VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

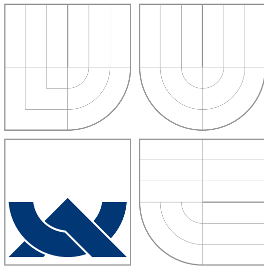
ROZŠÍŘENÍ GENERICKÉHO LADICÍHO NÁSTROJE V PROJEKTU LISSOM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

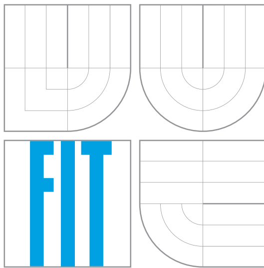
AUTOR PRÁCE
AUTHOR

Bc. PETR HONS

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ROZŠÍŘENÍ GENERICKÉHO LADICÍHO NÁSTROJE V PROJEKTU LISSOM

EXTENSION OF GENERIC DEBUGGER OF THE LISSOM PROJECT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETR HONS
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZDENĚK PŘIKRYL, Ph.D.
SUPERVISOR

BRNO 2014

Abstrakt
Práce se zabývá seznámením s problematikou ladění a ladicích nástrojů. Dále popisuje princip ladicích informací, speciálně formátu DWARF s důrazem především na Call Frame Information (CFI), které umožňují ladicím nástrojům vizualizovat zásobník volání. Zároveň byla navržena a implementována rozšíření ladicího nástroje projektu Lissom přidávající podporu zásobníku volání, historie hodnot a příkazů step return a step over.

Abstract
This thesis deals with an introduction to debugging and debuggers. The thesis describes principles of the debugging information, especially the DWARF format and its Call Frame Information (CFI), that enables a debugger to visualize the call stack. Furthermore, extensions of the debugger used in the Lissom project were designed and implemented. These extensions added support for call stack visualization, history value storage and step return and step over commands.

Klíčová slova
Ladicí informace, DWARF, Debugger, Call Frame Information, CFI, zásobník volání, call stack, Lissom.

Keywords
Debugging information, DWARF, Debugger, Call Frame Information, CFI, call stack, Lissom.

Citace
Petr Hons: Rozšíření generického ladicího nástroje v projektu Lissom, diplomová práce, Brno, FIT VUT v Brně, 2014

Rozšíření generického ladicího nástroje v projektu Lissom

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Zdeněk Přikryl, Ph.D. Další informace mi poskytli členové týmu Lissom. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Petr Hons
26. května 2014

Poděkování

Rád bych poděkoval panu Ing. Zdeňku Přikrylovi, Ph.D. za pomoc a konzultace mé diplomové práce, týmu projektu Lissom za otestování implementace a v neposlední řadě mé přítelkyni Natálii Klempové za trpělivost a podporu.

© Petr Hons, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Principy ladění a ladicí nástroje **4**
2.1 Klasifikace . . . 5
2.2 Ladění na úrovni zdrojového kódu . . . 6
2.2.1 Bod přerušení . . . 6
2.2.2 Ovládání běhu programu . . . 7
2.3 Architektura debuggerů . . . 8
2.3.1 Uživatelské rozhraní . . . 8
2.3.2 Jádro debuggeru . . . 10
2.3.3 HW a SW podpora pro ladění . . . 10
2.3.4 Existující debuggery . . . 11

3 Ladicí informace **12**
3.1 Formát DWARF . . . 13
3.2 Struktura formátu DWARF . . . 13
3.3 Kódování DWARF . . . 15
3.4 Informace o řádcích . . . 17
3.5 Call Frame Information (CFI) . . . 18
3.5.1 Struktura CFI informací . . . 21

4 Architektura projektu Lissom **22**
4.1 Třívrstvá architektura . . . 23
4.2 Debugger . . . 24
4.2.1 Rozhraní mezi debuggerem a simulátorem . . . 26

5 Návrh rozšíření **28**
5.1 Zásobník volání . . . 28
5.1.1 Zarážka . . . 30
5.1.2 Aktivní zásobník volání . . . 30
5.2 Návrh krokovacích příkazů . . . 31
5.2.1 Rozšíření bodů přerušení . . . 32
5.2.2 Step return . . . 32
5.2.3 Step over . . . 32
5.2.4 Řídicí inferior . . . 33
5.3 Historie hodnot . . . 33

6 Implementace rozšíření **35**
6.1 Knihovna libdwarf 35
6.2 Podpora ladění inicializace debuggeru 35
6.3 Čísla registrů 35
6.4 Vizualizace zásobníku volání 36
6.5 Aktivní zásobník volání 38
6.6 Problém s rekurzivními funkcemi 38
6.6.1 Řešení 39
6.7 Problém při vykročení z rámce 39
6.7.1 Řešení 39
6.8 Souběžné krokování v assembler a C kódu 39

7 Závěr **41**

A Obsah CD **45**

B Příklad výpisu DWARF CFI **46**

C Příklady rozhraní debuggerů **48**

D Vizualizace zásobníku volání **50**

Kapitola 1

Úvod

V dnešní době si neumíme představit život bez informačních technologií. Pomáhají nám v každodenním životě, v práci i v zábavě. Zároveň však prožívají tyto technologie rychlý vývoj a růst, jak v oblasti hardware, tak i software. Nejnovější aplikace obsahují velké množství funkcí, ale zároveň musí být jednoduché na ovládání. Při vývoji těchto aplikací však může vznikat velké množství chyb, které výrazně snižují kvalitu výsledného software nebo přímo znemožňují jeho použití. Z tohoto důvodu programátoři využívají řadu nástrojů pro odhalování těchto chyb, kontrolu běhu programu a testování.

Nejdůležitějším a nejzákladnějším nástrojem je *ladicí nástroj* (anglicky *debugger*[1], dále bude v textu diplomové práce používán tento anglický název[2]). Umožňuje programátorovi sledovat běh programu, analyzovat veškeré využívané struktury, případně dokonce měnit data za běhu. Díky tomuto nástroji pak lze lokalizovat, analyzovat a usnadnit nalezení většiny chyb v software.

Problémem však zůstává velké množství různých architektur, na kterých můžou aplikace běžet, přičemž každá má svá vlastní specifika, potřebné nástroje a funkcionalitu. Proto pod záštitou Vysokého učení technického v Brně vznikl projekt Lissom [28], který umožňuje vytvořit model architektury pomocí vysokoúrovňového jazyka CodAL [2]. Na základě tohoto modelu je poté možné generovat velké množství nástrojů, včetně výše zmíněného debuggeru, které umožní programátorům vytvářet a testovat programy pro danou architekturu a simulovat je.

Cílem této diplomové práce je nastudovat problematiku ladicích informací, upravit existující implementaci debuggeru projektu Lissom a přidat další užitečnou funkcionalitu případně podporu chybějících technologií. Zároveň bude řešení využíváno v projektu Lissom pro zlepšení možností při testování aplikací pomocí debuggeru.

Práce je rozdělena do sedmi hlavních kapitol. V kapitole 2 budou uvedeny základní principy ladění, rozdělení ladicích nástrojů a představena jejich architektura. V kapitole 3 bude představena problematika ladicích nástrojů, jejich účel a DWARF formát. V kapitole 4 bude následně popsána architektura projektu Lissom a jeho nástrojů se zaměřením na debugger. V kapitole 5 budou představena rozšíření debuggeru, která budou implementována v rámci této práce. Následně v kapitole 6 bude popsána implementace těchto rozšíření, popis problémů, které se objevily při této implementaci, včetně jejich řešení. Nakonec v poslední kapitole 7 budou shrnuty výsledky této práce a nastíněna možná budoucí rozšíření.

[1] Debugger z angl. spojením de-bug-er — doslovný překlad *odstraňovač bugů*, tj. chyb v programu.

[2] V textu je používán anglický pojem *debugger*, protože tento název je běžně užíván v praxi místo slovního spojení *ladicí nástroj*. Zároveň pojem *ladicí nástroj* může označovat i jiné podpůrné nástroje jako např. statické analyzátory kódu, aj.

Kapitola 2

Principy ladění a ladicí nástroje

Ladění je důležitým aspektem při vývoji jakékoli softwarové aplikace. Je definováno jako proces odstraňování chyb programu [27].

Původně byly používány jednodušší způsoby ladění programů, například úprava programu přidáním testovacích výpisů nebo analýza výstupů po ukončení programu [27]. S rostoucí složitostí programů však přestávaly tyto způsoby ladění být dostačující. Proto je dnes používán *debugger*, což je softwarový nástroj, který pomáhá zjistit, proč se daný program nechová korektně a pomáhá najít, izolovat a odstranit chyby v programu (tato definice byla převzata z [32]).

Ladicí nástroje jsou používány nejen pro hledání a opravu chyb, ale také pro sledování běhu programu, optimalizaci, případně údržbu a refaktorizaci[1] zdrojového kódu. Debugger umožňuje zastavovat běh programu na daných místech, sledovat a měnit hodnotu proměnných nebo analyzovat zásobník volání funkcí — tj. historie volání funkcí vedoucí k běhu aktuální funkce (viz. kapitola 3.5). Ladicí nástroje však nejsou používány pouze programátory, využívají je i testeři případně různé automatizované nástroje, nejčastěji pro zasílání hlášení o chybách programů (program *Dr. Watson* v operačním systému Windows [8]).

Při návrhu a vývoji debuggerů se setkáváme se třemi[2] základními principy [32] [26]:

1. **Heisenbergův princip** — *ladění daného programu nesmí pozměnit jeho chování oproti běhu bez debuggeru.* Tento princip je zajištěn nejčastěji operačním systémem, který poskytuje debuggeru rozhraní pro ovládání běhu jiného programu. Avšak jen existencí debuggeru v paměti může dojít ke změně adres v laděném programu, změně přidělování zdrojů počítače operačním systémem, atd. Tyto chyby jsou pak velice složitě odhalitelné, a proto se vývojáři operačních systémů a debuggerů snaží o eliminaci těchto problémů a co nejlepší dodržení Heisenbergova principu.

2. **Pravdivost** — *veškeré informace o laděném programu zobrazené uživateli musí být pravdivé.* Pokud by debugger zobrazoval špatné nebo zavádějící informace o běhu programu, mohl by uživatel strávit velké množství času hledáním chyby, která nemusí ani existovat. Zároveň by silně narušil důvěru v debugger samotný, což by znamenalo odmítnutí i správné informace uživatelem a ještě větší snížení efektivity ladění. Tento princip se však netýká pouze debuggeru. Většina ladicích informací je generována

[1] Refaktorizace = změna ve vnitřní struktuře programu s cílem učinit ji více srozumitelnou a snadněji modifikovatelnou, aniž by se změnilo jeho vnější chování [24].

[2] Zdroj zmiňuje čtyři principy, poslední princip popisuje, že debugger je vždy vývojově hodně pozadu oproti novým technologiím. Podle mého názoru však za 20 let došlo k posunu ve vnímání ladění a dnes již všechny technologie musí jít snadno ladit při uvedení na trh, jinak mají malou šanci uspět.

jinými nástroji, nejčastěji překladačem a assemblerem, a proto by se i tyto nástroje měly řídit tímto principem.

3. **Přehlednost** — *uživatel musí být vždy schopen zjistit, ve které části programu se aktuálně nachází — tzv. kontext.* Pokud se uživatel snaží opravit program, například po jeho pádu, potřebuje zjistit nejen kde k dané chybě došlo (např. jméno zdrojového souboru a jeho aktuální řádek), ale také jak se program k běhu dané funkce dostal (tj. jaké pořadí volání funkcí, které vedlo k volání aktuální funkce). Ve více-vláknových nebo více-procesových aplikacích musí být debugger zároveň schopen zobrazit uživateli aktuální vlákna a procesy včetně jejich stavu. A v neposlední řadě potřebuje uživatel přístup k aktuálním datům aplikace — k lokálním i globálním proměnným, datovým strukturám, případně přímo k paměti. Bez všech těchto funkcí ztrácí debugger svůj hlavní cíl pomáhat při opravě chyb v programu, protože uživatel není schopen zjistit, jak k dané chybě vůbec došlo.

2.1 Klasifikace

Při kompilaci jsou zdrojové soubory v daném zdrojovém jazyce transformovány na strojové instrukce v binární podobě. Protože však uživatel debuggeru potřebuje pozorovat běh zdrojového kódu, který je pro programátora více srozumitelný než-li samotné instrukce, bylo nutné vytvořit mapování mezi řádky zdrojového kódu a adresami strojových instrukcí [32] (řešení tohoto mapování je popsáno v kapitole 3.4). Starší debuggery však toto mapování neobsahovaly, takže zobrazovaly uživateli přímo strojové instrukce, což však nedostačovalo pro ladění složitějších aplikací. Proto moderní debugger vytváří iluzi provádění přímo zdrojového kódu místo instrukcí, což značně usnadňuje uživateli debuggeru hledání chyb.

Debuggery tedy lze rozdělit podle jejich úrovně pohledu na laděný program [32]:

1. **Úroveň strojového kódu** — debugger umožňuje pracovat a zobrazuje pouze strojové instrukce.

2. **Úroveň zdrojového kódu** — debugger mapuje prováděné strojové instrukce na řádky zdrojového kódu pomocí ladicích informací. Zároveň však umožňuje přepnutí uživatelem do pohledu strojového kódu, aby bylo možné ladit daný program i přímo po strojových instrukcích.

Nejčastěji dochází k ladění programů běžících na daném operačním systému. Občas však potřebují vývojáři ladit program běžící přímo v jádře operačního systému, nejčastěji při vývoji ovladačů. Debuggery tedy jde dělit i podle úrovně, ve které operují:

1. **Aplikační úroveň** — debugger ladí pouze aplikaci běžící nad daným operačním systémem. Je použita běžná instalace systému.

2. **Úroveň jádra operačního systému** — debugger neladí pouze aplikaci, ale i jádro operačního systému. Pro korektní ladění je nutné instalovat speciální edici operačního systému, která obsahuje ladicí informace pro jádro a při kompilaci tohoto systému nebyly použity optimalizace, které by znesnadnily jeho ladění. Protože často není schopen systém ladit sám sebe, je nutné využít vzdálené ladění, kdy počítač, na kterém běží laděný program a jádro, není stejný jako počítač, na kterém běží debugger.

Debuggery ale nejsou používány pouze pro ladění software, který běží na určitém operačním systému. S rostoucí složitostí hardware vznikly i standardy pro nízkoúrovňové ladění hardware, jako např. *JTAG* [6] (Joint Test Access Group). Proto lze rozlišovat následující typy debuggerů:

1. **Softwarový debugger** — ladí softwarové aplikace s pomocí operačního systému.
2. **Nízkoúrovňový debugger pro hardware** — debugger používající při ladění aplikace speciální hardwarové jednotky, které umožňují získat informace o nízkoúrovňových zdrojích počítače, jako např. registry, cache, paměti, aj.

Vývojáři debuggerů se snaží umožnit ladění aplikací na různých softwarových (operační systémy) a hardwarových (různé druhy procesorů a výpočetních jednotek) platformách, aby byl jejich nástroj co nejvíce použitelný. Avšak každá z těchto platforem má svá specifika a vlastní rozhraní, které může debugger využívat pro ladění jiného programu. Proto lze nakonec debuggery dělit i na následující typy:

1. **Specializovaný debugger** — debugger využívající vlastností daného systému, pro který byl navržen a implementován. Příkladem tohoto typu debuggerů je i GDB [19]. I když GDB podporuje velké množství platforem, v samotném jádře debuggeru jsou speciální funkce využívající rozhraní dané platformy (přesněji se jedná o strukturu v jazyce C – `struct gdbarch`). Samotný debugger poté využívá implementace této struktury pro danou platformu, kterých však podporuje konečné množství.
2. **Generický debugger** — debugger schopný běžet na všech architekturách. Tento typ debuggeru musí znát velké množství informací o dané architektuře. Tato diplomová práce popisuje rozšíření právě takového typu debuggeru. V projektu Lissom totiž může debugger využívat model celé platformy, ze kterého jsou poté získány požadované informace. Přesněji je architektura tohoto debuggeru popsána v kapitole 4.2.

2.2 Ladění na úrovni zdrojového kódu

Během kompilace zdrojového kódu jsou příkazy programovacího jazyka transformovány na sekvenci instrukcí daného procesoru uložených ve výsledném spustitelném programu. Po spuštění programu jsou poté postupně prováděny jeho instrukce. Avšak toto nízko-úrovňové provádění instrukcí není pro uživatele debuggeru vhodné, protože tyto instrukce vytvořil kompilátor a je obtížné z nich vypozorovat chování programu. Proto většina dnešních debuggerů vytváří iluzi provádění řádků zdrojového kódu, provádění instrukcí je skryto na pozadí a zobrazeno jen při explicitním požádání uživatelem. Pro implementaci této iluze jsou však potřebné další informace, tzv. *ladicí informace*, které jsou více popsány v kapitole 3.

Základní funkcionalita debuggerů je pozastavit a znovu obnovit běh programu. Protože je však běh programu ovládán z pohledu zdrojového kódu a pouhé pozastavení a znovu spuštění programu není dostačující, jsou používány pokročilejší techniky kontroly programu.

2.2.1 Bod přerušení

Bod přerušení (často nazývané z angl. *breakpoint*) je důležitým nástrojem pro kontrolu běhu programu a specifikuje, kde bude laděný program pozastaven a následně prohlížen [32]. Body přerušení jsou vkládány uživatelem na řádek zdrojového souboru, případně přímo

na adresu v paměti, ať už datovou nebo programovou. Při běhu laděného programu a zasažení nastaveného bodu přerušení dojde k pozastavení programu a uživatel je následně informován.

Body přerušení lze dělit na následující typy [29]:

1. **Logický** — bod přerušení specifikovaný uživatelem. Obsahuje informaci o souboru a řádku zdrojového kódu, kde je nastaven, případně o jménu funkce nebo absolutní adrese v binárním kódu programu. Zároveň může obsahovat podmínky, kdy dojde k aktivaci daného bodu přerušení, počet zasažení a průchodů nebo počet ignorovaných zasažení před aktivací daného bodu přerušení. V neposlední řadě obsahuje stav bodu přerušení, zda je aktivní nebo zda patří do některých z následujících typů.

2. **Fyzický** — bod přerušení specifikující obraz logického bodu přerušení v binárním kódu aplikace. Každý z aktivních bodů přerušení může mít konečný počet fyzických bodů přerušení, které ukazují přímo na instrukci, ve které dojde k pozastavení programu. Může však dojít i ke vztahu M:N mezi logickým a fyzickým bodem přerušení [32], například při použití šablon v programovacím jazyce C++. K úpravě, smazání nebo přidání fyzického bodu přerušení z logického dochází při editaci daného bodu přerušení uživatelem, k opačnému mapování dochází, když je bod přerušení zasažen při provádění instrukce na dané adrese fyzického breakpointu v debuggeru.

3. **Dočasný** — bod přerušení, který se automaticky smaže po zasažení. Často je tento typ bodu přerušení vytvářen debuggerem pro implementaci příkazů (například *step return*).

4. **Datový** — bod přerušení, který není nastaven na kód, ale na proměnnou. Často je také nazýván *watchpoint*. Při změně této proměnné dojde k pozastavení laděného programu a tímto přístupem lze ladit chyby způsobené nesprávnou hodnotou proměnných.

5. **Interní** — bod přerušení, který není viditelný uživatelem. Je vždy vytvářen debuggerem a velice často se jedná o dočasný bod přerušení.

6. **Nevyřešený** — bod přerušení, jehož fyzický bod přerušení nemohl být prozatím nalezen. Často nazýván *pending*, například v dokumentaci GDB [17]. Využívá se pro ladění dynamicky načítaných knihoven, jejichž kód není přístupný hned po spuštění programu, ale až po načtení knihovny.

2.2.2 Ovládání běhu programu

Kontrola běhu laděného programu je základním principem ladění pomocí debuggeru, při které debugger s podporou operačního systému a hardware přejímá kontrolu nad jiným procesem. Uživatelské rozhraní debuggeru poté umožňuje kontrolovat běh programu a provádět jednotlivé řádky zdrojového kódu programu. Tento způsob ovládání programu se také nazývá *krokování* programu [29].

Po pozastavení laděného programu může uživatel provést řadu příkazů debuggeru a běh programu kontrolovat. Základní příkazy jsou:

1. **Run** — Spustí laděný program od začátku. Pokud už program běží, spustí jej znovu.

2. **Stop** — Zastaví aktuálně laděný program.

3. **Continue** — Znovu spustí již pozastavený laděný program.
4. **Interrupt** — Pozastaví běh aktuálně běžícího programu.
5. **Step into** — Znovu spustí pozastavený laděný program a provede jeden řádek zdrojového kódu. Pokud v daném řádku dojde k volání funkce, je program pozastaven na začátku této funkce. Jinak je zastaven na novém řádku kódu.
6. **Step return** — Znovu spustí pozastavený laděný program a nastaví interní dočasný bod přerušení na návratovou adresu aktuální funkce. Z pohledu uživatele dojde k dokončení běhu aktuální funkce a k návratu za její volání.
7. **Step over** — Provede rozšířenou verzi příkazu *step into*. Jediným rozdílem je, že pokud dojde k volání funkce, není program pozastaven, ale debugger provede příkaz *step return* a následně opět pokračuje ve vykonávání příkazu *step over*. Z pohledu uživatele dojde k provedení aktuálního řádku zdrojového kódu včetně všech volání funkcí v tomto řádku a laděný program je pozastaven na řádku novém.

2.3 Architektura debuggerů

Architektura běžného debuggeru lze rozdělit do několika spolupracujících vrstev znázorněných na diagramu 2.1. Popisovaný debugger je specializovaný softwarový debugger, proto bude následně v kapitole 4.2 rozebrána achitektura generického debuggeru projektu Lissom a její hlavní rozdíly oproti běžnému debuggeru.

Nejdříve bude popsána vrstva nejblíže k uživateli – uživatelské rozhraní. Poté se budou popisovány nižší vrstvy až k samotnému hardware.

2.3.1 Uživatelské rozhraní

Uživatelské rozhraní je zobrazeno na obrázku 2.1 jako vnější vrstva debuggeru. Tato vrstva je zodpovědná za prezentování informací o aktuálně laděném programu.

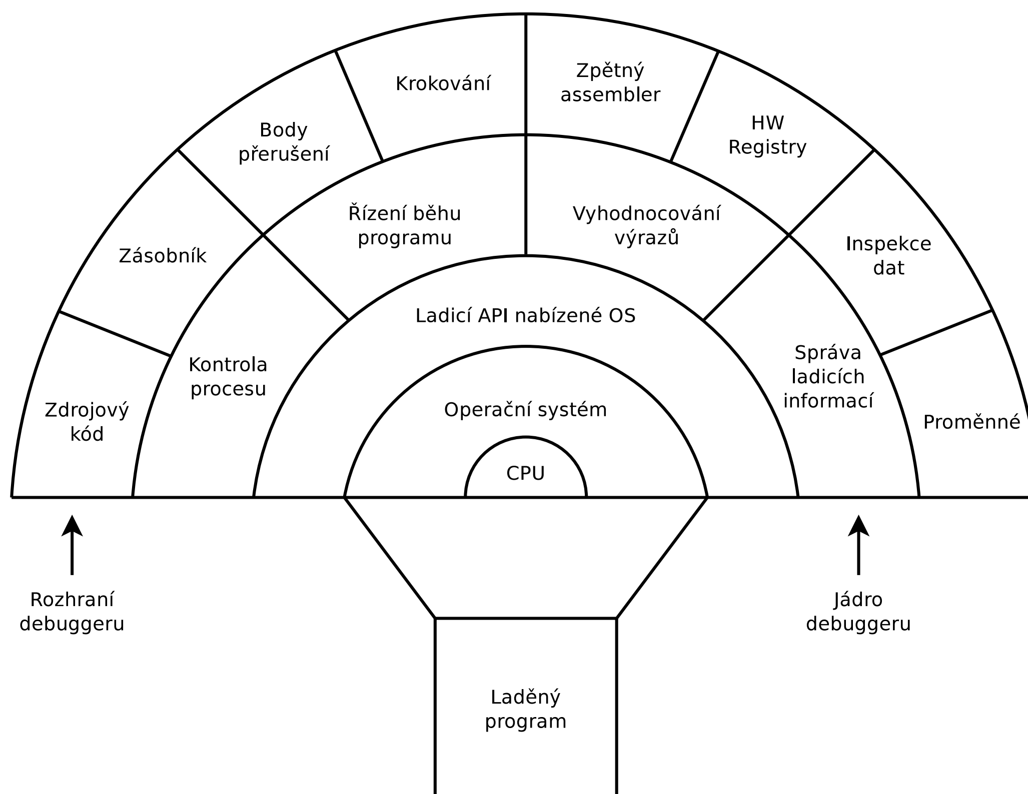
Základní prvky rozhraní se opakují ve všech nejpoužívanějších debuggerech. Ukázka grafických prostředí debuggeru *Eclipse* a *Visual Studio Debugger* je předvedena v příloze C. Obě prostředí jsou velice modifikovatelná, jak lze pozorovat v této ukázce, kde obě prostředí používají stejné rozvržení pohledů na laděný program. Více jsou tyto debuggery popsány v kapitole 2.3.4.

Nejdůležitějšími prvky rozhraní debuggeru jsou [32]:

Zdrojový kód — Zobrazuje prováděný zdrojový kód programu včetně zvýraznění aktuálně prováděného řádku. Velice často tento prvek rozhraní používá zvýraznění syntaxe programovacího jazyka, umožňuje nastavení bodů přerušení a ovládá samotný běh laděného programu.

Zásobník volání — Zobrazuje posloupnost volání funkcí, která vedla k zavolání aktuální funkce. Blíže je tento zásobník popsán v kapitole 3.5.

Seznam proměnných — Zobrazuje všechny proměnné validní na aktuálním řádku kódu včetně jejich hodnot. Tento prvek rozhraní je základním nástrojem pro porozumění chování laděného programu, protože umožňuje sledovat, zda mají proměnné správnou hodnotu, případně tuto hodnotu i měnit.



Obrázek 2.1: Typická architektura softwarového debuggeru [32].

Výrazy — Umožňuje zadávání příkazů ve zdrojovém jazyce, které jsou vypočteny při každém zastavení laděného programu.

Přehled bodů přerušení — Zobrazuje seznam všech bodů přerušení nastavených uživatelem.

Registry — Obsahuje seznam všech architekturálních[3] registrů daného procesoru a jejich hodnot. Tento prvek rozhraní je používán pouze při ladění nízkoúrovňových aplikací, běžné aplikace nejsou vázány na specifické uložení hodnot v registrech, ale místo uložení určuje kompilátor při kompilaci zdrojového kódu.

Podle způsobu přepínání zásobníku volání jej lze dělit do dvou základních typů:

1. **Pasivní** — zásobník volání je pouze vizualizován a hodnoty proměnných jsou zobrazeny pouze v nejvrchnějším rámci (pojem vysvětlen v kapitole 3.5). Tento typ debuggeru obsahoval projekt Lissom před implementací rozšíření v rámci této diplomové práce.

2. **Aktivní** — uživateli je umožněno přepínat mezi jednotlivými rámci. Seznam proměnných a jejich hodnot včetně prohlížeče zdrojového kódu je poté aktualizován pro vybraný rámec.

[3]Architekturální registry jsou registry, které jsou přístupné programátorům. V procesoru však existují i pomocné registry nutné pro korektní běh procesoru, které však nejsou programátorům přístupné (např. registry pro pipeline procesoru).

2.3.2 Jádro debuggeru

Prostřední vrstva ležící na rozmezí mezi uživatelským rozhraním a operačním systémem je jádro debuggeru. Tato část debuggeru ovládá laděný proces a získává z něj informace.

Nejdříve musí jádro debuggeru získat laděný proces. Nejběžnějším způsobem je spuštění programu přímo z debuggeru, čímž lze ladit i začátek běhu programu a jeho inicializaci. Občas je ale žádané ladit již běžící proces, a proto, pokud tuto funkcionalitu podporuje jádro operačního systému, lze debugger připojit i k již běžícímu procesu — například příkaz `attach` debuggeru GDB.

Po získání laděného programu poté debugger zpracuje dostupné ladicí informace. Tyto informace debugger potřebuje například pro mapování proměnných používaných ve zdrojovém kódu na umístění v paměti nebo v registru. Dále jsou tyto informace používány pro vizualizaci zásobníku volání. Debugger je také využívá pro vytvoření mapování adres instrukcí na řádky zdrojového kódu, čehož je poté například využito pro nastavování bodů přerušení nebo správné krokování. Blíže jsou tyto informace popsány v kapitole 3.

Většina jader debuggerů také obsahuje funkcionalitu vyhodnocování výrazů. Tyto výrazy jsou zapsány ve stejném programovacím jazyce jako zdrojový kód programu a při pozastavení laděného programu dojde k jejich vyhodnocení. Tato funkcionalita umožňuje uživatelům debuggeru testovat své představy o chování programu, například testovat návratové hodnoty funkcí nebo jiné složitější výrazy.

2.3.3 HW a SW podpora pro ladění

Softwarový debugger vyžaduje specifické rozhraní nabízené operačním systémem. Standardní přístup v operačních systémech je separace procesů od sebe navzájem, avšak debugger potřebuje přístup k datům a dokonce i kontrolovat běh jiného procesu. Proto musí operační systém nabízet podporu pro čtení a zápis paměti jiného procesu, pozastavení a opětovné spuštění jeho běhu, krokování po instrukcích, aj. Zároveň musí operační systém informovat debugger o událostech v laděném procesu, například o výjimkách, zastavení běhu procesu nebo zasažení bodu přerušení.

Na některých architekturách existují speciální instrukce (např. u procesorů Intel instrukce `INT`), které vyvolají přerušení procesoru a informování operačního systému. Ten může po zjištění, že k danému procesu je připojen debugger, informovat tento debugger o změně stavu procesu. Některé architektury (například *x86*) obsahují speciální registry pro nastavení bodů přerušení, a po dosažení dané adresy je vyvoláno přerušení. Avšak tato podpora není nezbytně nutná. Je možné krokovat proces po instrukcích a poté v debuggeru testovat, zda nedošlo k dosažení adresy nějakého bodu přerušení, i když hardwarové řešení je efektivnější.

Podpora ladění na systémech Linux

Základním nástrojem při vývoji debuggeru na operačních systémech Linux je systémové volání `ptrace` [9] [34].

Toto volání umožňuje připojení na daný proces (požadavek `PTRACE_TRACEME` pro podřízený proces nebo `PTRACE_ATTACH` pro připojení na běžící proces), získávání dat z procesu (`PTRACE_PEEKDATA` z paměti, `PTRACE_GETREGS` z registrů), ukládání dat daného procesu (`PTRACE_POKEDATA` do paměti, `PTRACE_SETREGS` do registrů) a kontrolu běhu (`PTRACE_CONT` pro pokračování běhu, `PTRACE_SINGLESTEP` pro provedení jediné instrukce). Čekání na událost je poté řešena pomocí systémového volání `wait` a přijetí signálu laděným procesem.

Podpora ladění na systémech Windows

Pro ladění procesu v operačním systému Microsoft Windows je použita speciální sada příkazů pro kontrolu procesu [1] [34].

Pro spuštění nového laděného procesu se využívá funkce `CreateProcess` s nastaveným příznakem `DEBUG_PROCESS`. Pro připojení na již běžící proces nabízí sytém Windows funkci `DebugActiveProcess`. Pro práci s pamětí laděného procesu jsou poté využity funkce `ReadProcessMemory` pro čtení a `WriteProcessMemory` pro zápis. Čekání na události z procesu je poté realizováno blokujícím voláním funkce `WaitForDebugEvent`, která vrací typ události, která nastala. Nakonec pro získání hodnot registrů je používána funkce `GetThreadContext`, pomocí které debugger získá kontext daného vlákna laděného procesu, a pro změnu hodnot registrů poté funkce `SetThreadContext`.

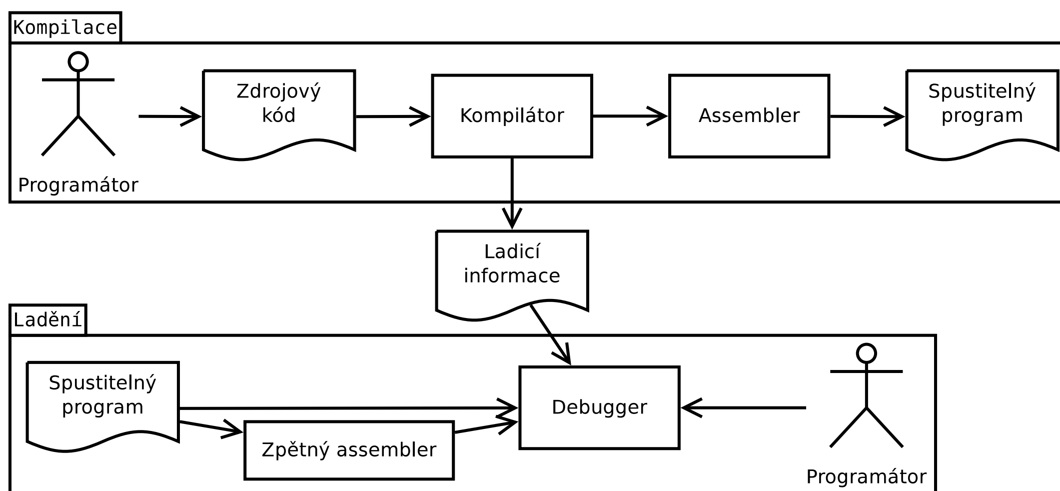
2.3.4 Existující debuggery

1. GDB [19] — nejběžnější debugger používaný v operačním systému Linux. Podporuje přes deset programovacích jazyků, z nejznámějších C, C++, Fortan, Java nebo Pascal. Umožňuje také ladění na dálku (z angl. *remote debugging* — ladění aplikací na jiném počítači, často používáno pro ladění jádra operačního systému) pomocí programu `gdbserver`. GDB však obsahuje jen textové rozhraní pomocí protokolu GDB/CLI (standardní pro ovládání uživatelem) a GDB/MI (specializované pro strojové zpracování), proto vzniklo větší množství grafických rozhraní využívající GDB jako svůj *back-end*[4], nejznámější:
   (a) DDD[5] — grafický debugger projektu GNU, podporuje nejen GDB, ale i jiné méně používané debuggery,
   (b) Eclipse[6] — platforma pro rozšiřitelné grafické vývojové prostředí různých programovacích jazyků (Java, C++, Python, PHP, aj.). Projekt CDT pro vývoj v C a C++ obsahuje i debugger využívající GDB debugger.
2. LLDB[7] — debugger vyvíjený jako součást *LLVM* projektu vyvíjející rozšiřitelný kompilátor jazyka C a C++. Jádro tohoto debuggeru využívá i samotný kompilátor, což umožňuje například podporu víceřádkových výrazů, aj.
3. Microsoft Visual Studio Debugger[8] — nejběžnější debugger používaný v operačním systému Microsoft Windows. Je součástí vývojového prostředí *Visual Studio*. Umožňuje ladění nativních C/C++ aplikací kompilovaných kompilátorem *Visual Studio C++ Compiler*, ale také podporuje jazyky využívající *.NET* knihovnu (např. C#, Visual Basic, aj.).
4. WinDbg[9] — víceúčelový debugger pro operační systém Microsoft Windows, který podporuje ladění nejen uživatelských aplikací, ale i ovladačů a jádra operačního systému. Zároveň obsahuje i grafické rozhraní a umožňuje ladění na dálku.

[4] Back-end je část aplikace, která je skrytá uživateli a je ovládána front-end aplikací. Tento přístup umožňuje odstínění jednotlivých vrstev aplikace a umožňuje tyto vrstvy na sobě nezávisle měnit, pokud komunikují pomocí předem daného rozhraní (u GDB například rozhraní GDB/MI).
[5] Domovská stránka na http://www.gnu.org/software/ddd/.
[6] Domovská stránka na http://www.eclipse.org/.
[7] Domovská stránka na http://lldb.llvm.org/.
[8] Dokumentace na http://msdn.microsoft.com/en-us/library/sc65sadd.aspx.
[9] Domovská stránka na http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

Kapitola 3

Ladicí informace



Obrázek 3.1: Obecný diagram kompilace s ladicími informacemi.

Pokud potřebuje programátor opravit chybu ve výsledném spustitelném programu, může jej pouze spustit, analyzovat výstup programu, případně použít pomocné výpisy na výstup programu pro přesnější určení chyby. Této technice se také říká ladění *post-mortem* [32]. Protože však tento přístup není efektivní, používají programátoři nástroj debugger, aby mohli ovládat, pozorovat a kontrolovat běh programu a jeho správnost.

Jak již bylo popsáno v kapitole 2.2, debugger se snaží vyvolat iluzi provádění programu po řádcích zdrojového souboru oproti skutečnému provádění instrukcí programu. Při kompilaci programu však dochází ke transformaci zdrojového kódu do sekvence instrukcí dané architektury, během které je ztraceno velké množství informací, protože výsledné instrukce umí pracovat pouze s pamětí, registry apod., ale ve zdrojovém kódu programátor využívá abstraktní prvky jako proměnné, funkce a struktury různého typu.

Proto kompilátor generuje ladicí informace, které následně debugger využívá pro zpětnou transformaci ze spustitelného programu na zdrojový kód. Ladicí informace jsou používány především pro:

1. mapování mezi adresou instrukce na řádek a soubor zdrojového kódu,

2. vizualizaci zásobníku volání funkcí,

3. získání aktuálně validních proměnných s jejich místem uložení (registr, paměť, aj.),
4. získání informací o funkcích, třídách, návratových typech a jiných abstrakcích zdrojového kódu.

Tyto informace jsou poté přiloženy k danému spustitelnému souboru, ať už přímo jako součást spustitelného programu nebo jako další pomocný soubor.

3.1 Formát DWARF

Během vývoje debuggerů a kompilátorů vzniklo v minulosti více formátů pro ukládání ladicích informací (např. STABS [30]). Později však vznikla potřeba pro vytvoření nového standardu pro tyto informace, který by splňoval základní požadavky pro moderní formát ladicích informací:

- nezávislost na platformě a zdrojovém jazyce,
- jednoduchý a efektivní formát uložení dat,
- nezávislost na místě uložení,
- vysoká efektivita vyhledávání a zpracování informací,
- možnost vytvoření rozšíření formátu,
- open-source.

Proto roku 1988 Brian Russell vyvinul v Bellovych laboratořích [16] první verzi standardu DWARF[1] [12]. Tento standard byl postupně vylepšován pod záštitou organizace PLSIG [23] — v roce 1993 vznikla 2. verze standardu DWARF s přidanou podporou jazyka C++. V roce 2005, již pod záštitou organizace DWARF Workgroup, vznikla 3. verze standardu s podporou jazyka Java, UTF-8 kódování, aj. Nakonec v roce 2010 byla vydána poslední 4. verze standardu DWARF přidávající podporu VLIW architektur, nových metod komprese, aj. Aktuálně je ve vývoji 5. verze standardu s předpokládaným vydáním v roce 2014.

Tento standard dnes využívá velké množství nejpoužívanějších vývojových nástrojů na unixových systémech (Linux, enterprise Unix, FreeBSD, apod) — především GCC [21] (GNU C/C++ compiler), GDB [19] (GNU debugger), LD [13] (GNU linker), aj.

Na platformě Windows není formát DWARF příliš rozšířený, protože společnost Microsoft používá svůj vlastní proprietární formát PDB [18].

3.2 Struktura formátu DWARF

DWARF ladicí informace jsou nejčastěji uloženy přímo ve spustitelném souboru, což je výhodné, protože není nutné distribuovat speciální soubor s ladicími informaci společně se spustitelným souborem. Nejčastěji je DWARF používán společně s formátem ELF[2],

[1] Název standardu *DWARF* je původně slovní hříčka, protože je komplementem k formátu spustitelných souborů ELF, se kterým byl společně vyvíjen [23]. Později se objevilo vysvětlení zkratky jako *Debugging With Attributed Record Formats*, avšak toto vysvětlení se nenachází v dokumentaci standardu [25].

[2] ELF — zkratka pro Executable and Linkable Format.

standardem pro formát souborů spustitelných aplikací, knihoven nebo objektových souborů. Tato kombinace ELF a DWARF standardů je také využívána v projektu Lissom.

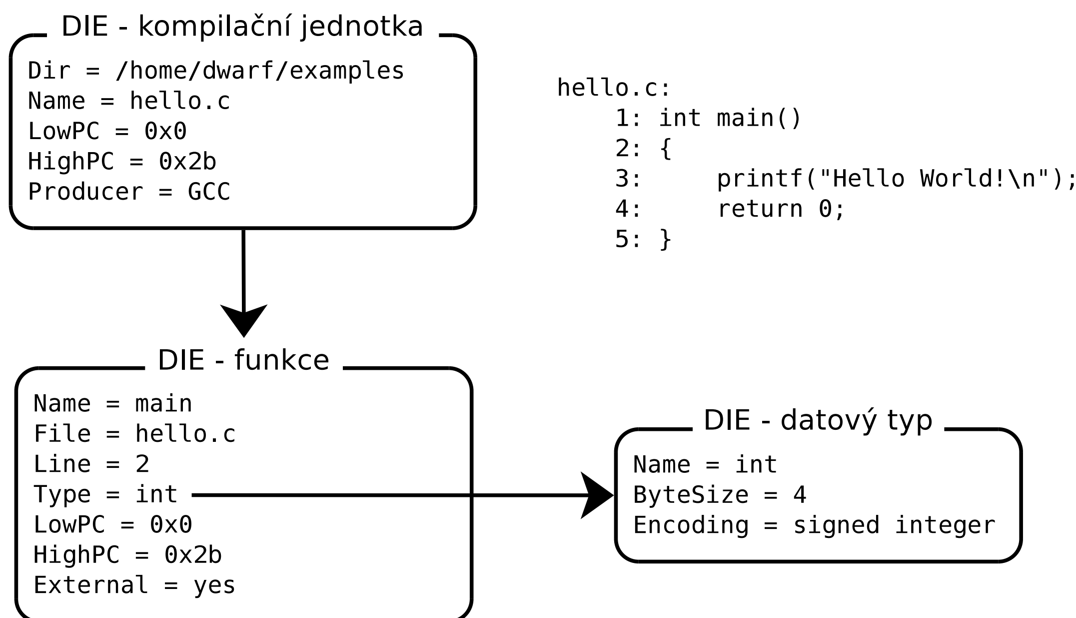
Většina moderních procedurálních jazyků je strukturována do lexikálních bloků, kdy jména proměnných definovaných v daném bloku jsou viditelné pouze v podřízených a aktuálním bloku. Tohoto stromového uspořádání je často využíváno při konstrukci tabulky symbolů v kompilátorech a využívá jej i formát DWARF.

Každý záznam ve DWARF formátu může obsahovat podřízené záznamy. Základním typem záznamu je *Debugging Information Entry (DIE)* obsahující 2 základní části:

tag určuje co daný DIE záznam popisuje. Například DIE s tagem `DW_TAG_variable` popisuje proměnnou nebo s tagem `DW_TAG_base_type` popisuje datový typ.

seznam atributů přesněji definuje vlastnosti elementu, který daný DIE záznam popisuje. Každý atribut obsahuje:

1. *jméno* — musí být unikátní v rámci jednoho DIE,
2. *hodnotu* — nejčastěji konstanta (jméno proměnné, velikost, ...), adresa, reference na jiný záznam, aj.



Obrázek 3.2: Příklad stromu DIE záznamů [23].

Výsledný strom DIE záznamů poté kopíruje strom bloků při kompilaci. Kořenový DIE záznam je označen jako *kompilační jednotka*, která popisuje každý samostatně zkompilovaný zdrojový soubor. Příklad části DIE stromu je znázorněn na obrázku 3.2.

Ladicí informace formátu DWARF jsou následně rozděleny do různých sekcí podle jejich typu. Pro každý typ DWARF ladicích informací je vytvořena sekce ve spustitelné aplikaci ve formátu ELF. Tyto sekce jsou pojmenovány podle vzoru `.debug_XXX` [23]:

`.debug_abbrev` — formát dat v `.debug_info` sekci,

`.debug_aranges` — vyhledávací tabulka pro mapování adres ze spustitelném souboru do kompilačních jednotek,

`.debug_frame` — obsahuje Call Frame Information, viz. kap. 3.5,

`.debug_info` — hlavní ladicí informace obsahující informace o proměnných, funkcích, datových typech, apod.,

`.debug_line` — informace pro mapování instrukcí spustitelného souboru na řádky zdrojového souboru,

`.debug_loc` — seznamy pro výpočet pozic v `DW_AT_location` atributech,

`.debug_macinfo` — informace o použitých makrech,

`.debug_pubnames` — vyhledávací tabulka pro mapování jmen objektů a funkcí na kompilační jednotky,

`.debug_pubtypes` — vyhledávací tabulka pro mapování jmen datových typů na kompilační jednotky,

`.debug_ranges` — rozsahy adres používané v `DW_AT_ranges` atributech,

`.debug_str` — řetězce používané v `.debug_info` sekci.

Pro ušetření místa v paměti jsou hodnoty některých atributů uloženy v jiných sekcích a v samotném záznamu je poté reference. Tohoto přístupu je využito například při ukládání řetězců (sekce `.debug_str`), což zaručuje, že DWARF informace nebudou obsahovat duplikáty stejných řetězců.

3.3 Kódování DWARF

Data formátu DWARF jsou popsány ve stromové struktuře DIE záznamů. Každý záznam poté obsahuje *tag*, který určuje typ daného záznamu.

Pro linearizaci stromových dat DWARF formátu je použit *pre-order* (někdy také nazýván *prefix order*) průchod stromem. Tento způsob uložení stromu nejdříve uloží obsah kořene stromu a následně postupně za sebou dětské uzly stromu. Pokud daný uzel (DIE záznam) neobsahuje dětské uzly, následuje za ním jeho vlastní *sourozenec*[3].

Zároveň se formát DWARF snaží komprimovat ladicí informace, aby velikost spustitelných souborů příliš nenarostla. Většina kompilátorů generuje stejné vlastnosti pro stejný typ DIE záznamů, čehož je využito v sekci zkratek `.debug_abbrev`. Tato sekce obsahuje seznam atributů, jejich typů a způsob uložení pro jednotlivý typ DIE záznamů daný identifikačním číslem zkratky. Toto číslo je poté použito v sekci `.debug_info` pro každý DIE záznam. Bez této speciální sekce by musel každý atribut každého záznamu ukládat svoje jméno, typ uložení a následně data. Při kompresi definované ve standardu DWARF se celý záznam DIE v sekci `.debug_info` odkazuje pouze na položku v sekci `.debug_abbrev` a poté již následují jen data atributů.

Na příkladu 3.3 a 3.4 lze demonstrovat toto kódování. Příklad 3.3 popisuje ladicí informace pro funkci jménem `fact` na řádku 1 souboru 1, která se nachází v binárním kódu programu na adresách `0x70`–`0x134`. Na prvním řádku výpisu ladicích informací je poté uvedeno identifikační číslo zkratky 2. Na příkladu 3.4 je poté uveden výpis ladicích informací obsahující záznam ze sekce `.debug_abbrev` s číslem 2. Tento záznam popisuje seznam atributů a k ním odpovídajícím typům uložení.

[3] Dva uzly ve stromové struktuře jsou navzájem sourozenci, pokud mají stejný přímý nadřazený uzel.

Základní typy atributů a jejich uložení [25]:

1. DW_FORM_data1, DW_FORM_data2, DW_FORM_data4, . . . — číslo o velikosti 1B, 2B, 4B, atd.,
2. DW_FORM_ref1, DW_FORM_ref2, DW_FORM_ref4, . . . — 1B, 2B, 4B, atd. reference na jiný DIE záznam,
3. DW_FORM_strp — adresa v sekci .debug_str obsahující řetězec ukončený nulovým bytem,
4. DW_FORM_addr — adresa v binárním kódu programu,
5. DW_FORM_flag_present — 1B dat obsahující hodnotu 0 nebo 1 popisující přítomnost nebo absenci vlastnosti.

```
<1><22>: Abbrev Number: 2 (DW_TAG_subprogram)
   <23>   DW_AT_name        : (indirect string, offset: 0xce): fact
   <27>   DW_AT_decl_file   : 1
   <28>   DW_AT_decl_line   : 1
   <29>   DW_AT_prototyped  : 1
   <29>   DW_AT_type        : <0x68>
   <2d>   DW_AT_external    : 1
   <2d>   DW_AT_low_pc      : 0x70
   <31>   DW_AT_high_pc     : 0x134
   <35>   DW_AT_frame_base  : 1 byte block: 51  (DW_OP_reg1 (r1))
```

Příklad 3.3: DIE záznam v sekci .debug_info.

```
  2      DW_TAG_subprogram    [has children]
   DW_AT_name          DW_FORM_strp
   DW_AT_decl_file     DW_FORM_data1
   DW_AT_decl_line     DW_FORM_data1
   DW_AT_prototyped    DW_FORM_flag_present
   DW_AT_type          DW_FORM_ref4
   DW_AT_external      DW_FORM_flag_present
   DW_AT_low_pc        DW_FORM_addr
   DW_AT_high_pc       DW_FORM_addr
   DW_AT_frame_base    DW_FORM_block1
```

Příklad 3.4: DIE záznam v sekci .debug_abbrev.

3.4 Informace o řádcích

Jednou z nejdůležitějších částí ladicích informací jsou informace o řádcích zdrojového kódu uložené standardně v sekci `.debug_line`. Debugger pro ladění na úrovni zdrojového kódu popsaný v kapitole 2.2 totiž vyžaduje mapování mezi adresami instrukcí v programu a řádkem a souborem zdrojového kódu. Toto mapování je důležité například pro krokování programu po řádcích nebo nastavování bodů přerušení na daném řádku souboru.

```
The File Name Table:
 Entry Dir Time Size Name
 1     0   0    0    /devel/project/studio/application/main.c

Line Number Statements:
 Extended opcode 2: set Address to 0x70
 Set basic block
 Extended opcode 4: set Discriminator to 1
 Copy
 Advance PC by 28 to 0x8c
 Advance Line by 1 to 2
 Set prologue_end to true
 Copy
 Advance PC by 32 to 0xac
 Advance Line by 1 to 3
 Set basic block
 Extended opcode 4: set Discriminator to 2
 Copy
 Advance PC by 56 to 0xe4
 Advance Line by 1 to 4
```

Příklad 3.5: Část výpisu informací o řádcích v sekci `.debug_line`.

DWARF informace o řádcích zároveň obsahují speciální informace jako konec *prologu* nebo začátek *epilogu* funkce. *Prolog* je počáteční část instrukcí dané funkce, které se starají o zálohu registrů a nastavování speciálních registrů pro danou funkci, jako například *base pointer* registr. *Epilog* je poté koncová část instrukcí funkce, ve které dochází ke zpětné obnově registrů zálohovaných v prologu. Při ladění na úrovni zdrojového kódu však nedochází během provádění instrukcí prologu a epilogu k provádění žádného řádku zdrojového kódu, prolog i epilog je automaticky generován kompilátorem na základě aktuální architektury a její ABI (viz. kapitola 3.5). Proto debugger využívá tyto informace pro přeskakování těchto bloků instrukcí.

Pokud by mapování obsahovalo pro každou adresu instrukce řádek a soubor, bylo by uložení velmi neefektivní a velikost ladicích informací by extrémně narostla. Proto DWARF ukládá řádkové informace jako instrukce pro konečný automat. Tento automat pracuje se sadou registrů, které mění svými instrukcemi [25]:

1. adresa instrukce programu,
2. jméno zdrojového souboru (není ukládáno jako řetězec, ale jako index do tabulky zdrojových souborů),
3. řádek ve zdrojovém souboru,

4. sloupec ve zdrojovém souboru,

5. příznaky konce a začátku prologu, epilogu a lexikálního bloku[4].

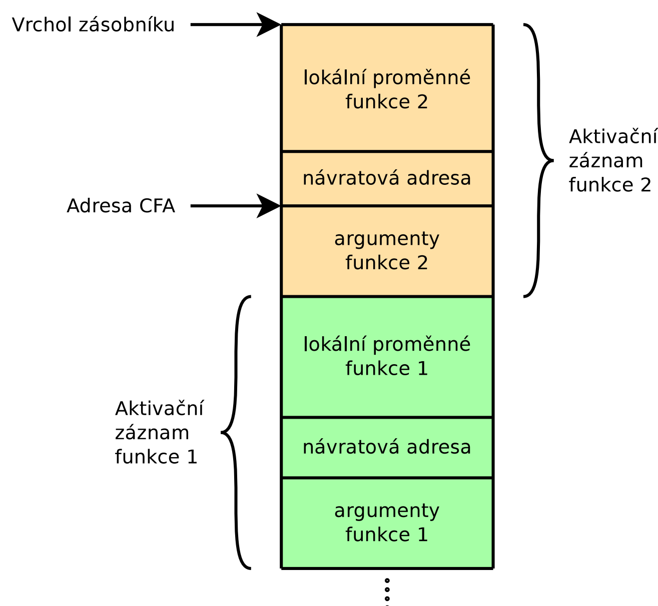
Výše popsané registry mají definované své počáteční nastavení a instrukce poté popisují pouze změny mezi jednotlivými řádky a sloupci zdrojového kódu, jak je ukázáno i na příkladu 3.5. Tento princip ukládání obsahující pouze diference mezi jednotlivými záznamy je mnohem efektivnější a je ve formátu DWARF využit vícekrát, například i u CFI informací popsaných v následující kapitole.

3.5 Call Frame Information (CFI)

Dnešní moderní procesory využívají zásobník, na který ukládají *aktivační rámce* (někdy nazývané *aktivační záznamy* nebo angl. *call frame*) pro každou zavolanou funkci. Tento rámec na zásobníku obsahuje nejčastěji tyto informace:

- argumenty aktuální funkce,
- návratovou adresu aktuální funkce,
- adresu začátku předchozího rámce, pozice aktuálního rámce je často nazývaná *frame pointer* a je uložena ve speciálním registru,
- lokální proměnné aktuální funkce.

Tento zásobník je nejčastěji nazýván *zásobník volání* (angl. *call stack*). Na obrázku 3.6 je ukázán příklad zásobníku volání, kde funkce 2 byla zavolána z funkce 1.



Obrázek 3.6: Příklad zásobníku volání.

[4] Lexikální blok (v angl. *scope*) je část zdrojového kódu, která vymezuje platnost lokálních proměnných. V jazyce C++, C ale i Java, Javascript a mnoha dalších je pro vyhrazení lexikálního bloku používána dvojice znaků { }.

Pozice těchto informací na zásobníku velice závisí na platformě (především na *ABI*[5]) a optimalizacích kompilátorů — např. pro tzv. *leaf funkce*, které nevolají žádnou další funkci, není vytvořen aktivační záznam a je využit záznam volající funkce [22]. Zároveň hodnoty některých registrů musí být zachovány při volaní nové funkce — tzv. *callee-saved*, nejčastěji registry obsahující lokální proměnné. Jiné registry zachovány být nemusí — tzv. *caller-saved*, nejčastěji registry obsahující dočasné proměnné a řešení zachování jejich hodnot musí řešit kompilátor (nejčastěji zálohou na zásobník).

Při ladění programu však debugger potřebuje vizualizovat zásobník volání, aby mohl programátor zjistit argumenty funkce, přepínat mezi jednotlivými rámci na zásobníku, získat přehled o hiearchii volání funkcí, která vedla k aktuálnímu stavu programu, aj. (viz. kapitola 2.3.1). Debugger také využívá návratovou adresu funkce pro implementaci některých krokovacích metod – například *step return* nebo *step over* – které byly popsány v kapitole 2.2.2.

Proto sekce `.debug_frame` formátu DWARF obsahuje *Call Frame Information* (dále *CFI*) obsahující veškeré potřebné informace pro správnou rekonstrukci a vizualizaci zásobníku volání. Tyto informace nejsou však používány pouze při ladění, využívají je i nástroje pro reportování chyb po nesprávném ukončení programu nebo jsou také nedílnou součástí specifikace C++ jazyka, kde jsou využity pro řešení zachytávání výjimek [4] (proto některé kompilátory ukládají CFI do sekce `.eh_frame`[6], formát uložení CI informací se však lehce liší).

Každý aktivační záznam na zásobníku (tj. blok dat na zásobníku) odpovídá *aktivaci* dané funkce — tj. nejen blok dat, ale i další informace k danému rámci, mapování na funkci programu, aj. Každá aktivace funkce obsahuje následující složky [25]:

- **Adresa instrukce**, kde byla přerušena volající funkce voláním aktuální funkce nebo zastavením na bodu přerušení nebo kroku debuggeru. Tato adresa je použita pro návrat a pokračování běhu předchozí funkce.

- **Adresa aktivačního rámce** na zásobníku, kde se nachází aktivační záznam dané funkce. Tato adresa je nazývána *Canonical Frame Address* (dále CFA). Je definována jako adresa vrcholu zásobníku v předchozím rámci (adresa vrcholu zásobníku před voláním aktuální funkce). Na některých architekturách může být CFA definována jako adresa vrcholu zásobníku při vstupu do aktuálního rámce, avšak není to možné vždy, protože instrukce volání funkce na některých architekturách (například x86) můžou upravovat zásobník uložením nových dat (v případě x86 se jedná o návratovou adresu, viz. předchozí složka aktivace). Tato adresa se nerovná hodnotě ukazatele *frame pointer*, i když velice často je výpočet CFA na tomto ukazateli založen. Hlavním rozdílem je, že CFA je konstantní v rámci celé aktivace dané funkce, zatímco hodnota frame pointeru se mění, nejčastěji v prologu a epilogu funkce.

- **Zálohy registrů**, protože obsah některých registrů musí být zachován (callee-saved registry). Tyto zálohy jsou poté použity, pokud chceme zobrazit hodnotu těchto registrů v aktivaci předchozí funkce.

Aby si mohl programátor prohlížet předchozí aktivace, musí debugger *virtuálně rozvinout* (v angl. *unwind*) zásobník volání, dokud nenarazí na požadovanou aktivaci. Debugger

[5] ABI = z angl. Application Binary Interface, obsahuje pravidla určující jak mají být funkce volány, jak se při volání funkcí předávají argumenty, kde je uložena návratová hodnota funkce, jak mají být řešena volání operačního systému aj.

[6] zkratka *eh* ve jménu sekce `.eh_frame` značí *Exception Handler*, tj. z angličtiny *obsluha výjimek*.

vždy začíná od aktuální aktivace, tj. aktuální instrukce (nejčastěji uložena v registru *program counter*) a posledního aktivačního záznamu. Poté jsou obnoveny veškeré zálohované registry v aktuální aktivaci a vypočtena CFA přechozí aktivace. Hodnota ukazatele na aktuální instrukci se změní na návratovou adresu aktuální aktivace. Tímto je simulován návrat z aktivace aktuální funkce do předchozí aktivace. Opakováním tohoto postupu až k požadované aktivaci dochází k vytváření obrazu zásobníku volání a k jeho *rozvinutí*.

Při rozvinutí zásobníku volání potřebuje debugger zjistit hodnotu CFA v předchozí aktivaci, kde jsou uloženy zálohy registrů apod., nezávisle na architektuře. Výpočet CFA a místa uložení registrů obsahují CFI informace uložené v sekci `.debug_frame`. CFI lze abstraktně reprezentovat jako tabulku 3.1.

adresa	CFA	R0	R1	...	RN
L0					
L1			*Pravidla*		
...					
LN					

Tabulka 3.1: Abstraktní struktura tabulky CFI

První sloupec (v tabulce `L0` – `LN`) určuje adresu, pro kterou jsou další sloupce tabulky platné. Další sloupce obsahují pravidla pro výpočet obsahu daného registru.

Druhý sloupce tabulky (první sloupec pravidel) obsahuje výpočet hodnoty CFA pro danou programovou adresu.

Ostatní pravidla určují výpočet hodnoty registru daného čísla v předchozí aktivaci. Tyto registry zahrnují i speciální registry jako *stack pointer* (ukazatel na vrchol zásobníku) nebo *base pointer* (ukazatel na aktivační záznam). Zároveň v pravidlech `R0`–`RN` může být virtuální registr, který neexistuje v hardware. Této techniky je využito pro výpočet návratové adresy funkce, protože na některých architekturách existuje registr s návratovou adresou funkce (např. registr `$ra` v procesorech MIPS) a na jiných architekturách je uložen například na zásobníku (např. x86).

Typy pravidel pro výpočet hodnot registrů (sloupce `CFA`, `R0`–`RN`) jsou [25]:

`nedefinovaná` — Daný registru není v předchozí aktivaci použit.

`stejná hodnota` — Hodnota daného registru se nezměnila.

`offset(N)` — Předchozí hodnota je uložena na adrese CFA+N.

`val_offset(N)` — Předchozí hodnota je rovna součtu CFA+N (málo používané).

`register(R)` — Předchozí hodnota je uložena v registru R.

`expression(E)` — Předchozí hodnota registru je uložena na adrese vypočítané pomocí výrazu E.

`val_expression(E)` — Předchozí hodnota registru je rovna výsledku výrazu E.

3.5.1 Struktura CFI informací

Kompletní CFI tabulka by byla pro běžné aplikace obrovská, její velikost by mnohonásobně převyšovala velikost samotného spustitelného kódu. Proto je tabulka reprezentována pouze rozdíly mezi jednotlivými adresami. CFI záznamy jsou ve DWARF sekci `.dwarf_debug` uloženy ve dvou formách:

`CIE (Common Information Entry)` — obsahuje společné vlastnosti pro více `FDE`, musí existovat alespoň jeden `CIE` záznam v neprázdné `.dwarf_debug` sekci. Obsahuje tyto hlavní složky:

1. `CIE_id` — unikátní číselný identifikátor `CIE`,
2. `return_address_register` — číslo sloupce v CFI tabulce obsahující pravidlo pro výpočet návratové adresy aktuální funkce,
3. `initial_instructions` — instrukce pro vytvoření prvního řádku tabulky, většinou jsou dané architekturou a ABI.

`FDE (Frame Description Entry)` – obsahuje rozdíly mezi jednotlivými řádky CFI tabulky (viz. tabulka 3.1). Nejčastěji `FDE` odpovídá jedné funkci ve zdrojovém kódu. `FDE` obsahuje tyto složky:

1. `CIE_pointer` — unikátní číselní identifikátor `CIE` nadřazaný danému `FDE`,
2. `initial_location` — adresa spustitelného kódu, od které je dané `FDE` platné,
3. `address_range` — počet bytů spustitelného kódu, po které je dané `FDE` platné,
4. `instructions` — instrukce vytvářející jednotlivé řádky tabulky.

Díky kompresi tabulky do podoby instrukcí, které implementují pouze změny mezi jednotlivými řádky CFI tabulky, došlo k velkému zmenšení CFI informací uložených ve spustitelném souboru.

Pro výpis CFI informací v daném spustitelném souboru lze využít aplikace `objdump` s parametrem `-g`. Ukázka výpisu je demonstrována v příloze B. Veškeré typy instrukcí nebudou v této práci rozebírány, protože je jejich seznam obsáhlý a lze je vyhledat ve standardu formátu DWARF [25].

Kapitola 4

Architektura projektu Lissom

Projekt Lissom [28] se zabývá hardware/software co-designem ASIP[1] procesorů pod záštitou Vysokého učení technického v Brně. Pomocí vysokoúrovňového jazyka *CodAL* lze vytvořit model procesoru, na jehož základě jsou poté automaticky generovány nástroje pro následující vývoj a testování daného procesoru.

Při testování pokročilých hardware/software systémů se můžeme setkat se dvěma způsoby testování funkcionality před samotným vytvořením cílového systému [11]:

Simulátor modeluje nějaké prostředí nebo systém, na kterém jsou prováděny experimenty (například simulátor řízení letadla). Podle výsledků těchto experimentů je následně model systému upravován, aby se více blížil potřebám návrháře.

Emulátor přesně kopíruje chování nějakého jiného systému, avšak způsob dosažení tohoto chování může být rozdílný (například emulátor staré herní platformy). Emulátor umožňuje testovat daný systém a integrovat jej do jiného systému bez nutnosti přístupu k reálnému systému.

Testovací nástroj projektu Lissom lze chápat jako simulátor (na základě jeho modelu jsou testovány aplikace, podle kterých je opět upravován model) i jako emulátor (pokud použijeme generovaný testovací nástroj v jiném systému, emuluje chování modelovaného procesoru). V rámci této diplomové práce však budeme považovat tento nástroj za simulátor, protože tato práce se nezabývá integrací testovacího nástroje do jiných platforem.

Kromě simulátoru umožňuje projekt Lissom generovat i další pomocné nástroje, především *assembler*, *zpětný assembler* a *kompilátor* jazyka C. Nakonec po ukončení vývoje daného systému je možné exportovat model do podoby jazyka používaného pro syntézu hardware do skutečného zařízení (příkladem tohoto jazyka je například VHDL).

Návrhář využívající projekt Lissom modeluje systém ze dvou pohledů:

ASIP — model jediného procesorového jádra obsahující registry, signály, porty, instrukční sadu, sémantiku instrukcí, atd. Na základě tohoto modelu je poté generován specifický assembler, zpětný assembler a kompilátor jazyka C.

Platforma — model propojení jednotlivých procesorů s ostatními periferiemi jako paměť, sběrnice, externí porty či jiné specializované komponenty. Na základu modelu platformy je poté generován simulátor.

[1]ASIP = Aplikačně specifický instrukční procesor — procesor s instrukční sadou specializovanou pro danou funkcionalitu.

Toto rozdělení do dvou typů modelů umožňuje odstínění modelu procesoru od svého okolí a jednoduší víceprocesorovou simulaci. V dřívějších verzích projektu Lissom totiž každý procesor obsahoval vlastní paměť a vlastní simulátor. Následná synchronizace těchto simulátorů byla poté velice obtížná a neefektivní, protože probíhala pomocí sdílené paměti nebo TCP/IP protokolu. Využití platformy však umožňuje jednodušší modelování složitějších systémů, protože při jiném zapojení procesorů není nutné upravovat jejich modely, ale pouze celkový model platformy. Zároveň synchronizace je mnohem efektivnější, protože je generován jediný spustitelný simulátor pro kompletní platformu, simulátory pro jednotlivé procesory jsou poté reprezentovány objekty uvnitř programu simulátoru.

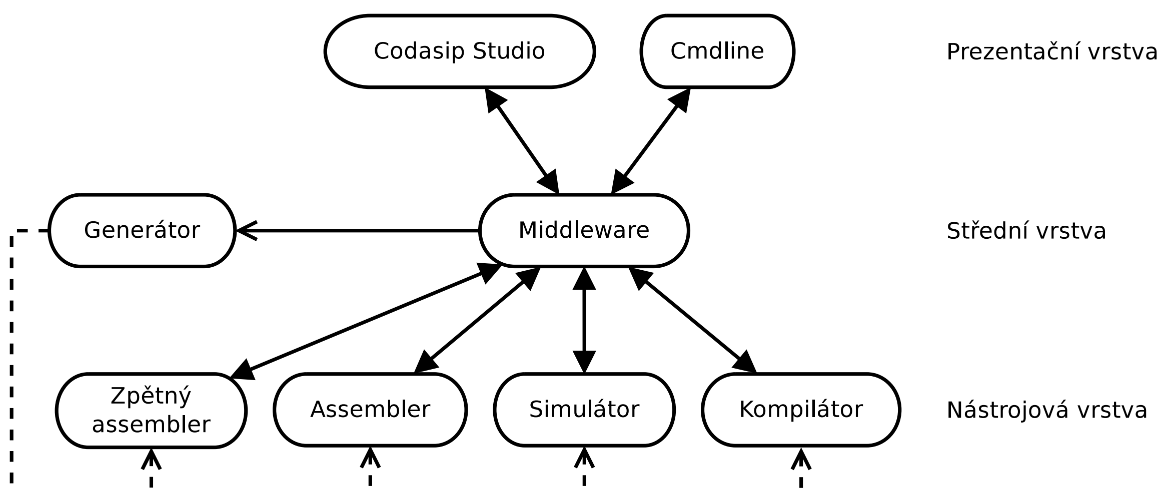
4.1 Třívrstvá architektura

Architektura projektu Lissom se skládá ze tří oddělených vrstev:

Prezentační vrstva zprostředkovává rozhraní pro uživatele.

Střední vrstva zpracovává požadavky od uživatele, spouští generátory a obsluhuje generované nástroje a simulaci.

Nástrojová vrstva je generována na základě modelu procesoru.



Obrázek 4.1: Třívrstvá architektura projektu Lissom.

Na obrázku 4.1 jsou znázorněny vrstvy projektu Lissom. Plné šipky ukazují směr komunikace mezi jednotlivými vrstvami. Přerušované šipky ukazují směr volání a generování jednotlivých nástrojů.

Prezentační vrstva umožňuje klientům připojit se na server *Middleware* a jeho prostřednictvím pracovat s nástroji projektu Lissom. Uživatel může využít klienta příkazové řádky *cmdline* pro základní rozhraní nebo například pro tvorbu skriptů. Projekt Lissom nabízí i pokročilejší grafické rozhraní *Codasip Studio*, které je postavené na platformě *Eclipse* [3]. Oba klienti umožňují posílat příkazy serveru *Middleware*, přes který můžou spouštět generátor, generované nástroje nebo samotnou simulaci. Při generování simulátoru s podporou ladění může klient ovládat debugger stejnými příkazy jako standardní GDB [19] nebo z grafického rozhraní stejným způsobem, jako je ovládáno ladění C++ aplikací.

Střední vrstva obsahuje server *Middleware*, který je spojujícím článkem celého projektu Lissom. Jeho základním úkolem je ověřovat identitu uživatelů a zpracovávat jejich příkazy, na jejichž základě jsou spouštěny nástroje projektu Lissom. Zároveň je tento server rozšiřitelný pomocí pluginů. Druhou částí střední vrstvy je *generátor*, který umožňuje na základě modelu procesoru nebo platformy generovat požadované nástroje.

Nástrojová vrstva je poté specifická pro každý procesor vyvíjený uživatelem. Generátor na základě modelu generuje zdrojové soubory v jazyku C++. Společná funkcionalita nástrojů, která je nezávislá na samotném modelu procesoru, je předem zkompilována do podoby statických knihoven (binární soubor s příponou .a), čímž je výrazně urychlena doba kompilace. Nakonec jsou zkompilované generované C++ soubory a knihovny spojeny ve výsledný spustitelný soubor. Pokud klient požaduje přímé spouštění nástrojů, je možné exportovat všechny spustitelné nástroje (soubor nástrojů je nazýván *toolchain*) zpět klientovi v podobě archívu.

4.2 Debugger

V této kapitole bude popsána architektura debuggeru v projektu Lissom, jehož rozšířením se zabývá tato diplomová práce. Jedním ze základních cílů tohoto debuggeru je podpora kompletního rozhraní debuggeru GDB. Hlavním důvodem je možnost využití velkého množství grafických rozhraní, která využívají GDB jako svůj *back-end* (viz. kapitola 2.3.4).

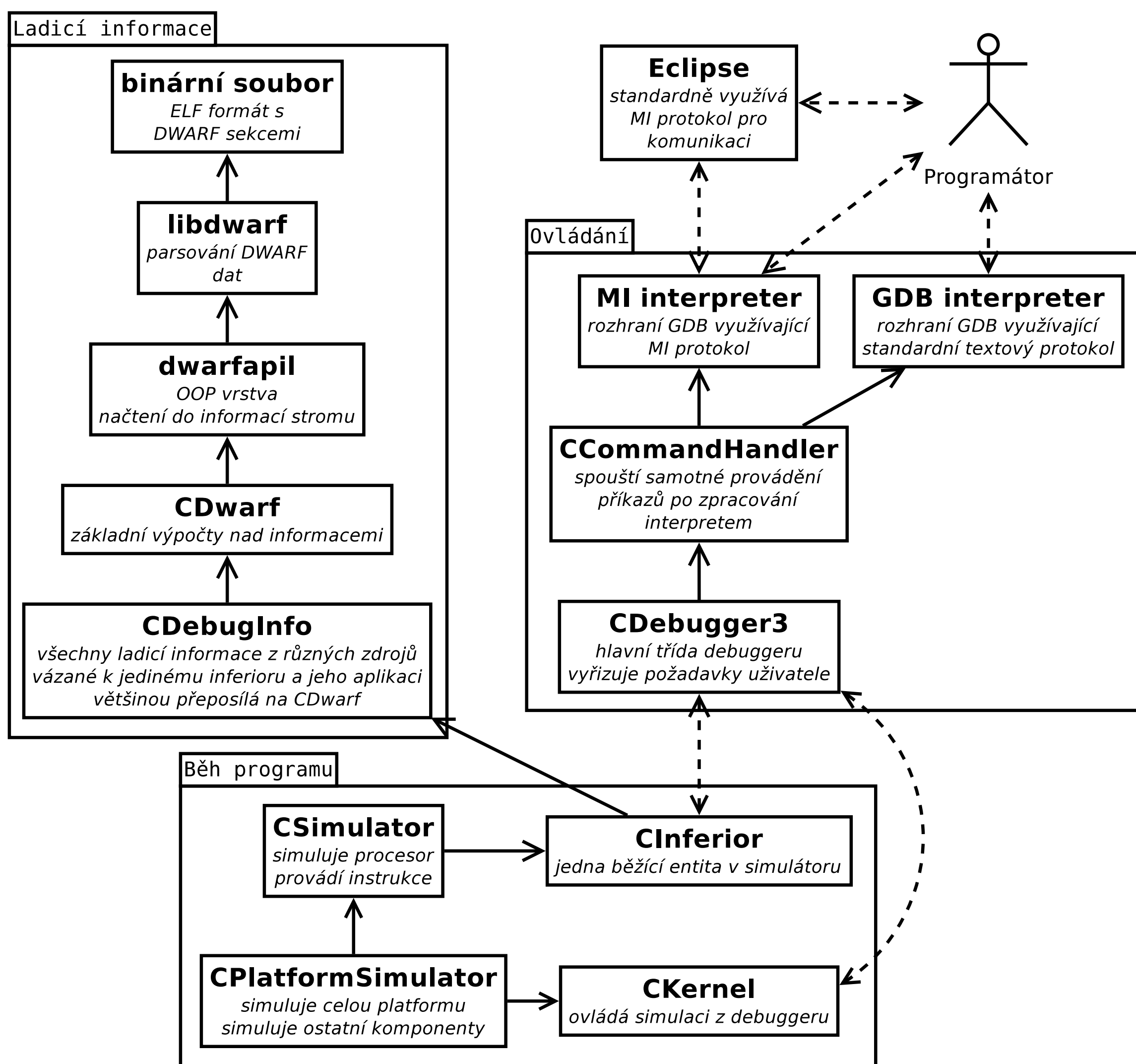
Na obrázku 4.2 je znázorněna část architektury debuggeru, šipky na obrázku znázorňují směr, kterým jsou objekty vytvářeny a inicializovány, přerušované šipky poté směr zasílání požadavků.

Debugger je silně vázán k simulátoru, neexistuje jako samostatná aplikace, protože pro tuto funkcionalitu lze využít standardní GDB. Debugger projektu Lissom je generický (viz. kapitola 2.1) a proto jeho zdrojové soubory nejsou závislé na modelu procesoru, takže není nutné jej generovat. Samotné jádro debuggeru je předem zkompilováno do podoby knihovny `dbg3l`, která je poté přilinkována k simulátoru. Pokud je simulátor generován s podporou ladění, jsou v jeho kódu generovány volání obslužných rutin debuggeru, jako obsluha zasažení bodu přerušení nebo volání obslužné rutiny po provedení instrukce při zapnutém krokování (rozhraní je blíže specifikováno v kapitole 4.2.1).

Debugger komunikuje s programátorem (případně s jiným externím programem) pomocí dvou protokolů:

GDB/CLI — standardní textové rozhraní GDB, někdy nazýváno pouze *Console*. Na obrázku 4.2 implementováno třídou `GDB interpreter`. Toto rozhraní je uzpůsobeno pro ovládání člověkem z příkazové řádky, obsahuje jednoduché snadno zapamatovatelné příkazy (např. `run`, `step`, aj.). Toto rozhraní však není příliš vhodné pro ovládání debuggeru z externího programu, protože jeho výstup není uzpůsoben pro strojové zpracování, nemá podporu pro asynchronní zprávy, atd.

GDB/MI — novější strojově uzpůsobené textové rozhraní [17]. Na obrázku 4.2 implementováno třídou `MI interpreter`. Toto rozhraní podporuje asynchronní zprávy, umožňuje číslovat pořadí zpráv a výstupy z příkazů jsou snadno strojově zpracovatelné, proto jej využívá například vývojové prostředí Eclipse. Zároveň posílá aktuální kontext pro každý příkaz (tj. vybraný inferior, vlákno a rámec), takže jádro debuggeru si nemusí tento kontext ukládat. Není vyloučeno ovládání přímo programátorem, avšak syntaxe je složitější, a proto je pro přímé ovládání vhodnější spíše standardní textové rozhraní. Ukázka komunikace je předvedena na příkladě 4.3.



Obrázek 4.2: Architektura debuggeru projektu Lissom.

Architekturu debuggeru na obrázku 4.2 lze z pohledu ladicích informací rozdělit do 3 oddělených částí:

Ladicí informace — třídy starající se o načtení, zpracování a analýzu ladicích informací. Při načtení daného spustitelného souboru debuggerem jsou z tohoto souboru načteny DWARF informace pomocí C knihovny `libdwarf` (viz. kapitola 6.1), tyto informace jsou v knihovně `dwarfapil` transformovány do záznamů uspořádaných ve stromech kopírujících specifikaci DWARF formátu. Následně při dotazu debuggeru na specifickou informaci nebo při požadavku na výpočet či vyhledání nějaké entity volá debugger funkce knihovny `CDebugInfo`, která přeposílá požadavek do knihovny `CDwarf`, která daný požadavek vyhodnotí. Knihovna `CDebugInfo` je využívána, protože obecně může existovat více druhů ladicích informací z různých zdrojů. Tato diplomová práce se však zabývá pouze DWARF informacemi.

```
-> -exec-run
<- ^running
<- (gdb)
<- *stopped,reason="breakpoint-hit",disp="keep",bkptno="1",
   thread-id="0",frame={addr="0x08048564",func="main",
   args=[{name="argc",value="1"},{name="argv",value="0xbfc4d4d4"}],
   file="myprog.c",fullname="/home/nickrob/myprog.c",line="68"}
<- (gdb)
-> -exec-continue
<- ^running
<- (gdb)
<- *stopped,reason="exited-normally"
<- (gdb)
```

Příklad 4.3: Příklad spuštění aplikace a zasažení bodu přerušení v protokolu GDB/MI

Ovládání — třídy starající se o zpracování a vykonání požadavků uživatele debuggeru. Při spuštění debuggeru je specifikován typ rozhraní, který bude použit. Při přijetí zprávy přes MI rozhraní nebo standardní GDB rozhraní je tento požadavek a jeho parametry zpracovány v daném interpretu, který jej přepošle do třídy `CCommandHandler`. Tato třída již daný požadavek přímo provádí s pomocí hlavní třídy `CDebugger3`, ladicích informací a dat ze simulátoru.

Běh — třídy starající se o samotné provádění spustitelného souboru. Třída `CSimulator` je generována na základě modelu daného procesoru a provádí samotný běh programu, zároveň však umožňuje ostatním částem debuggeru přístup k získání nebo změně hodnot registrů a paměti. Při ladění je simulátorem každého ASIPu zapouzdřen v tzv. *inferioru* implementovaném třídou `CInferior`, která zapouzdřuje jeden běžící proces v debuggeru. Inferior poté může mít více vláken, avšak pořád se jedná o jediný proces. Třída `CPlatformSimulator` je generována na základě modelu celé platformy, řídí běh a synchronizaci procesorů s jejich okolím a při své inicializaci vytváří objekty specifických tříd `CSimulator` pro každý ASIP uvnitř platformy a také objekty obsluhující všechny ostatní prvky platformy. Při ladění je celý platformní simulátor zapouzdřen do objektu třídy `CKernel`, který obsluhuje komunikaci mezi debuggerem a platformním simulátorem.

4.2.1 Rozhraní mezi debuggerem a simulátorem

Pro implementaci generického debuggeru je klíčovou částí rozhraní mezi debuggerem a běžícím procesem, v případě projektu Lissom simulátorem. Toto rozhraní musí umožňovat ovládání jakéhokoli procesoru přes stejný generický debugger. Simulátor simulující jednotlivý ASIP nabízí následující rozhraní využívané debuggerem a jeho částmi:

1. Získání unikátního jména simulátoru v rámci platformy. Toto jméno je specifikováno v modelu platformy.
2. Načtení spustitelného programu z daného souboru.
3. Ovládání běhu simulace — spuštění a zastavení simulace, spuštění pouze jednoho cyklu nebo restart simulace.

4. Získání adresy aktuální instrukce (hodnota registru *program counter*) a počtu již provedených cyklů.

5. Čtení a zápis zdrojů — registrů, portu, aj.

6. Čtení a zápis slova dat paměti na dané adrese, velikost slova je daná modelem a musí být také zjistitelná.

7. Nastavení nebo zrušení bodu přerušení (i datového) na dané programové adrese, jedná se pouze o příznak bodu přerušení, samotné řešení podmínek a jiných vlastností obstarává debugger.

8. Zapnutí a vypnutí módu krokování — pokud je tento mód zapnutý, simulátor informuje debugger po každém cyklu simulace.

9. Informace o zdrojích — například bitové šířky a jména registru, statistiky, aj.

Jádro generického debuggeru (třída `CDebugger`) musí na druhé straně nabízet obecné rozhraní pro všechny druhy procesorů. Následující rozhraní je navíc využíváno simulátorem pro platformu i ASIP:

1. Spuštění debuggeru s daným rozhraním. Pokud je simulace řízená přes Middleware, je specifikován i port, protože příkaz jsou posílány přes TCP/IP kanál.

2. Registrace platformního simulátoru pro přístup k platformě.

3. Obslužné rutiny reagující na následující změny stavu inferiorů:

   (a) Registrace nového inferioru,

   (b) Spuštění nebo ukončení běhu programu v daném inferioru,

   (c) Vytvoření nebo ukončení běhu vlákna v daném inferioru.

4. Hlavní obslužná rutina volaná při pozastavení simulátoru (metoda `HandleDebugger`). Tato funkce obdrží seznam důvodů pozastavení platformního simulátoru — seznam zasažených adres bodů přerušení v daném inferioru a inferiory se zapnutým krokovacím módem. Debugger poté na základě všech dostupných informací rozhodne, zda má k zastavení simulace opravdu dojít. Například podmínka bodu přerušení nemusí být splněna nebo při krokování nedošlo ke změně zdrojového řádku kódu, tyto informace však nejsou simulátoru známy, proto nemusí být simulace při volání `HandleDebugger` vždy pozastavena.

Kapitola 5

Návrh rozšíření

Cílem této diplomové práce je rozšíření již existujícího debuggeru o chybějící funkcionalitu, aby se více přiblížila GDB [19] debuggeru. V této kapitole bude popsána existující implementace debuggeru projektu Lissom a následně budou předvedena rozšíření debuggeru implementované v rámci této diplomové práce.

Základní nedostatky původní implementace debuggeru:

1. zobrazení pouze posledního rámce v zásobníku volání funkce,
2. nebylo možné prohlížet proměnné předchozích rámců, protože byl zobrazen pouze jediný rámec,
3. podpora krokování pouze pomocí příkazu *step into*,
4. chybějící podpora pro interní body přerušení,
5. nebylo možné nastavit speciální obslužné rutiny debuggeru, které by byly zavolány při zasažení bodu přerušení.

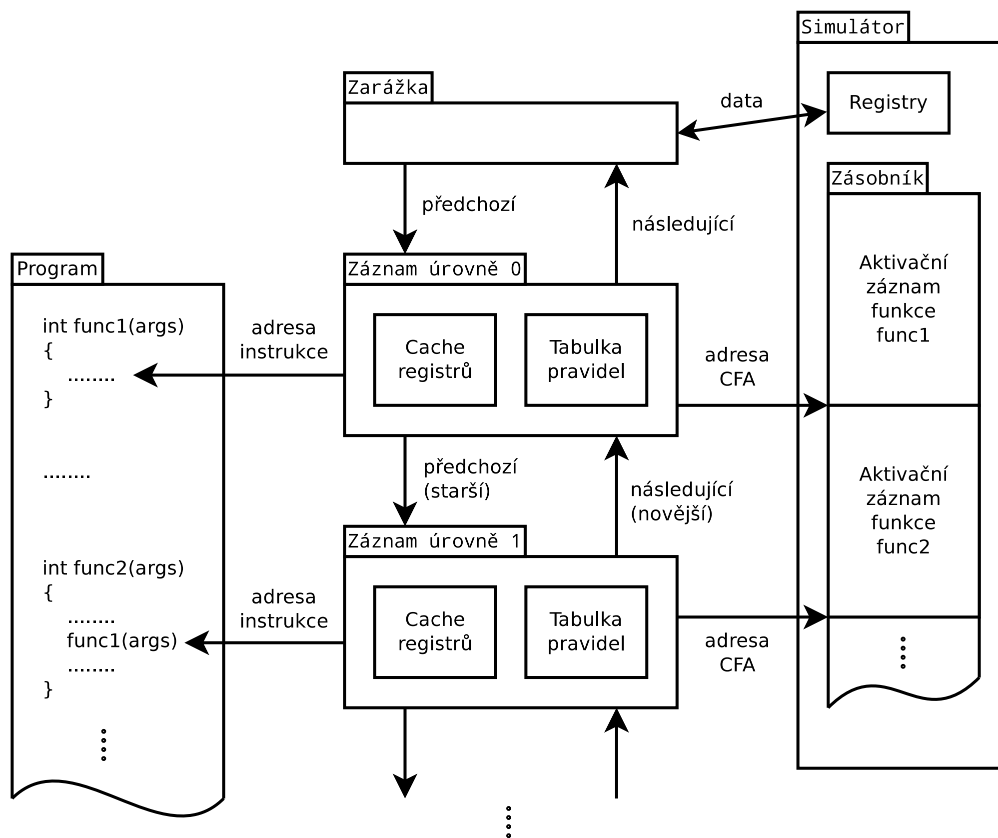
Kromě řešení výše zmíněných nedostatků došlo i k optimalizaci již existujících funkcí debuggeru pro snížení režie ladění.

5.1 Zásobník volání

Základem pro všechna rozšíření debuggeru o nové krokovací příkazy je využití vizualizace zásobníku volání a následná modifikace běhu simulace.

Po načtení spustitelné aplikace zpracuje debugger ladicí informace z DWARF sekcí uložených přímo v ELF souboru aplikace. Pro načítání těchto informací v binární formě je využita knihovna `libdwarf`, která odstiňuje vývojáře od binárního kódování DWARF informací (viz. kapitola 3.3). Tyto informace obsahují seznamy *FDE* a *CIE* záznamů, které jsou následně uloženy v knihovně `dwarfapil` pro pozdější využití. Při zastavení simulátoru je zrekonstruován zásobník volání podle aktuálního stavu simulátoru s pomocí adresy aktuální instrukce, hodnot registrů, zásobníku a načtených CFI.

Každý rámec obsahuje informace pro návrat programu do předchozího rámce (zálohy registrů, adresa instrukce předchozího rámce, adresa předchozího rámce na zásobníku), protože na začátku běhu funkce daného rámce jsou v prologu zálohovány obsahy registrů. Této vlastnosti je využito v architektuře reprezentace zásobníku volání v debuggeru.



Obrázek 5.1: Architektura reprezentace zásobníku volání.

Jednotlivé rámce jsou uspořádány v obousměrném seznamu, aby mohl každý rámec získat předchozí (volající) nebo následující (volaný) rámec. Pokud daný rámec potřebuje získat informace o hodnotě svých registrů, předá tento požadavek svému následujícímu rámci, který poté na základě CFI a svých uložených záloh registrů vrátí zpět požadovanou hodnotu. Aby tento princip nebyl narušen, je pro následující záznam nejvrchnějšího (nejnovějšího) záznamu využita zarážka, která pro požadavky nejvrchnějšího záznamu vrací aktuální hodnoty registrů ze simulátoru.

Z důvodu zmenšení režie debuggeru je pro každý rámec využita cache ukládající hodnoty registrů vrácených z následujícího rámce, protože by pro každý nový požadavek obnovy registru mohl být dotazován následující rámec, který by mohl dotazovat svůj následující rámec, atd.

Podle diagramu na obrázku 5.1 obsahuje každý rámec následující informace:

1. Ukazatel na adresu instrukce v programu, kde se daný rámec nachází. Pro nejvrchnější rámec se jedná o adresu aktuální instrukce programu, pro předchozí rámce o adresu, kde došlo k volání funkce následujícího rámce.

2. CFA — tj. adresa aktivačního rámce na zásobníku, jejíž výpočet je uložen v CFI.

3. Ukazatel na předchozí a následující rámec.

4. Ukazatel na vlákno procesu aplikace, ke kterému daný rámec patří,

5. Cache hodnot registrů získaných z následujícího rámce.

Tyto informace jsou uloženy v objektu třídy `CFrame`, který popisuje jednotlivý rámec. Tento objekt poté debugger získává z objektu `CThread` popisující jednotlivé vlákno simulátoru. Každé vlákno totiž provádí svůj kód nezávisle, takže využívá svůj vlastní zásobník volání.

Tato reprezentace zásobníku volání v podobě obousměrného seznamu objektu třídy `CFrame` je vytvářena při každém pozastavení simulátoru. Pro snížení režie je však vždy vytvořen pouze nejvrchnější rámec a předcházející rámce až při prvním dotazu na rozvinutí jejich následujícího rámce. Každý takto rozvinutý objekt třídy `CFrame` je poté uložen jako prvek obousměrného seznamu zásobníku volání, takže při příštím požadavku jej nemusí debugger znovu vytvářet.

Debugger velice často potřebuje získat vizualizaci kompletního zásobníku volání, například při výpisu všech rámců příkazem `backtrace`. Pro rozvinutí kompletního zásobníku volání je iterativně získáván předchozí rámec k aktuálnímu rámci, až k nejstaršímu rámci, který již svůj předchozí rámec nemá. Tímto vznikne kompletní vizualizace zásobníku volání, která je následně vrácena uživateli.

5.1.1 Zarážka

Pro zjednodušení rozhraní a sjednocení algoritmu tvorby vizualizace zásobníku volání, je využit princip *zarážky* (anglicky *sentinel*).

Standardně získává rámec hodnoty svých registrů dotazem následujícího rámce. Avšak toto neplatí pro nejvrchnější (nejnovější) rámec, který hodnoty registrů získává přímo ze simulátoru.

Proto je vytvořen nultý skrytý rámec, který je sám sobě následujícím rámcem a jeho předchozím rámcem je nejvrchnější rámec. Pokud poté tento nejvrchnější rámec požádá zarážku o hodnotu svých registrů, nebude zarážka tento požadavek šířit dále, ale načte hodnotu přímo ze simulátoru.

Zároveň tato zarážka vytváří reprezentaci nejvrchnějšího rámce jako svého předchozího rámce s pomocí aktuální adresy instrukce simulátoru (často také čítač instrukcí, anglicky *program counter*, vždycky přítomný registr).

5.1.2 Aktivní zásobník volání

Jak bylo popsáno v kapitole 2.3.1, zásobník volání může být aktivní nebo pasivní. Původní implementace debuggeru obsahovala pasivní zásobník, protože zobrazovala pouze jediný rámec, takže nebylo možné přepnout zásobník volání na jiný rámec. S rozšířením funkcionality debuggeru v rámci této diplomové práce však bylo možné implementovat i aktivní zásobník volání.

Základním principem je obnovení stavu laděného programu (v případě projektu Lissom simulátoru) do stavu v předchozím rámci. Zároveň však musí být tento nový stav reverzibilní, aby mohlo dojít k dalšímu běhu programu v nejvrchnějším rámci. Přepínání mezi jednotlivými rámci nesmí jakýmkoli způsobem ohrozit běh samotného programu (viz. Heisenbergův princip v kapitole 2).

Kompilátor jazyka C projektu Lissom negeneruje úplně validní CFI informace při zapnutých optimalizacích zdrojového kódu (parametr kompilace `-O1` až `-O3`). Debugger tedy není vždy schopen správně vizualizovat zásobník volání, čímž narušuje princip pravdivosti (viz. kapitola 2). Proto při chybách během vizualizace zásobníku volání debugger zobrazuje pouze nejvrchnější rámec.

Kompilátor projektu Lissom však generuje dostačující CFI informace pro ladění kódu bez optimalizací, takže lze v rámci ukázky na CD demonstrovat chování aktivního zásobníku.

Pro navržení spolehlivé změny stavu simulátoru a jeho opětovné navrácení do původního stavu byla využita stejná technika využití cache jako pro ukládání hodnot z registrů z následujících rámců (viz. kapitola 5.1). Při požadavku na obnovu simulátoru do stavu v daném rámci, uloží objekt třídy `CFrame` daného rámce aktuální stav simulátoru — hodnoty všech jeho registrů včetně registru adresy aktuální instrukce (tzv. *program counter*). Následně nahradí hodnoty všech registrů simulátoru hodnotami registrů v daném rámci. Tyto hodnoty jsou získány rozvinutím zásobníku volání pomocí CFI informací.

Pokud uživatel požaduje přepnutí na jiný rámec poté, kdy byl zásobník volání již přepnut na jiný rámec, je nejdříve stav simulátoru obnoven a až následně dojde k novému nahrazení stavu simulátoru za stav v požadovaném rámci. Tímto přístupem je zajištěna trvalá korektnost hodnot registrů v simulátoru.

Pokud bude následně uživatel vyžadovat opětovné spuštění programu, debugger informuje objekt třídy `CFrame`, aby opět obnovil původní stav simulátoru. Až poté je obnoven běh laděného programu.

5.2 Návrh krokovacích příkazů

V původní implementaci debuggeru již byl příkaz *step into* podporován. Debugger umožňuje krokování podle různých úrovních pohledu na laděný program [31] — po řádcích zdrojového C kódu, po instrukcích assembleru nebo přímo po jednotlivých cyklech simulace. Při načtení laděného spustitelného souboru je načtena DWARF sekce `.debug_line` s informacemi o řádcích. Tyto informace jsou poté použity pro mapování mezi adresami instrukcí a řádkem v daném C souboru.

Původní implementace příkazu poté fungovala v následujících krocích:

1. uživatel požádá přes rozhraní debuggeru o provedení kroku *step into* (v klientovi příkazové řádky příkazem `step`, v grafickém rozhraní pomocí daného příkazu z menu nebo tlačítka),
2. debugger po přijetí daného příkazu uloží řádek a C soubor pro aktuální instrukci zjištěný pomocí řádkových informací (viz. kapitola 3.4),
3. simulátor je přepnut do krokovacího módu a opět spuštěn,
4. simulátor provede jedinou instrukci a informuje debugger voláním obslužné rutiny `HandleDebugger` (rozhraní popsáno v kapitole 4.2.1),
5. debugger zjistí řádek a soubor pro adresu nové instrukce, porovná jej s řádkem a souborem uloženém v kroku 2, čímž může dojít ke dvěma stavům:
   (a) aktuální řádek se liší od uloženého — příkaz *step into* je tedy ukončen, simulátor pozastaven a uživatel informován.
   (b) aktuální řádek je shodný s uloženým — příkaz *step into* je opět spuštěn, běh simulátoru je opět obnoven a algoritmus pokračuje od kroku 4.

Původní implementace příkazu *step into* však neřešila prolog funkcí (pojem vysvětlen v kapitole 3.4). C kompilátor projektu Lissom totiž negeneruje úplně validní CFI a ladicí informace, kompilátor předpokládá, že uživatel nebude krokovat program v prologu a epilogu,

pouze až po nastavení registrů pro danou funkci. Proto je při krokování zjištěna aktuální funkce pomocí ladicích informací v sekci `.debug_info` a následně je zjištěna adresa konce prologu. Debugger poté při krokování tento prolog přeskakuje.

5.2.1 Rozšíření bodů přerušení

Při implementaci složitějších příkazů debuggeru jsou potřebné interní body přerušení. Tyto body přerušení nejsou viditelné uživatelem, jsou vytvářeny a spravovány debuggerem samotným.

V debuggeru projektu Lissom správu bodů přerušení řeší *manažer bodů přerušení*, který je implementován třídou `CBreakpointManager`. Tento manažer umožňuje vytváření bodů přerušení a datových bodů přerušení na dané adrese, úpravu jejich vlastností a také mapování mezi logickým a fyzickým bodem přerušení (vysvětlení v kapitole 2.2.1). Manažer také řeší vyhledání zasažených bodů přerušení a vyhodnocování jejich podmínek. Bylo navrženo rozšíření tohoto manažeru o novou funkci `CreateInternBreakpoint` včetně úpravy rozhraní debuggeru o filtrování interních bodů přerušení, aby nebyly zasílány zpět uživateli (pro výpis uživatelských bodů přerušení je v rozhraní GDB/CLI používán příkaz `info breakpoints`).

V rámci úprav manažeru bodů přerušení byla taktéž přidána speciální vlastnost logického bodu přerušení, která umožňuje specifikovat obslužnou rutinu, která je volána při zasažení daného bodu přerušení. V této metodě lze poté implementovat speciální chování, které přepíše standardní chování bodu přerušení. Standardně totiž debugger při zasažení bodu přerušení pozastaví simulátor a informuje uživatele. Tento přístup umožňuje pokročilejší využití bodů přerušení, například v rámci implementace příkazu *step over*.

5.2.2 Step return

Tento příkaz je základním a nejjednodušším příkazem debuggeru využívající CFI. Pro tento příkaz musí být simulátor pozastavený. Po spuštění příkazu *step return* je běh programu opět obnoven a zastaven po ukončení aktuální funkce a návratu do místa volání této funkce. Přesněji je simulátor zastaven na instrukci následující za instrukcí volání aktuální funkce. Příkaz *step return* je spuštěn pomocí příkazu `finish` v GDB/CLI rozhraní nebo `-exec-finish` v rozhraní GDB/MI.

Tohoto chování je docíleno nastavením interního bodu přerušení na návratovou adresu aktuálního rámce. Poté je simulátor znovu spuštěn a po zasažení nově vytvořeného bodu přerušení znovu pozastaven. Z pohledu uživatele dojde k dokončení výpočtu aktuální funkce a vrácení k předchozí funkci, přičemž na zásobníku volání je zrušen nejvrchnější rámec.

Jádrem rozšíření je získání návratové adresy funkce aktuálního rámce. Ta je získána jako adresa instrukce předchozího rámce k nejvrchnějšího rámci. Proto stačí rozvinout aktuální rámec pomocí algoritmu popsaném v kapitole 6.4 a využít získanou reprezentaci předchozího rámce.

5.2.3 Step over

Příkaz *step over* je nejsložitějším příkazem pro kontrolu běhu programu z debuggeru. Spojuje sémantiku standardního krokování *step into* a příkazu *step return*. Příkaz *step over* je spuštěn pomocí příkazu `next` v GDB/CLI rozhraní nebo `-exec-next` v rozhraní GDB/MI.

Pro zahájení příkazu *step over* musí být simulátor pozastaven. Po spuštění příkazu *step over* je běh programu opět obnoven a zastaven po dosažení následujícího řádku v zdrojovém

kódu. Pokud při provádění instrukcí aktuálního řádku dojde k volání další funkce, je nejdříve tato funkce ukončena (provede se příkaz *step return*) a opět se pokračuje v provádění *step over*. Z pohledu uživatele debuggeru dojde k provedení aktuálního řádku a všech volání funkcí z tohoto řádku zdrojového kódu.

Základem příkazu *step over* je detekce volání funkce. Tato funkcionalita je řešena přepnutím simulátoru do krokovacího módu, kdy simulátor informuje debugger po provedení jediné instrukce. V debuggeru je poté prováděna kontrola aktuálního rámce, při které může dojít ke dvěma možnostem:

1. *Nedošlo ke změně rámce* — Simulátor je opět spuštěn v krokovacím módu a provede další simulační cyklus.

2. *Došlo ke změně rámce* — Bude proveden příkaz *step return* pro vykročení z nově zavolané funkce. Po ukončení příkazu *step return* bude provádění příkazu *step over* pokračovat, dokud se nezmění aktuální řádek zdrojového kódu.

5.2.4 Řídicí inferior

Generický debugger projektu Lissom musí být schopen ovládat i více procesů zároveň, protože v rámci projektu lze modelovat i víceprocesorové systémy v podobě platformy (viz. kapitola 4). V rámci debuggeru každý proces modeluje *inferior*, který spravuje simulátor daného procesoru a na něm běžící program včetně jeho vláken. V aktuální verzi projektu Lissom pracují všechny simulátory v režimu *all-stop* — provádějí společně jednotlivé cykly celé platformy, takže při zastavení jediného simulátoru je zastaven běh celé platformy.

Tento přístup však přinesl problém v podobě krokování v různých aplikacích. Obecně může platforma obsahovat více simulátorů různých procesorů, na kterých běží rozdílné aplikace. Při požadavku na provedení kroku od uživatele může poté tento krok trvat rozdílný počet cyklů.

Například máme platformu obsahující 2 různé procesory — P1 a P2. Na těchto procesorech běží různé aplikace A1 a A2. Uživatel si přeje provést příkaz *step over*. V aplikaci A1 dojde k volání nové funkce, příkaz *step over* provede vykročení z této funkce a následně zastaví simulátor P1, celkem například 50 cyklů simulace. V aplikaci A2 dojde k provedení jen jediného řádku kódu, celkem například 5 cyklů. Všechny simulátory ale simulují současně takt za taktem, čímž vzniká spor v počtu cyklů, který by daný příkaz *step over* měla provést.

Řešením tohoto problému je nastavení řídicího inferioru při spuštění příkazu debuggeru. Tento inferior je nastaven na aktuálně vybraný inferior při spuštění příkazu — v rozhraní GDB/CLI příkaz `inferior číslo`, v rozhraní GDB/MI je řídicí inferior zasílaný v rámci příkazu (pro *step over* syntaxe `-exec-next --thread-group číslo_inferioru`). Funkcionalita daného příkazu je poté prováděna pouze na řídicím inferioru (například pro příkaz *step over* detekce zavolání funkce, vykračování z funkce, aj.) a ostatní inferiory (tj. i jim odpovídající simulátory) neprovádějí žádné příkazy a pouze simulují cykly v synchronizaci s řídicím inferiorem.

5.3 Historie hodnot

Při návrhu příkazu *step return* vznikl požadavek na zobrazení návratové hodnoty funkce, ze které je příkaz proveden. Visual Studio Debugger umožňuje zobrazení návratové hodnoty v seznamu proměnných pod jménem dané funkce. GDB na druhé straně neupravuje samotné

proměnné, ale obsahuje robustnější funkcionalitu *Value History* [10] (překlad z angl. *historie hodnot*).

Výstupy příkazů pro vypisování hodnot proměnných (`print` v rozhraní GDB/CLI) a návratové hodnoty funkcí při provedení příkazu *step return* jsou ukládány do historie hodnot. Uživatel k nim může přistupovat pomocí výrazů začínajících dolarem a číslem. Uživatel může použít také výraz `$` pro přístup k poslední uložené hodnotě.

Debugger projektu Lissom však nebyl schopný zjistit, na jaké adrese, případně v jakých registrech, je uložena návratová hodnota funkce. Debugger GDB řeší tento problém speciálním nastavením pro danou architekturu a funkci. Tento přístup však nemohl být použit v projektu Lissom, protože ten se zabývá návrhem právě těchto architektur a proto musí debugger podporovat všechny možné kombinace. Z tohoto důvodu byl kompilátor jazyka C rozšířen o generování nových pomocných direktiv pro dva základní typy uložení návratové hodnoty funkce:

1. V registru (nebo i registrech) — návratová hodnota je uložena v jednom nebo více registrech. Kompilátor generuje na začátku prologu každé funkce pomocnou direktivu `codasip_retval_regs` následovanou čísly registrů. Více registrů je využíváno při návratu struktury z funkce, protože kompilátor může rozložit složky této struktury do registrů a nepoužít následující druhý způsob návratu hodnoty. K tomu nejčastěji dochází při zapnutí pokročilejších optimalizací při kompilaci.

2. V paměti — návratová hodnota (v tomto případě často struktura) je uložena v paměti na adrese uložené v registru. Kompilátor generuje pro funkci pomocnou direktivu `codasip_retstruct_reg` následovanou číslem registru, který obsahuje adresu v paměti (většinou se jedná o adresu na zásobníku).

Assembler projektu Lissom poté využije tyto pomocné direktivy a vytvoří novou binární sekci `.codasip_retval` v ELF spustitelném souboru. Tato sekce obsahuje binárně zakódované informace o daných funkcích. Identifikace jednotlivých funkcí je v této sekci řešena pomocí uložení rozpětí adres, které daná funkce zabírá v programu.

Při návrhu podpory historie hodnot v debuggeru Lissom byl vytvořen *manažer historie* implementovaný pomocí třídy `CHistoryManager`. Debugger informuje tento manažer po dokončení příkazu *step return*, kdy je na základě informací ze sekce `.codasip_retval` zkopírována návratová hodnota ze simulátoru do alokované paměti v manažeru. Toto řešení umožňuje zobrazovat návratovou hodnotu i když byly registry nebo paměť obsahující tuto hodnotu přepsány.

Kapitola 6

Implementace rozšíření

V této kapitole bude popsána implementace rozšíření debuggeru projektu Lissom. Zároveň budou představeny problémy včetně jejich řešení, které vyvstaly při implementaci a testování těchto rozšíření.

Projekt Lissom je implementován v jazyce C++ s podporou nástrojů operačního systému Linux (např. skripty v jazyce Bash), GNU projektu Binutils [5] a s pomocnou POSIX vrstvou MinGW [7] pro operační systém Windows.

6.1 Knihovna libdwarf

Protože je formát DWARF relativně složitý, vznikla knihovna `libdwarf` [14] v jazyce C, která nabízí přístupné rozhraní pro práci s DWARF formátem. Tato knihovna umožňuje generování i analýzu ladicích informací ve formátu DWARF [33] a je využívána v projektu Lissom.

6.2 Podpora ladění inicializace debuggeru

Při implementaci rozšíření vznikla potřeba ladit vytvářený debugger. Debugger GDB se umí navázat na již běžící proces (příkaz `attach`) a ladit tak jeho běh, avšak tímto přístupem nelze ladit začátek běhu procesu. Tento způsob nebyl dostačující, protože velká část zpracovávání ladicích informací probíhá hned při spuštění debuggeru po načtení aplikace.

Proto byl simulátor upraven přidáním nového parametru `--stop-on-start`, který způsobí zastavení simulátoru ihned po spuštění před inicializací hlavních objektů čekáním na POSIX signál `SIG_USR1`. Při čekání na signál lze poté navázat GDB na nově spuštěný proces simulátoru a následně simulátor opět spustit pomocí příkazu `kill -10 <pid procesu>` nebo `pkill -10 simulator`, kde `simulator` je jméno spustitelného souboru simulátoru. Pomocí této metody pak lze ladit inicializaci debuggeru, jeho interních struktur včetně načítání a analýzy ladicích informací, i když samotný simulátor byl spouštěn z grafického prostředí Eclipse.

6.3 Čísla registrů

Specifikace DWARF formátu [25] využívá jednotný přístup pro popis uložení proměnných, popis uložení položek struktur, případně výpočty adres v rámci CFI. Tyto popisy umístění

velice často pracují s registry. DWARF informace se však na registry odkazují číslem, přestože registry při popisu architektury mají často své jméno nebo v případě registrových polí jméno a velikost tohoto pole. Tento popis registrů je využit i v modelu procesoru v jazyce CodAL, který je používán v projektu Lissom.

Standardně je mapování mezi DWARF číslem registru a jeho skutečným jménem případně indexem obsaženo přímo ve specifikaci dané architektury, jako například u procesorů ARM [15]. Kompilátor i debugger poté může využívat toto pevně dané mapování.

V projektu Lissom lze však modelovat různorodé architektury s různými registry, proto muselo být využito robustnější řešení. Kompilátor jazyka C projektu Lissom tedy generuje toto mapování do speciální sekce `.codasip_regmap`. V této sekci jsou uloženy záznamy obsahující:

1. DWARF číslo registru,
2. jméno registru z modelu procesoru,
3. index registru, pokud dané jméno popisuje registrové pole, nebo 0.

Tyto informace jsou poté assemblerem přeloženy do binární podoby a uloženy ve speciální sekci `.codasip_regmap` spustitelného ELF programu. Při načítání ladicích informací je pak načteno i toto mapování. Knihovna `dwarfapil` popsaná v kapitole 4.2 poté překládá DWARF čísla registrů na jméno a index registru simulátoru, takže samotné jádro debuggeru pracuje pouze se skutečnými jmény registrů.

6.4 Vizualizace zásobníku volání

Implementace zásobníku volání je realizována s co nejmenším zásahem do jádra debuggeru. Základní rozhraní, které debugger využívá, je získání nejvrchnějšího rámce daného vlákna pomocí metody `GetTopFrame` třídy `CThread` popisující vlákno simulátoru. Tento nejvrchnější rámec lze poté rozvinout (tento princip je popsán v kapitole 3.5) pro získání předchozího rámce. Pokud rozvinutí vrátí hodnotu `NULL`, je tento rámec nejstarším nebo došlo k chybě při rozvinutí. Zároveň tento princip umožňuje odstínění debuggeru od chybějících CFI informací, poté nejvrchnější rámec nelze rozvinout a vizualizace zásobníku volání obsahuje jen tento jediný nejvrchnější rámec.

Základní částí implementace vizualizace zásobníku volání je získání záloh registrů ze svého následujícího (tj. novějšího rámce). Pokud tento následující rámec obdrží požadavek pro vrácení hodnot registru od svého předchozího rámce, provede následující kroky:

1. Je vytvořen požadavek na získání FDE a CIE záznamu pro adresu instrukce daného rámce (popis těchto záznamů je uveden v kapitole 3.5).
2. Pomocí knihovny `CDebugInfo` je získán soubor všech ladicích informací patřících k danému rámci a jemu odpovídajícímu vláknu procesu. Následně je požadavek předán knihovně `CDwarf` a `dwarfapil`.
3. Knihovna `dwarfapil` s pomocí knihovny `libdwarf` vyhledá FDE záznam pro zaslanou adresu instrukci. Při nalezení odpovídajícího FDE záznamu vyhledá i jemu odpovídající CIE záznam.
4. S pomocí funkce `dwarf_get_fde_info_for_all_regs3` [33] jsou získány výpočetní pravidla pro všechny registry, tj. řádek CFI (pravidla jsou blíže popsána v kapitole 3.5).

5. Následně jsou všechna získaná pravidla, včetně prvního pravidla pro výpočet hodnoty CFA, převedeny na lépe použitelnou formu a DWARF čísla registrů jsou převedena na odpovídající jména registrů v simulátoru.

6. Ve standardu DWARF je uložení návratová adresy funkce specifikováno speciálním registrem v CFI tabulce. Tento registr však může být i virtuální (neexistuje k němu odpovídající hardwarový registr). Číslo tohoto registru je uloženo v záznamu CIE přiřazeném k FDE záznamu získaném v kroku 3. Pravidlo pro výpočet, získané již v kroku 5, je poté uloženo separátně od běžných registrů pro pozdější využití.

7. Pravidla pro výpočet všech registrů, CFA i návratové adresy jsou poté společně uloženy do tabulky pravidel, která je vrácena rámci. Rámec si získanou tabulku pravidel uloží pro pozdější opětovné použití, čímž je snížena režie.

8. Velice často jsou zálohy registrů uloženy na zásobníku — je nutné znát CFA adresu (tj. adresa vrcholu zásobníku pro daný rámec). Z tohoto důvodu je v tabulce pravidel vyhledáno pravidlo pro výpočet CFA a daná adresa je vypočtena.

9. Nakonec rámec využije tabulku pravidel, vyhledá odpovídající záznam k požadovanému registru, dané pravidlo s pomocí získané CFA adresy vypočítá, čímž získá obsah požadovaného registru, který je následně zaslán zpět předchozímu rámci.

Druhou základní částí je poté rozvinutí rámce a získání reprezentace jeho předchozího rámce. Reprezentace zásobníku volání je popsána v kapitole 5.1. Tato část probíhá v následujících krocích:

1. Daný rámec získá tabulku pravidel, viz. výše v kroku 7.

2. V tabulce pravidel je nalezeno pravidlo pro výpočet uložení návratové adresy funkce, které je poté vypočteno. Velice často je pro výpočet nutná i hodnota CFA.

3. Nakonec je vytvořena reprezentace předchozího rámce s adresou instrukce rovnou návratové adrese funkce daného rámce.

4. Nově vytvořený rámec je poté vložen jako předchozí prvek obousměrného seznamu k danému rámci.

Pokud již daný rámec vytvořil reprezentaci předchozího rámce dříve, není výše popsaný postup použit a je přímo vrácena dříve získaná reprezentace předchozího rámce.

Druhé důležité rozhraní využívané debuggerem je invalidace zásobníku volání. Po provedení jediné instrukce simulátoru může dojít ke změně zásobníku volání, takže jeho vizualizace již není aktuální. Proto debugger volá metodu `InvalidateFrames` vlákna daného inferioru, která kompletně zruší celou vizualizaci zásobníku volání a uvolní alokovanou paměť. Při dalším přístupu na nejvrchnější rámec pomocí metody `GetTopFrame` dojde k opětovnému vytvoření reprezentace nejvrchnějšího rámce a z něj lze postupným rozvinutím získat novou reprezentaci celého zásobníku volání.

Protože však k volání `InvalidateFrames` může docházet z různých funkcí debuggeru, je využita optimalizace, která zabrání invalidaci zásobníku volání, pokud se aktuální počet cyklů simulátoru nezměnil. Tato optimalizace snižuje počet zbytečného znovuvytváření zásobníku volání, pokud nemohlo dojít k jeho změně.

6.5 Aktivní zásobník volání

V kapitole 5.1.2 je popsán princip a návrh aktivního zásobníku volání. V rámci implementace bylo upraveno rozhraní vlákna inferioru, aby umožňovalo měnit aktuální rámec v zásobníku volání daného vlákna. Debugger poté využívá toto upravené rozhraní při požadavku uživatele na změnu aktuálního rámce.

Samotné vlákno je zodpovědné za správnou manipulaci se stavem simulátoru. Při přepnutí na daný rámec volání dojde k volání metody `UnwindSimulator`, která zálohuje aktuální stav registrů simulátoru a nahradí jej stavem registrů v daném rámci. Při následné změně na jiný rámec dojde nejdříve k volání metody `RestoreSimulator` předchozího vybraného rámce, která opět vrátí simulátor do původního stavu před přepínáním rámců a až poté může být opět volána metoda `UnwindSimulator`.

6.6 Problém s rekurzivními funkcemi

Při implementaci a testování příkazu *step return* se objevil problém rekurzivních funkcí. Jak bylo popsáno v kapitole 5.2.2, při provádění příkazu *step return* je vytvářen interní bod přerušení. Tento bod přerušení umožňuje zastavení na daném řádku zdrojového kódu, avšak již nekontroluje, v jakém rámci došlo k zastavení. Zároveň se tento problém týká i příkazu *step over*, protože také využívá sémantiky příkazu *step return*.

```
1  int factorial(int n)
2  {
3    if (n == 0)
4      return 1;
5    else
6      return(n*factorial(n-1));
7  }
8
9  int main() {
10   factorial(5);
11 }
```

factorial(3):2
factorial(4):6
factorial(5):6
main():10

(a) Před *step return*

factorial(3):6
factorial(4):6
factorial(5):6
main():10

(b) Po *step return*

Obrázek 6.1: Příklad C programu.

Obrázek 6.1 obsahuje demonstraci problému na programu pro výpočet faktoriálu v jazyce C. Simulátor byl pozastaven na řádku 2 (začátek funkce) ve funkci `factorial` s parametrem `n` rovným 3. Vizualizace zásobníku volání je znázorněna na obrázku 6.1a (čísla za dvojtečkou znázorňují číslo řádku, kde se nachází aktuální instrukce daného rámce).

Uživatel debuggeru následně provede příkaz *step return*. Debugger potřebuje zjistit návratovou adresu funkce aktuálního rámce. Funkce `factorial(3)` byla volána z funkce `factorial(4)`, návratová adresa je tedy adresou instrukce na řádku 6. Proto debugger vytvoří interní bod přerušení na řádku 6 a opět spustí simulaci.

Funkce `factorial` s parametrem 3 běží dále, provádí řádky 3, 5 a 6 a následně je simulátor opět pozastaven po zasažení bodu přerušení na řádku 6 jak je znázorněno na obrázku 6.1b. I když došlo k zasažení správného bodu přerušení, nacházíme se na špatném rámci té samé funkce, takže příkaz *step return* nebyl proveden správně.

6.6.1 Řešení

Při zasažení interního bodu přerušení vytvořeného při spuštění příkazu *step return* je nutné kontrolovat, zda je nejnovější rámec požadovaným rámcem — tj. předchozí rámec při spuštění příkazu *step return*. Pokud dojde k zasažení bodu přerušení na nesprávném rámci, je bod přerušení znovu aktivován a simulátor opět spuštěn.

Pro implementaci tohoto rozšíření bylo nutné upravit implementaci bodů přerušení v debuggeru projektu Lissom, jak bylo popsáno v kapitole 5.2.1. Byla přidána podpora obslužné rutiny, která je zavolána při zasažení bodu přerušení. Internímu bodu přerušení vytvářením při provádění akce *step return* je poté přiřazena obslužná rutina provádějící kontrolu popsanou v předchozím odstavci, čímž je tento problém vyřešen.

6.7 Problém při vykročení z rámce

Při provádění příkazu *step over* může dojít k provádění jiné funkce, než ze které byl příkaz *step over* spuštěn, avšak přitom nebyla žádná nová funkce volána. Nejobvyklejším případem je provedení příkazu *step over* nad příkazem `return` jazyka C.

Tento příkaz jazyka C provede ukončení aktuální funkce s vrácením dané hodnoty. Pokud k tomuto ukončení dojde při běhu příkazu *step over*, je detekováno zavolání nové funkce a je spuštěn příkaz *step return* jak bylo popsáno v kroku 2 v kapitole 5.2.3. Toto chování pak má za následek nekorektní doběhnutí funkce předchozího rámce k rámci, odkud byl příkaz *step over* volán.

6.7.1 Řešení

Řešením tohoto problému je úprava spuštění příkazu *step over*. Během spuštění tohoto příkazu je uložena identifikace aktuálního rámce. Jako identifikace je použita adresa začátku rámce na zásobníku. Pokud poté dojde k detekci volání jiné funkce, je prohledána kompletní vizualizace zásobníku volání, zda uložený rámec ještě existuje, nebo již bylo jeho provádění ukončeno. Tuto funkcionalitu obstarává funkce `FindPrevFrameByStackAddress` objektu třídy `CFrame` reprezentujícím rámec, ze kterého je prohledávání spuštěno.

Pokud dojde k nalezení uloženého rámce, je provádění příkazu *step over* obnoveno a debugger provede příkaz *step return* pro návrat z nově volané funkce. Pokud však uložený rámec nalezen není, jako například v modelujícím případě, kdy příkaz `return` ukončí aktuální funkci a její rámec je odstraněn, dojde k ukončení příkazu *step over* a pozastavení simulace.

6.8 Souběžné krokování v assembler a C kódu

V průběhu kompilace zdrojového souboru v jazyce C v projektu Lissom dojde k vytvoření assembler souboru a následně jeho transformaci generovaným assemblerem do binární spustitelné podoby. Protože však v rámci modelu procesoru jsou specifikovány instrukce včetně jejich syntaxe v assembleru daného procesoru, není takto generovaný assembler soubor zahozen, ale je vrácen zpět klientovi.

Proto grafické rozhraní projektu Lissom *Studio* popsané v kapitole 4.1 umožňuje přepínání mezi zdrojovým C souborem a vygenerovaným assembler souborem, místo standardního výpisu instrukcí pomocí zpětného assembleru (tento přístup využívá běžný debugger v grafickém prostředí Eclipse).

Před implementací rozšíření:

```
*stopped,reason="breakpoint-hit",disp="keep",bkptno="1",
   frame={addr="0x19c",func="testik",args=[],
      file="main.c",fullname="....../main.c",line="19"
   },thread-id="0",stopped-threads="all"
```

Po implementaci rozšíření:

```
*stopped,reason="breakpoint-hit",disp="keep",bkptno="1",
   frame={addr="0x19c",func="testik",args=[],
      file="main.c",fullname="....../main.c",line="19",
      file2="main.asm",fullname2="main.asm",line2="172"
   },thread-id="0",stopped-threads="all"
```

Příklad 6.2: Úprava GDB/MI protokolu podporující dvojí řádkové informace.

Z tohoto důvodu obsahuje binární spustitelný soubor projektu Lissom dvojí informace o řádcích — první obsahují řádkování pro zdrojový C soubor ve standardní sekci `.debug_line`, druhé potom řádkování pro generovaný assembler soubor ve speciální sekci `.codasip_line`. V běžné komunikaci pomocí protokolu GDB/MI jsou však klientovi posílány pouze jediné řádkové informace — například číslo řádku zdrojového kódu, kde došlo k zasažení bodu přerušení (viz. první část příkladu 6.2). Studio muselo posílat nový dotaz pro získání řádkových informací o assembleru s pomocí standardního GDB/MI příkazu `-file-list-exec-source-file` [20] pro každou řádkovou informaci zdrojového C souboru, kterou přijalo.

Pro zvýšení efektivity komunikace bylo proto navrženo a implementováno rozšíření GDB/MI protokolu, které upravuje syntaxi zasílaných řádkových informací přidáním druhých řádkových informací označených číslem 2 (viz. druhá část příkladu 6.2). Pokud však aktuální adresa instrukce obsahuje pouze řádkové informace o assembleru (například při programování v assembleru bez kompilátoru C) jsou posílány pouze jediné řádkové informace. Zároveň toto rozšíření pořád umožňuje využití debuggeru projektu Lissom grafickým rozhraním vyžadujícím standardní GDB/MI rozhraní, protože jsou schopny ignorovat nepodporované informace (testováno na debuggeru grafického prostředí Eclipse).

Kapitola 7

Závěr

Cílem diplomové práce bylo seznámit se s rozhraním ladicího nástroje GNU Debugger (GDB), formátem ladicích informací DWARF a implementací generického ladicího nástroje v rámci projektu Lissom.

V textu této práce byl popsán obecný princip a architektura ladicích nástrojů, jejich klasifikace a způsob ladění na úrovni zdrojového kódu. Následně byl představen formát ladicích informací DWARF, který je využíván v rámci projektu Lissom, se zaměřením na ladicí informace CFI, které jsou nedílnou součástí vizualizace zásobníku volání. Poté byla popsána architektura projektu Lissom, především jeho generického ladicího nástroje. V rámci této práce byla prezentována rozšíření tohoto nástroje včetně jejich implementace a problémů, které se objevily při testování. Ladicí nástroj byl rozšířen o podporu nových příkazů *step over* a *step return*, možnost rozvinutí zásobníku volání funkcí včetně jeho aktivního přepínání, ukládání historie hodnot a korektní ladění při simulaci víceprocesorových systémů. Nakonec došlo i k optimalizaci a vylepšení již existujícího příkazu *step into* a GDB/MI zpráv debuggeru pro snížení režie ladění.

Problematika DWARF formátu a CFI informací byla přednesena na konferenci *Language theory with applications 2013* (*LTA 2013*) na Vysokém učení technickém v Brně.

Vývoj generického ladicího nástroje bude dále pokračovat se snahou se co nejvíce přiblížit funkcionalitě ladicího nástroje GDB. Implementace zásobníku volání lze nadále optimalizovat, například znovu vytvoření jen změněné části vizualizace zásobníku při pozastavení simulace oproti opětovnému znovuvytváření kompletní vizualizace. Zároveň po rozšíření generovaných simulátorů a o podporu více vláken bude nutné dále upravovat architekturu a implementaci ladicího nástroje pro korektní a efektivní práci. Nakonec může být historie hodnot rozšířena o podporu příkazu `print`, což by umožnilo snadnější používání mezivýsledků při výpočtu výrazů při ladění.

Ladicí nástroj projektu Lissom umožňuje ladění libovolné modelované architektury, avšak zároveň nabízí srovnatelnou funkcionalitu s běžně dostupnými a používanými nástroji. Díky neustálému rozvoji informačních systému však dochází ke zvyšování komplexity aplikací a jejich běhu na více výpočetních jednotkách různých architektur. Projekt Lissom umožňuje tyto architektury modelovat a upravovat podle aktuálních potřeb, generovat pro tyto modely specializované nástroje a simulovat samotný běh víceprocesorového systému. Ladicí nástroj musí s tímto vývojem držet krok, protože je základním nástrojem umožňujícím ladění chyb a testování aplikací v těchto systémech.

Literatura

[1] *Basic Debugging* [online]. [cit. 12. 5. 2013]. Dostupné na: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms679276(v=vs.85).aspx>.

[2] *CodAL Architecture Description Language* [online]. [cit. 30. 12. 2013]. Dostupné na: <http://www.codasip.com/products/codal/>.

[3] *Eclipse - The Eclipse Foundation open source community website* [online]. [cit. 19. 5. 2013]. Dostupné na: <http://www.eclipse.org/>.

[4] *GCC Exception Frame* [online]. [cit. 19. 5. 2013]. Dostupné na: <http://www.airs.com/blog/archives/166>.

[5] *GNU Binutils* [online]. [cit. 19. 5. 2013]. Dostupné na: <http://www.gnu.org/software/binutils/>.

[6] *Introduction to JTAG* [online]. [cit. 12. 5. 2013]. Dostupné na: <http://www.embedded.com/electronics-blogs/beginner-s-corner/4024466/Introduction-to-JTAG>.

[7] *MinGW — Minimalist GNU for Windows* [online]. [cit. 19. 5. 2013]. Dostupné na: <http://www.mingw.org/>.

[8] *Popis nástroje Dr. Watson pro systém Windows (Drwtsn32.exe)* [online]. [cit. 11. 5. 2013]. Dostupné na: <http://support.microsoft.com/kb/308538/cs>.

[9] *Ptrace(2): process trace - Linux man page* [online]. [cit. 12. 5. 2013]. Dostupné na: <http://linux.die.net/man/2/ptrace>.

[10] *Value History - Debugging with GDB* [online]. [cit. 19. 5. 2013]. Dostupné na: <https://sourceware.org/gdb/onlinedocs/gdb/Value-History.html>.

[11] *What is the difference between Emulator vs Simulator ?* [online]. [cit. 19. 5. 2013]. Dostupné na: <http://hemantcnb.blogspot.cz/2013/08/what-is-difference-between-emulator-vs.html>.

[12] *The DWARF Debugging Standard* [online]. 2007–2013 [cit. 30. 12. 2013]. Dostupné na: <http://www.dwarfstd.org/>.

[13] *GNU linker ld* [online]. 2009 [cit. 30. 12. 2013]. Dostupné na: <https://sourceware.org/binutils/docs-2.21/ld/>.

[14] *Libdwarf - DWARF debugging information* [online]. 2009 [cit. 30. 12. 2013]. Dostupné na: <`http://libdwarf.sourceforge.net/`>.

[15] *DWARF for the ARM®Architecture* [online]. Listopad 2012. Dostupné na: <`http://infocenter.arm.com/help/topic/com.arm.doc.ihi0040b/IHI0040B_aadwarf.pdf`>.

[16] *Bell Labs* [online]. 2013 [cit. 30. 12. 2013]. Dostupné na: <`http://www.alcatel-lucent.com/bell-labs`>.

[17] *Debugging with gdb* [online]. 2013 [cit. 30. 12. 2013]. Dostupné na: <`http://www.sourceware.org/gdb/onlinedocs/gdb.html`>.

[18] *PDB Files (C++)* [online]. 2013 [cit. 30. 12. 2013]. Dostupné na: <`http://msdn.microsoft.com/en-us/library/yd4f8bd1(v=vs.90).aspx`>.

[19] *GDB: The GNU Project Debugger* [online]. 2013-12-08 [cit. 30. 12. 2013]. Dostupné na: <`https://www.gnu.org/software/gdb/`>.

[20] *GDB/MI – Debugging with GDB* [online]. 2013-12-08 [cit. 19. 5. 2013]. Dostupné na: <`https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html`>.

[21] *GCC, the GNU Compiler Collection* [online]. 2013-12-20 [cit. 30. 12. 2013]. Dostupné na: <`http://gcc.gnu.org/`>.

[22] Coutant, D., Hammond, C. a Kelley, J. Compilers for the New Generation of Hewlett-Packard Computers. *Hewlett-Packard Journal*. 1986. S. 4–18.

[23] Eager, M. J. *Introduction to the DWARF Debugging Format* [online]. 2012 [cit. 30. 12. 2013]. Dostupné na: <`http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf`>.

[24] Fowler, M. *Refactoring: Improving the Design of Existing Code*. [b.m.]: Addison-Wesley, 1999. ISBN 0-201-48567-2.

[25] Free Standards Group. *DWARF Debugging Information Format, Version 3* [online]. 2005. Dostupné na: <`http://dwarfstd.org/doc/Dwarf3.pdf`>.

[26] Gramlich, W. C. Debugging Methodology: Session Summary. In *Proceedings of the Symposium on High-level Debugging*. [b.m.]: ACM, 1983. SIGSOFT '83. ISBN 0-89791-111-3.

[27] Hayes, B. The Information Age: The Discovery of Debugging. *The Sciences*. 1993, roč. 33, č. 4. S. 10–13.

[28] HRUŠKA, T. *Lissom* [online]. [cit. 30. 12. 2013]. Dostupné na: <`http://www.fit.vutbr.cz/research/groups/lissom`>.

[29] Korvas, P. *Rekonfigurovatelný ladicí nástroj na úrovni zdrojového kódu*. Vysoké Učení Technické v Brně, 2013. Diplomová práce.

[30] Menapace, J., Kingdon, J. a MacKenzie, D. *The "stabs"representation of debugging information* [online]. 1992–2013 [cit. 30. 12. 2013]. Dostupné na: <`https://sourceware.org/gdb/current/onlinedocs/stabs.html`>.

[31] Přikryl, Z. *Advanced Methods of Microprocessor Simulation.* Vysoké Učení Technické v Brně, 2011. Disertační práce.

[32] Rosenberg, J. B. *How Debuggers Work: Algorithms, Data Structures, and Architecture.* [b.m.]: Wiley Computer Publishing, 1996. ISBN 0-471-14966-7.

[33] UNIX International Programming Languages Special Interest Group. *A Consumer Library Interface to DWARF* [online]. 2002. Dostupné na: <`ftp://ftp.software.ibm.com/software/os390/czos/dwarf/libdwarf2.1.pdf`>.

[34] Wilczák, M. *Ladicí nástroj generických simulátorů procesorů.* Vysoké Učení Technické v Brně, 2010. Diplomová práce.

Příloha A

Obsah CD

Přiložené CD obsahuje:

1. Adresář `doc` obsahuje tuto technickou zprávu se zdrojovým souborem v LaTeXu.
2. Adresář `sim` obsahuje generované C++ soubory pro kompilaci simulátorů procesorů codix (32bit) a adop (16bit).
3. Adresář `app` obsahuje zdrojové C soubory včetně zkompilovaných spustitelných aplikací pro oba modelované procesory.
4. Adresář `src` obsahuje zdrojové soubory pro generický debugger projektu Lissom.
5. Adresář `inc` obsahuje vyžadované hlavičkové soubory projektu Lissom.
6. Adresář `lib` obsahuje předem zkompilované knihovny projektu Lissom.

Pro vytvoření spustitelných simulátoru spusťte příkaz `make`. Je podporován pouze 64bitový operační systém Linux a 32 i 64bitový operační systém Windows.

Pro kompilaci je vyžadována knihovna OpenSSL, pthread, libedit a zlib.

Pro kompilaci a běh na operačním systému Windows musí být nainstalováno MinGW (viz. [7]) s požadovanými knihovnami. Platformu MinGW lze nainstalovat i pomocí instalátoru na CD.

CD obsahuje dvě testovací aplikace přeložené pro oba procesory:

1. `fact.xexe` — jednoduchá aplikace pro testování rekurzivních funkcí a zásobníku volání,
2. `bitcnt.xexe` — aplikace pro testování výkonnosti, která sčítá počet nastavených bitů v poli.

Pro každou z aplikací je přiložen i zdrojový C soubor (`.c`), assembler soubor (`.asm`) a inicializační assembler soubor (`loader.s`).

Simulace procesoru je nakonec spuštěna pomocí příkazu:

1. pro procesor codix `simulator-codix --input app/codix/fact.xexe`,
2. pro procesor adop `simulator-adop --input app/adop/fact.xexe`.

Příloha B

Příklad výpisu DWARF CFI

Zdrojový C soubor:

```
int test() {
   register int c = 123;
   register int d = -15;
   return c+120;
}

int main( int argc, const char* argv[] ) {
   register int a = 23;
   register int b = a+12;
   a = test();
   return a+b+22;
}
```

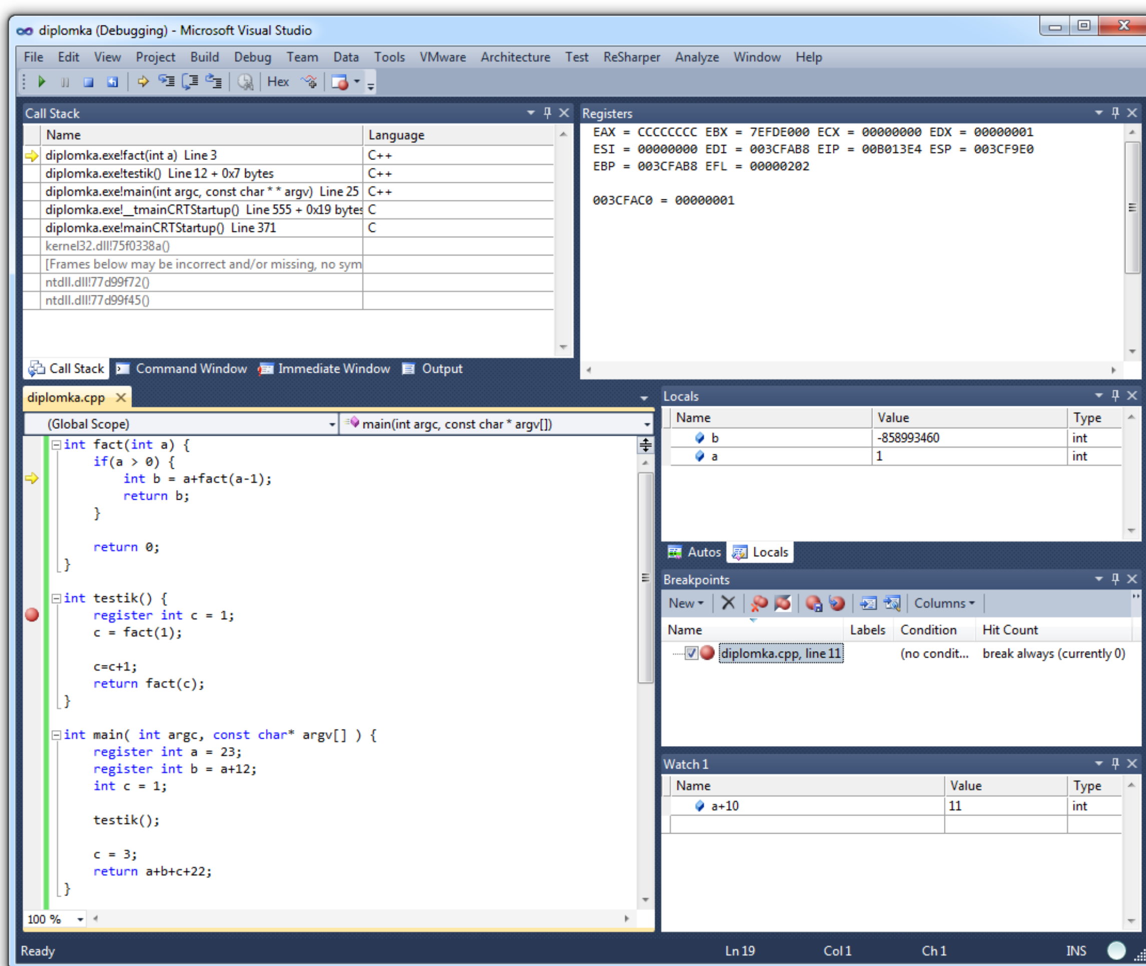
Část odpovídajícího výstupu `objdump -g` s CFI informacemi:

```
Contents of the .debug_frame section:

00000000 0000000c ffffffff CIE
  Version:               4
  Augmentation:          ""
  Pointer Size:          4
  Segment Size:          0
  Code alignment factor: 1
  Data alignment factor: 1
  Return address column: 30

  DW_CFA_nop

00000010 00000040 00000000 FDE cie=00000000 pc=0000008c..000000e8
  DW_CFA_same_value: r30
  DW_CFA_def_cfa_sf: r1 ofs 0
  DW_CFA_advance_loc4: 4 to 00000090
  DW_CFA_def_cfa_offset_sf: 8
  DW_CFA_advance_loc4: 8 to 00000098
```

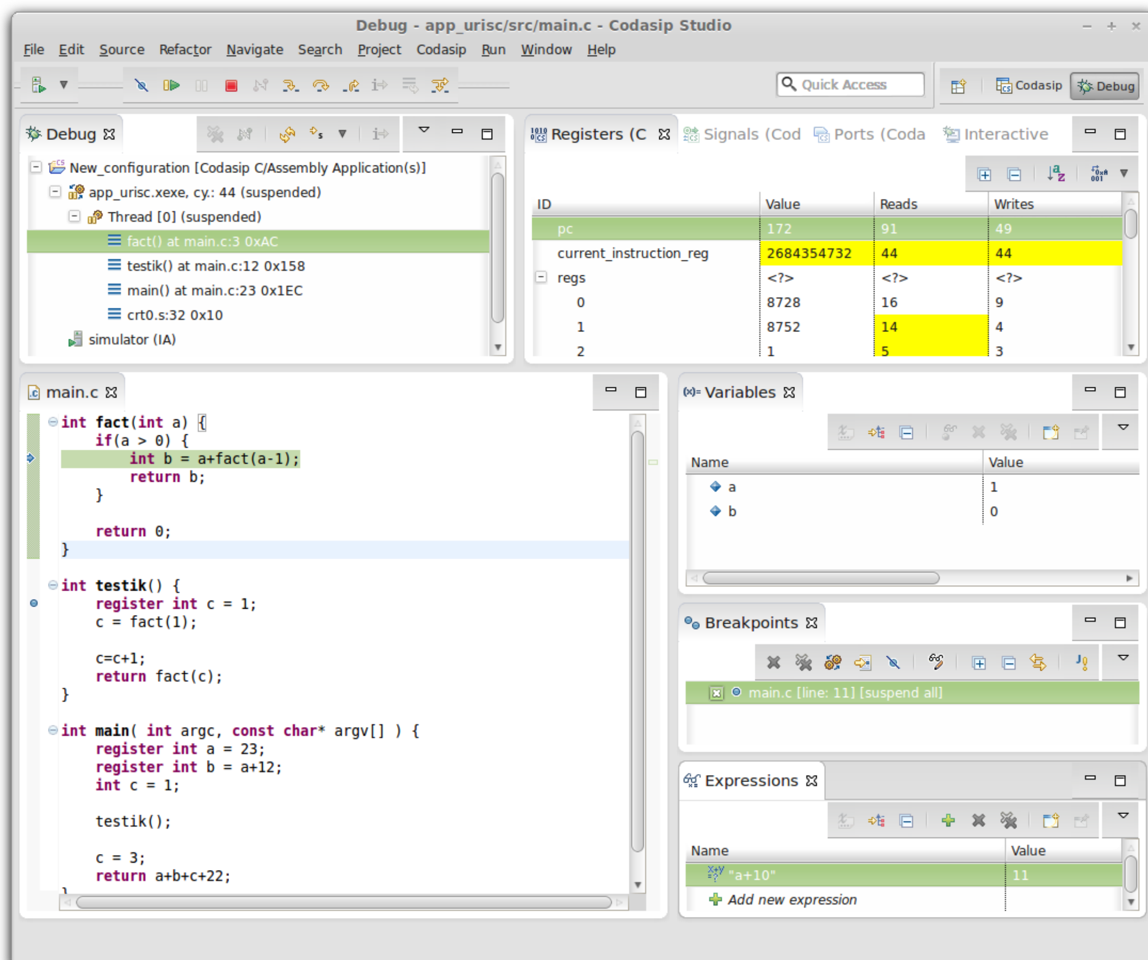
```
  DW_CFA_offset_extended_sf: r30 at cfa-4
  DW_CFA_advance_loc4: 4 to 0000009c
  DW_CFA_offset_extended_sf: r2 at cfa-8
  DW_CFA_advance_loc4: 4 to 000000a0
  DW_CFA_def_cfa_sf: r2 ofs 8
  DW_CFA_advance_loc4: 60 to 000000dc
  DW_CFA_def_cfa_sf: r1 ofs 8
  DW_CFA_advance_loc4: 8 to 000000e4
  DW_CFA_def_cfa_offset_sf: 0
  DW_CFA_nop

00000054 0000000c ffffffff CIE
  Version:               4
  Augmentation:          ""
  Pointer Size:          4
  Segment Size:          0
  Code alignment factor: 1
  Data alignment factor: 1
  Return address column: 30

  DW_CFA_nop

00000064 00000050 00000054 FDE cie=00000054 pc=000000e8..0000016c
  DW_CFA_def_cfa_sf: r1 ofs 0
  DW_CFA_same_value: r30
  DW_CFA_advance_loc4: 4 to 000000ec
  DW_CFA_def_cfa_offset_sf: 8
  DW_CFA_advance_loc4: 8 to 000000f4
  DW_CFA_offset_extended_sf: r30 at cfa-4
  DW_CFA_advance_loc4: 4 to 000000f8
  DW_CFA_offset_extended_sf: r2 at cfa-8
  DW_CFA_advance_loc4: 4 to 000000fc
  DW_CFA_def_cfa_sf: r2 ofs 8
  DW_CFA_advance_loc4: 12 to 00000108
  DW_CFA_offset_extended_sf: r3 at cfa-16
  DW_CFA_advance_loc4: 4 to 0000010c
  DW_CFA_offset_extended_sf: r4 at cfa-20
  DW_CFA_advance_loc4: 84 to 00000160
  DW_CFA_def_cfa_sf: r1 ofs 8
  DW_CFA_advance_loc4: 8 to 00000168
  DW_CFA_def_cfa_offset_sf: 0
  DW_CFA_nop
```

Příloha C

Příklady rozhraní debuggerů



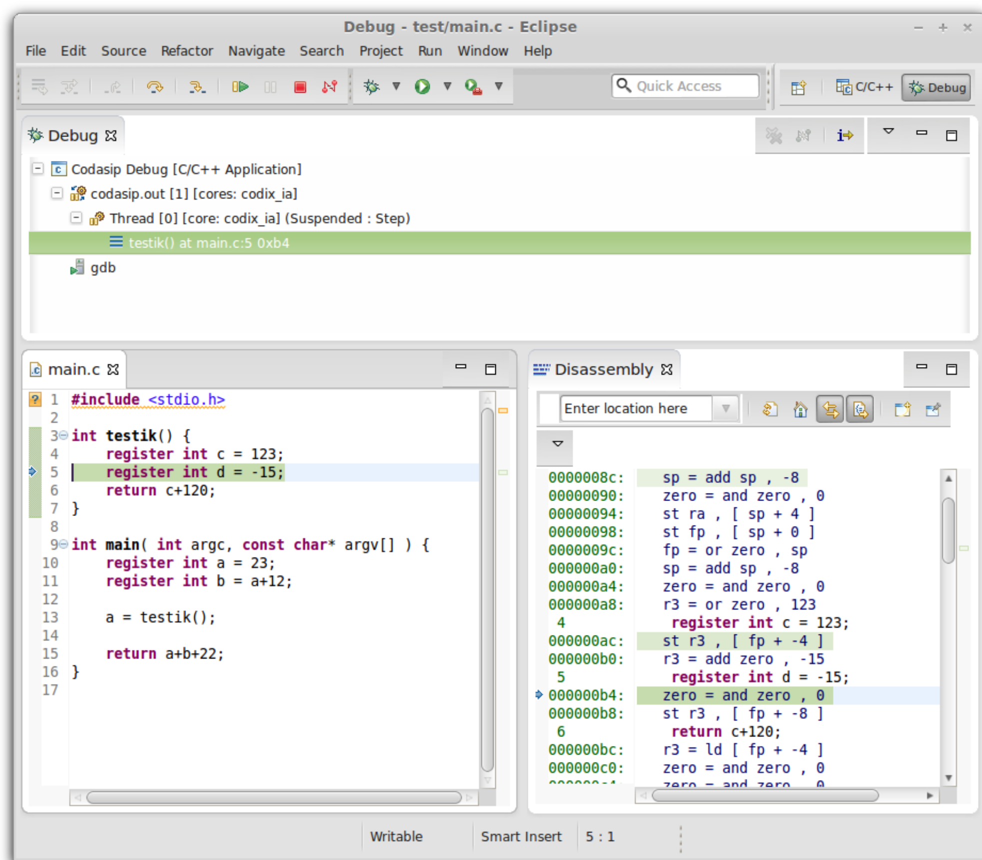
Obrázek C.1: Grafické rozhraní Microsoft Visual Studio 2010 debuggeru.



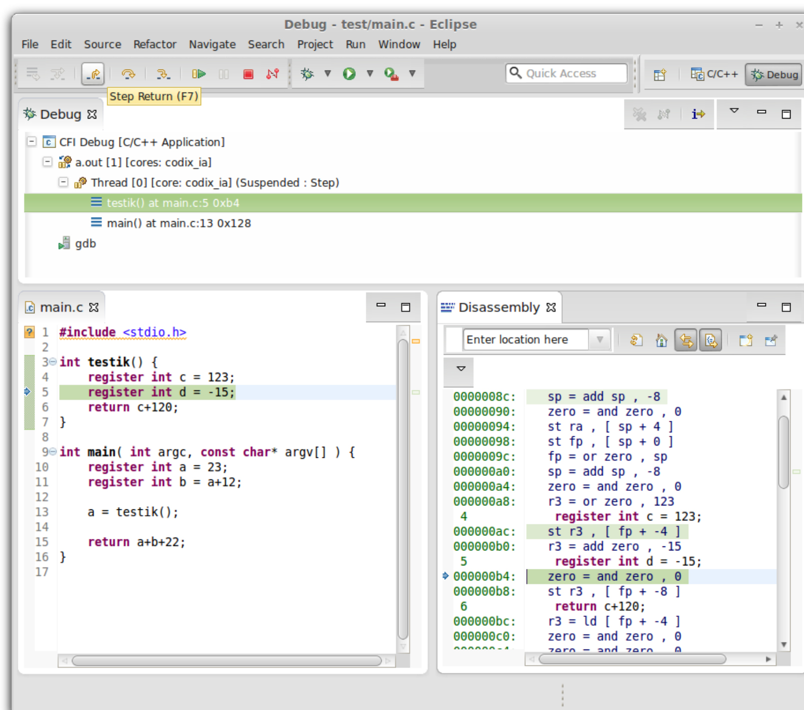
Obrázek C.2: Grafické rozhraní Codasip Studio debuggeru.

Příloha D

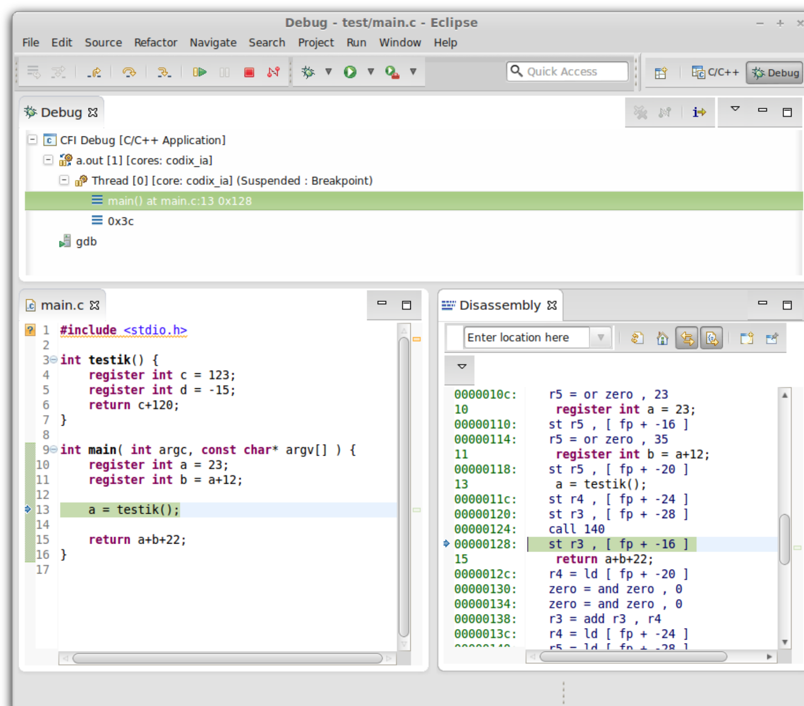
Vizualizace zásobníku volání



Obrázek D.1: Rozhraní debuggeru v prostředí Eclipse bez podpory zásobníku volání.



Obrázek D.2: Rozhraní debuggeru v prostředí Eclipse s podporou zásobníku volání před provedením akce *step return.*



Obrázek D.3: Rozhraní debuggeru v prostředí Eclipse s podporou zásobníku volání po provedení akce *step return.*