého procesu překladu překladače LLVM, jenž poskytuje propojení úvodní fáze a koncové fáze. Tím umožňuje překladači LLVM zpracovávat širokou škálu vstupních jazyků a generovat velké množství typů strojového kódu (viz oddíl 2.2.1 o architektuře LLVM). Kromě propojení těchto dvou fází je jazyk místem kde se mohou provádět optimalizace a analýzy nad překládaným programem.

2.3.1 Jazyk LLVM IR

LLVM IR je silně staticky typovaný jazyk, který dodržuje pravidla SSA (viz odstavec 2.3.1). Každý výsledek instrukce je v LLVM IR ukládán do nového registru. Množství nabízených registrů zde není limitováno (virtualizace) na rozdíl od množství fyzických registrů CPU. Vzniklé virtuální registry jsou při generování strojového kódu mapovány na fyzické registry CPU.

Pro instrukce pracující s daty se v LLVM IR používá tříadresný kód. Instrukce má dva zdrojové operandy a pozici, kam se uloží výsledek.

Instrukční sada LLVM IR má plochou strukturu (flat IR). V reprezentaci kódu to znamená, že iterace a větvení jsou zajišťovány instrukcemi skoku. Další z vlastností vyplývající z této reprezentace instrumentační sady je lepší převod v koncové fázi LLVM IR na strojový kód. Blízkost LLVM IR jazyka ke strojovému jazyku je tak velká, že je téměř nemožné namapovat části IR na původní kód. Tyto vlastnosti zajišťují snadnější optimalizace, ale ztěžují přenášení informací o živosti proměnných a výjimkách (LLVM IR správu si zajišťuje ve vlastní režii). Podobně jako jazyk LLVM IR používá plochou instrukční sadu například také jazyk PTX.

Důležitými třídami při tvorbě LLVM IR kódu jsou třídy odpovídající struktuře vnitřního překladu 2.7. K dalším důležitým třídám patří:

- IRBuilder — pomáhá s vytvářením IR kódu.
- Type — nadtřída ke všem LLVM konkrétním typům.
- PassManagerBuilder — zařazuje zvolené průchody kódem v rámci běhu pLLVM IR je páteří celého procesu překladu překladače LLVM, jenž poskytuje propojení úvodní fáze a koncové fáze. Tím umožňuje překladači LLVM zpracovávat širokou škálu vstupních jazyků a generovat velké množství typů strojového kódu (viz oddíl 2.2.1 o architektuře LLVM). Kromě propojení těchto dvou fází je jazyk místem kde se mohou provádět optimalizace a analýzy nad překládaným programe

  rogramu.
- ExecutionEngine — rozhraní pro JIT překladač.

LLVM IR umožňuje optimalizaci během sestavování programu, kvůli čemuž si při překladu ukládá LLVM IR kód na disk ve formátu bitcode (podobně jako bytecode u Javy). Sloučením více modulů do stejného souboru je umožněno provedení mezi-procedurální optimalizace.
Kód LLVM IR se může vyskytovat běžně v těchto reprezentacích:

- LLVM jazyk symbolických adres čitelný pro člověka (`.ll` soubor).
- Bitcode binární reprezentace (.bc soubor) - efektivní co do spotřeby místa na disku.
- C++ třídy (reprezentace v paměti).

Každá z těchto reprezentací má své výhody, jako jsou např. optimalizace místa na disku, nebo rychlost překladu. Mezi jednotlivými fázemi (LLVM jazyk symbolických adres a Bitcode binární reprezentací) je možné přecházet pomocí nástrojů (`llvm-as` a `llvm-dis`).

LLVM IR kód má kanonickou formu[6] — např. vstup a výstup ze základního bloku.

SSA (Static Single Assignment) je typ formy, která vyžaduje splnění následujících podmínek:

- Do registru (proměnné) může být hodnota přiřazena pouze jednou.
- Každá proměnná musí být definována dříve, než se použije.

[6] kanonická norma označuje jednoznačný tvar primitiv kódu.

Bez SSA bychom museli pro každý use-def řetězec[7] provádět analýzu diagramu datových toků (data flow), abychom mohli použít standardní optimalizace. Kód dodržující SSA pravidla umožňuje optimalizace, jako jsou např. propagace konstant, aritmetické zjednodušení, skládání konstant a propagace kopií [19].

Porovnání kódu jazyka C a LLVM IR. Na ukázkách kódu 2.2, C 2.1 jsou ilustrovány podobnosti a rozdíly mezi jazykem C a LLVM IR.

```
int sum(int a, int b) {
    return a~+ b;
}
```

Výpis 2.1: Ukázka kódu znázorňuje příklad funkce sčítání v jazyku C.

```
define i32 @sum(i32 %a, i32 %b) #0 {
entry:
    %a.addr = alloca i32, align 4
    %b.addr = alloca i32, align 4
    store i32 %a, i32* %a.addr, align 4
    store i32 %b, i32* %b.addr, align 4
    %0 = load i32* %a.addr, align 4
    %1 = load i32* %b.addr, align 4
    %add = add nsw i32 %0, %1
    ret i32 %add
}
```

Výpis 2.2: Na ukázce kódu je znázorněn kód funkce sčítání v LLVM IR po převodu funkce z výpisu 2.1.

Jak je vidět v ukázkách kódů, tak LLVM IR je v některých syntaktických konstrukcích podobný jazyku C. Po klíčovém slovu `define` je uveden návratový typ a v závorkách typované parametry podobně jako v hlavičce funkce jazyka C. Za funkcí, podobně jako u jazyka C++, jsou uvedeny atributy formou odkazu (#0) na strukturu atributů na konci souboru.

Pro práci s pamětí se v LLVM IR využívá architektury load/store, kde tyto dvě operace jsou jediné, které mohou pracovat s pamětí, a proto jsou napříč celým kódem hojně využívány. Další z důležitých instrukcí je instrukce `alloca`, která doplňuje tuto dvojici tím, že alokuje místo na zásobníku.

Proměnné se značí podobně jako registry v jazyku symbolických instrukcí. Každá proměnná začíná symbolem "%", po kterém následuje řetězec alfanumerických znaků. Tvar jména není pevně stanovený, ale standardně se využívá číselných názvů proměnných.

V jazyku LLVM IR lze nastavit zarovnávání dané proměnné v paměti. Děje se tak pomocí atributu `align`, který určuje, na kolik bytů se má zarovnat, a to na násobek zvoleného čísla (možné jsou jen násobky dvou).

[7] Use-def řetězec je datová struktura obsahující množinu užití proměnných a množinu definování proměnných.

LLVM IR disponuje základními datovými typy:

- Celočíselný — iN, kde N udává počet bitů (standardně 32, 64, 128)
- S plovoucí čárkou — float (32bitů) a double (64 bitů)
- Vektor — je ve formátu `<počet_elementů x typ_elementů>` (např. `<4 x i32>`)

2.3.2 Metadata LLVM IR

Jazyk LLVM IR využívá metadata k připojení informací o kódu k instrukcím a funkcím programu, které jsou následně využity pro optimalizace a generování kódu. Metadata mohou sloužit jako ladící informace. Všechna metadata jsou na svém začátku syntakticky označena symbolem "!". V metadatech se též používá klíčové slovo `distinct`, které znamená, že dané matadata nejsou unikátní.

Informace k metadatům pochází z dokumentace ladícího formátu Dwarf [2], která popisuje jeho aktuální verzi 5. Taktéž čerpá z dokumentace projektu LLVM [5], kde jsou popsány všechny typy metadat a jejich umístění v kódu.

Metadata se dělí podle toho, k jaké kódové části poskytují dodatečné informace. Data obsažená u jednotlivých částí se liší — u souborů obsahují jiné položky než u proměnných, či metadat polohy kódu (viz výpis 2.3). Metadata u těchto částí kódů se mohou vyskytovat na vícero místech. Většinou metadata zaujímají pozici za instrukcí, ke které se vztahují. Jednou z výjimek jsou pomocné funkce, generované přímo LLVM IR, kde jsou metadata na pozicích argumentů funkce.

```
!0 = !DIGlobalVariableExpression(var: !1, expr: !DIExpression())
!1 = distinct !DIGlobalVariable(name: "global_var", scope: !2, file: !3,
line: 6, type: !12, isLocal: false, isDefinition: true)
!2 = distinct !DICompileUnit(language: DW_LANG_C, file: !3, producer:
"clang version 9.0.1", emissionKind: FullDebug, enums: !4, globals: !5)
!3 = !DIFile(filename: "src/file.cpp", directory: "/home/user/tmp_dir")
```

Výpis 2.3: Kód znázorňuje část metadat po výpisu ladících informací (pomocí přepínače -g) k překládanému programu. Jednotlivé číslice za symbolem "!"znázorňují názvy jednotlivých metadat. Za rovnítkem můžeme vidět příslušný typ metadat a v závorce pak informace, které tato metadata nesou.

Kapitola 3

Další instrumentační nástroje

V této kapitole jsou popsány existující řešení pro instrumentaci programu během překladu. Jednotlivé podkapitoly představují možné způsoby řešení této problematiky. Ke každému řešení je rovněž připojeno shrnutí jeho výhod a nevýhod.

Prvním z těchto řešení je nástroj XRay (viz podkapitola 3.1), který je součástí překladače LLVM. Druhým řešením je přepínač `-finstrument-instruction` (více lze nalézt v podkapitole 3.3), který je standardním nástrojem pro instrumentaci funkcí u vícero překladačů. Třetí podkapitola se zabývá sadou nástrojů Score-P (více viz podkapitola 3.2). Čtvrté řešení SBT-instrumentation (viz podkapitola 3.4) umožňuje instrumentaci konfigurovanou pomocí pravidel zadaných v souboru JSON. Poslední řešení představuje CSI Framework (viz podkapitola 3.5), který poskytuje komplexní instrumentaci mnoha různých typů primitiv v kódu (funkce, operace s pamětí, atd.). Poslední podkapitola shrnuje představená řešení (viz podkapitola 3.6) a zaměřuje se na srovnání jednotlivých nástrojů s aplikací vyvíjenou v této práci.

3.1 XRay

První z existujících řešení je nástroj pro sledování volání funkcí XRay. Tento nástroj umožňuje instrumentaci funkcí během překladu. Nástroj též poskytuje knihovnu, která umožňuje za běhu programu získávat data o instrumentovaných funkcích. Tato knihovna dokáže dynamicky za běhu programu povolit nebo zakázat vykonávání vloženého kódu v rámci instrumentovaných funkcí. Dostupné informace k tomuto nástroji pochází z článku tvůrců tohoto nástroje [58] a jeho dokumentace [59].

Nástroj umožňuje provádět instrumentaci vstupních a výstupních bodů funkcí. Při procesu instrumentace jsou identifikována místa — kde budou následně vloženy instrumentované funkce — pomocí sledů bytů, které samy o sobě neslouží jako vykonavatelný příkaz, jedná se tedy v podstatě o `no-op`[1]. Tyto sledy jsou poté zaznačeny do tabulky, která je zakódována v rámci objektového souboru. Při zapnutí instrumentace jsou tyto sledy bytů nahrazeny *uživatelským kódem*[2].

[1] no-op - no operation

[2] Tímto termínem budeme označovat kód, který se vkládá na instrumentovaná místa v programu.

Nástroj XRay může být použit těmito způsoby:

- Pro instrumentaci aplikací napsaných v jazycích C/C++/Objective-C/Objective-C++ — funkce, které mají být instrumentovány, musí být označeny ve zdrojovém souboru atributy. Atributy mohou být definovány za deklarací funkce, nebo dvojicí hranatých závorkách před deklarací funkce (podle normy C++11). K instrumentaci samotné je potom nutné spustit překladač s dodatečnými přepínači pro instrumentaci.

- Generování LLVM IR s příslušnými atributy funkcí — funkce jsou instrumentovány přímo v LLVM IR kódu. Podobně jako u předešlého způsobu se deklarace funkce osazují atributem, ve kterém určujeme zda bude instrumentace povolena nebo zakázána.

- Speciálním souborem se seznamem funkcí — jedná se o speciální soubor s vlastní syntaxí, kde jsou deklarovány funkce, které se budou instrumentovat a které ne. Při překladu je pak tento soubor specifikován pomocí přepínače.

Výhody a nevýhody řešení První důležitou výhodou nástroje Xray je přítomnost tohoto nástroje přímo v základní distribuci LLVM. Díky této skutečnosti je nástroj kontinuálně podporován a vyvíjen spolu s překladačem. Vývojáři tohoto nástroje rozumí do detailu vnitřní implementaci překladače a mohou tak nástroj optimalizovat, jak do rychlosti, tak do množství prostoru na disku, který zaberou výsledné instrumentované programy. Dle experimentů provedených vývojáři nástroje se při spuštění instrumentace zvýší využití CPU o 20-40% a navýší se velikost binárního souboru o 2%.

Jednou z dalších výhod nástroje XRay je podpora příkazů běžně dostupných u analytických nástrojů [59] pro trasování[3] jako jsou `extract, convert, graph, stack, atd.` Nástroj rovněž garantuje podporu u vícevláknových programů.

Nevýhodou nástroje XRay je rozsah možností instrumentace, který je omezen pouze na instrumentaci vstupních a výstupních bodů funkcí. Pokrývá tak pouze malou část funkcionality požadované v této práci.

3.2 Score-P

Score-P je sada měřicích nástrojů, které umožňují analýzu zkoumaného programu profilováním, sledováním událostí a online analýzou HPC[4] aplikací. Díky zmíněným analýzám poskytuje tato sada nástrojů možnost instrumentace funkcí. K analytickým nástrojům se lze připojit pomocí rozhraní zásuvných modulů (*plugins*) a tím využít (případně rozšířit) jejich funkcionalitu. Podrobnosti o této sadě nástrojů vychází z článku [57] — kde je popisována instrumentace pomocí zásuvných modulů skrze tuto sadu nástrojů — z oficiálních stránek tohoto produktu [63] a její dokumentace [1].

Score-P umožňuje uživateli instrumentovat funkce těmito způsoby (nástroj podporuje spoustu druhů instrumentace, ale pro obsah této práce jsou důležité jen některé z nich):

- Instrumentace všech funkcí podobně jako v případě funkce překladače finstrument-functions (viz 3.3).

- Poloautomatická instrumentace — rozsah instrumentace je vymezen pomocí použití direktiv rodiny jazyka C (pro jazyky postavená na jazyku C) nebo jiným ohraničením regionu, který má být instrumentován.

[3] Analytické nástroje pro trasování - trace analysis tools
[4] HPC — high-performance computing - vysoce náročné výpočty

- Instrumentace definovaná parametry — pro určitou část, vyznačenou definovanými body, je možnost instrumentovat pouze vybrané funkce na základě jejich parametrů. Parametry mohou nabývat hodnot celočíselných typů (integer) a typu řetězce (string).

- Instrumentace pomocí PDT[5] — uživatel poskytne soubor se seznamy funkcí, které se nemají či mají instrumentovat[6]. Nástroj pak zajistí předzpracování instrumentovaného zdrojového kódu do objektového souboru. Dle vyjádření autorů má tento přístup zatím svá omezení a nachází se v experimentální fázi.

- Instrumentace knihoven — Score-P umožňuje instrumentovat volání knihovních funkcí pomocí tzv. *obálky* (*wrapper*). Díky tomuto mechanismu tak není nutné mít k dispozici zdrojové kódy dané knihovny a provádět opětovné sestavení knihovny s využitím Score-P nástroje.

Score-P instrumentace využívá LLVM průchod (*LLVM pass*) k instrumentaci funkcí na jejich vstupních a výstupních bodech. Zpřístupňuje navíc i metadata (např. jméno volající funkce a pozici volané funkce ve zdrojovém kódu).
Score-P podporuje dvě možnosti připojení zásuvných modulů pomocí rozhraní [43]:

- *Metric Plugin* — toto rozhraní umožňuje přidávat zásuvnému modulu sledovaná data do *monitorovací infrastruktury*[7]

- *Substrate Plugin* — rozhraní zásuvného modulu, které dovoluje specifikovat vlastní způsob zpracování dat ze Score-P

Výhody a nevýhody řešení První z výhod je snadná orientace v dokumentaci, která svou vizuální formou připomíná LLVM dokumentaci. Druhou výhodou je možnost využití tohoto nástroje na vícero typech překladačů (LLVM, GCC, IBM xlf, PGI), a to ve třech podporovaných jazycích (C, C++ a Fortran).

Další výhodou je možnost rozšíření funkcionality Score-P nástrojů použitím zásuvných modulů, které mohou čerpat data z této sady nástrojů a nebo jim je poskytovat.

Sada nástrojů se řídí specifikací *Score-P Governance Model*, která vymezuje šíři, vlastnosti a funkcionalitu nástroje[8]. Existence specifikace tak poskytuje výhodu této sadě nástrojů.

Poslední výhodou Score-P je PDT instrumentace, při níž je možné specifikovat seznam funkcí, které mají být instrumentovány (jako v nástroji XRay 3.1, ale trochu odlišným způsobem) — na rozdíl od přepínače finstrument 3.3, který tuto možnost nepodporuje.

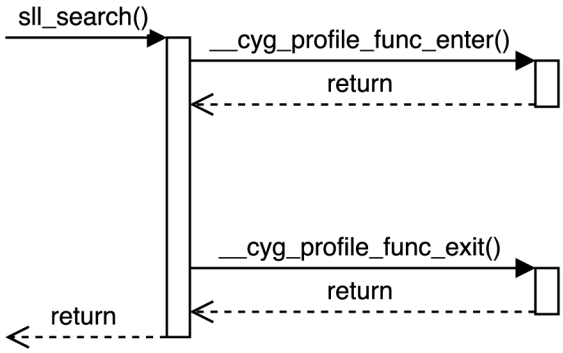
Nevýhodou nástroje je omezená šíře instrumentace, tzn. nástroj dokáže instrumentovat pouze funkce, což je pro potřeby této práce nevyhovující.

3.3 Instrumentace pomocí přepínače finstrument-functions

Tento funkce (jedná se pouze o přepínač) je součástí překladače LLVM, stejně jako GCC[9] překladače. Nástroj slouží k instrumentaci funkcí v jejich vstupních a výstupních bodech

[5] PDT - PDToolkit
[6] Formát souboru viz http://www.cs.uoregon.edu/research/tau/docs/newguide/bk01ch01s03.html
[7] datové typy pro sadu nástrojů
[8] https://www.vi-hps.org/cms/upload/packages/scorep/scorep_gov.pdf
[9] GCC - GNU Compiler Collection

(viz obrázek 3.1). Množství instrumentovaných funkcí lze regulovat dodatečným přepínačem *-finstrument-functions-exclude-function-list*, pomocí kterého je možné vynechat instrumentaci některých funkcí. Seznam vyňatých funkcí[10] specifikuje, které funkce nemají být instrumentovány. Informace o této funkci překladače pochází z dokumentace o argumentech příkazové řádky [4] a z bakalářské práce [45], která daný nástroj využívá ve sběru dat. Výhody a nevýhody řešení tohoto nástroje byly konzultovány s autorem bakalářské práce.



Obrázek 3.1: Obrázek ukazuje princip instrumentace pomocí přepínače finstrument. Konkrétně volání funkcí bezprostředně po vstupu do těla funkce sll_search, respektive před výstupem z něj. Logiku těchto instrumentačních funkcí je ale nutné implementovat ve vlastní režii. Obrázek převzán z diplomové práce [45].

Výhody a nevýhody řešení Hlavní výhodou této funkce je její podpora už v rámci samotného LLVM překladače. Další výhodou je snadnost použití — postačuje definovat dvě funkce s pevně daným předpisem[11]. Výhodou je též, že není nutné modifikovat stávající zdrojový kód (nicméně stále je nutné dodat soubor s definicemi zmiňovaných funkcí).

První z nevýhod je množství nadbytečně instrumentovaných funkcí. Počet instrumentovaných funkcí výrazně vzroste, pokud je použita například STL knihovna — v tomto případě se může jednat až o řádové zvýšení počtu instrumentovaných funkcí. Tento nárůst počtu instrumentovaných funkcí negativně ovlivňuje režii, která se výrazně zvýší (čerpáno z dokumentu [57] z tabulky 2, kde je srovnání doby běhu programu při instrumentaci jednotlivými nástroji).

Další z nevýhod nástroje je omezená možnost specifikovat funkce, které mají být instrumentovány. Nástroj umožňuje definovat pouze seznam funkcí, které nemají být instrumentovány. Uživatel by tak měl uvést seznam všech funkcí, které si nepřeje instrumentovat (včetně funkcí v použitých knihovnách).

Velkou nevýhodou je implementační omezení, při němž jsou pro všechny instrumentované funkce dostupné pouze dva standardizované předpisy funkcí pro vložení uživatelského kódu — jeden pro vstupní bod instrumentované funkce a jeden pro její výstupní bod. Toto citelně omezuje možnosti definovat různý uživatelský kód pro konkrétní funkce, aniž by byla znatelně navýšena časová režie.

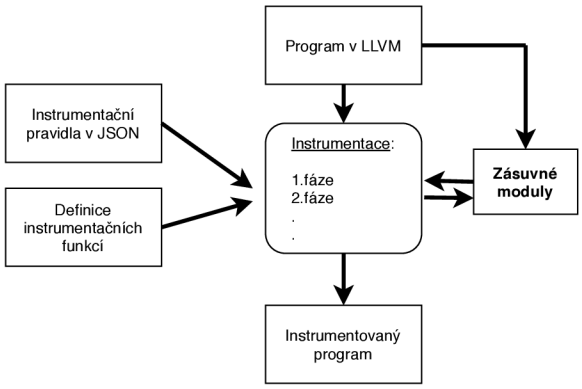
[10] Seznam vyňatých funkcí- exclude list

[11] `void __cyg_profile_func_enter (void *this_fn, void *call_site)`
`void __cyg_profile_func_exit (void *this_fn, void *call_site)`

Poslední nevýhodou je úzké zaměření nástroje, které má za následek, že lze instrumentovat pouze funkce. Vzhledem k velkému množství nevýhod nebyl tento přístup dále zvažován pro vytvoření instrumentačního frameworku.

3.4 SBT-instrumentation

SBT-instrumentation je sada nástrojů pro konfigurovatelné instrumentování kódu v překladači LLVM. SBT instrumentuje přímo kód LLVM IR, se kterým rovněž pracuje. K identifikaci částí kódu, které se mají instrumentovat, slouží konfigurační soubor ve formátu JSON. Kromě konfiguračního souboru je zde též soubor obsahující definice vkládaných funkcí (viz obrázek 3.2). Informace o nástroji jsou čerpány z diplomové práce [60], která se zabývá jeho návrhem a implementací. O nástroji je též vydán článek [62], který tento nástroj představuje a vysvětluje jeho využití. Posledním zdrojem je úložiště GIT [61], které obsahuje aktuální informace o rozvoji nástrojů.



Obrázek 3.2: Schéma fungování konfigurovatelného nástroje SBT. Na vstup instrumentace vstupují dva soubory s konfigurací (jeden s definicema funkcí, druhý s pravidly ve formátu JSON) a program pro instrumentaci. Modul SBT se připojuje skrze rozhraní a provádí samotnou instrumentací. Výsledkem je instrumentovaný program na výstupu. Obrázek je přepracován podle vzoru v článku [62].

Princip nástroje SBT-instrumentation spočívá ve vytvoření vlastní specifikace bez zásahu do původního zdrojového kódu. Primárně byla tato sada nástrojů vytvořena pro kontrolu bezpečnosti paměti[12]. Nástroj dokáže detekovat nevalidní přístup do paměti[13], nevalidní uvolnění paměti a paměťové úniky. Podpora uživatelských zásuvných modulů rozšiřuje možnosti použití sady i k jiným účelům. Důsledkem tohoto návrhu nástroje jsou další zásuvné moduly implementující nové techniky analýzy (lze pozorovat v GIT úložišti [61]).

Samotná instrumentace se skládá z jednotlivých fází, kdy pro každou fázi jsou definována nová instrumentační pravidla — přičemž počet fází si může uživatel zvolit podle svých potřeb. Sada nástrojů využívá k přístupu k částem LLVM IR vlastního specifikačního jazyka (definovaný v [60]), jehož základním stavebním blokem je instrumentační pravidlo. V rámci specifikačního jazyka lze pomoci zásuvných modulů rozšiřovat specifikace svými vlastními podmínkami (což umožňuje blíže specifikovat filtraci). Sada umožňuje předávat

[12] bezpečnost paměti - memory safety
[13] nevalidní přístup do paměti - dereference

informace z jedné fáze do druhé, díky čemuž dokáže řetězit specifikace. Instrumentační pravidla udávají vzory, dle kterých se v LLVM IR kódu hledají jim odpovídající bloky. Po nalezení odpovídajícího LLVM IR kódu je ověřeno jestli je splněna podmínka. Po splnění podmínky může příslušné pravidlo zajistit vložení nového uživatelského kódu na nalezenou pozici v LLVM IR (zatím je implementované pouze vložení volání funkce).

Sada nástrojů umožňuje též zúžení oblasti, kde se má část LLVM IR kódu hledat. Podporované možnosti hledání jsou:

- Pouze ve funkci `main`.
- Ve všech funkcích programu.

Nástroj využívá záznamový soubor[14], do kterého ukládá záznamy o vykonaných událostech, jako jsou například začátky a konce fází, vložení funkcí, nahrávání zásuvných modulů, atd.

do kterého ukládá záznamy o vykonaných událostech se použitím fází a statické analýzy dramaticky sníží počet vložených instrukcí, a to až o 85%.

Výhody a nevýhody řešení První z výhod sady nástrojů SBT je vlastní specifikační jazyk, který zvyšuje flexibilitu vkládání funkcí. Nevýhodou je nutnost naučit se tento specifikační jazyk a též komplikovanější a pracnější zápis ve formátu JSON. Tato vlastnost se může projevit především při automatizaci, kdy by každá změna pravidel způsobila nutnost opětovného sestavení JSON kódu a poté celou analýzu znovu spustit.

Další z výhod SBT je vynechání instrumentace nedosažitelného kódu, čímž se sníží množství celkově instrumentovaných funkcí.

Výhodou je též jednoduchá rozšiřitelnost nástroje o nové techniky analýzy — proces se skládá z definice vlastních podmínek a potřebných funkcí k jejich vyhodnocení, které jsou umístěny v zásuvném modulu.

Nevýhodou je omezené množství podmínek, kterými jde filtrovat vyhledávané pravidla. V základní implementaci je jich pouze hrstka, další potřebné podmínky musí si uživatel sám nadefinovat v přídavném zásuvném modulu. V základní verzi není implementovaná podmínka pro kontrolu primitiva podle typu, což způsobí zvýšení množství instrumentovaných funkcí.

Jednou z největších nevýhod je chybějící podpora pro práci s lokálními proměnnými, která je důležitou součástí práce. Na základě zde uvedených nevýhod byl nástroj vyhodnocen jako nevhodný pro použití v této práci.

3.5 The CSI Framework for Compiler-Inserted Program Instrumentation

CSI Framework poskytuje statickou instrumentaci programu, kterou dále využívají nástroje pro dynamickou analýzu. Komunikace nástroje CSI s překladačem je prováděna pomocí vlastního rozhraní. Nástroj CSI vkládá do instrumentovaného programu své instrukce, které jsou záchytným místem při instrumentaci. Informace k tomuto frameworku byly převzaty z článku tvůrců [46].

Nástroj CSI umožňuje instrumentaci mnoha událostí, jakými jsou přístupy do paměti, volání funkce, vstup a výstup z funkce, atd. Všechny prvky dostávají svou vlastní identifikaci (ID) a jsou zapsány do interní ploché reprezentace, která umožňuje pracovat s prvky

[14] záznamový soubor - log file

nezávisle na jejich typu. Nástroj bohužel umožňuje pouze instrumentaci jednotlivých typů primitiv, bez možnosti instrumentovat pouze jejich částí.

Výhody a nevýhody řešení První z výhod je možnost komunikovat s frameworkem skrze rozhraní, což poskytuje kompatibilitu s více překladači — rozhraní však musí být nejprve implementováno na straně daného překladače.

Další z výhod frameworku je široký výběr instrumentovatelných prvků, jako jsou např. funkce, paměťové operace, atd. Instrumentace této množiny prvků poskytuje výhodu oproti jiným nástrojům zabývajících se instrumentací během překladu. Instrumentace operací s pamětí je však limitována omezeními popsanými v závěrečné kapitole [46]).

Z dalších výhod CSI lze uvést automatické odstranění (v průběhu fáze sestavování programu[15]) funkcí vložených instrumentací, pro které však nebylo definováno jejich obsloužení.

Nevýhodou je nemožnost instrumentace částí primitiv daného typu (load, store, atd.). Výběr jednotlivých instancí musí probíhat až v těle obslužné funkce pomocí podmínky — tímto se zvýší počet vložených funkcí.

Hlavní nevýhodou CSI je nutnost modifikace LLVM překladače. Překladač musí být rozšířen o rozhraní, které umožní nástroji CSI vkládat záchytné body (hooks) do LLVM IR programu. Kvůli této úpravě zdrojového kódu LLVM překladače je nutno dbát na kompatibilitu frameworku CSI s novými verzemi překladače. Tvůrci usilovali o začlenění frameworku do LLVM překladače za účelem zajištění kompatibility s novějšími verzemi, avšak neúspěšně. Poslední implementace rozhraní CSI frameworku pro LLVM překladač je z jara roku 2017, což odpovídá verzi 3.9, která je již zastaralá — nástroj tak už není udržován.

3.6 Shrnutí existujících řešení

V kapitole 3 bylo představeno několik zástupců z nástrojů pro instrumentaci programů během překladu. Přestože zde byly popsány především nástroje podporující univerzálnější instrumentaci primitiv (které jsou relevantní pro tuto práci), existuje i mnoho úzce specializovaných instrumentačních řešení pro konkrétní podmnožiny primitiv.

Všechny zde zmíněná řešení jsou distribuovaná pod hlavičkou open-source licencí. Konkrétně nástroj XRay, přepínač finstrment (oba patřící pod LLVM) a taktéž CSI framework jsou vydávány s licencí Apache 2.0. Další z nástrojů, Score-P a SBT, jsou distribuovány s licencemi BSD a MIT.

Celkový přehled funkcionality jednotlivých nástrojů znázorňuje tabulka 3.1.

Vysvětlení tabulky 3.1 Tabulka srovnává řešení z hlediska šíře instrumentovaných primitiv, kompatibility s překladačem a aktuálností nástrojů. V následujících pár odstavcích je popsána legenda kritérií.

- **Function** dovoluje instrumentaci vstupního a výstupního bodu funkce.
- **Call** znamená instrumentaci před (popřípadě za) voláním funkce.
- **Basic block** umožňuje instrumentovat všechny základní bloky programu.
- **Mem operation** poskytuje instrumentaci operace load a store.

[15] link phase

- **Identificators** skýtají možnost práce s informacemi o jednotlivých lokálních proměnných ve funkci.
- **Return value** poskytuje možnost pracovat s návratovou hodnotu funkce.
- **No edit SC** udává, jestli lze instrumentaci provést tak, aby nebyly pozměněny stávající zdrojové kódy programu. U XRay je tato funkcionalita podporována u jednoho z nastavení.
- **Easy update LLVM** vyjadřuje, jestli je nástroj snadno převeditelný na další verzi překladače LLVM (je dodáván pomocí zásuvných modulů) a není pevně spoután s danou verzí překladače (zásah přímo do zdrojových kódů překladače).
- **Last update** vyjadřuje, kdy byl nástroj naposledy aktualizován. Toto datum vypovídá o tom, zda-li je nástroj stále vyvíjen. U záznamů, které mají v datu vynechaný rok, se implicitně počítá s rokem 2019. U funkce finstrument se počítá s datem úpravy překladače.

Informace jsou platné k datu 5. 12. 2019. Případné pozdější aktualizace nejsou v této tabulce zohledněny.

	XRay	Finstrument	Score-P	SBT	CSI	T-force
Function entry/end	✓	✓	✓	✓	✓	×
Call	×	×	×	✓	✓	✓
Basic block	×	×	×	×	✓	×
Mem operation	×	×	×	✓	✓	✓
Identificators	×	×	×	×	×	✓
Return value	×	×	×	✓	×	✓
NO EDIT SC	✓	✓	✓	✓	✓	✓
Easy update LLVM	✓	✓	✓	✓	×	✓
Last update	15.8.	5.12.	26.10.	30.10.	14.4.2017	4.12.

Tabulka 3.1: Tabulka popisuje možnosti instrumentace jednotlivých primitiv, kompatibilitu s překladačem a aktuálnost nástrojů (viz popis tabulky 3.6).

Závěr ze shrnutí Po prostudování některých existujících řešení pro instrumentaci programu během překladu je patrné, že žádný z nástrojů nesplňuje požadavky ze specifikace zadání diplomové práce a nebyl tak vhodný ani pro jeho dílčí využití.
Hlavní důvody nezvolení některého z již existujících řešení jsou tyto:

- Zakomponování nástroje by vyvíjený program spíše zkomplikovalo, především z důvodu složitosti rozhraní a rozdílnosti ve zpracování dat.
- Dalším důvodem nevyužití nástrojů je jejich rozličná funkcionalita. Tato rozličnost je způsobena často jiným zaměření nástroje.
- Nástroj má být udržitelný v rámci výzkumné skupiny Testos (viz kapitola 7.5), což znamená, že se do budoucnosti počítá s rozšiřováním funkcionality a úpravami. Použitím nástroje 3. strany by se tento projekt stal závislý na podpoře a vývoji nástroje.

- Plánovaná rozšíření, které by nebylo možné zakomponovat do stávajících nástrojů. Mezi tyto rozšíření patří centralizovaná paralelní instrumentace, atd.

Především z těchto důvodů tak v této práci představujeme nový instrumentační framework. Konkrétně kapitoly 4, 5 a 6 pojednávají o analýze, návrhu a implementaci nástroje (Tforc).

Kapitola 4

Analýza způsobů řešení instrumentace

Tato kapitola popisuje analýzu problému vytvoření instrumentačního frameworku. Kapitola čerpá potřebné informace pro analýzu problému z teorie zmíněné v kapitole 2.

V podkapitole 4.1 je popsána specifikace požadavků, které musí instrumentační framework splňovat. Podkapitola 4.2 je věnována způsobům, jak řešit volbu vhodných přístupů ve výběru způsobu tvorby nástroje. Jsou zde nastíněné jednotlivé přístupy a se zdůvodněním vybrán jeden způsob, který se jevil jako nejvhodnější.

4.1 Specifikace požadavků

V této podkapitole jsou specifikovány požadavky, které musí vyvinutý nástroj splňovat. Specifikace vychází ze zadání diplomové práce a posléze po konzultaci s vedoucím práce jsou konkretizovány jednotlivé požadavky. Požadavky specifikace jsou následovné:

1. Program musí umožňovat snadnou konfiguraci instrumentace testovaných programů.
2. Program musí umožnit instrumentaci funkce,
   - 2.1. před jejím voláním a dokázat předat vkládané funkci argumenty instrumentované funkce,
   - 2.2. po jejím volání a dokázat předat vkládané funkci argumenty instrumentované funkce a její návratovou hodnotu.
3. Program musí umožnit instrumentovat instrukce pro přístup do paměti, jak u lokálních, tak globálních proměnných. U operace `store` před uložením do registru a u operace `load` po načtení z registru. Funkce vložená během instrumentace musí umět poskytnout:
   - 3.1. adresu, na které se proměnná nachází,
   - 3.2. hodnotu, kterou nyní proměnná nabývá a
   - 3.3. identifikace části zdrojového kódu, kde se operace vyskytuje.
4. Program musí být být vyzkoušen na testovací sadě, která bude ověřovat hlavní funkcionalitu nástroje.
5. Program musí být schopen integrace do nástroje Spectra v platformě Testos.

4.2 Analýza způsobu realizace frameworku

V této podkapitole jsou analyzovány způsoby přístupu k vyvíjenému nástroji a jeho instrumentaci. V prvním oddíle 4.2.1 je analyzována a rozebrána volba přístupu k tvorbě nástroje. Druhý oddíl 4.2.2 se zabývá analýzou jednotlivých možností instrumentace v rámci LLVM průchodu.

4.2.1 Výběr přístupu k instrumentaci, volby překladače a způsobu tvorby nástroje

Zadání práce vyžadovalo vytvoření nástroje, který dokáže instrumentovat funkce a přístup do paměti. Pro instrumentaci se nejčastěji využívají dva přístupy: buďto instrumentovat kód již při překladu, nebo ho instrumentovat až v jeho binární podobě (viz oddíl 2.1.3).

Pro instrumentaci byl zvolen přístup instrumentace za překladu, kvůli již existujícímu řešení problému pomocí binární instrumentace. Avšak toto řešení vykazovalo při svém využívání velkou režii a neumožňovalo instrumentovat přístupy do paměti s nepřímou adresací, která je pomocí binární instrumentace těžko implementovatelná. Instrumentace za překladu by měla snížit režii instrumentace, zvýšit rychlost (menší množství vloženého kódu) a umožnit zisk většího množství informací, které může poskytnout uživateli díky překladači. Instrumentace za překladu také umožňuje využívat optimalizace vestavěné v překladači, a tím redukovat množství instrukcí a zrychlit běh programu.

Pro instrumentaci při překladu připadaly v úvahu dva překladače, a to GCC a LLVM díky jejich používanosti mezi uživateli a nabízeným možnostem rozšíření.

Z těchto dvou překladačů byl vybrán LLVM díky možnosti instrumentovat pomocí zásuvných modulů a mezikódu LLVM IR (viz podkapitola 2.3), který umožňuje využití široké škály již existujících optimalizací.

Po zvolení přístupu instrumentace (formou instrumentace během překladu) a cílového překladače (LLVM) byl zvažován způsob, jak tento nástroj realizovat s vybraným překladačem. Instrumentace kódu během překladu s překladačem LLVM je možná realizovat několika možnými způsoby:

1. Vytvořit si kopii současného překladače a v něm provést změny zdrojového kódu zajišťující instrumentaci. Výhodou je instrumentace přímo v rámci překladu bez přebytečné režie. Tato možnost má velkou nevýhodu v nutnosti udržovat kopii překladače v aktuální (používané) verzi původního překladače, což zabírá vývojáři takového nástroje hodně prostředků. Příkladem využití tohoto způsobu je nástroj framework CSI (viz podkapitola 3.5).

2. Vytvořit k překladači zásuvný modul, který umožňuje rozšířit funkcionalitu překladače. Tento zásuvný modul umožňuje vytvořit průchod, ve kterém je možné provést instrumentaci kódu. Výhody tohoto způsobu jsou: možnost sekvenčního procházení programu po instrukcích, dále možnost využít prostředky optimalizace překladače LLVM a vysoká míra nezávislosti na verzi překladače (závislost je zde pouze na rozhraní a datových typech). Nevýhodou je zvýšená režie oproti instrumentaci v kopii překladače (1. možnost).

3. Vytvořit vlastní nástroj, který by procházel kód po mezipřekladu do LLVM IR. Zde by nástroj kód instrumentoval a posléze by ho skrze překladač přeložil do binární podoby. Jediné výhody této možnosti jsou: nezávislá instrumentace na překladači (lze ji provádět bez jeho přítomnosti — na jiném stroji) a nezávislost na struktuře

a rozhraní překladače. Nevýhodou je nutnost získávat všechny informace z mezikódu LLVM IR jeho analýzou, která přinese velkou režii kvůli kompletní důkladné analýze LLVM IR kódu.

4. Upravit překladač (1. možnost) a poté se pokusit ho předat vývojářům LLVM, aby ho začlenili jako část stávajícího překladače. Takto vytvořený nástroj má výhody první možnosti. Hlavní nevýhodou je velmi obtížné, dlouhé a komplikované začleňování do překladače LLVM pomocí komunity starající se o něj, které se někdy nemusí vydařit (neúspěch framework CSI viz podkapitola 3.5). Jedním ze zástupců, kterému se podařil tento proces, je nástroj pro instrumentaci funkcí XRay (viz podkapitola 3.1), jenž se stará o instrumentaci funkcí.

5. Využít stávajících nástrojů na instrumentaci a k nim vytvořit nástroj doplňující chybějící funkcionalitu. Výhody jsou určeny použitým nástrojem. Kromě nevýhod plynoucích z kombinace využitých nástrojů je další nevýhodou závislost vzniklého nástroje na vývoji a podpoře ze strany vývojářů. Každá změna architektury, výstupů, atd. může ovlivnit funkčnost nástroje vytvořeného s využitím jiného nástroje. Další nevýhodou jsou omezené možnosti předávání informací o instrumentaci mezi nástroji — neposkytuje takové možnosti jako samotný překladač.

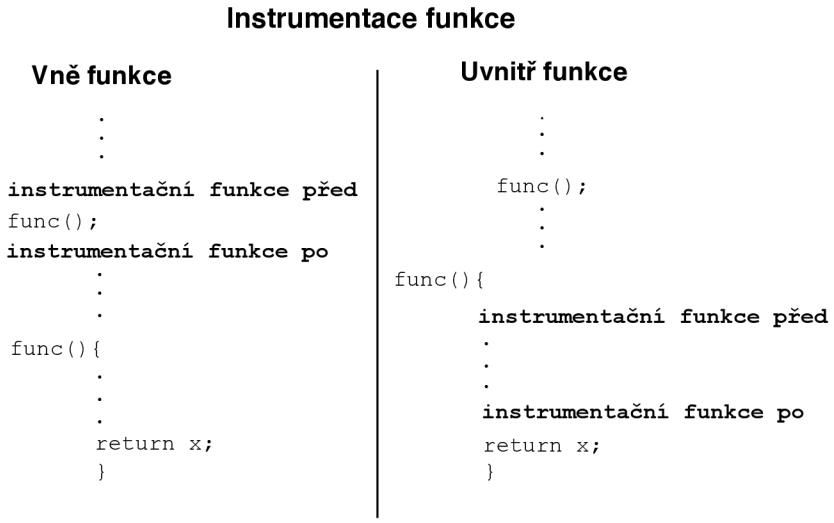
Pro řešení nástroje vzniklého v této práci byla zvolena možnost zásuvného modulu formou průchodu (*Pass*). Hlavním důvodem volby zásuvného modulu je snadná udržovatelnost programu s aktuální verzí LLVM překladače. Dalším důvodem je možnost v budoucnosti samostatně vyvíjet tento modul projektem Testos (viz podkapitola 7.5) vlastním směrem bez závislostí na jiných projektech. Posledním důvodem volby je možnost přístupu k mnoha informacím z překladače a snadnější práce s metadaty LLVM IR (viz oddíl 2.3.2).

4.2.2 Analýza možností instrumentace v modulu

Průchod kódem programu probíhá sekvenčně podle pořadí funkcí a instrukcí v nich, z čehož plyne, že každá funkce/instrukce je zpracována právě jednou. Díky tomu je možné ke každé instrukci odpovídající zkoumanému artefaktu vygenerovat a vložit volání funkce. Při samotném přidávání instrukcí před/za zpracovávanou instrukcí se musí brát v potaz možné zpracování vložených instrukcí během pokračování průchodu programu. Vygenerované instrukce musí být buďto přeskočeny nebo ignorovány.

Zadání diplomové práce vyžaduje instrumentaci funkcí a přístupů do paměti. Průchod instrukcemi poskytuje možnost provedení této instrumentace a navíc v budoucnosti možnost rozšířit množství sledovaných artefaktů.

Pozice vkládané funkce při instrumentaci funkce. Instrumentace funkce nabízí dvě možnosti výběru místa, na kterém je ji možno realizovat. První z možností je umístit funkci, která se instrumentací vkládá před (popř. za) voláním funkce (tzn. vně funkce). Druhou možností je umístit vkládanou funkci dovnitř funkce. Zde se umísťuje vkládaná funkce na začátku instrumentované funkce nebo před všechny příkazy `return`. Možnosti umístění vkládaných funkcí jsou znázorněné na obrázku 4.1. Na základě zkoumání již existujících nástrojů lze usoudit, že většina z nich dává přednost instrumentaci uvnitř instrumentované funkce (viz tabulka 3.1).



Obrázek 4.1: Znázornění dvou možností instrumentace funkce. Ve vnější instrumentaci se vkládaná funkce v instrumentaci umísťuje před a za voláním funkce. V druhém případě, u vnitřní instrumentace, se funkce umísťuje na začátek instrumentované funkce a před příkaz `return`. Tečky v obrázku reprezentují libovolný kód.

Každý z těchto dvou způsobů instrumentace funkce má své klady a zápory. Vnější instrumentace dokáže čerpat z kontextu, ve kterém je zasazena (informace o prostředí) a také má k dispozici nepozměněný zásobník programu (nebyla ještě zavolána funkce, nebo už byla uklizena ze zásobníku). Nevýhodou je vkládání instrumentační funkce ke každé instanci instrumentovaného volání funkce.

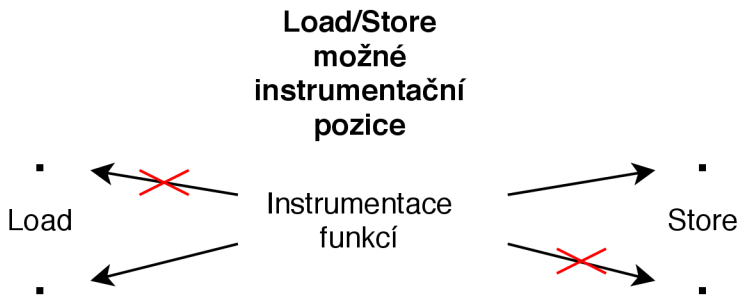
Naopak u instrumentace probíhající uvnitř definice instrumentované funkce stačí vložit pouze jedno volání této funkce. Je to způsobeno tím, že definice funkce je definovaná jen na jednom místě. Další výhodou vnitřní instrumentace je schopnost zachytit volání funkce z jiného modulu, který nebyl instrumentován v rámci tohoto nástroje (popř. modulu třetí strany). Nevýhodou tohoto způsobu instrumentace je nutnost vkládat instrumentační funkci před každý příkaz `return`. Je to způsobené tím, že každý příkaz `return` provede skok na konec funkce a tím pádem je posledním příkazem ve funkci. Taktéž nevýhodou je pozměněný zásobník funkce a nedostupnost informací vně definice funkce.

Pozice instrumentace přístupu do paměti Pro přístup do paměti jsou v procházeném kódu LLVM IR hlavní dvě instrukce zajišťující čtení a zápis do paměti. Jedná se o instrukce `load` a `store`. Pro instrumentaci těchto dvou funkcí (před a za) existují v kombinaci možností čtyři místa, kde by se daly vložit instrumentační funkce.

Dvě z těchto možností jsou kvůli absenci některých informací v nich oslabeny ve své použitelnosti. První možnost vložit funkci před operaci `load`, která postrádá některé informace, je před nahráním do paměti. V této chvíli nejsou vkládané funkci k dispozici žádné informace o instrumentované části paměti (kromě adresy).

Druhá z možností umísťuje vkládanou funkci za operaci `store`. Jsou zde k dispozici všechny informace o proměnné, ale její volání je až po provedení samotné operace, takže neumožňuje kontrolu zápisu.

Po vyloučení těchto dvou variant zbývají dvě pozice, a to za operací `load` a před operací `store`. Tyto možnosti byly vybrány v nástroji proto, že poskytují informace, které neposkytovaly vyloučené varianty. Pro operaci `load` přístup ke všem informacím o načtené proměnné a u operace `store` možnost kontroly před zápisem do proměnné. Obrázek zvažovaných možností instrumentace je zobrazen na obrázku 4.2.



Obrázek 4.2: Schéma znázorňuje umístění funkce vkládané v instrumentaci k operacím `load` a `store`. Jsou zde znázorněny čtyři možné pozice umístění funkce. Dvě z nich (přeškrnuté šipky) neposkytují některé informace. První, vkládající funkci před operaci `load`, postrádá přístup k informacím o načítané proměnné. Druhá možnost umísťuje funkci za operaci `store`, kde jsou k dispozici všechny informace, ale neumožňuje kontroly před zápisem. Po vyloučení těchto dvou možností, zbývají dvě pozice, a to za operací `load` a před operací `store`, které jsou v nástroji využity.

Nepřímá adresace u přístupu do paměti. Při přístupu do paměti, kde nevíme, jestli se nám v programu objeví nepřímá instrumentace, nemůžeme během překladu zjistit, jestli některá proměnná není ukazatelem na hlídanou proměnnou. Pro přístup do paměti nepřímou adresací existují následující přístupy, jak se s touto adresací vypořádat:

1. Zakázat nepřímou adresaci uživateli.
2. Instrumentovat všechny přístupy do paměti.
3. Pomocí statické analýzy zjistit množinu přístupů do paměti, u kterých není jisté, zda nemodifikují hlídané proměnné. Poté je instrumentovaná pouze tato množina.
4. Instrumentovat všechny přístupy do paměti kromě přímých přístupů.

Ze zmíněných možností byla zvolena možnost instrumentovat všechny přístupy do paměti, které nejsou přímými přístupy. Tento způsob byl zvolen za účelem omezení zbytečného vkládání funkcí před již známé přístupy (oproti instrumentaci všech přístupů do paměti). Statická analýza nebyla použita kvůli přílišnému zvýšení komplexnosti vyvíjeného instrumentačního frameworku bez výrazného snížení režie instrumentovaného programu.

Pro výrazné snížení režie instrumentovaného programu je výhodné umožnit uživateli zakázat nepřímou adresaci (uživatel ví, že v programu není nepřímá adresace, kterou by chtěl sledovat).

Kapitola 5

Návrh instrumentačního frameworku

V kapitole je popsán návrh vytvořeného frameworku pro instrumentaci. Framework je pojmenován Tforc jako zkratka z Test *Framework for Instrumentation during Compilation*. Celý instrumentační framework je složen ze dvou základních částí. První částí je průchod, kde je realizovaná samotná instrumentace. Druhá část zajišťuje správu paměti. Tato část je přidána ke vznikajícímu programu během fáze sestavování programu a poskytuje v rámci běhu instrumentovaného programu informace o použitých proměnných.

Kapitola začíná podkapitolou 5.1 navrhující komunikaci mezi programy a soubory zasahující z vnějšího prostředí do procesu instrumentace. Další podkapitola 5.2 je věnovaná vnitřní struktuře frameworku, kooperaci jednotlivých jejich částí a využití souborů v něm. V podkapitole 5.3 je ukázaná struktura a možnosti konfigurace frameworku Tforc. Podkapitola 5.4 rozebírá dekoraci jmen a návrh formátu specifikačního jazyku pro definování instrumentovaných artefaktů. Tento jazyk slouží pro popis sledovaných funkcí a přístupů do paměti, a také dovoluje využití komentářů. V poslední podkapitole 5.5 je ukázáno možné využití navrženého nástroje. Na jednom příkladu je zde ukázáno, jaké zásahy uživatele frameworku jsou nutné vykonat pro provedení instrumentace.

5.1 Struktura frameworku a kooperace programů

V rámci této podkapitoly je popsáno, které jednotlivé programy či soubory z vnějšího okolí zasahují (a jakým způsobem) do běhu instrumentačního frameworku.

Hlavním programem, který zasahuje do instrumentačního frameworku, je překladač LLVM. Tento nástroj umožňuje frameworku provádět samotnou instrumentaci — v rámci zásuvného modulu průchodu — formou sekvenčního průchodu jednotlivých instrukcí. Překladač přijímá zdrojové kódy překládaného programu a v rámci svého zpracování umožní zapojit vlastní průchod.

V průchodu, který je částí frameworku, se využívá **soubor se specifikacemi** (viz podkapitola 5.4). Soubor definuje, které funkce/proměnné mají být v průchodu instrumentované. Informace, ze kterého souboru má průchod čerpat specifikace, je udaná v **souboru konfigurace** (viz podkapitola 5.3).

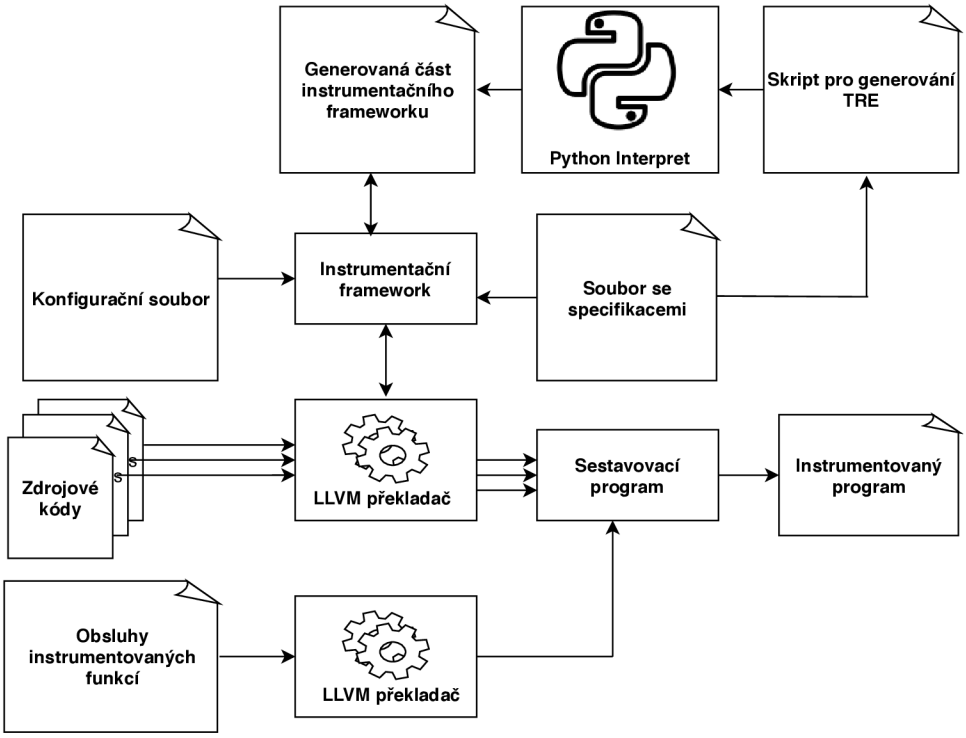
Díl části instrumentačního frameworku pro správu paměti je vytvářen až během překladu programu. Díl je vytvořen pomocí interpretace (interpretem `Python`) `skriptu pro generování TRE (Tforc Runtime Engine)`. Skript ze specifikací v rámci **souboru se spe-**

cifikacemi vygeneruje díl obsahující obsluhu funkcí pro instrumentování nepřímých přístupů do paměti.

Po překladu a instrumentaci zdrojových kódů je instrumentovaný kód v sestavovacím programu složený z:

- části starající se o správu paměti ve frameworku,
- části s definicemi obsluh instrumentovaných funkcí
- a samotného instrumentovaného kódu.

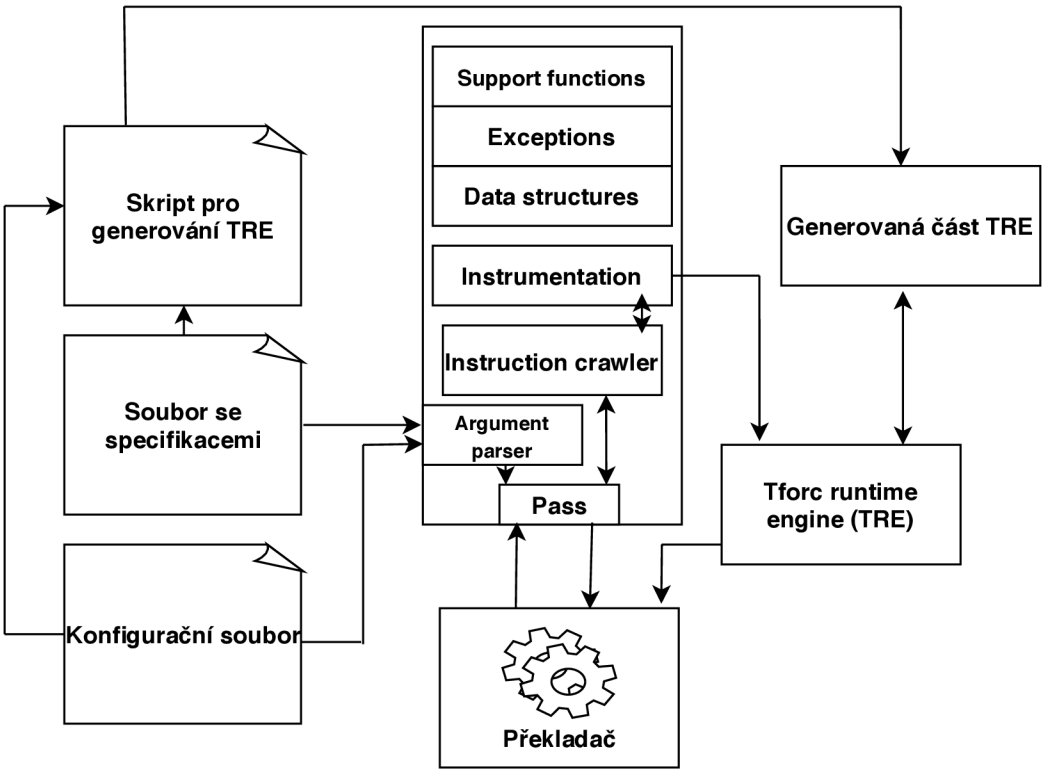
Výstupem sestavovacího programu je instrumentovaný spustitelný program. Znázornění celého propojení nástrojů je zobrazeno na obrázku 5.1.



Obrázek 5.1: Schéma zobrazuje kooperaci mezi jednotlivými soubory a nástroji v rámci provádění instrumentace souboru. Zdrojové kódy vstoupí instrumentací do překladače LLVM, kde v rámci průchodů v překladači se dostanou mimo jiné též na přidaný průchod Tforc (Instrumentační framework). V tomto průchodu jsou artefakty, které jsou zadané v souboru se specifikacemi, instrumentovány. Cesta k souboru se specifikacemi je zadána v konfiguračním souboru. Ze souboru se specifikacemi se pomocí skriptu pro generování TRE generuje díl části frameworku. Po instrumentaci souboru v sestavovacím programu se spojí instrumentovaný kód se samostatně přeloženými obsluhami instrumetovaných funkcí, a tak vznikne instrumentovaný program.

5.2 Struktura průchodu

V této podkapitole je navrženo rozložení jednotlivých částí frameworku, využívaných souborů a komunikace mezi nimi navzájem. Jednotlivá komunikace a umístění jednotlivých modulů je znázorněno na obrázku 5.2.



Obrázek 5.2: Schéma zobrazuje vnitřní návrh instrumentačního frameworku. Dělí se na části v rámci průchodu, které zpracovávají vstupní soubory s konfigurací (`Parser configuration`, dále dvojici částí, jež se starají o projití všech instrukcí (`Instruction crawler`) a jejich instrumentaci (`Instrumentation`). Poslední skupinou jsou moduly, jež poskytují jmenovaným částem společné datové struktury, konstanty a funkce. Kromě samotného průchodu je součástí frameworku též `TRE` — složený z pevné a generované části — jenž se stará o správu paměti a obsluhu nepřímé adresace.

V rámci překladu prochází překladač fází optimalizací (optimalizační průchody), v niž lze pomocí zásuvného modulu vložit vlastní průchod instrukcemi `Pass`. Zásuvný modul je složený z více částí (tříd), jež se starají o jednotlivou funkcionalitu instrumentačního frameworku.

První část průchodu **`Argument parser`** načte konfiguraci (viz 5.3) z **konfiguračního souboru** a poté zpracuje specifikace (viz 5.4) ze **souboru se specifikacemi** do vnitřní reprezentace. Tato vnitřní reprezentace slouží pro další části průchodu jako zdroj informací o sledovaných artefaktech.

Tyto získané informace, ve vnitřní reprezentaci, jsou poprvé využity v části **`Instruction crawler`**, která má na starosti průchod všemi instrukcemi v rámci všech funkcí. Během průchodu jednotlivých funkcí se sleduje, zda se mezi instrukcemi vyskytují artefakty, které si

přeje uživatel sledovat. Pokud je takový prvek nalezen, je předán části `Instrumentation`. Část `Instruction crawler` kromě průchodu funkcí s instrukcemi též prochází a získává informace o globálních proměnných, které mohou být taktéž sledovanými artefakty.

Část **`Instrumentation`** zabezpečuje vkládání volání instrumentačních funkcí, které jsou určena částí `Instruction crawler`, na požadovaná místa. Pro instrumentaci jsou důležité argumenty, které se vkládají do vytvářených funkcí. Informace o těchto argumentech se čerpají z vnitřní reprezentace získané z části `Argument parser`.

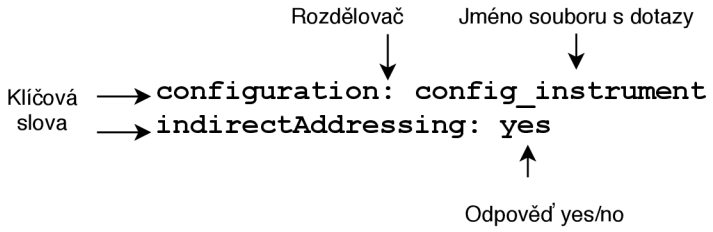
Výše zmíněné části průchodu využívají skupinu částí, jež jim umožňuje využívat funkcionalitu a rozhraní, která jsou pro ně společná. V první části **`Exceptions`** jsou definovány vnitřní výjimky. Další část **`Support functions`** poskytuje pomocné funkce využívané napříč všemi částmi. V poslední části **`Data structures`** jsou poskytnuté datové struktury, konstanty a definice datových struktur používaných v průchodu.

Kromě samotného průchodu se instrumentační framework skládá též z části TRE (Tforc runtime engine). TRE se v sestavovacím programu připojí k instrumentovanému programu. Tato část se stará v rámci běhu programu o správu proměnných a informací o nich. Správa paměti je složena z pevné části, jež má na starosti obsluhu instrumentovaných funkcí v rámci průchodu (rozhraní s průchodem) **`Tforc runtime engine`** a variabilní části **`Generated TRE`**, jež je generována skriptem **`TRE maker`**. Variabilní část slouží pro správu nepřímé adresace generováním funkcí.

5.3 Konfigurační soubor Tforc

Tato podkapitola popisuje formát konfiguračního souboru frameworku Tforc. V souboru jsou jednotlivé konfigurace uváděny za klíčovými slovy, za nimiž následuje oddělovač ve formě dvojtečky. V konfiguračním souboru záleží na pořadí konfiguračních záznamů. V tomto souboru lze v případě rozšiřování frameworku přidat další možné konfigurace, a tak skrze něj přenést informace do průchodu. Na prvním řádku — za klíčovým slovem **`configuration`** — jsou obsaženy informace o cestě k souboru se specifikacemi určujícími, které prvky mají být instrumentovány. Na druhém řádku — za klíčovým slovem `inderectAddressing` — je udané, jestli jsou přístupy do paměti s nepřímým adresováním povoleny odpovědí **yes/no**. Příklad konfigurace je znázorněn na obrázku 5.3.

Zapnutí instrumentace přístupu do paměti s nepřímou adresací může mít za následek mnohonásobné zvýšení počtu vkládaných funkcí. Tímto se může výrazně ovlivnit rychlost instrumentovaného programu. Při zvolené přímé adresaci nejsou přístupy do paměti pomocí nepřímé adresace instrumentovány, a tím může být program rychlejší.



Obrázek 5.3: Znázornění příkladu konfigurace. Vždy jsou zde dva řádky, kdy první z nich obsahuje cestu k souboru se specifikacemi k instrumentaci. Na druhém řádku je možnost volby, zda-li instrumentovat i přístupy do paměti přes nepřímou adresaci.

5.4 Specifikační jazyk Tforc

V této podkapitole se pojednává o návrhu jazyka definujícího, které funkce a které přístupy do paměti se budou instrumentovat. Specifikační jazyk Tforc je dán gramatikou v příloze A. V instrumentačním frameworku Tforc je tento jazyk využit v **souboru se specifikacemi**.

V oddíle 5.4.1 je popsáno, jak vypadají, a k čemu se využívají, dekorovaná jména funkcí. Oddíl 5.4.2 představuje možnosti použití komentářů v specifikačním jazyce.

Oddíl 5.4.3 zobrazuje způsob, jakým jsou instrumentovány funkce a jakým způsobem je možné instrumentovat všechny funkce pomocí žolíka. Poslední oddíl 5.4.4 popisuje možnosti instrumentace proměnných a vyjmenovává možné argumenty, které může uživatel využít při instrumentaci.

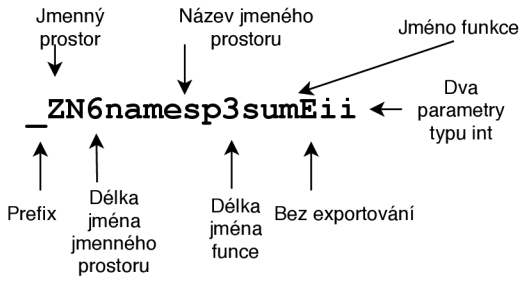
5.4.1 Dekorování jména funkce

Tento oddíl nastiňuje způsob kódování jmen pomocí dekorace funkce (v angličtině *name mangling*), která sama o sobě je komplexní oblastí. Pro potřeby instrumentace bude ale pouze zapotřebí znát část základních principů. Více o dekoraci funkce je uvedeno v kapitole knihy [26].

Dekorace jmen se využívá ve specifikacích specifikačního jazyka Tforc. Tento tvar jména umožňuje konkretizovat danou funkci a předat informace o jejím jmenném prostoru a jejich argumentech. Dekorované jméno je možné získat z přeloženého kódu do LLVM IR nebo taktéž z tabulky symbolů objektového souboru příkazem `nm`.

Dekorování jmen u funkcí se věnuje zachycení celého předpisu funkce (jmenný prostor, název a parametry) do jednoho identifikátoru. Tato operace je reverzibilní a díky tomu jde i zpětně rekonstruovat původní předpis (viz nástroje llvm-cxxfilt [8] a online nástroj GCC and MSVC C++ Demangler [9]).

Pro vysvětlení většiny principů potřebných k používání dekorování jmen funkcí stačí vysvětlení na ilustračním obrázku 5.4.



Obrázek 5.4: Znázornění a popsání jednotlivých částí dekorované funkce. V tomto tvaru jsou zachyceny důležité informace o funkci `sum`. V úvodu této funkce je část, která udává, že je funkce v jmenném prostoru `namesp`. Na konci výrazu jsou vyznačeny parametry, které funkce přijímá. Konkrétně u této funkce jsou to dvě hodnoty typu `int`.

Jména jednotlivých podporovaných typů a jejich tvar po provedení dekorování jsou umístěny v tabulce podporovaných typů 6.3. Při manuální tvorbě sufixů funkcí obsahujících zkratky parametrů funkce může kvůli chybě zadávání vzniknout problém při překladu. Při vícenásobném využití zkrácení typu `char *` je potřeba od druhého použití využívat zkratku `S_` místo `Pc`.

5.4.2 Komentáře a mezery ve specifikačním jazyce

Tento oddíl pojednává o významu bílých znaků a možnostech komentářů v Specifikačním jazyku Tforc.

Ve specifikačním jazyce je libovolné množství bílých znaků bráno jako jedna mezera, proto využívání těchto znaků neovlivňuje sémantiku specifikací. Volné řádky mezi jednotlivými specifikacemi jsou ignorovány. Mezery se ve specifikačním jazyku využívají na oddělení jednotlivých slov, kde není použit jiný oddělovač jako např. ":", ","nebo závorky "[]".

Jazyk též umožňuje používat jednořádkové komentáře. Tyto komentáře jsou uvozeny znakem "#", od kterého se ignoruje text umístěný napravo od něho. Možnosti využití komentářů jsou znázorněné ve výpisu 5.1.

```
# Instrumenting functions
#func: beforeCountDown before _Z9countDownii [1], [number]
func: afterCountDown after _Z9countDownii [0,1] # comment
```

Výpis 5.1: Znázornění využití komentářů ve specifikačním jazyku. Díky komentáři můžeme ignorovat řádky s textem, jak je vidět na prvních dvou řádcích. Nebo může být ignorován jen popisek za platnou specifikací.

5.4.3 Instrumentace funkce

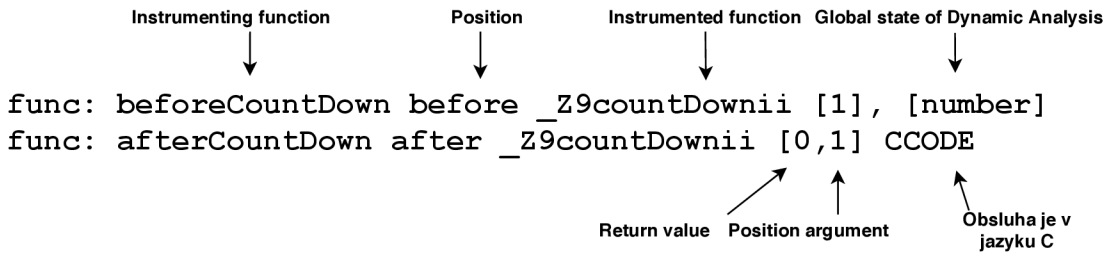
Oddíl představuje první možnost instrumentace, a to funkce dané dekorovaným názvem pomocí specifikačního jazyku Tforc.

Specifikace je uvozena klíčovým slovem `func` a oddělovačem ":". Tento prefix určuje, že se bude jednat o instrumentaci funkce. Po tomto prefixu následuje název vkládané funkce. Tato funkce je zadána prostým názvem funkce (což ulehčuje uživatelskou specifikaci) — je možné sestrojit dekorované jméno během překladu.

Za názvem vkládané funkce následuje určení, kde bude funkce umístěna, jestli před či za instrumentovanou funkci. Toto umístění může nabývat hodnot `before` pro vložení funkce před instrumentovanou funkci, nebo `after` pro vložení funkce za instrumentovanou funkci. Klíčová slova a název pozic umístění funkcí nejsou závislé na velkých a malých písmenech.

V poslední části specifikace se pomocí jména určuje, jaká funkce se bude instrumentovat. Název této funkce je dekorovaný. Za instrumentovanou funkcí jsou zadány poziční argumenty ve dvojici hranatých závorek, které udávají argumenty předávané z instrumentované funkce do vkládané. Zvláštní význam zde má argument s indexem 0, který označuje předání návratové hodnoty. Volitelně může být přidána též druhá dvojice hranatých závorek, jež udává předávané globální stavy dynamické analýzy (globální proměnné). Do přídavných argumentů lze vložit také klíčové slovo `LOCATION`, jež vytvoří výpis o instrumentované funkci. Na obrázku 5.5 jsou znázorněny ukázky specifikací instrumentace funkcí a popsány jejich části.

Za samotnou specifikací se může nacházet klíčové slovo **CCODE**, které dodává frameworku informaci o tom, že funkce vkládaná během instrumentace je napsaná v jazyku C.



Obrázek 5.5: Znázorňuje vysvětlení specifikace instrumentace funkcí. Jsou zde instrumentovány dvě funkce — první před voláním a druhá po volání funkce `countDown` (s dekorovaným jménem). V prvních hranatých závorkách jsou pozice argumentů, které mají být do vkládaných funkcí předány. Druhá dvojice závorek dovoluje předávat do vkládané funkce globální stavy dynamické analýzy. Tato poslední dvojice může být v zápisu vynechána. Za specifikací může být doplněné klíčové slovo **CCODE**, které dodává frameworku informaci o tom, že funkce vkládaná během instrumentace je napsaná v jazyku C.

Divoká karta (*wildcard*) kromě instrumentace jednotlivých funkcí lze místo jména instrumentované funkce zadat žolíka `"*"`, který zajistí instrumentaci všech volání uživatelem definovaných funkcí. Je možné také použít druhý typ žolíka `"**"`, který způsobí instrumentaci všech volání jak uživatelem definovaných funkcí, tak funkcí systémových a knihovních. Kvůli možnosti instrumentování vícevláknových programů jsou některé funkce starající se o zamykání a obsluhu vláken z instrumentace vyňaty (viz oddíl 7.3.3).

Pro instrumentující funkci jsou frameworkem pevně dané parametry, které musí obsahovat. Tyto parametry umožní předat některé dodatečné informace vkládané funkci o instrumentované funkci. Informace jsou složeny ze jména funkce, řádku, na kterém je volána funkce, a názvu (cesty) souboru. Pro vytvoření funkce je nutné mít typy parametrů `char*, int, char*` v dekorovaném tvaru `"PciS_"`.

5.4.4 Instrumentace proměnné

V tomto oddíle je uvedena další z možností instrumentace jazyku frameworku Tforc — přístupy do paměti, konkrétně operace `load` a `store`.

Specifikace na instrumentaci přístupu do paměti začíná klíčovým slovem `load` nebo `store`, které určuje, jestli se budou instrumentovat operace čtení z paměti nebo zápisu do ní. Pokud jsou potřeba instrumentovat obě operace, musí se pro každou z nich napsat vlastní specifikace. Za tímto klíčovým slovem následuje oddělovač dvojtečky.

Proměnná, která se bude při vybrané operaci hlídat, je určena výrazem, který může nést název globální proměnné, nebo může být složen z dvojice názvu dekorované funkce (`linkage`) a ze jména lokální či globální proměnné. Tato dvojice je od sebe oddělena znakem dvojtečky.

Po výrazu s proměnnou je umístěno dekorované jméno funkce, která bude vkládána k instrumentovanému přístupu do paměti. Do této vkládané funkce lze přenést informace o proměnné pomocí předem definovaných klíčových slov argumentů, umístěných v hranatých závorkách. Klíčová slova pro definici argumentů jsou následující:

- **`ADDRESS`** — adresa, na které je uložena proměnná.
- **`VALUE`** — hodnota přečtené/zapsané proměnné.

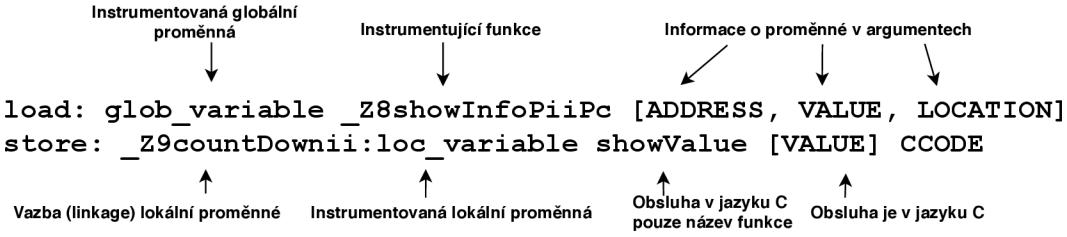
- `LOCATION` — řetězec s informacemi o pozici operace ve zdrojovým kódu (řádek, funkce a soubor).

Tato klíčová slova jasně určují, které parametry může mít vložená funkce. Argumenty funkce musí být seřazeny ve stejném pořadí jako jsou tato klíčová slova zadána ve specifikaci při instrumentaci. Pokud není použit žádný argument ve vkládané funkci, pak se v tomto případě dává explicitní typ `void` (v dekoraci pod zkratkou `v`). Argumenty mají pevně daný typ (v jazyku C) a to:

- `ADDRESS` — int * (dekorace `Pi`).
- `VALUE` — podle typu, který nabývá zkoumaná proměnná. U ukazatelů je to jednotně int *.
- `LOCATION` — char * (dekorace `Pc`).

Za samotnou specifikací se může nacházet klíčové slovo **CCODE**, které dodává frameworku informaci o tom, že funkce vkládaná během instrumentace je napsaná v jazyku C. Při využití tohoto klíčového slova je nutné zadat vkládanou funkci bez dekorování jména.

Příklad instrumentace proměnné s popisem jednotlivých částí a ukázka výše popisovaných argumentů je znázorněn na obrázku 5.6.

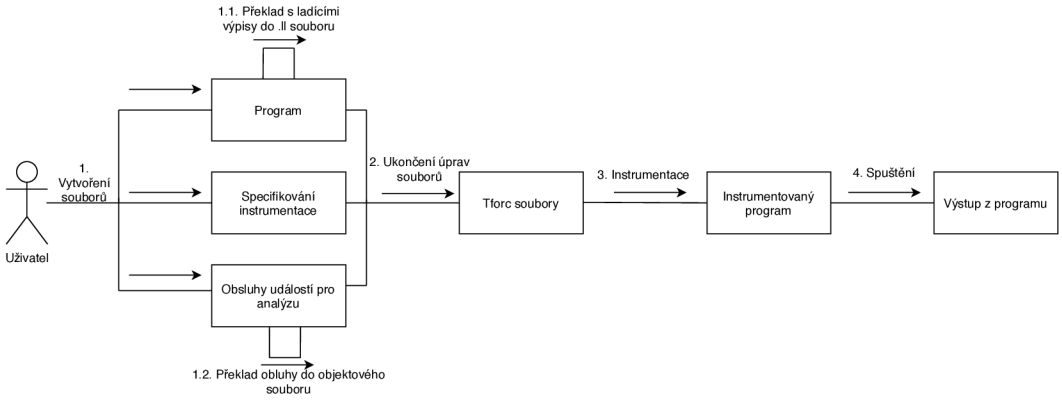


Obrázek 5.6: Vysvětluje specifikaci pro instrumentaci přístupů do paměti. Existují zde dva typy operací dané klíčovými slovy (pro čtení a zápis), za nimiž následuje oddělovač ":". Za ním je umístěna sledovaná proměnná daná buďto názvem globální proměnné, nebo vazba hledané proměnné s jejím názvem — oddělené dvojtečkou. Po ní následuje dekorované jméno vkládané funkce. Za názvem vkládané funkce následuje dvojice hranatých závorek, jež v sobě obsahuje klíčová slova. Slova definují, jaké informace se předají do vkládané funkce. Za samotnou specifikací může být doplněné klíčové slovo **CCODE**, které dodává frameworku informaci o tom, že funkce vkládaná během instrumentace je napsaná v jazyku C.

Nepřímá adresace se řeší přes volání funkce v Tforc frameworku, které zjistí za běhu programu, jestli daný ukazatel ukazuje na hlídanou proměnnou. Návrh nepřímé adresace je více rozveden v oddílu 5.3.

5.5 Způsob použití frameworku

Tato podkapitola se věnuje návrhu, jak používat framework Tforc, jak by se měl framework chovat a jaké jsou potřebné zásahy ze strany uživatele (ukázky kódu). Návrh zobrazuje též snadnost obsluhy. Na diagramu kolaborace na obrázku 5.7 lze vidět procesy, které uživatel musí využít, aby dostal výstup z programu.



Obrázek 5.7: Diagram kolaborace, zahrnující i uživatele, znázorňuje posloupnost úkolů, které musí uživatel vykonat, aby dostal výstup z instrumentovaného programu. Prvním vstupním procesem je vytvoření třech souborů, které jsou potřeba na vstupu frameworku, a to konkrétně samotného programu, specifikace instrumentace a obslužných událostí pro analýzu. Po tomto procesu je možnost provést částečný překlad programu do `.ll` souboru nebo překlad obslužných souborů do objektového souboru. Druhým povinným procesem je ukončení úprav souborů, čímž vznikají připravené soubory pro spuštění instrumentace (Tforc soubory). Třetím procesem se provede sama instrumentace a vznikne instrumentovaný program. Tento program se ve čtvrtém procesu spustí se svými parametry a vytvoří výstup z programu.

Jednotlivé procesy prováděné uživatelem jsou:

1. Prvním procesem je **vytvoření souborů**. Je v něm nutné vytvořit tři soubory (program, specifikaci instrumentace a obsluhy události pro analýzu) před samotnou instrumentací.

   Program. Uživatel dodá zdrojový kód, u kterého chce instrumentovat funkce či přístupy do paměti. Soubor je napsán v jazyku C/C++. Příklad zdrojového kódu je ilustrován výpisem 5.2.

   Uživatel má též možnost dodat zdrojový kód, který již byl přeložen do mezikódu LLVM IR (`.ll` soubor). Díky této možnosti lze instrumentovat i komplexnější programy pomocí nástroje `llvm-link`, který umožňuje sestavení více souborů `.ll` do jednoho. Pro využití této možnosti je nutné provést všechny překlady souborů s ladícími informacemi (přepínačem -g). Více o této možnosti je napsáno v odstavci 7.2.

```
#include <iostream>
int main(){
    int a~= 5;
    int b = 3;
    int c = a~+ b;
    std::cout << "result: " << c << std::endl;
    return 0;
}
```

Výpis 5.2: Zdrojový kód programu pro sečtení dvou konstant a jejich vytisknutí.

Specifikace instrumentace V této části je uživatelem zadán ve specifikačním jazyku (viz podkapitola 5.4) požadavek na instrumentaci dané funkce nebo proměnné. Příklad zadaného požadavku je zobrazen ve výpisu 5.3.

```
load: main:b _Z9printInfoiPiPc [VALUE, ADDRESS, LOCATION]
store: main:b _Z9printInfoiPiPc [VALUE, ADDRESS, LOCATION]
```

Výpis 5.3: Ve specifikačním souboru je zadáno sledovaní proměnné `b` ve funkci `main`. Po čtení/zápisu se vloží za/před něj instrumentační funkce `_Z9printInfoiPiPc`.

Obsluhy událostí pro analýzu Pro instrumentaci je nutné definovat funkce, jejichž volání se budou vkládat na místa definovaná ve specifikacích. Pokud není některá funkce definována, tak při sestavování programu překladač selže. Příklad definice takové funkce je znázorněn na výpisu 5.4.

```
void printInfo(int value, int *address, char * location){
    std::cout << location << " s~hodnotou " << value
    << " na adrese " << address << "." << std::endl;
}
```

Výpis 5.4: Ukázka kódu znázorňuje definici funkce, která vypíše informace o instrumentované proměnné.

1.1. Uživatel má též možnost udělat část překladu do tvaru souboru `.ll`. Musí však pro správnou funkčnost instrumentačního frameworku použít přepínač `"-g"`. Je zde též možnost nahrát rovnou soubor `.ll`, který byl získán překladačem z jiného jazyku než C/C++.

1.2. Přeložení funkcí může uživatel nechat na programu `Make`, nebo dodat předem přeložený objektový soubor s funkcemi pro instrumentaci.

2. **Ukončí se úpravy všech souborů.** Soubory jsou v konečném tvaru a jsou připraveny pro spuštění další fáze instrumentace.

3. Třetím procesem je **instrumentace**, jež se spouští programem `Make` se správně vyplněnými proměnnými prostředí udávající cesty k jednotlivým souborům potřebným k instrumentaci. Jeden z příkladů takového zadání je ve výpisu 5.5.

```
SOURCE_FILE=file.cpp CONFFILE=tforc_config.conf USRFUNC=inst_func.cpp
BIN=instrumented_program make
```

Výpis 5.5: Znázornění nastavení proměnných prostředí pro spuštění programu `Make`, který provede instrumentaci souboru `file.cpp`.

4. Po dokončení instrumentace stačí **spustit** instrumentovaný program a dostaneme výstup z programu. Spuštění programu je znázorněné ve výpisu 5.6.

```
./instrumented_program <arg1> <arg2> <arg3>
```

Výpis 5.6: Spuštění instrumentovaného programu se vstupními argumenty.

Kapitola 6

Implementace instrumentačního frameworku

Tato kapitola pojednává o implementacích stěžejních částí programu zajišťujících jeho klíčovou funkcionalitu. Jednotlivým částem je věnována vždy celá jedna podkapitola. Instrumentační framework Tforc je implementován v jazyku C++. Pro některé nezávislé dílčí části vypomáhají se zpracováním a generováním skripty v jazycích Python a Bash.

Podkapitola 6.1 udává, jak je ve frameworku realizované ukládání záznamů o instrumentovaných prvcích. Podkapitola 6.2 se zabývá implementací průchodu přes program po jednotlivých instrukcích a filtrováním instrukcí, které mají být instrumentované. Podkapitola 6.3 ukazuje, jakým způsobem jsou vytvářena volání vkládaných funkcí. V podkapitole 6.4 je ukázán způsob komunikace mezi LLVM průchodem a části TRE, jež běží za běhu programu. Podkapitola 6.5 obsahuje vysvětlení principu správy adres ve frameworku Tforc. Poslední podkapitola 6.6 shrnuje implementační závislosti a omezení implementace.

Implementace je umístěna na přenosném médiu, jehož adresářový strom je znázorněn v příloze E. Pro bližší popsání jednotlivých funkcí je k dispozici dokumentace vytvořená programem `Doxygen`. Tuto dokumentaci lze vygenerovat po spuštění programu `Make` s argumentem `documentation`. Celá implementace je k dispozici na `Gitlab` stránce výzkumné skupiny `Testos` pod názvem `Tforc` [64].

6.1 Správa záznamů pro instrumentaci

Tato podkapitola popisuje, jakým způsobem jsou zpracovány uživatelské specifikace (viz podkapitola 5.4) do záznamů. Zpracování specifikací probíhá podle gramatiky v příloze A. Dále pojednává o tom, jak jsou tyto záznamy uchovávány a jak se k nim přistupuje z ostatních částí frameworku.

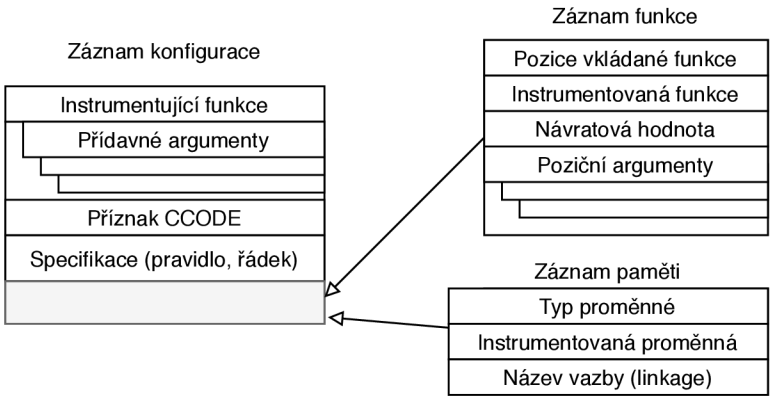
Zpracování specifikací probíhá na začátku celého průchodu a má ho na starosti modul `ArgumentParser`. Při chybné syntaxi specifikace se vyvolá výjimka, která ukončí překlad a uživatele upozorní, kde udělal chybu.

Zpracování specifikací instrumentace funkce a přístupu do paměti se liší v syntaxi specifikace. Z tohoto důvodu je na začátku zpracování rozpoznáno — podle klíčového slova na první pozici — o jaký druh specifikace se jedná. Zpracování jednotlivých typů specifikací se liší svými metodami.

V obou typech specifikací se některé informace shodují, a proto je vytvořena specializace z jednoho obecného záznamu do dvou konkrétních záznamů. Díky tomu se mohou některé

metody použít pro oba typy záznamů. Jednotlivé položky záznamů a znázornění dědičnosti je zobrazeno na obrázku 6.1.

Pro uložení záznamů je vytvořena datová struktura typu `DataConfigStorage`, ve které jsou uloženy samostatně jak záznamy o instrumentaci funkcí a přístupu do paměti, tak informace, jestli je povolená nepřímá adresace (nahrána z konfiguračního souboru). Tato datová struktura se předává mezi objekty průchodu Tforc.



Obrázek 6.1: Schéma znázorňuje specializace jednotlivých druhů záznamů instrumentace. Obecný záznam konfigurace obsahuje shodující se položky záznamů (název instrumentující funkce, informace o pozici specifikace, příznak CCODE a vektor přídavných argumentů). Specializované specifikace se dělí na záznam funkce a záznam paměti. Záznam funkce obsahuje od obecného záznamu navíc pozici, kam se má vložit instrumentovaná funkce, název instrumentační funkce, návratovou hodnotu a vektor pozičních argumentů. Druhý typ — záznam paměti — obsahuje typ instrumentované proměnné, její jméno a u proměnných zkoumaných v určitém rozsahu též *linkage* — dekorované jméno funkce.

6.2 Průchod a filtrování instrukcí

Tato podkapitola pojednává o tom, jak probíhá průchod přes celý vstupní program instrukci po instrukci. Tento průchod kontroluje podle záznamů (viz podkapitola 6.1), která z procházených instrukcí má být instrumentovaná. Průchod taktéž zajišťuje odeslání zprávy frameworku při každé deklaraci proměnné a při povolené nepřímé adresaci rovněž vložení funkcí komunikujících se správou paměti (viz podkapitola 6.4).

Prvně, než začne samotný průchod instrukcemi, se provede zpracování deklarací globálních proměnných, které mají ladící informace. Výsledkem zpracování globálních proměnných je vložení funkcí (jejichž jména začínají na `__initDeclare`), jež deklarují globální proměnné před první instrukcí funkce `main`. Funkce zajišťují předání informací o globální proměnné frameworku. Ukázka kódu LLVM IR ve výpisu 6.1 znázorňuje vložení funkce na začátek funkce `main`.

Realizace průchodu všemi instrukcemi je zajištěna strukturou LLVM programu (viz obrázek 2.7), kde se sekvenčně prochází přes všechny funkce, ve kterých je možné iterovat přes všechny základní bloky (*Basic block*) a v nich přes jednotlivé instrukce. Hlavní funkcionalita této části se odehrává v metodě `processInstruction`, jež filtruje instrukce podle jejich typů, získává z nich konkrétní informace (využívá se dědičnosti typů z třídy `Instruction`) a rozpoznává, jestli se má daná funkce instrumentovat. Jednotlivé rozeznané

typy instrukcí lze pro ladění programu sledovat pomocí nastavení konstanty (přepínače) `EVENT_PRINTER` na nenulovou hodnotu. Přepínač je definován v modulu `DataStructures`. Povolené sledování rozeznaných typů zprostředkovává informace o instrukcích, které jsou zkoumány.

```
define dso_local i32 @main() #6 !dbg !947 {
  call void @_Z13__initDeclarePcPiS_S_ic(i8* getelementptr inbounds
  ([7 x i8], [7 x i8]* @0, i32 0, i32 0), i32* @glob_i, i8* null,
  i8* getelementptr inbounds
  ([19 x i8], [19 x i8]* @1, i32 0, i32 0), i32 0, i8 1)
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  ...
```

Výpis 6.1: Ukázka kódu znázorňuje vložení funkce `_Z13__initDeclarePcPiS_S_ic` na začátek funkce `main` pro předání informací o globální proměnné `glob_i` frameworku Tforc.

Pokud metoda `processInstruction` rozpozná nějaký přístup do paměti pomocí instrukcí pro četbu či zápis do paměti, tak prozkoumá, jestli se nejedná o proměnnou ve sledovaných záznamech. Pokud zjistí, že se jedná o proměnnou ze záznamů, předá řízení instrumentaci (viz podkapitola 6.3). Výjimkou je případ, kdy uživatel vyžaduje přístup do paměti i pomocí nepřímé adresace. Potom musí být instrumentovaná každá funkce, u které se nenajde záznam.

Lokální proměnná neobsahuje jméno proměnné, která je v ní uložena, a proto je nutné získat jméno jiným způsobem. K pozdějšímu získání jména slouží ukládání záznamů do vektoru `nameOfVariableMapToRegister`, jež mapují číslo registru na lokální proměnnou. Vektor se po každém skoku z funkce vynuluje, protože lokální proměnné mají platnost jen pro danou funkci. Sekvenční procházení funkcí není pro ukládání problém, protože se děje v každé funkci nezávisle na jiných.

Dalším z možných typů, které může metoda `processInstruction` rozpoznat, je volání funkce. První ze sledovaných funkcí je funkce vznikající při překladu souboru se zapnutými ladícími informacemi `llvm.dbg.declare`. Tato funkce se váže na každou deklaraci proměnné. Pro zachycení každé deklarace proměnné je `llvm.dbg.declare` instrumentována funkcí (s názvem začínajícím na `__initDeclare`), jež posílá informace o vznikající proměnné frameworku. Další sledované volání funkcí jsou zkoumané uživatelské funkce. Pokud jsou tyto funkce definované v záznamech, provede se instrumentace.

Instrumentace funkcí probíhá též tehdy, pokud je v záznamech přítomný žolík. Pokud se jedná o žolíka "*", tak se instrumentuje každá uživatelská funkce. V případě žolíka "**" jsou mimo uživatelské funkce instrumentované i knihovní/systémové funkce.

Poslední ze sledovaných typů je instrukce návratu z funkce `ret`. Tento typ instrukce je instrumentován pouze tehdy, pokud je povolena nepřímá adresace. O instrumentaci této instrukce pojednává blíže podkapitola 6.4.

6.3 Instrumentace funkcí a proměnných

Podkapitola se zabývá instrumentací funkcí a proměnných. Konkrétně se věnuje vytvoření volání funkcí, které se vkládají na místo instrumentace a jsou definované v záznamech.

Často využívanou třídou LLVM je v implementaci třída `Context`. Tato třída se stará o globální data modulu a LLVM infrastruktury zahrnující typy a konstanty.

Aby se během překladu nemusely stále získávat informace z třídy `Context` (jež poskytuje vždy celé objekty LLVM), byla pro vnitřní práci s typy vytvořena interní reprezentace. Implementace reprezentace je provedena pomocí výčtu podporovaných typů `InstructionType`.

Instrumentace funkcí probíhá ve všech typech stejným způsobem. Prvním krokem je vytvoření objektu typu `FunctionCallee`, jenž slouží jako typ pro volání funkce. K tomuto volání funkce je potřeba prvně definovat jeho parametry. Vložení typů do vektoru se děje pomocí funkcí (jejichž název začíná na `Initialize`), které vrátí typ funkce (`FunctionType`) v objektu. Potom se vektor argumentů pomocí funkcí (jejichž název začíná na `Assign`) naplní argumenty příslušné operace. Poté se naplní objekt `FunctionCallee` předpisem z objektu `FunctionType`, který definuje jakého typu mají být parametry a jeho návratová hodnota. Dále probíhá samotná instrumentace, jež vkládá toto volání funkce na zvolené místo mezi existující instrukce a přidává argumenty, které má vkládaná funkce mít.

Při instrumentaci pomocí funkce napsané v jazyce C (vytvořené pomocí `extern "C"`) — s pozitivním příznakem **CCODE** — je nutné vytváření vkládané funkce pozměnit. U výše zmíněného postupu není možné vkládat funkce, které nemají dekorované jméno. Kvůli tomu byl vytvořen postup pro vkládání volání funkcí se jmény bez dekorací. Postup v prvním kroku vytvoří pro zpracovávanou funkci prototyp. Funkce s již vytvořenými prototypy se ukládají do hashovací tabulky `nameOfCCodeMapToInstrumentedFunction`. Z této tabulky jsou při dalším použití přečteny, aby se nevytvářel prototyp podruhé. K získanému typu funkce jsou posléze připojeny argumenty a je vytvořena instrukce volání funkce, jež je vložena na zvolené místo.

Vkládané funkce se vkládají vždy hned před, nebo za funkci. Proto při vložení více funkcí je poslední vložený prvek umístěn těsně před/za instrumentovaným voláním funkce. Výpis 6.2 znázorňuje uspořádání více vložených funkcí před (a za) instrumentovanou funkci.

```
1. funkce pred
2. funkce pred
instrumentovana funkce
2. funkce po
1. funkce po
```

Výpis 6.2: Znázornění pořadí vkládání funkcí před a po instrumentované funkci. Nově vkládaná funkce je vždy co nejblíže instrumentované funkci.

Kvůli nemožnosti využití šablon — definice funkcí jsou známé až při sestavování programu — musí každá funkce obsluhující průchod být deklarovaná určitými typy. To má za následek situaci, kdy se musí pro každý typ zpracovávaný funkcemi vytvářet vlastní obslužná funkce. Pro rozumný rozsah množství funkci byly funkce redukovány pouze na typy bez ukazatelů a na ukazatel samotný, který generalizuje (zastřešuje) všechny ukazatele.

Pro předání řetězce v rámci kódu LLVM IR je nutné vytvořit v LLVM IR konstantu obsahující řetězec. Pro zamezení zbytečného opakování konstant řetězců se při vytváření řetězce kontroluje, zda-li už byl definován. Pokud už byl definován, tak místo vzniku nové konstanty se předá pouze odkaz na již existující konstantu. Informace o již použitých konstantách jsou v hashovací tabulce `mapOfGlobalString`.

6.4 Komunikace mezi průchodem a Tforc runtime engine

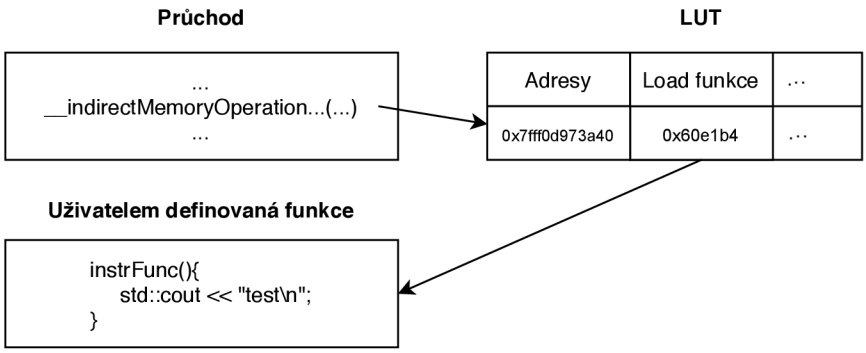
V této podkapitole je ukázána komunikace mezi průchodem a TRE (Tforc runtime engine). Komunikace probíhá skrze jednosměrné zasílání zpráv pomocí vkládání funkcí do instrumentovaného kódu.

Pro minimalizování možnosti záměny uživatelských funkcí s funkcemi komunikace frameworku jsou funkce užívané uvnitř frameworku umístěny ve jmenném prostoru `TRE`. Pro funkce používané komunikací mezi průchodem a frameworkem je přidán prefix ___, který standardně předznamenává interní funkce.

Prvním důvodem komunikace mezi průchodem a TRE je správa záznamů proměnných. Po každé deklaraci proměnné je vložené volání funkce (se jménem začínajícím na `__initDeclare`), která zajišťuje přenos informací o proměnné do běhu programu.

Druhým a hlavním důvodem komunikace mezi těmito částmi je nepřímá adresace proměnných. Pokud je tato možnost povolena v konfiguraci (viz podkapitola 5.3), tak jsou vkládány funkce zabezpečující zanořovaní úrovní datových struktur (správa zásobníku), správné uvolňování záznamů a zpracování operací s nepřímou adresací.

Při nepřímé adresaci, pokud není proměnná hlídána některou konkrétní specifikací, je vloženo volání podle typu proměnné ze skupiny funkcí `__indirectMemoryOperation`. Toto volání se zpracovává ve frameworku, kde dokáže určit, jestli daná proměnná odkazuje na některou hledanou proměnnou, nebo skrze jinou proměnnou ukazuje na hledanou proměnnou (viz obrázek 6.2).



Obrázek 6.2: Znázornění volání funkce nepřímé adresace. Tato funkce je podle odkazované adresy porovnána s některou položkou ze seznamu adres v LUT tabulce. Pří shodě adres je zavolána příslušná uživatelem definovaná funkce k instrumentaci hlídané proměnné.

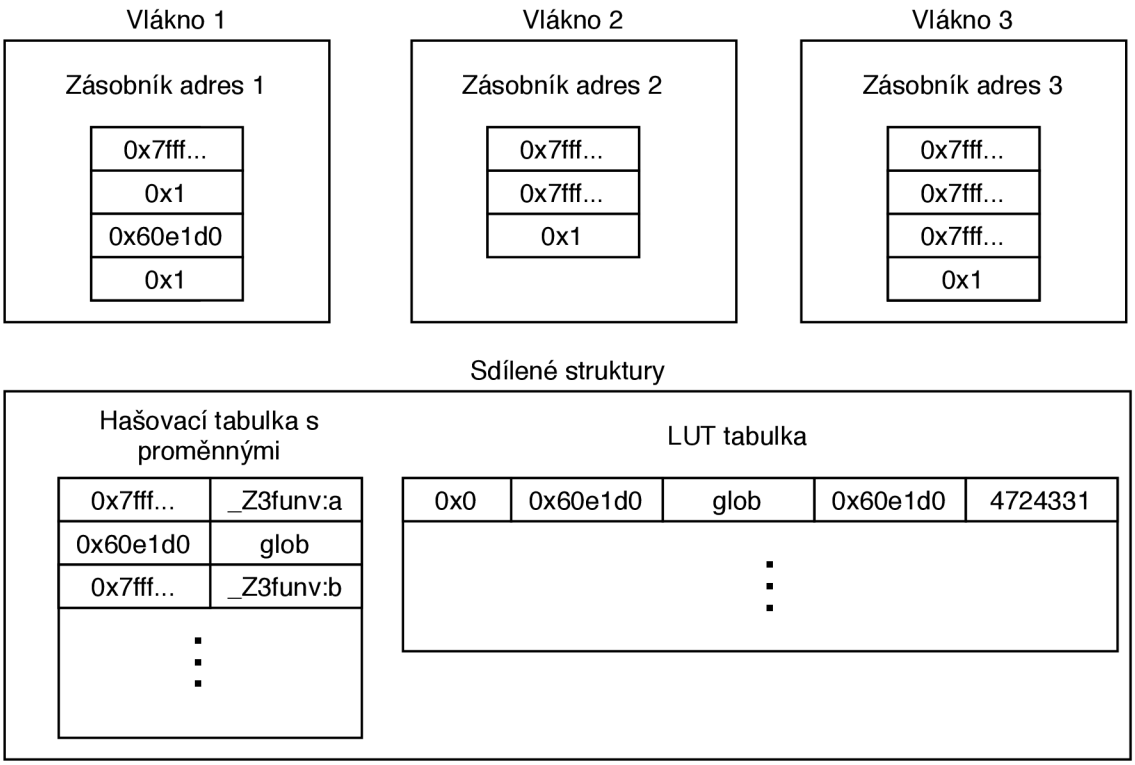
Kromě funkcí starajících se o nepřímou adresaci se do programu vkládají též funkce starající se o správu paměti. První z nich `__addStackToVariableAddressStack` přidává do zásobníku nový rámec funkce, který se stará o platnost proměnných. Tato funkce je volána vždy před uživatelskou funkcí, a tím pro ni vymezuje nový rámec proměnných. Druhá funkce se stará o smazání proměnných funkce v zásobníku proměnných. Funkce `__removeStackFromVariableAddressStack` je vkládaná před každou instrukci `ret`, která vždy ukončuje funkci. O správě paměti pojednává více podkapitola 6.5.

6.5 Správa adres v Tforc

Podkapitola pojednává o správě adres, jejich uložení a práci v TRE se spravovanými adresami za běhu programu. Je zde popsána jak správa paměti u přímé adresace, tak hlavně u nepřímé adresace.

Framework Tforc používá pro uložení dat o proměnných a adresách tři typy struktur, znázorněné na obrázku 6.1, které jsou blíže představeny v oddílech níže. První z nich, hašovací tabulka proměnných (viz oddíl 6.5.1), je sdílená napříč vlákny a ukládá k jednotlivé adrese informace o proměnné, která na ní leží. Druhou strukturou je zásobník adres (viz oddíl 6.5.2), který uchovává aktuálně deklarované proměnné ve vláknu. Každé vlákno má vlastní instanci této struktury (skrze kvalifikátor thread local). Poslední strukturou je LUT tabulka (viz oddíl 6.5.3), starající se o mapování adres na proměnné. LUT taktéž poskytuje ukazatele funkcí k obsluhám funkcí instrumentovaných proměnných. Tato tabulka je sdílena napříč vlákny.

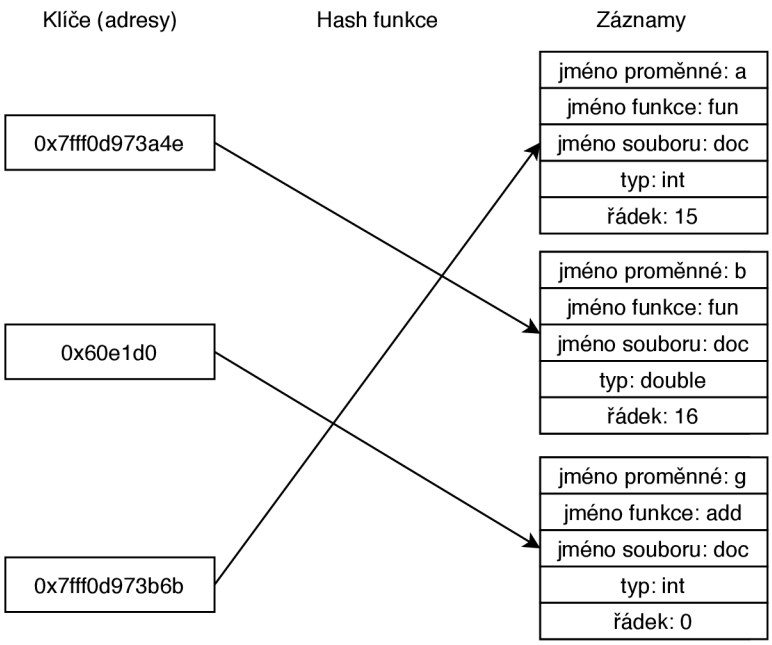
Ke sdíleným strukturám lze přistoupit pouze s výlučným přístupem, který obstarává binární semafor `__tforc_memory_mutex`. Tento semafor se využívá ve všech funkcích pracujících se sdílenými strukturami. Díky této implementaci mohou vlákna pracovat paralelně, jen při práci se sdílenými strukturami musí pracovat výlučně. Pro výpis programu do souboru je využit taktéž binární semafor `__tforc_log_mutex`, který umožňuje zapisovat do souboru vždy pouze jednomu vláknu.



Tabulka 6.1: Znázornění struktur starajících se o uložení informací o proměnných. Struktury LUT tabulky a hašovací tabulky jsou sdílené pro všechna vlákna na rozdíl od zásobníku adres, kde každé vlákno má svůj zásobník.

6.5.1 Hašovací tabulka s proměnnými

Správu adres pro přímou adresaci obstarává výhradně hashovací tabulka `__mem`, jež přiřazuje adresu k záznamu proměnné. V záznamu proměnné jsou informace o jménech proměnné, funkce a souboru. Dále název typu proměnné (typy ukazatele jsou redukovány na typ `pointer`) a řádek, na kterém byla proměnná deklarovaná. Znázornění mapování a jednotlivé položky záznamu jsou zobrazeny na obrázku 6.3.



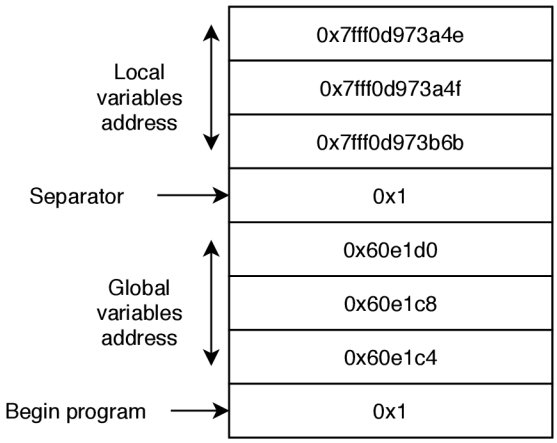
Obrázek 6.3: Znázornění mapování jednotlivých adres na záznamy obsahující informace o proměnných. Záznam proměnné obsahuje jméno proměnné, jméno funkce a souboru, ve kterém je použita, typ proměnné a řádek, na kterém byla deklarovaná (řádek 0 je vyčleněn pro globální proměnné).

6.5.2 Zásobník adres

Další důležitou strukturou pro správu adres je zásobník udržující informaci o tom, které proměnné (adresy) jsou deklarovány pro současnou funkci. Zásobník je plněn adresami deklarovaných funkcí. Při vstupu do programu se vloží na zásobník zarážka s hodnotou `0x1`. Toto se děje i před prvním přístupem nového vlákna ke svému zásobníku. Hlavním účelem vkládané zarážky je vyznačit předěl mezi proměnnými deklarovanými v jednotlivých funkcích. Proto se zarážka vkládá též před zavoláním každé funkce. Hodnota adresy zarážky je vybrána s ohledem na to, že případná manipulace s obsahem paměti (zápis/čtení) zajistí konec programu při možné chybě (díky ochraně paměti). Při každém skončení funkce se vymažou adresy až do zarážky. V posledním kroku se vymaže i sama zarážka. Znázornění naplněného zásobníku v rámci běhu programu je zobrazeno na obrázku 6.4.

Tento zásobník se dále též využívá pro udržování aktuálních hodnot v adresách LUT tabulky (viz tabulka 6.2). Hodnoty, které jsou vkládány na zásobník, se zároveň zapisují k odpovídajícím zápisům v LUT tabulce. K záznamu v LUT tabulce se též přidává identifikátor vlákna, jež jako jediné může smazat tento záznam. Při mazání záznamů ze zásobníku

se též mažou hodnoty adres z LUT tabulky a uvolňují se odpovídající záznamy proměnných i v hashovací tabulce.



Obrázek 6.4: Ukázka zásobníku starajícího se o udržování informací o adresách proměnných, které jsou deklarovány v jednotlivých zanořeních funkcí. Před první funkcí jsou uloženy adresy globálních proměnných se svojí zarážkou. Po ní je pro každou novou funkci vložena zarážka `0x1`, která značí začátek nové funkce a za ní příslušné adresy deklarované v dané funkci.

6.5.3 LUT tabulka

Hlavní datovou strukturou pro nepřímou adresaci je tabulka LUT (Look Up Table), ze které čerpá TRE informace o provázání nepřímých adres s voláním obslužné funkce. Příklad takové tabulky je vidět na obrázku s tabulkou 6.2.

V tabulce jsou uloženy identifikátory sledovaných proměnných (jméno proměnné, popřípadě i s bližším určením pomocí dekorované funkce, ve které se nachází) a k nim uloženy adresy (ve vektoru) s identifikátorem vlákna, které proměnnou vložilo. Tyto adresy ukazují přímo či nepřímo na proměnnou v záznamu. První adresa je přidána při deklaraci proměnné. Další adresy jsou přidávány pomocí kontroly každé nepřímé adresace, v jejichž hodnotě je typ ukazatel. U těchto nepřímých adresací se zkoumá, zda neukazují na již existující adresu z LUT tabulky. Pokud ano, přidá se adresa ukazatele do tabulky.

Ke každé proměnné je uložen alespoň jeden ukazatel na obsluhu operací `load/store` podle definice v souboru se specifikacemi. Když daný ukazatel na funkci není zadaný, je adresa vyplněná nulovou hodnotou.

Pro ladění přístupu a správy paměti je možné spustit ladící výpisy pomocí přednastavení konstanty `DEBUG_ADDRESS_CONTROL` na nenulovou hodnotu. Tato možnost bude vypisovat stav LUT tabulky a též položky v zásobníku starající se o životnost proměnných.

LUT (Look UP Table) pro nepřímou adresaci

Ukazatel na funkci load	Store ukazatel na funkci	ID proměnné	Vektor adresa - vlastník							...
0x0	0x60e1b0	glob_var	0x60e1c4 - 47243310069504							...
0x60e1b4	0x0	_Z3addii:sum	0x7fff0d973a4f - 47243312170752							...
0x60e1bb	0x60e1b8	_Z3addii:tmp	0x7fff0d973a40 - 47243312170752							...

Tabulka 6.2: Tabulka znázorňuje uživatelem hlídané proměnné a adresy (s identifikátorem vlákna, které ji zde zapsalo), které ukazují přímo či nepřímo na danou proměnnou. K této dvojici proměnné a adres jsou připojeny dva ukazatele na uživatelem dané funkce pro instrumentaci zápisu a čtení z paměti. Slouží pro zavolání adekvátní funkce při přístupu k proměnné záznamu.

6.6 Implementační závislosti a omezení

Tato podkapitola podává informace o závislostech a omezeních, které má implementace frameworku pro instrumentaci Tforc.

Architektura frameworku je postavena na aplikaci návrhu řízeného zodpovědností, není postavena na žádné robustní architektuře (na vrstvách), proto případná rozšiřitelnost tohoto frameworku může být omezená. Nástroj je dekomponován do modulů (tříd) podle jejich účelu, což by se při větším rozšiřování mohlo stát problematickým (z důvodu přehlednosti).

Obsluhy funkcí mohou být uživatelem psány v jazyku C/C++ nebo dodány v objektovém souboru (možnost psaní obsluhy v jiných jazycích). U jazyku C++ není podporované přetěžování funkcí (každá z definovaných funkcí musí mít jedinečný název).

Datový typ `long double` je v různých architekturách využíván s rozdílnou bitovou délkou. Proto je typ `x86_fp80` možné využít pouze u architektur podporujících formát s rozšířenou přesností x86.

Datový typ `__int128` nelze použít v argumentech instrumentované funkce. Nemožnost předání argumentů je způsobena návratovým typem překladače, jenž vrací namísto typu `{ i64 , i64 }` pouze typ `i64`.

Informace o typu proměnné je u ukazatele ukládána jednotně jako `pointer`. Proto není možné zjistit zpětně více informací o typu ukazatele, a jestli byl ukazatelem na jiný ukazatel, nebo hodnotu.

Při volání sestavovacího programu `Make` je nutné nastavit přepínače překladače do proměnné prostředí `CPPFLAGS`. Taktéž je nutné nastavit cesty k jednotlivým souborům. Dalším z omezení je to, že na vstup je možné v jednu dobu přivést pouze jeden soubor.

Ve verzích před LLVM 9 nastává problém kvůli nekompatibilitě s dřívějšími verzemi, protože verze 9 a novější využívá jiný typ návratové hodnoty `FunctionCalee` ve funkci `getOrInsertFunction` ve třídě Module.

Spojení uživatelských obsluh se samotným programem probíhá až v programu ke skládání (*link*). Kvůli tomu je před prvním voláním nutné nejprve deklarovat prototypy funkcí z TRE, které jsou využívány k získávání informací z proměnných na zvolené adrese. Stejně tak pokud se využívá operátor "<<" pro typ `__int128`, je nutno deklarovat jeho prototyp (tato operace není definována ve standardní knihovně). Prototypy funkcí získávající informace z TRE jsou umístěny v příloze B.

Použitelné typy. Ve frameworku jsou typy limitovány na základní typy LLVM. Toto omezení je možné vyřešit následovně:

- přetypováním všech nekompatibilních typů v programu,
- vytvořením nástroje, který by přepsal deklarace typů ve zdrojových kódech na podporované typy,
- nebo přidáním těchto typů do implementace frameworku Tforc

Ve frameworku Tforc jsou podporované tyto typy:

LLVM typy	C/C++ typy	Dekorovaný typ (zkratka)
i1	bool	b
i8	char	c
i16	short	s
i32	int	i
i64	long long	x
i128	__int128	n
float	float	f
double	double	d
x86_fp80	long double	e

Tabulka 6.3: První dva sloupce znázorňují ekvivalentní typy v rámci jazyků C/C++ a LLVM IR. Poslední sloupec znázorňuje, jak se typy kódují během dekorování jmen.

Kapitola 7

Evaluace instrumentačního frameworku

Tato kapitola se věnuje evaluaci vyvinutého frameworku formou validace, experimentů, příkladu použití frameworku a jeho začlenění do platformy Testos. První podkapitola 7.1 udává závislost frameworku na jednotlivých nástrojích třetích stran. Další podkapitola 7.2 znázorňuje postup při využití vyvinutého frameworku a podporované možnosti nastavení proměnných prostředí pro program `Make` ze strany uživatele (viz odstavec 7.2). V podkapitole 7.3 jsou popsané omezení frameworku, validace programu a způsob ověření pomocí testů. Podkapitola 7.4 představuje výsledky časové efektivnosti frameworku v porovnání s neinstrumentovaným programem. V podkapitole 7.5 je představena sada nástrojů Testos, která má automatizovat části testování softwaru. V navazující podkapitole 7.6 je popsaný způsob integrace frameworku Tforc s nástrojem Spectra, jímž se zařazuje mezi nástroje platformy Testos.

7.1 Závislosti vývojového prostředí

Podkapitola udává, které nástroje musí být přítomny na zařízení, kde se bude spouštět vyvinutý testovací framework, aby všechny části fungovaly správně.

Prvním nutným programem je nástroj `Make`, který poskytuje automatizované provedení sledu překladových operací a zavolání dalších odpovídajících programů.

V druhé řadě je potřebné mít na zařízení sadu nástrojů LLVM 9.0, které poskytuje překladač `Clang` pro provedení instrumentačního průchodu programem. Jedná se konkrétně o balíčky `clang`, `llvm` a `llvm-devel`.

Dalším důležitým nástrojem je interpret jazyka Python ve verzi 3.6.9. Na této verzi byl vyvinutý skript, kterým se ve frameworku generuje část, která se zaměřuje na práci s nepřímou adresací.

Tři poslední nástroje se využívají v testování a jsou potřeba k úspěšnému otestování instrumentačního frameworku. První z nich je nástroj `bc`, jenž umožňuje počítat s desetinným místem v příkazovém řádku. Poté nástroj `valgrind`, jenž se využívá jak v měření úniků paměti, tak v testech pro zjištění chyb v testu paralelního programu. Poslední nástroj `diff` v balíčku `diffutils` zajišťuje možnost porovnávání dvou souborů.

7.2 Použití frameworku Tforc

Tato podkapitola ukazuje na vzorovém příkladu, jakým způsobem se provádí instrumentace pomocí frameworku Tforc.

Část vytváření souborů potřebných k instrumentaci programu je již znázorněna v podkapitole 5.5. Použití frameworku navazuje na postup zmíněný v této podkapitole. Po splnění kroků ze zmíněné podkapitoly je nutné ještě správně zvolit možnost v programu `Make`. O možnostech tohoto programu při instrumentaci je pojednáno v následujícím odstavci.

Nastavení programu `Make` Program Make zajišťuje překlad ze souborů do instrumentovaného programu pomocí aplikací třetích stran.
Pro využití automatizované instrumentace existují tři způsoby využití:

1. Prvním způsobem je provést překlad ze zdrojového kódu v `.cpp` souboru (`FILE_NAME`) do binární formy (`BINARY`) instrumentovaného programu. Uživatelské funkce obsluhy jsou dodány v `.cpp` souboru (`USR_FUNC`). Tuto variantu lze spustit příkazem **`make`**.

2. Druhý způsob je jako vstupní zdrojový kód použít soubor `.ll`, který je už částečně přeložen (`LLVM_IR_ORIGINAL`). Pro správný chod instrumentace je nutné, aby zdrojový kód byl přeložen s **ladícími informacemi** (pomocí přepínače `-g`). Uživatelské funkce jsou dodány pomocí objektového souboru `.o` nebo `.a` (`CALLBACKS_LIB`). Tuto variantu lze spustit příkazem **`make instrumentate`**.

3. Třetí způsob je podobný jako druhý způsob s tím rozdílem, že soubor s uživatelskými funkcemi (`USR_FUNC`) je ve tvaru `.cpp`. Před použitím druhého způsobu je nutné spustit nástroj `Make` příkazem **`make compile_user_functions`**.

Jednotlivé proměnné prostředí, kterými lze ovlivnit program `Make`:

- SOURCE_FILE — jméno zdrojového kódu programu.
- LLVM_IR_ORIGINAL — jméno částečně přeloženého zdrojového kódu v LLVM IR (`.ll` formátu).
- CPPFLAGS — přepínače překladače zdrojového kódu.
- objs — pracovní adresář pro vytváření souborů.
- BINARY — jméno binárního souboru s instrumentovaným programem.
- CONF_FILE — jméno konfiguračního souboru.
- GEN_TRE_FILE — vygenerovaná část TRE.
- LOG — jméno souboru pro výpisy z překladu a instrumentace.
- USR_FUNC — jméno souboru s definicemi instrumentačních funkcí.
- CALLBACKS_LIB — jméno k objektovému souboru s obsluhami.

Jméno souboru použité v tomto výčtu proměnných prostředí zastupuje jméno souboru s relativní cestou.

Pokud chceme, aby soubor vypisoval interní informace do souboru namísto standardního chybového výstupu, je nutné nastavit název souboru do proměnné prostředí `_TFORC_OUTPUT`. Tuto proměnnou prostředí můžeme nastavit pomocí příkazu `export` pro daný terminál, nebo spustit program s nastavením proměnné prostředí.

Pokud je ve specifikaci funkce obsluhy instrumentace, která není definována v souboru s obsluhami, nastane chyba během sestavování programu kvůli chybějícímu symbolu v tabulce symbolů.

7.3 Validace

Tato podkapitola se věnuje validaci programu, která je provedena pomocí testů. Testy jsou zde popsány (viz oddíl 7.3.1) a je zde vysvětlena struktura souborů a adresářů, které se využívaji při testování (viz oddíl 7.3.2). V poslední části se podkapitola zabývá důvody, kvůli nimž nelze framework v některých případech využít (viz oddíl 7.3.3).

Pro provedení samotného testování je potřeba zadat v domovském adresáři projektu příkaz `make test`, který spustí všechny testy nad instrumentačním frameworkem a provede jejich vyhodnocení. Výsledky testů se uloží do souboru `tests_result.out` v podadresáři `tests`. Pokud chceme provést pouze jeden test, je potřeba přejít do podadresáře `tests`, zde nastavit proměnou prostředí `TEST` a použít příkaz `make test`. Instrumentovaný program z testu se objeví ve složce `bin`.

Při testování přeložených programů bylo demonstrováno, že instrumentace nezavedla žádný únik paměti. Test správy paměti je možné provést za pomocí příkazu `make mem_check` v podadresáři `tests`. Provedení tohoto příkazu vytvoří výpis `mem_result.out`, v němž jsou pro jednotlivé testy vypsány korespondující přidělení a uvolnění paměti v programu.

7.3.1 Funkcionalita testů

V tomto oddíle je popsán význam jednotlivých skupin testů podle kontrolované funkcionality, která je definována v podkapitole specifikace (viz podkapitola 4.1). Testy se dělí na skupiny následovně:

- t001-t015 — instrumentace funkcí.
- t016-t021 — instrumentace přímého přístupu do paměti.
- t022-t028 — instrumentace nepřímého přístupu do paměti.
- t029 — testování využití funkcí pro získávání informací z TRE.
- t030-t032 — testování žolíků a rekurze.
- t033-t036 — testování instrumentace funkcí psaných v jazyku C.
- t101-t122 — kontrola, jestli je správně ošetřena špatně zadaná syntaxe specifikace.
- t200 — testování změny chování programu podle vstupních argumentů. Deklaruje, že výsledky instrumentace nepřímé adresace se vyhodnocují až v rámci běhu programu.
- t300 — testování vícevláknového programu.

Popis jednotlivých testů je popsán ve podadresáři `tests` v souboru s názvem `Readme.md`.

7.3.2 Struktura testovacích souborů

V tomto oddíle je vysvětlena struktura testovacích souborů. Samotné testy jsou děleny podle prefixu na:

a) testy, které se přeloží a spustí, jejich výsledek je poté porovnán (dopadnou úspěšně). Testy jsou označeny prefixy "t0","t2","t3".

b) testy, které selžou při překladu s předem očekávaným chybovým kódem. Tyto testy začínají prefixem "t1".

Adresářový strom struktury testovacích souborů je zobrazen v příloze E. Význam jednotlivých adresářů a souborů je:

- src — adresář se soubory zdrojových kódů testů.
- queries — adresář se soubory se specifikacemi pro instrumentaci.
- configuration — adresář se soubory s konfigurací daného testu.
- instrumentation_functions — adresář se zdrojovými kódy funkcí obsluh.
- log — adresář se záznamovými soubory s chybovým výstupem z instrumentace v překladači.
- bin — adresář se soubory s instrumentovanými binárními soubory.
- references — adresář s referenčními výsledky testů, vůči kterým se porovnávají výsledky vzniklé prováděním testů.
- results — adresář se soubory výsledků jednotlivých spuštění instrumentovaného programu. V souborech s koncovkou .txt jsou výstupy ze standardního výstupu a v souborech s koncovkou .log jsou výstupy z chybového výstupu.
- tmp — adresář se soubory, které vznikají během překladu.

7.3.3 Důvody omezení funkčnosti

Framework Tforc nebude správně fungovat v těchto případech:

- Není k dispozici zdrojový kód programu.
- Program nelze přeložit pomocí překladače Clang.
- Program není přeložen s ladícími informacemi (přepínač -g).
- Program má příliš komplikovanou infrastrukturu překladu — mnoho konfigurací překladu pomocí programu `Make` — nelze z něj vytvořit jeden soubor v LLVM IR mezikódu.

Program funguje i nad vícevláknovými programy. Některé funkce starající se o zamykání a správu vláken jsou filtrovány před instrumentací. Je to způsobené tím, že tyto funkce jsou využívány v obsluze nepřímé adresace. Při jejich instrumentaci by nastalo uváznutí (*deadlock*) kvůli rekurzivnímu volání funkcí nepřímé adresace. O filtraci instrumentace těchto funkcí se stará funkce `isInMutexFunction`.

Pří využívání standardních výstupních proudů (stdout, stderr) v obslužných funkcích vícevláknových programů je potřeba dávat pozor na synchronizaci výpisů.

7.4 Experimenty

Tato podkapitola pojednává o experimentech, jež byly provedeny nad frameworkem Tforc pro instrumentaci. Jsou zde porovnány časy běhů programů, jež byly instrumentovány třemi různými způsoby. První typ vynechával instrumentaci (originální program). Druhým typem byla instrumentace pouze s přímou adresací. Poslední typ využíval instrumentaci s nepřímou adresací.

Kvůli krátkému trvání programu byl každý program spuštěn 1000krát ve smyčce pro každé měření. Každé měření programu pak bylo opakováno 10krát a bylo vynecháno minimum a maximum z výsledných hodnot. Výstupy ze spuštění byly ignorovány. Pro měření času byl využit program `/usr/bin/time`.

Režie spjata s prostředím, ve kterém se experimenty prováděly (skriptovací jazyk bash), byla naměřena jako zanedbatelná (neprojevila se ani setinou sekundy ve výsledku). Kvůli tomu nebyla započítána v měření. Naproti tomu při měření má spouštění každé instance programu významnou režii. Tato režie však nejde jednoduše eliminovat ve výsledcích.

Experimenty byly prováděné nad třemi různými programy (výsledky jsou znázorněné v tabulce 7.1). První, na němž proběhly první čtyři experimenty, je implementací hledání minima a maxima v neseřazeném poli (viz kód v příloze C).

Druhý z kódů byl program `Lift`, jenž byl převzat z testovacích příkladů z integrace s nástrojem `Spectra` (viz podkapitola 7.6). Třetí byl nástroj `head` ze sady nástrojů `coreutils`. Způsob, jak získat LLVM IR kód z překladu `coreutils` je uveden v příspěvku [13].

Jednotlivé konfigurace specifikací při překladu a význam jednotlivých specifikací v kontextu programů jsou zapsané v příloze D.

	Bez instrumentace	S přímou adresací	S nepřímou adresací
Prázdná konfigurace	1.06s	1.12s	1.29s
Zápis proměnné	1.16s	1.23s	1.38s
Čtení z proměnné	1.12s	1.29s	1.41s
Volání funkce	1.14s	1.15s	1.30s
Lift	1.24s	1.66s	1.69s
Core uttils (head)	1.03s	1.23s	1.24s

Tabulka 7.1: Porovnání časové náročnosti programu bez instrumentace a s instrumentací pomocí frameworku v nastavení s přímou a nepřímou adresací. Srovnání běhů bylo provedeno na několika příkladech s několikanásobným opakováním.

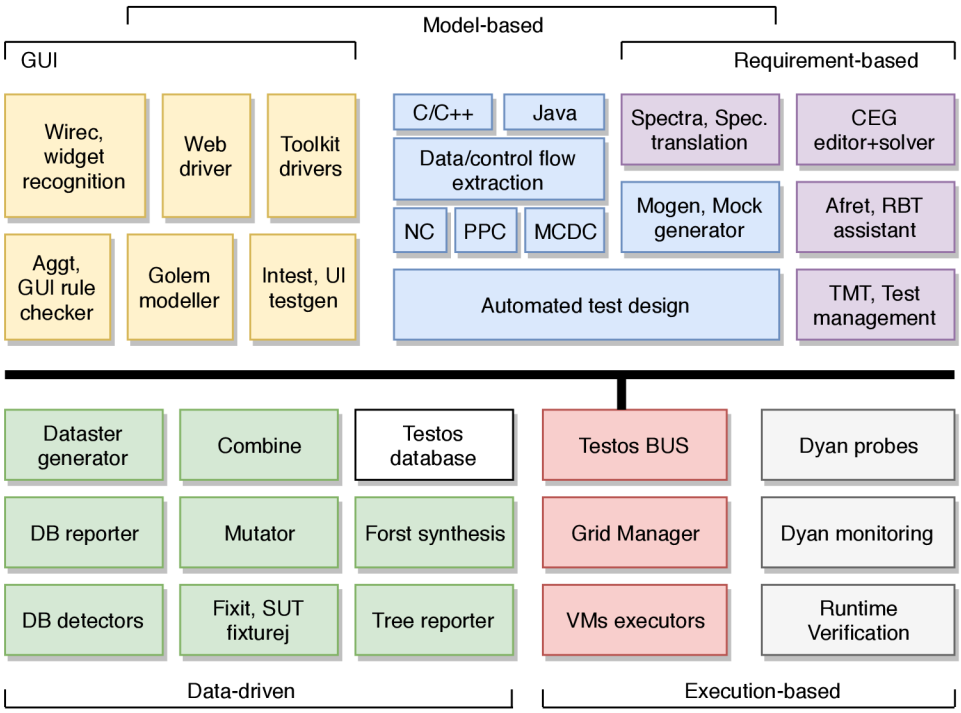
Experimenty byly prováděny na zařízení s konfigurací:

- Operační systém: Ubuntu 18.04.4 LTS
- Procesor: Intel i7-4600U
- Operační paměť: 11,4 GiB
- Cache: 4096 KB

Z výsledků experimentů je patrné, že instrumentace pomocí frameworku Tforc přidává únosnou míru režie. Je vidět, že pokud je vypnuta možnost nepřímé adresace, režie je menší. V experimentech s pouze přímou adresací vznikla průměrná časová režie 14 % oproti neinstrumentovanému běhu. Naopak experimenty s nepřímou adresací způsobily nárůst časové režie na 23 %.

7.5 Testos

V této podkapitole je popsán projekt Testos (Test Tool Set) [49], jehož hlavním cílem je vytvoření sady nástrojů podporujících automatizované testování softwaru. Nástroje v platformě Testos (viz obrázek 7.1) kombinují různé úrovně testování a lze je řadit do několika kategorií: testování založené na modelech (Model-based), testování založené na požadavcích (Requirement-based), testování grafického uživatelského rozhraní (GUI), testování založené na datech (Data-based) a dynamická analýza (Execution-based).



Obrázek 7.1: Modulární schéma infrastruktury projektu Testos, která se dělí na základních pět částí: testování založené na modelech (Model-based), testování založené na požadavcích (Requirement-based), testování grafického uživatelského rozhraní (GUI), testování založené na datech (Data-based) a dynamická analýza (Execution-based). Obrázek je převzat z [49]

Nástroj vyvinutý v rámci diplomové práce je na hranici dvou částí, a to na testování založeném na modelech a na dynamické analýze.

V sekci nástrojů pro testování založené na modelech jsou nástroje pro extrakci grafů toku řízení (CFG) ze zdrojových kódů jazyka C/C++ [34], nástroj pro hledání požadavků na testy z CFG, tj. cest v CFG [35] a nástroj pro tvorbu testovacích dat pro jazyky C/C++ [55].

V sekci nástrojů pro testování dynamickou analýzou se nachází nástroj Spectra (Specification translation) [48], v němž je spojeno testování založené na modelech s dynamickou analýzou.

Spectra je nástroj umožňující verifikaci za běhu. Využívá k tomu transformace formální specifikace do strukturovaných dat. Z těchto dat jsou vytvářeny sondy v jazyku C, které jsou následně přeloženy s testovaným programem. Instrumentace sond zatím probíhá pouze ručně (ruční oprava kódu). Pro specifikaci nástroj využívá ptLTL logiku (past-time LTL), která zde slouží pro popis požadovaného chování. Tento nástroj slouží jako podpora pro hledání a opravu sémantických chyb v implementaci.

Jedním z využití diplomové práce je integrace vyvíjeného frameworku do nástroje Spectra. Instrumentační framework Tforc zde nahradí ruční opravy kódu automatickou instrumentací podle konfigurace instrumentace (viz v podkapitole 5.4).

7.6 Integrace s nástrojem Spectra

Podkapitola se zabývá integrací nástroje Tforc do platformy Testos, konkrétně do nástroje Spectra. V tomto nástroji slouží pro automatické instrumentování požadovaných funkcí či přístupů do paměti. Program Spectra nabízí sondy pro kontrolu různých vlastností softwaru v knihovně `libspectra.a`.

V následující části je popsán postup, ve kterém je na příkladu z programu Spectra demonstrováno, jak lze využít framework Tforc v automatické instrumentaci. Jedná se o příklad Lift v podadresáři `examples`. Postup integrace je následující:

1. Prvním bodem je vytvoření adresáře, kde bude integrace probíhat.

   ```
   mkdir integration
   ```

2. Ve druhém bodu je stáhnutí jednotlivých nástrojů z úložiště platformy Testos.

   ```
   cd integration/
   git clone https://pajda.fit.vutbr.cz/testos/spectra.git
   git clone https://pajda.fit.vutbr.cz/testos/tforc.git
   ```

3. Dalším bodem je sestrojení nástroje Spectra. Zvolený příkaz nejen že sestaví program, ale též inicializuje zvolený příklad, na kterém bude ukázána integrace.

   ```
   cd spectra/
   make test
   ```

4. Po inicializaci může v adresáři tohoto příkladu proběhnout samotné vytvoření sond pomocí nástroje spectra (lift.spec — specifikace v ptLTL logice a lift.h — hlavičkový soubor programu). Vytvoří se, mimo jiné, též statická knihovna `libspectra.a` se sondami.

   ```
   cd examples/Lift/
   ./spectra --tpc -s lift.spec -d lift.h
   ```

5. Před samotným překladem programu je potřeba ve zdrojových kódech `lift.cpp-good` a `lift.cpp-bad` vymazat ručně vloženou instrumentaci funkce `monitorVerify`, aby se mohl vyzkoušet nástroj Tforc pro automatickou instrumentaci.

6. Pomocí příkazu `ln` lze vytvořit alternativní jméno k zadanému souboru. Díky tomu se může nastavit do souboru `lift.cpp` buďto program se správným (první možnost) nebo chybným (druhá možnost) během.

   ```
   ln -sf lift.cpp-good lift.cpp
   ln -sf lift.cpp-bad lift.cpp
   ```

7. Po přípravě zdrojových kódů může proběhnout samotný překlad programu do LLVM IR. Program je nutné překládat s ladícími výpisy (přepínač -g).

   ```
   clang++ -S -emit-llvm -g lift.cpp -o lift.ll
   ```

8. Nyní je potřeba přesunout vytvořené soubory do domovského adresáře programu Tforc, kde bude provedena instrumentace. A poté do tohoto adresáře přejít.

   ```
   cp lift.ll libspectra.a ../../../tforc/
   cd ../../../tforc/
   ```

9. Pro nastavení instrumentace je nutné napsat specifikaci do souboru `config_instrument`, v němž specifikujeme, co chceme instrumentovat.

   ```
   func: monitorVerify after _Z4stepv [] CCODE
   ```

10. Po nastavení specifikace může proběhnout samotná instrumentace, pro kterou je potřeba nastavit následující proměnné:

    - `LLVM_IR_ORIGINAL` — jméno llvm souboru (s ladícími informacemi),
    - `CALLBACKS_LIB` — knihovna s definicemi obsluh a
    - `BINARY` — s názvem výstupního instrumentovaného binárního souboru.

    Příkaz spouštějící instrumentaci:

    ```
    make LLVM_IR_ORIGINAL=lift.ll CALLBACKS_LIB=libspectra.a
    BINARY=lift.out instrumentate
    ```

11. Po vytvoření instrumentovaného programu je možné vyzkoušet funkčnost pomocí skriptu (test-good.sh v případě zvolením zdrojového kódu se správným během, v opačném případě test-bad.sh) z adresáře příkladu `Lift`. Skript je nutno přesunout ze stávajícího projektu do pracovního adresáře. Poté je potřeba tento skript upravit. Úprava spočívá ve smazání prvních deseti řádků (platí pro oba skripty), které slouží pro jiné účely, než je testování. Dále pak přejmenování spouštěného programu lift na jméno vytvořeného instrumentovaného programu. Poté je možné tento skript spustit.

    ```
    cp ../spectra/examples/Lift/test-bad.sh .
    # uprava skriptu
    ./test-bad.sh
    ```

    V případě dynamické analýzy správného běhu by neměla nastat chyba. V případě chybného běhu může chyba nastat. Pokud při prvním spuštění skriptu není nalezena žádná chyba (což je způsobeno závislostí programu `Lift` na výstupu pseudonáhodného generátoru čísel), je zapotřebí spustit skript znovu.

    Výsledky z prováděných testů jsou zaznamenány v souboru `lift-bad.out` (popř. `lift-good.out`), kde v případě chyby je uvedeno i pravidlo, které bylo v definovaných pravidlech ltLTL logiky porušeno.

Kapitola 8

Závěr

Cílem této diplomové práce bylo navrhnout a implementovat framework pro instrumentaci kódu během překladu, který by umožňoval instrumentovat přístupy do paměti a volání funkcí. Cíl daný zadáním se podařilo v této práci naplnit.

Na úvod práce byly představeny teoretické a odborné podklady řešeného problému. Konkrétně zde byly představeny: dynamická analýza, testování, instrumentace, překladač LLVM a nízkoúrovňový jazyk LLVM IR. Dále zde byly analyzovány již existující nástroje zabývající se tématem instrumentace během překladu.

Další části se zabývaly prvně analýzou způsobu realizace, kterým byl nástroj implementován. Následně návrhem jednotlivých částí frameworku. Posléze byly popsány implementační detaily vývoje nástroje. Poslední část se věnovala zhodnocení výsledků.

V práci se povedla vyřešit– kromě funkcionality dané specifikací zadání práce– také podpora vícevláknových programů, možnost konkretizace rozsahu sledování proměnné (jak globální, tak lokální), získávání informací o proměnných zpětně z frameworku a instrumentaci všech funkcí pomocí žolíků. Přestože nástroj povoluje instrumentaci pouze jednoho souboru, poskytuje možnosti dodání programu již v mezikódu LLVM IR, do kterého je možné převést celý program. Díky možnosti přijetí programu v mezikódu LLVM IR dovoluje framework instrumentovat programy psané v jiném jazyce než C/C++.

Po provedení experimentů nad frameworkem bylo zjištěno, že při vypnuté nepřímé adresaci program zabere průměrně o 14 % více času. Při zapnuté nepřímé adresaci se průměrný čas programu zvýší o 23 %.

Projekt by mohl být v budoucnosti obohacen o instrumentaci instrukcí větvení. Pomocí nich by tak bylo možné sledovat, jestli program prošel určitou větví, nebo při instrumentaci všech větví měřit různé metriky pokrytí kódu. Dalšími instrumentovanými typy instrukcí, které jsou vhodné pro instrumentaci, jsou základní bloky nebo atomické operace.

Pro zvýšení pochopitelnosti při tvorbě analyzátorů u vícevláknových programů může být využita možnost centralizovaného paralelismu. Pro funkčnost tohoto paralelismu je potřeba instrumentovaný kód vložit do objektu a odeslat jej do vzdáleného programu, který řeší synchronizaci přístupu do paměti centralizovaně. Také by bylo možné ve vícevláknovém programu změnit způsob zamykání souboru. Nahradit stávající globální zámek sdílených struktur *read–write* zámky nebo decentralizaci sdílené struktury v kombinaci s *read–write* zámky.

Další možné rozšíření instrumentačního frameworku je umožnění instrumentace této funkce uvnitř svého těla (definice), jež může být pro některé analyzátory vhodnější. Posledním z navrhovaných rozšíření je odstranit omezení v práci s celočíselným 128bitovým číslem, které je popsáno výše.

Literatura

[1] *Dokumentace k sadě nástrojů Score-P* [online]. VI-HPS [cit. 2019-12-03]. Dostupné z: http://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/.

[2] *DWARF Debugging Information Format Version 5* [online]. DWARF Debugging Information Format Committee, únor 2017 [cit. 2019-12-17]. Dostupné z: http://www.dwarfstd.org/doc/DWARF5.pdf.

[3] *The Architecture of Open Source Aplication* [online]. LLVM Project, prosinec 2019 [cit. 2019-12-31]. Dostupné z: https://www.aosabook.org/en/llvm.html.

[4] *Clang Argumenty příkazového řádku* [online]. LLVM Project, 2019 [cit. 2019-12-03]. Dostupné z: https://clang.llvm.org/docs/ClangCommandLineReference.html.

[5] *Dokumentace k překladači LLVM/Clang* [online]. LLVM Project, prosinec 2019 [cit. 2019-12-02]. Dostupné z: http://llvm.org/docs/.

[6] *GCC vs. LLVM Clang vs. AOCC Compiler Benchmarks On The AMD EPYC 7742 2P Linux Server* [online]. Phoronix, srpen 2019 [cit. 2019-12-31]. Dostupné z: https://www.phoronix.com/scan.php?page=article&item=epyc-7742-compilers&num=1.

[7] *The LLVM Compiler Framework and Infrastructure* [online]. David Koes, září 2019 [cit. 2020-1-4]. Dostupné z: http://symbolaris.com/course/Compilers12/LLVM.pdf.

[8] *C++ filt* [online]. LLVM Project, duben 2020 [cit. 2020-4-18]. Dostupné z: https://llvm.org/docs/CommandGuide/llvm-cxxfilt.html.

[9] *GCC and MSVC C++ Demangler* [online]. Duben 2020 [cit. 2020-4-27]. Dostupné z: https://demangler.com/.

[10] *LLVM's Analysis and Transform Passes* [online]. LLVM Project, leden 2020 [cit. 2020-1-4]. Dostupné z: https://llvm.org/docs/Passes.html.

[11] *Writing an LLVM Pass* [online]. LLVM Project, leden 2020 [cit. 2020-1-4]. Dostupné z: http://llvm.org/docs/WritingAnLLVMPass.html.

[12] Adve, V., Lattner, C., Brukman, M., Shukla, A. a Gaeke, B. LLVA: A Low-Level Virtual Instruction Set Architecture. In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. USA: IEEE Computer Society, 2003, s. 205. MICRO 36. ISBN 076952043X.

[13] Balatsouras, G. *Compiling Autotooled projects to LLVM Bitcode* [online]. [cit. 2020-5-19]. Dostupné z: http://gbalats.github.io/2015/12/10/compiling-autotooled-projects-to-LLVM-bitcode.html.

[14] Ball, T. The Concept of Dynamic Analysis. *SIGSOFT Softw. Eng. Notes*. New York, NY, USA: ACM. říjen 1999, roč. 24, č. 6, s. 216–234. Dostupné z: http://doi.acm.org/10.1145/318774.318944. ISSN 0163-5948.

[15] Barach, D. R., Taenzer, D. H. a Wells, R. E. A Technique for Finding Storage Allocation Errors in C-language Programs. *SIGPLAN Not.* New York, NY, USA: ACM. květen 1982, roč. 17, č. 5, s. 16–24. Dostupné z: http://doi.acm.org/10.1145/947923.947925. ISSN 0362-1340.

[16] Bartocci, E., Falcone, Y., Francalanza, A. a Reger, G. Introduction to Runtime Verification. In: *Lectures on Runtime Verification. Introductory and Advanced Topics*. Springer, 2018, s. 1–33. Lecture Notes in Computer Science, sv. 10457. Dostupné z: https://hal.inria.fr/hal-01762297. ISBN 9783319756325.

[17] Beizer, B. *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN 0-442-20672-0.

[18] Bernat, A. R. a Miller, B. P. Anywhere, Any-time Binary Instrumentation. In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. New York, NY, USA: ACM, 2011, s. 9–16. PASTE '11. Dostupné z: http://doi.acm.org/10.1145/2024569.2024572. ISBN 978-1-4503-0849-6.

[19] Braun, M., Buchwald, S., Hack, S., Leiundefineda, R., Mallon, C. et al. Simple and Efficient Construction of Static Single Assignment Form. In: *Proceedings of the 22nd International Conference on Compiler Construction*. Berlin, Heidelberg: Springer-Verlag, 2013, s. 102–122. CC'13. Dostupné z: https://doi.org/10.1007/978-3-642-37051-9_6. ISBN 9783642370502.

[20] Bruening, D., Duesterwald, E. a Amarasinghe, S. Design and Implementation of a Dynamic Optimization Framework for Windows. In: *ACM Workshop on Feedback-Directed and Dynamic Optimization*. Austin, Texas: [b.n.], Dec 2001. Dostupné z: http://groups.csail.mit.edu/commit/papers/01/RIO-FDDO.pdf.

[21] Chisnall, D. *Modern Intermediate Representations* [online]. University of Cambridge, červen 2017 [cit. 2020-1-6]. Dostupné z: https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf.

[22] De Bus, B., Chanet, D., De Sutter, B., Van Put, L. a De Bosschere, K. The Design and Implementation of FIT: A Flexible Instrumentation Toolkit. In: *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. New York, NY, USA: ACM, 2004, s. 29–34. PASTE '04. Dostupné z: http://doi.acm.org/10.1145/996821.996833. ISBN 1-58113-910-1.

[23] Ding, C. a Zhong, Y. Predicting Whole-program Locality Through Reuse Distance Analysis. *SIGPLAN Not.* New York, NY, USA: ACM. květen 2003, roč. 38, č. 5, s. 245–257. Dostupné z: http://doi.acm.org/10.1145/780822.781159. ISSN 0362-1340.

[24] Dinning, A. a Schonberg, E. Detecting Access Anomalies in Programs with Critical Sections. *SIGPLAN Not.* New York, NY, USA: ACM. prosinec 1991, roč. 26, č. 12, s. 85–96. Dostupné z: http://doi.acm.org/10.1145/127695.122767. ISSN 0362-1340.

[25] Feng, M. a Leiserson, C. E. Efficient Detection of Determinacy Races in Cilk Programs. In: *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*. New York, NY, USA: ACM, 1997, s. 1–11. SPAA '97. Dostupné z: http://doi.acm.org/10.1145/258492.258493. ISBN 0-89791-890-8.

[26] Fog, A. *5. Calling conventions*. Technical University of Denmark, Dec 2019. Dostupné z: https://www.agner.org/optimize/calling_conventions.pdf.

[27] Graham, S. L., Kessler, P. B. a Mckusick, M. K. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.* New York, NY, USA: ACM. červen 1982, roč. 17, č. 6, s. 120–126. Dostupné z: http://doi.acm.org/10.1145/872726.806987. ISSN 0362-1340.

[28] Greathouse, J. a Austin, T. Position Paper: The Potential of Sampling for Dynamic Analysis. Červen 2011.

[29] He, Y., Leiserson, C. E. a Leiserson, W. M. The Cilkview Scalability Analyzer. In: *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2010, s. 145–156. SPAA '10. Dostupné z: http://doi.acm.org/10.1145/1810479.1810509. ISBN 978-1-4503-0079-7.

[30] Jalan, R. a Kejariwal, A. Trin-Trin: Who's Calling? A Pin-Based Dynamic Call Graph Extraction Framework. *International Journal of Parallel Programming*. 2012, roč. 40, s. 410–442.

[31] Jorgensen, P. C. *Software Testing: A Craftsman's Approach*. 2nd. Boca Raton, FL, USA: CRC Press, Inc., 2002. 359 s. ISBN 9780429170034.

[32] Kajan, M. *Metody a organizace testování software*. Disertační práce.

[33] Kamdar, J. *Program to find Maximum and minimum number in C++* [online]. [cit. 2020-5-19]. Dostupné z: https://www.studymite.com/cpp/examples/finding-maximum-and-minimum-number-in-array-in-cpp/.

[34] Kondula, V. *Extrakce grafu toku řízení z formátu LLVM IR*. Brno, CZ, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/19239/.

[35] Kraut, D. *Generování modelů pro testy ze zdrojových kódů*. Brno, CZ, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21628/.

[36] Larus, J. R. a Schnarr, E. EEL: Machine-independent Executable Editing. *SIGPLAN Not.* New York, NY, USA: ACM. červen 1995, roč. 30, č. 6, s. 291–300. Dostupné z: http://doi.acm.org/10.1145/223428.207163. ISSN 0362-1340.

[37] Lattner, C. a Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. USA: IEEE Computer Society, 2004, s. 75. CGO '04. ISBN 0769521029.

[38] Lopes, B. C. a Auler, R. *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014. ISBN 1782166920, 9781782166924.

[39] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A. et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* New York, NY, USA: ACM. červen 2005, roč. 40, č. 6, s. 190–200. Dostupné z: http://doi.acm.org/10.1145/1064978.1065034. ISSN 0362-1340.

[40] Mellor Crummey, J. On-the-fly Detection of Data Races for Programs with Nested Fork-join Parallelism. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: ACM, 1991, s. 24–33. Supercomputing '91. Dostupné z: http://doi.acm.org/10.1145/125826.125861. ISBN 0-89791-459-7.

[41] Myers, G. J. a Sandler, C. *The Art of Software Testing*. 2. vyd. USA: John Wiley; Sons, Inc., 2004. ISBN 0471469122.

[42] Nethercote, N. a Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: *PLDI*. ACM, 2007, s. 89–100. Dostupné z: http://dblp.uni-trier.de/db/conf/pldi/pldi2007.html#NethercoteS07. ISBN 978-1-59593-633-2.

[43] Niethammer, C., Gracia, J., Hilbrich, T., Knüpfer, A., Resch, M. M. et al. *Tools for High Performance Computing 2016*. 1. vyd. Springer International Publishing, 2017. ISBN 978-3-319-56701-3.

[44] Pandey, M. a Sarda, S. *LLVM Cookbook*. Packt Publishing, 2015. ISBN 178528598X, 9781785285981.

[45] Pavela, J. *Knihovna pro profilování datových struktur programů C/C++*. Božetěchova 2 Brno, 2016. Bakalářská práce. Fakulta informačních technologií Vysoké učení technické v Brně.

[46] Schardl, T., Denniston, T., Doucet, D., Kuszmaul, B., Lee, I.-T. et al. The CSI Framework for Compiler-Inserted Program Instrumentation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*. Prosinec 2017, roč. 1, s. 1–25.

[47] Serebryany, K., Bruening, D., Potapenko, A. a Vyukov, D. AddressSanitizer: A Fast Address Sanity Checker. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2012, s. 28–28. USENIX ATC'12. Dostupné z: http://dl.acm.org/citation.cfm?id=2342821.2342849.

[48] Sečkařová, P. *Ověřování temporálních vlastností konečných běhů programů*. Brno,CZ, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21762/.

[49] Smrčka, A. *Testos (Test Tool Set)* [online]. FIT VUT, 2018 [cit. 2020-4-9]. Dostupné z: testos.org.

[50] Smrčka, A. *Testování a dynamická analýza*. FIT VUT Brno, 2019. [texty k přednáškám].

[51] Smrčka, A. *Automatizované testování a dynamická analýza*. FIT VUT Brno, 2020. [texty k přednáškám].

[52] Sommerville, I. *Software Engineering.* 9th. USA: Addison-Wesley Publishing Company, 2010. ISBN 0137035152, 9780137035151.

[53] Srivastava, A. a Eustace, A. ATOM: A System for Building Customized Program Analysis Tools. *SIGPLAN Not.* New York, NY, USA: ACM. červen 1994, roč. 29, č. 6, s. 196–205. Dostupné z: http://doi.acm.org/10.1145/773473.178260. ISSN 0362-1340.

[54] Sugumar, R. A. a Abraham, S. G. Efficient Simulation of Caches Under Optimal Replacement with Applications to Miss Characterization. *SIGMETRICS Perform. Eval. Rev.* New York, NY, USA: ACM. červen 1993, roč. 21, č. 1, s. 24–35. Dostupné z: http://doi.acm.org/10.1145/166962.166974. ISSN 0163-5999.

[55] Sušovský, T. *Generování testovacích vstupů podle stopy programu.* Brno, CZ, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/22088/.

[56] Tikir, M. M. a Hollingsworth, J. K. Efficient Instrumentation for Code Coverage Testing. *SIGSOFT Softw. Eng. Notes.* New York, NY, USA: ACM. červenec 2002, roč. 27, č. 4, s. 86–96. Dostupné z: http://doi.acm.org/10.1145/566171.566186. ISSN 0163-5948.

[57] Tschüter, R., Ziegenbalg, J., Wesarg, B., Weber, M., Herold, C. et al. An LLVM Instrumentation Plug-in for Score-P. *ArXiv.org.* Ithaca: Cornell University Library, arXiv.org. 2017. Dostupné z: https://deepai.org/publication/an-llvm-instrumentation-plug-in-for-score-p.

[58] Veitch, A., Berris, D., Nevin Heintze, E. A. nad a Wang, N. *XRay: A Function Call Tracing System* [online]. Google, duben 2016 [cit. 2019-11-29]. Dostupné z: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45287.pdf.

[59] Veitch, A., Berris, D., Nevin Heintze, E. A. nad a Wang, N. *XRay Instrumentation* [online]. Google, listopad 2016 [cit. 2019-11-29]. Dostupné z: https://llvm.org/docs/XRay.html.

[60] VITOVSKÁ, M. *Instrumentation of LLVM IR [online].* 2018 [cit. 2019-12-09]. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. Dostupné z: https://is.muni.cz/th/nhd8u/.

[61] Vitovská, M. a Chalupa, M. *Configurable instrumentation of LLVM bitcode* [online]. Statica, 2019 [cit. 2019-12-04]. Dostupné z: https://github.com/staticafi/sbt-instrumentation.

[62] Vitovská, M., Chalupa, M. a Strejček, J. *SBT-instrumentation: A Tool for Configurable Instrumentation of LLVM Bitcode.* 2018.

[63] wesarg, B., Bielert, M., Hünich, D., Brendel, R., Jitschin, L. et al. *Score-P* [online]. Vi-Hps, 2019 [cit. 2019-12-2]. Dostupné z: https://www.vi-hps.org/projects/score-p/.

[64] Ševčík, V. *Tforc.* Vysoké učení technické v Brně, Apr 2020. Dostupné z: https://pajda.fit.vutbr.cz/testos/tforc.

Příloha A

Gramatika k specifikačnímu jazyku Tforc

Pravidla specifikačního jazyka Tforc je popsána v Gramatice G. Gramatika G je definovaná čtveřicí G(N,T,P,E).
N={<start>, <query>, <position>, <function>, <mangled_function>, <position_arguments>, <position_arg>, <other_position_arg>, <additional_part>, <additional_arguments>, <additional_arg>, <other_additional_arg>, <memory>, <function>, <mangled_function>, <location>, <variable_name>}
T={ε, newline, argument, before, after, string, int, func, load, store}

	Přechod P
1	<start> → <query> newline <start>
2	<start> → newline
3	<start> → ε
4	<query> → func: <function> <position> <mangled_function> [<position_arguments>] <additional_part>
5	<query> → <memory>: <location> <mangled_function> [<additional_arguments>]
6	<position> → before
7	<position> → after
8	<memory> → store
9	<memory> → load
10	<function> → string
11	<mangled_function> → string
12	<position_arguments> → <position_arg> <other_position_arg>
13	<position_arguments> → ε
14	<other_position_arg> → , <position_arg>
15	<other_position_arg> → ε
16	<position_arg> → int
17	<additional_part> → , [<additional_arguments>]
18	<additional_part> → ε
19	<additional_arguments> → <additional_arg> <other_additional_arg>
20	<additional_arguments> → ε
21	<other_additional_arg> → , <additional_arg>

	Přechod P
22	<other_additional_arg> $\rightarrow \varepsilon$
23	<additional_arg> $\rightarrow$ string
24	<location> $\rightarrow$ <variable_name>
25	<location> $\rightarrow$ <mangled_function> : <variable_name>
26	<variable_name> $\rightarrow$ string

Příloha B

Prototypy funkcí poskytující uživateli informace z TRE

```
std::string getNameOfVariableAtAddress(int *address);
std::string getTypeOfVariableAtAddress(int *address);
std::string getFunctionNameOfVariableAtAddress(int *address);
int getLineDeclarationOfVariableAtAddress(int *address);
std::string getFileNameOfVariableAtAddress(int *address);
std::ostream& operator<<(std::ostream& os, const __int128 i) noexcept;
```

Výpis B.1: Prototypy funkcí poskytující uživateli přístup k informacím na základě adresy proměnné. Poslední z funkcí je prototyp operace « pro typ ___int128.

Příloha C

Zdrojový kód hledání extrémů

```
#include <iostream>
using namespace std;
int FindMax(int a[],int n){ //function to find largest element
        int i, max = a[0];//assume that first element is max
        for(i=1;i<n;i++) {
                if(a[i]>max) //if currentelement is greater than max
                        max =a[i]; //assign that number as max now
        }
        return max; //returns the largest number to main function
}
int FindMin(int a[],int n){ //function to find smallest element
        int i, min = a[0];// assuming first element as minimum
        for(i=1;i<n;i++){
                if(a[i]<min)// If current element is smaller than min
                        min =a[i];//assigning the smaller number to min
        }
        return min; //returns the smallest number to main function
}
int main(){
    int size = 30;
    int i, array[]{2, 31, 74, 79, 79, 50, 35, 33, 54, 93, 32, 39, 98,
        22, 76, 13, 41, 44, 99, 42, 57, 17, 67, 27, 33, 36, 95, 46, 50,
        36}, max, min;

    max = FindMax(array,size); //calls the max function
    min = FindMin(array,size); //calls the min function

    cout<<"Maximum element in the array is:" << max << "\n";
    cout<<"Minimum element in the array is:" << min << "\n";
    return 0;
}
```

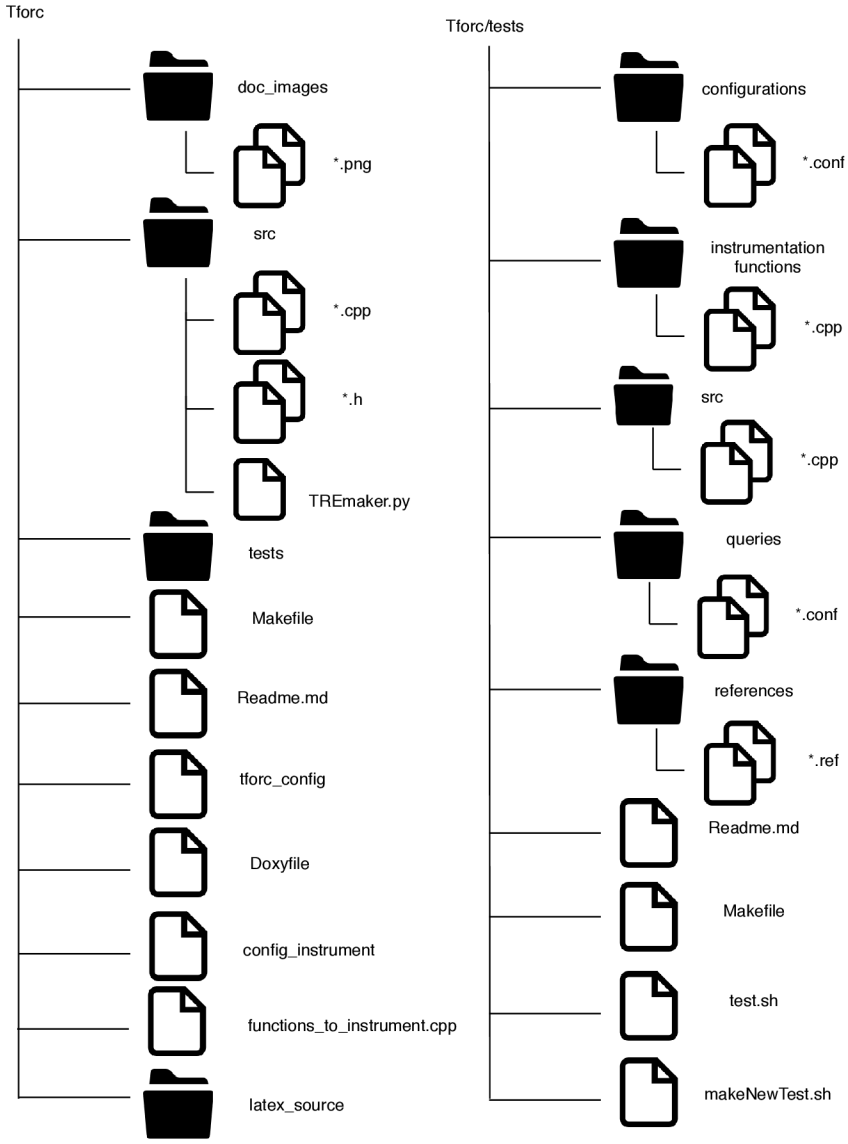
Výpis C.1: Zdrojový kód programu pro vypsání extrémů (minima a maxima) z pole hodnot. Tento příklad slouží jako referenční program pro část experimentů. Program byl částečně přejat ze stránky Studymite [33].

Příloha D

Konfigurace specifikací experimentů

```
# 1. experiment - empty

# 2. experiment - store
store: _Z7FindMaxPii:max _Z31printValueAndLocationOfVariableiPiPc
[VALUE, ADDRESS, LOCATION]

# 3. experiment - load
load: _Z7FindMaxPii:max _Z31printValueAndLocationOfVariableiPiPc
[VALUE, ADDRESS, LOCATION]

# 4. experiment - function
func: displayMin after _Z7FindMinPii [0]

# 5. experiment - Lift - function
func: monitorVerify after _Z4stepv[] CCODE

# 6. experiment - head - coreutils
store: main:n_units _Z31printValueAndLocationOfVariablexPiPc
[VALUE, ADDRESS, LOCATION]
```

Výpis D.1: Konfigurace specifikací jednotlivých experimentů. Jednotlivé specifikace jsou napsány každá na jednom řádku. První konfigurace je prázdná, druhá specifikuje sledování zápisů do lokální proměnné `max` ve funkci `FindMax`. Třetí sleduje stejnou proměnnou, ale namísto operace zápisu sleduje čtení z proměnné. Čtvrtá konfigurace sleduje výstupní hodnotu z funkce `FindMin`. Pátá specifikuje vložení funkce `monitorVerify` v jazyku C za funkci `step`, která provádí krok v programu `Lift`. Poslední konfigurace sleduje všechny zápisy do lokální proměnné `n_units`, která udává v programu `head` množství řádků, které se mají vypsat.

Příloha E

Obsah přiloženého média