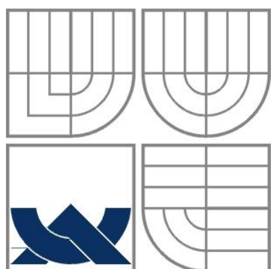
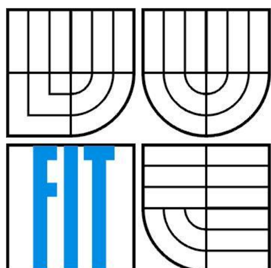


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYUŽITÍ FORMÁTU DWARF PŘI LADĚNÍ GENERICKÝCH SIMULÁTORŮ MIKROPROCESORŮ

USAGE OF A DWARF FORMAT IN DEBUGGER FOR GENERIC MICROPROCESSOR SIMULATORS

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE PAVEL JANEČKA
AUTHOR

VEDOUCÍ PRÁCE prof. Ing. TOMÁŠ HRUŠKA, CSc.
SUPERVISOR

BRNO 2011

Abstrakt

Práce poskytuje základní seznámení s laděním na úrovni zdrojového kódu, formátem DWARF a jeho možným použitím. Dále představuje čtenáři s projektem Lissom, v jehož rámci práce vznikla. Náplní práce dále bylo využití získaných poznatků při rozšíření funkcí debuggeru v projektu Lissom.

Abstract

This paper gives basic introduction to source-level debugging, DWARF debugging information format and it’s possible applications. Further it presents the Lissom project to the reader. The goal of this paper was also to draw on gained knowledge in order to extend Lissom’s debugger.

Klíčová slova

Ladicí nástroj, DWARF, debugger, multiplatformní, Lissom

Keywords

Debugger, DWARF, cross-platform, Lissom

Citace

Janečka Pavel: Využití formátu DWARF při ladění generických simulátorů mikroprocesorů, bakalářská práce, Brno, FIT VUT v Brně, 2011

Využití formátu DWARF při ladění generických simulátorů mikroprocesorů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc.
Další informace mi poskytl Ing. Jakub Křoustek.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Pavel Janečka
18.5.2011

Poděkování

Tímto bych rád poděkoval prof. Ing. Tomáši Hruškovi, CSc. za cenné rady. Rovněž děkuji panu Ing. Jakubu Křoustkovi za odbornou pomoc a trpělivost.

© Pavel Janečka, 2011
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

Obsah 1
1 Úvod 3
2 Ladění 4
 2.1 Překlad 4
 2.2 Ladicí nástroje 5
 2.3 Ladění na úrovni zdrojového kódu 7
  2.3.1 Breakpointy 8
  2.3.2 Typy ladicích informací 9
  2.3.3 Optimalizovaný kód 9
 2.4 Existující ladicí nástroje 11
3 Formáty ladicích informací 12
 3.1 Přehled 12
 3.2 Formát ladicích informací DWARF 13
  3.2.1 Organizace ladicích informací 14
  3.2.2 Informace o řádcích 15
  3.2.3 Vyhledávání 15
  3.2.4 Seznam lokací 16
  3.2.5 Popis rámce 16
4 Projekt Lissom 17
 4.1 Jazyk ISAC 17
 4.2 Architektura projektu 17
  4.2.1 Komunikační protokol 18
 4.3 LLVM 19
 4.4 Formát objektového souboru 19
 4.5 Současný stav ladicího nástroje 20
  4.5.1 Příkazová řádka 20
  4.5.2 Ladicí knihovna 21
5 Vlastní řešení 22
 5.1 Zpracování formátu DWARF 22
  5.1.1 Využití knihovny SymtabAPI 23
 5.2 Jak debugger pracuje 25

5.3 Implementace 25
5.3.1 Podpora DWARFu 25
5.3.2 Ladicí informace 26
5.3.3 Komunikace 27
5.3.4 Další vývoj 27
6 Závěr 28

1 Úvod

Každý člověk jistě někdy za svůj život udělal chybu. Pravděpodobně jich bylo několik, jejich počet zpravidla nikdo nezapisuje. Říká se, že chybovat je lidské. Pomineme-li etickou a duchovní stránku tohoto tvrzení, jsme ponecháni na pospas faktu, že jakýkoliv výsledek tvorby člověka je ohrožen svou možnou chybovostí. Vyrábí-li člověk nějaký produkt, potom aby tato výroba měla smysl, tedy odběratele, musí být z produktu odstraněny vady a zajištěna jeho bezchybnost.

Programování lze do jisté míry přirovnat k výrobnímu procesu. Máme zdroje, tedy pracovní stanice a lidi, kteří na nich vyvíjí programy. Jako každý výrobní proces ovšem i tento může produkovat chyby, je to jeho přirozenou vlastností. Čím rozsáhlejší a složitější je daný produkt, tím obtížnější je udržet jej v provozuschopném stavu po dobu jeho vývoje a zaručit jeho bezchybnost. Je tedy třeba zajistit jistý kontrolní mechanismus, s jehož pomocí by kontrola vyvíjeného produktu byla snadnější.

V oblasti programování a vývoje se proto ujímá této role tzv. ladění. Z historického hlediska šlo nejprve o analýzu post-mortem výpisů (výpisy stavů registrů, obsahu paměti a dalších informací po ukončení programu), umisťování testovacího kódu a výpisů do laděné aplikace a nakonec použití specializovaných ladicích nástrojů. V současnosti je testování a ladění ve vývojovém cyklu produktu programování nedílnou, náročnou a kritickou součástí, je tedy důležité poskytnout ladicí nástroje schopné zobrazit jejich uživatelům co nejvíce možných informací o laděném programu (více o historii v [6]).

Práce vznikla v rámci projektu Lissom. Jde o výzkumnou skupinu zabývající se oblastí souběžného návrhu hardware a jeho programovým vybavením.

Cílem této práce je stručně čtenáře seznámit s problematikou ladění, speciálně se zaměřením na řešení nezávislé na architektuře procesoru a vhodným způsobem rozšířit existující ladicí nástroj v projektu Lissom. Nejprve bude probrán přehled obecných ladicích nástrojů, dále se práce zaměří na ladění na úrovni zdrojového kódu a vybrané související problémy. Další kapitola se bude zabývat ladicími informacemi a jejich použitím, se zaměřením na formát DWARF. Čtenář bude zběžně seznámen s projektem Lissom. Nakonec budou probrány provedené změny v rámci této práce a diskutován další možný vývoj.

Práce vychází z poznatků získaných z [6], [1], [5] a [4].

2 Ladění

Nejprve je třeba říci něco o ladění obecně. To prošlo v historii jistým vývojem a je možné jej v současnosti provádět několika způsoby. U malých projektů (přibližně do 1000 řádků s nepříliš složitou vnitřní logikou) se tedy můžeme setkat s postupem známým jako kontrolní výpisy, kde se na místa v kódu, kde se chce uživatel přesvědčit o stavu daných proměnných nebo o tom, zda se v daném místě provádění programu vůbec nachází, vloží příslušné konstrukce způsobující vizualizaci požadovaných informací. Tento postup se bohužel ukázal zvláště při práci s většími nebo náročnějšími projekty jako nevhodný, jelikož narušuje strukturu původního kódu, je obtížné jej efektivně spravovat a z hlediska zdrojů je práce s ním časově příliš náročná. Z těchto důvodů je tedy lepší použít alternativního postupu při detekci a odstraňování chyb v programu. Vhodným postupem je užití ladicího nástroje (debuggeru).

2.1 Překlad

Abychom mohli dále pokračovat v probírání problematiky ladění, je třeba zmínit některé základní informace o překladu zdrojového kódu, jelikož právě při tomto procesu je možné generovat ladicí informace.

Proces překladu zdrojového kódu do binárního, strojově zpracovatelného, lze rozložit do šesti základních fází, jsou to lexikální analýza, syntaktická analýza, sémantická analýza, generování vnitřní formy programu, optimalizace a generování cílového programu. Lexikální analýza je proces, při kterém probíhá čtení zdrojového programu a dochází k jeho rozčlenění na jednotlivé lexikální jednotky (lexémy). Ty má poté za úkol zpracovat syntaktický analyzátor, který kontroluje, jak může z názvu vyplynout, právě syntaktickou správnost zdrojového programu, respektive posloupnosti lexémů. V případě této správnosti předává syntaktický analyzátor k sémantické kontrole tzv. derivační strom (ten reprezentuje syntaktickou strukturu zdrojového programu). Zde potom dochází ke kontrolám různých aspektů, jako jsou například deklarace proměnných nebo typová správnost. Sémanticky zkontrolovaný a korektní derivační (syntaktický) strom je posléze předán části překladače, která jej převede do interní reprezentace programu. Tento převod slouží zejména k usnadnění případné následné optimalizace programu a představuje hlavní mezistupeň při převodu zdrojového programu na strojově čitelný kód. Z vnitřní reprezentace zdrojového programu překladačem (vnitřní kód) se tedy generuje výsledný cílový program.

Celý proces můžeme také rozdělit do tří základních částí, kde první nazýváme front-end, druhou middle-end a poslední back-end. Front-end (přední část) zahrnuje zmíněnou lexikální, syntaktickou, sémantickou analýzu a generování vnitřní reprezentace zdrojového programu. Právě s pomocí frond-endu je možné získat během překladu zdrojového kódu ladicí informace, jejich

generování se přitom zpravidla nařizuje speciálním přepínačem daného překladače. Následující část middle-end a back-end (výstupní část) se po řadě specializují na optimalizaci vnitřního kódu a generování výsledného strojově čitelného kódu.

Z tohoto popisu lze tedy soudit, že překlad zdrojového kódu na strojově čitelný program je možné pojmout jako sérii transformací na postupně nižší úrovně. Podrobnější informace lze získat v [4] a [13].

2.2 Ladicí nástroje

Debugger je poměrně komplexní softwarový nástroj, který jeho uživateli slouží hlavně k inspekci či řízení běhu programu a hledání případných chyb. Často bývá také používán jako schopná pomůcka pro programátora, který potřebuje zjistit, jak daný program funguje, nebo například testery.

K úspěšnému fungování procesu ladění je nutná podpora ze strany operačního systému. Základní princip procesu ladění programu s pomocí ladicího nástroje je následující. Nejprve je spuštěn debugger. Poté přichází na řadu samotný laděný program. Jsou extrahovány a analyzovány ladicí informace, zatímco se debugger připojí ke zmíněnému laděnému programu. Další vykonávání laděného programu je poté v řízení ladicího nástroje (díky podpoře ze strany operačního systému). Nastane-li při vykonávání programu k nějakému výjimečnému stavu (vyvolání výjimky či bodu přerušení), je na rozdíl od klasického průběhu informován nejprve debugger. V této fázi (např. dosažení breakpointu) je možné prohlížet informace poskytované ladicím nástrojem a lze z ní opět obnovit provádění laděného programu. S tímto postupem souvisí také krokování, kde je možné nechat laděný program zastavit po provedené instrukci nebo řádku kódu.

Vlastní průběh ladění je potom takový, kde uživatel debuggeru spouští program, který chce zkontrolovat či analyzovat, a na problematické nebo jinak významné části programu umisťuje zvláštní body, takzvané breakpointy (body přerušení). Jakmile exekuce dorazí na některé z těchto míst, provádění programu se pozastaví a debugger je informován o vzniklé situaci, často lze hovořit o řízení událostmi, kde je debugger spraven o aktuálním stavu zprávou.

Na ladicí nástroje je kladena řada nároků. Jedním z nich je Heisenbergův princip. Ten tvrdí, že debugger by měl vykonávání analyzovaného programu ovlivňovat co nejméně – ideálním případem by byl stav, kdy by ladicí nástroj program absolutně neovlivňoval. Toho však nelze dosáhnout hned z několika důvodů. Už jenom fakt, že je debugger společně s laděným programem nahraný ve společném paměťovém prostoru, ono vykonávání programu ovlivňuje. Dále je zde skutečnost, že debugger a sledovaný program jsou dva procesy, které se musí dělit o zdroje, jako je například čas na procesoru. V neposlední řadě jsou zde potom možnosti, které vycházejí ze samé podstaty ladicího nástroje, totiž schopnost řídit proces vykonávání zkoumaného programu. Cílem tedy je ovlivnění laděného programu debuggrem co nejvíce snížit.

Dalším požadavkem je takzvané pravdivé ladění. Jde v podstatě o problém, že ladicí nástroj nesmí jeho uživateli poskytovat nesprávné informace. Dopad nesplnění tohoto požadavku by byl takřka katastrofický. Lze si poměrně snadno představit některé následky takové situace, například nedůvěru v daný nástroj (více se tímto aspektem zabývá [6]). Častými oblastmi, kde se takovéto problémy mohou vyskytnout, jsou například ladění optimalizovaného kódu nebo sledování tzv. stack back-trace.

Nakonec je zde požadavek, který lze s jistotou pokládat za základní pilíř funkcionality ladicích nástrojů – chceme především interpretovat ladicí informace. Je velmi důležité při inspekci běhu programu exaktně vědět, kde se provádění právě nachází, případně v jakém místě došlo k eventuálnímu pádu aplikace. Když tyto informace máme k dispozici, může nás dále zajímat, jak jsme se do daného místa dostali a jaké okolnosti jsou přítomny. Okolnostmi přitom máme na mysli obsahy jednotlivých proměnných nebo paměťových míst, registrů aj.

Nyní se je vhodné zmínit se o dělení debuggerů. Jelikož, jak bylo v úvodu podkapitoly zmíněno, jsou debuggery značně složité nástroje, lze je členit mnoha způsoby. Prvním členění můžeme založit na vizuální interpretaci ladicího procesu. Z tohoto hlediska dělíme debuggery na debuggery příkazové řádky (známým příkladem je GDB) a na ladicí nástroje s grafickým uživatelským rozhraním (např. DDD). Zatímco u nejdříve zmiňovaného typu je proces ladění řízen uživatelem pomocí psaní příkazů a následném zisku požadovaných informací, vizualizační schopnosti ladicích nástrojů s grafickým uživatelským rozhraním (GUI) pochopitelně značně převyšují prvně jmenovaný typ. Přesto debuggery příkazové řádky lze upřednostnit v některých specifických případech, například nemáme-li kvůli okolnostem přístupný grafický zobrazovací režim (typicky jde o vzdálené připojení, nicméně v debuggerech s GUI je v poslední době znám také fenomén vzdáleného ladění). Zvláštním případem ladicích nástrojů s grafickým uživatelským rozhraním je potom integrované vývojové prostředí (IDE), které představuje komplexní a efektivní kombinaci překladače, debuggeru, zvýrazňovače syntaxe a dalších nástrojů.

Ladicí nástroje se mohou dále specializovat na základě svého určení. Lišit se například budou debuggery, které mají za úkol sledovat program z pohledu procesoru a ty, které mají přiblížit provádění původnímu zápisu. Podle určení tedy členíme ladicí nástroje následovně (více v [4] a [6]):

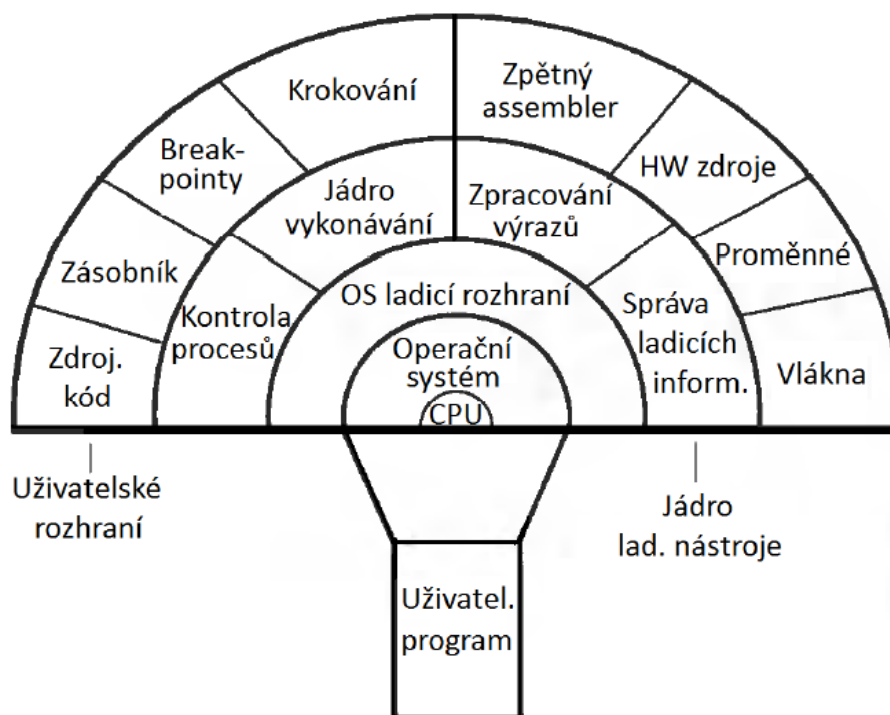
1. Machine level debugger (ladicí nástroj pracující na úrovni strojového kódu) Tento způsob ladění je vhodný zejména ve chvíli, kdy není možné ladit program na vyšší úrovni, což může být způsobeno absencí ladicích informací nebo nemožností naleznout soubor se zdrojovým kódem aj. Zvláštním případem použití debuggeru na této úrovni je často reverzní inženýrství. Aby uživatel mohl co nejvíce využít této metody ladění, potřebuje být dobře obeznámen s informacemi o cílové architektuře (zejména instrukční sadě procesoru, registrech), na kterých je strojový kód závislý. Na této úrovni je nicméně ladění složitého programu velmi obtížné.

2. Source level debugger (ladicí nástroj pracující na úrovní zdrojového kódu) umožňuje uživateli odhalovat chyby a zkoumat běh programu na vysoké úrovni abstrakce od cílové platformy – na úrovni zdrojového kódu. Takovéto ladění je založeno na zisku a interpretaci ladicích informací, které jsou produkovány během překladu zdrojového kódu aplikace. Díky tomu může debugger mapovat instrukce programu na původní zdrojový kód a je možné v požadovaném okamžiku zjistit například obsah jednotlivých proměnných nebo stav zásobníku.
3. Kernel debugger (ladicí nástroj pro jádro) slouží pro ladění jádra operačního systému, případně jeho ovladačů.

První dva typy ladicích nástrojů se velmi často kombinují. Jelikož je ladění na úrovni zdrojového kódu v současnosti nejběžnější způsob detekce chyb a kontroly běhu programu, budeme se nadále zabývat právě jím.

2.3 Ladění na úrovni zdrojového kódu

Od nástroje na této úrovni je očekávána řada funkcí. Co se týče zisku informací o laděném programu, uživatel by měl mít možnost znát místo, ve kterém se v kódu právě nachází, hodnoty proměnných programu, stav a obsah paměti a registrů, stav a informace týkající se jednotlivých vláken v případě vícevláknové aplikace, stav zásobníku (pro zjištění posloupnosti volání funkcí a případně předávání argumentů). V jistých situacích je žádoucí mít možnost přepnout na ladění na úrovni strojového kódu. Aby bylo možné tyto informace využít, je potřeba přístup k laděné aplikaci v podobě spuštěného procesu. To se děje na základě komunikace mezi operačním systémem (ten musí poskytovat podporu pro ladění) a jádrem ladicího nástroje (podrobnější popis architektury ladicího nástroje lze nalézt v [6]).



Obrázek 1. Pochází ze zdroje [4]. Představuje hierarchii při ladění uživatelského programu ladicím nástrojem. Vrstva uživatelské rozhraní znázorňuje informace poskytnuté uživateli, vrstva „Jádro ladicího nástroje" zprostředkovává komunikaci mezi operačním systémem a uživatelským rozhraním.

V následujících podkapitolách budou probrány skutečnosti a jevy související s laděním na úrovni zdrojového kódu.

2.3.1 Breakpointy

Jedním z dalších požadavků na ladicí nástroj je umožnění krokování programu a vkládání bodů přerušení (neboli breakpointů). Breakpointy typicky vkládáme do zdrojového kódu, kde očekáváme, že se vykonávání sledovaného programu bude nacházet a požadujeme jeho pozastavení. Body přerušení lze dále členit, jsou známy například logické a fyzické, kde logický breakpoint představuje možnost přerušení běhu programu podmiňovat, například lze pozastavení vykonávání laděného programu navázat na stav konkrétní proměnné, počet běhů daným místem aj., zatímco fyzické jsou body přerušení již reálně umístěné do kódu analyzovaného programu. Tyto dva druhy breakpointů spolu přitom souvisejí – lze je vzájemně mapovat (fyzickým bodům přerušení dodávají logické podmínky, za kterých jsou platné, pokud je to požadováno, zatímco naopak je logickým breakpointům poskytnuta informace o tom, kde leží jejich pole působnosti).

Breakpointy potom umožňují existenci souvisejícím funkcím debuggeru, jako jsou například *step over function call*, *step into*, *step out*. Často jde přitom o využití dočasných bodů přerušení. Důležité je, že po každém přerušení aplikace je nutné umožnit pokračování v jejím provádění.

2.3.2 Typy ladicích informací

Pro co nejúspěšnější proces ladění programu je potřeba, aby uživatel debuggeru měl k dispozici co nejvíce relevantních a pravdivých informací, které mohou popsat okolnosti a samotný běh programu. Jejich zisk a zejména interpretace jsou hlavním smyslem debuggeru, proto se v této podkapitole seznámíme s některými druhy ladicích informací, se kterými se lze setkat (podrobnější informace lze získat v [6]). Je nutné podotknout, že za existenci a formu ladicích informací pro daný program je zodpovědný překladač, ze kterého program vzešel. Bez těchto informací by bylo možné program ladit pouze na úrovni strojového kódu (případně pomocí ladicích výpisů).

Jedním z důležitých faktorů, který hraje při ladění roli, je takzvaný kontext. Jedná se o souhrn informací, které dávají uživateli debuggeru vědět, kde se vykonávání programu právě nachází. Z jistého pohledu lze brát kontext jako prostou pozici ve zdrojovém kódu, zvýraznění aktuálního řádku a zobrazení okolních, přičemž je zásadní, aby bylo možné řádky zdrojového kódu přiřadit instrukcím strojového a naopak. Toto je jedna ze stěžejních funkcí každého ladicího nástroje. Na kontext lze ale také nahlížet poněkud komplexněji, a to jako na soubor informací o stack back-trace (zásobník volání funkcí), stavech registrů, paměti a případně pozici ve strojovém kódu.

Stack back-trace informace poskytují přehled o tom, pomocí jaké posloupnosti volání funkcí bylo ve vykonávání programu dosaženo aktuálního stavu, a získává se ze zásobníku, kam se typicky ukládá nejen návratová adresa funkce, do které se vnořuje, ale například i informace o jejích argumentech.

Z pohledu kontextu tedy dále sledujeme stav registrů a paměti. Tyto informace představují narušení abstrakce zavedené laděním na úrovni zdrojového kódu, v jistých situacích je ale ladění s jejich pomocí nevyhnutelné, bez znalosti cílové architektury se zde neobejdeme. S tímto souvisí i možnost přepnout do režimu ladění na úrovni zdrojového kódu, kde se zpravidla strojový kód programu převádí do asembleru (disassembly).

Nakonec je potřeba zmínit další z důležitých funkcí debuggeru. Jde o sledování proměnných. U tohoto druhu ladicích informací se naskýtá řada možností. Základní funkcí je zobrazení hodnoty proměnné v návaznosti na její jméno, pod nímž je známa ve zdrojovém kódu. Kromě zjevné možnosti vizualizace jednotlivých obsahů se zde otevírá nastavování podmíněných breakpointů, některé debuggery umožňují také obsahy proměnných za běhu editovat. Navíc lze s jejich pomocí provádět různá vyhodnocování výrazů.

2.3.3 Optimalizovaný kód

Dosud zmíněné informace se týkaly obecně ladění na úrovni zdrojového kódu. Mohou nastat situace, kdy se může ladění programu znesnadnit. Typickým případem je právě ladění optimalizovaného kódu. Tento problém sice přesahuje rámec této práce, může být ale vhodné jej zmínit.

V současnosti používané překladače nabízejí různé optimalizace, které lze provést během překladu, díky nimž je zrychlen běh výsledného programu nebo snížena jeho paměťová náročnost. Tyto optimalizace, hovoříme-li o optimalizacích za překladu, musí v místech, kde je to třeba, narušit vazby na původní zdrojový kód a změnit výkon některých příkazů. Z toho potom musí vzejít funkčně ekvivalentní strojově čitelný kód. Děje se tak ale za cenu odlišností od výsledné neoptimalizované posloupnosti instrukcí, na kterou lze jinak bez větších obtíží namapovat zdrojový kód. Tato skutečnost má dále vliv i na interpretaci ladicích informací.

Je zde možnost sice ladit neoptimalizovaný kód, nicméně v případě, kdy se má distribuovat optimalizovaná aplikace, je žádoucí zajistit či kontrolovat její funkčnost právě při zapnutých optimalizacích. Je tedy třeba, aby ladicí informace byly schopny podat přehled o průběhu programu i při provedených optimalizacích. Některé jejich typy jsou (levý sloupec znázorňuje kód před optimalizací, pravý sloupec po ní):

- Zabalení konstanty (vyhodnocení výrazu složeného pouze z konstant během překladu)

```
a = 0;                              c = 1;
b = 1;
c = b - a;
```

- Šíření konstanty (přiřazení hodnoty k proměnné za překladu)

```
a = 0;                              c = 0;
b = a;
c = b;
```

- Kopírování proměnné (analogické se šířením konstanty)

```
a = b;                              d = a;
c = a;
d = c;
```

- Výrazové invarianty v cyklu (připomíná matematické „vytknutí před závorku“, výraz konstantní v průběhu cyklu je vyhodnocen mimo cyklus)

```
for (i = 1; i < 10; i++)            tmp = a+b-c;
  F[i] = (a+b-c)/i;                 for (i = 1; i < 10; i++)
                                      F[i] = tmp/i;
```

- Rozbalení cyklu (nahrazení cyklu sekvencí prováděných akcí)

```
for (i = 2; i < 4; i++)             Print(2);
  Print(i);                         Print(3);
```

- Eliminace mrtvého kódu (odstranění kódu, který nemůže být proveden, nebo kódu, který neprovádí žádnou činnost)

```
while(false)                        //zde nebude žádný kód
    Print(„test“);
```

```
b = b;
```

Informace pocházejí z [10].

U prvních tří příkladů optimalizace by nastaly potíže, požadovali-li bychom inspekci posloupnosti příkazů, které byly zapsány v levém sloupci. Ve skutečnosti by totiž bylo možné provést vizualizaci pouze výsledku optimalizace (pravý sloupec), příkladu zabalení konstanty by se tedy přeskočil řádek `b = 1;`, u příkladu rozbalení cyklu by nebylo možno vizualizovat iterační proměnnou atd.

Základním problémem zmíněných operací je tedy zejména to, že nelze úplně přiřadit posloupnost strojového kódu konstrukcím zapsaným ve zdrojovém kódu a naopak. Rovněž vyhodnocování nemusí probíhat na místech patrných ze zdrojového kódu.

Existují i další možnosti optimalizací zdrojového kódu a souvisejících potíží, například u architektur zaměřujících se na vyšší míru paralelismu (více v [4]). Ty ale již nebudou zmíněny.

2.4 Existující ladicí nástroje

Jedním z nejznámějších ladicích nástrojů je bezesporu Microsoft Visual Studio. Jedná se o vývojové prostředí, jehož nedílnou součástí je také ladicí nástroj. Podporuje zejména jazyky C#, C/C++, Visual Basic a další, je vázán na operační systémy Microsoft Windows. Debugger je přehledný a schopný, bohužel neumožňuje ladit samotné rutiny či funkce jádra operačního systému.

Zajímavým ladicím nástrojem je LDB [18], [19]. Jako u každého ladicího nástroje i zde je možné nastavovat breakpointy a sledovat stavy proměnných. Pozoruhodným debuggerem jej dělá skutečnost, že tyto funkce poskytuje napříč cílovými platformami, a to za pomoci podpory ze strany překladače, vestavěného interpretu nezávislého na cílové architektuře a minimalizaci kódu závislého na cílové platformě. Tento ladicí nástroj bylo v plánu rozšířit tak, aby kromě programů napsaných v jazyce C podporoval i C++. Používá specializovaného formátu ladicích informací založeného na PostScriptu, nevýhodou ovšem je, že tento formát není žádným jiným překladačem podporován. Dalším úskalím je fakt, že tento nástroj spolupracuje pouze s překladačem LCC. Z těchto důvodů se LDB dnes prakticky nepoužívá.

Poměrně často používanými nástroji pro ladění jsou také programy GDB [20] a DDD, které je jeho grafickou nadstavbou (GDB je základem mnoha jiných debuggerů v IDE, například Code::Blocks). Tento ladicí nástroj v současnosti podporuje jazyky, jako jsou C, C++, Objective-C, Pascal a další, umožňuje vzdálené ladění a funguje v unixových operačních systémech i Microsoft Windows. Nově (od září 2009) obsahuje možnost zpětného ladění.

V projektu Lissom, v rámci kterého tato práce vznikla, disponuje vlastním ladicím nástrojem, který bude společně s projektem probrán v příslušné kapitole.

3 Formáty ladicích informací

Ladicí informace lze teoreticky generovat v jakékoliv možné podobě, je-li to technicky proveditelné. Stejně ale jako klademe požadavky na programovací jazyky, aby vyhovovali normám, tedy k zápisu programu používám již ty existující, je vhodné generovat ladicí informace v podobě již standardizovaného formátu. Tato skutečnost potom může usnadnit práci s těmito informacemi, neboť je potom možné použít stávajících nástrojů k jejich zpracování. Z tohoto důvodu se v této části práce nebudeme zabývat tvorbou vlastního formátu ladicích informací, naopak krátce probereme některé dosud známé formáty a zdůvodníme výběr formátu DWARF, který se jeví pro účel této práce jako nejvýhodnější.

3.1 Přehled

V současnosti lze tedy evidovat řadu forem ukládání ladicích informací. Můžeme zmínit například STABS, DWARF, dále existují informace pevněji svázané s objektovým souborem, například COFF nebo jeho varianty XCOFF a PE/COFF, IEEE-695 nebo OMF. Velmi často používaným a známým formátem je rovněž PDB.

Začněme krátkým představením typu ladicích informací STABS [3]. Formát, za jehož autora je považován Peter Kessler, byl původně zamýšlen pro použití s jazykem Pascal, dnes je ale možné jej použít i s jazyky, jako jsou například COBOL nebo C. STABS je znám především ve spolupráci s formáty objektových souborů XCOFF, COFF a ECOFF, přičemž u posledních dvou toto umožňují nástroje GNU. Symbol Table Strings (STABS) ukládá ladicí informace jako textové řetězce. V současnosti je náročnost, respektive složitost tohoto formátu takové, že jej nelze efektivně a hlavně konzistentně spravovat. Příklad formátu záznamu: `.stabx „řetězec složený z deskriptoru symbolu a informace o typu"`, hodnota, `typ záznamu, 0`

Dalším formátem ladicích informací, který bude nastíněn, je PDB [15], [16], [4], [1]. Jde o formát používaný společností Microsoft, respektive jejími nástroji pro vývoj či tvorbu software (Visual Studio). Na rozdíl od většiny ostatních formátů se tyto ladicí informace zpravidla generují do separátního souboru (i když jejich generování lze v jisté omezenější míře provést i do objektového souboru [16]). Tento formát se používá v kombinaci s programovacími jazyky C/C++, C# nebo například Visual Basic. Jedná se nicméně o proprietární standard a tudíž není možné jej plně využívat.

Rovněž lehce zmíníme formáty objektových souborů, které obsahují přímo v rámci své specifikace specializované struktury pro uchování ladicích informací. Jde konkrétně o již zmíněné formáty XCOFF, PE/COFF, COFF, IEEE-695 a OMF [4], [1], [17]. Způsob uložení se liší v závislosti na objektovém formátu a v některých z nich je označován jako zastaralý či nepoužívaný.

Posledním formátem, který byl jmenován, je standard DWARF [2], [1]. Jedná se o prostředek, který je podporuje jazyky jako například C, C++ nebo Fortran, přičemž je navržen, aby jej bylo možné rozšířit pro funkčnost s jinými programovacími jazyky. Zároveň je nejčastěji znám ve spojitosti s objektovým formátem ELF, přesto lze DWARF rozšířit pro spolupráci s jinými typy objektových souborů. Rovněž je možné tento formát používat v rámci různých cílových architektur. Standard Debugging With Atttributed Record Formats (DWARF) je rozšiřitelný, kompaktní a konzistentnější než již zmíněný formát STABS [12]. Navíc je podporován většinou novějších překladačů a přechází na něj většina velkých firem.

Z výše zmíněných důvodů lze usoudit, že nejvhodnější způsob uložení ladicích informací poskytuje právě formát DWARF. Jeho přední vlastnosti, jako například nezávislost na cílové platformě, jsou potom velmi důležitými výhodami při použití v rámci projektu Lissom. Dalším důvodem pro jeho užití je skutečnost, že v rámci projektu Lissom je používám překladač LLVM, který generování ladicích informací formátu DWARF podporuje. Tímto standardem se v práci budeme dále zabývat.

3.2 Formát ladicích informací DWARF

Přestože byl tento formát navržen jako nezávislý na architektuře procesoru nebo programovacím jazyku (a je tedy teoreticky možné jej použít s každým formátem objektového souboru za předpokladu podpory ze strany překladače a umožnění použití pojmenovaných sekcí v objektovém souboru), nejčastěji je znám ve spojitosti s formátem objektového souboru ELF.

Poskytuje základní informace, jako jsou informace o lokálních a globálních proměnných, návratové hodnoty funkcí, rozsah platnosti, viditelnost, pro vyšší programovací jazyky použitelné informace o blocích zaměřených na zpracování výjimek. Velmi důležitými informacemi pro proces ladění jsou také poskytnuté informace o pozici v kódu (identifikace řádku) a o pozici na řádku. Podrobné informace o tomto formátu jsou k nalezení v [2].

Záznamy ladicích informací DWARF jsou uloženy v objektovém souboru v sekci se jménem .debug_info. Posloupnost těchto záznamů potom určuje nízkoúrovňovou reprezentaci zdrojového programu. Každý záznam je potom tvořen svou jmenovkou (tag) a svými atributy. Ty dále můžeme členit do tříd, jako jsou adresa (která se odkazuje na jisté místo v adresovém prostoru), blok (počet neinterpretovaných dat), konstanta (neinterpretovaná data specifikované délky nebo data kódované ve formátu LEB128 [2]), příznak (flag, indikátor přítomnosti atributu), odkaz (reference, odkazuje na některý záznam formátu DWARF) a řetězec (posloupnost znaků ukončená nulou).

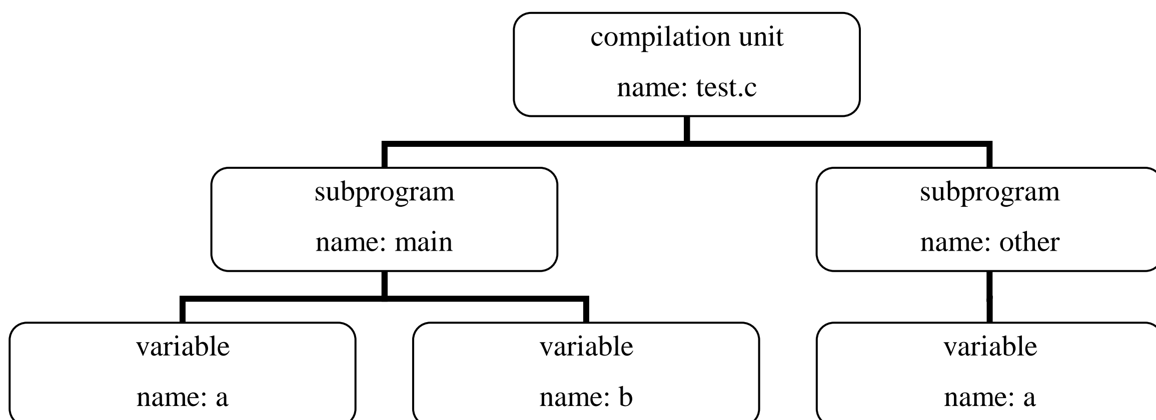
Formát DWARF dále poskytuje interní reprezentaci registrů cílové architektury. Díky tomu lze ladicí informace abstrahovat a oprostit je tak od závislostí na platformě.

3.2.1 Organizace ladicích informací

Většina v současnosti používaných programovacích jazyků je blokově organizována. Zjednodušeně lze říct, že kód v nich napsaný se skládá z entit, např. může jít o třídy, objekty, funkce nebo jiné ohraničené bloky. Ty lze dále zanořovat a vytvářet tak různé lexikální rozsahy platnosti. Samotný formát DWARF je logicky organizovaný podobně. Skládá se ze záznamů, které popisují informace o programu. Tyto záznamy obsahují atributy (minimálně identifikaci svého typu a své jméno) a jsou uspořádány do stromové struktury, přičemž její kořen je záznam představující zdrojový soubor, ke kterému se dané ladicí informace vztahují. Tento speciální záznam se nazývá kompilační jednotka. Dále je hierarchie taková, že uzel stromu přestavuje entitu ladicích informací (pro příklad funkci) a jeho potomek potom vnořenou entitu (může jít například o lokální proměnnou této funkce), každá entita je ve skutečnosti ladicí záznam.

Dalšími typy jsou záznamy o modulu, podprogramu vstupním bodu a lexikálním bloku (v jazyce C jde o blok kódu ohraničený složenými závorkami). Tyto záznamy obsahují především informace na své úrovni působnosti. U modulu se navíc často uvádí jeho název, případně adresa inicializačního kódu, u podprogramů se pak zmiňují například název a návratový typ.

Zůstaňme u příkladu s funkcí. Předpokládejme, že funkce ve zdrojovém souboru již není ničím obalena, pak je její záznam potomkem kořenového uzlu. Měli-li bychom mimoto v tom samém zdrojovém souboru další jinou funkci na stejné úrovni jako předchozí, byla by potom rovněž potomkem kořenového uzlu (více v [1], [2]).



Obrázek 2. Naznačuje hierarchické uspořádání ladicích informací ve formátu DWARF. Každý objekt představuje jeden záznam. Kořenem je kompilační jednotka mající dva potomky – dvě funkce, které obsahuje. Každá tato funkce má pak své potomky – své lokální proměnné. Stojí za povšimnutí, že jelikož jsou obě proměnné jménem "a" v různých disjunktních rozsazích platnosti, nedochází k jejich konfliktu. Schéma není přesné, slouží pouze jako ilustrace.

Jednou z výhod DWARFu je mj. také to, že je zaměřen na úsporu místa. Daný strom záznamů ladicích informací je proto uložen s prefixovou notací, dále se používá tzv. zkratek. Do tabulky

zkratek se odkazují jednotlivé záznamy, čímž se šeří místo, které by jinak bylo zaplněno například řetězcem ([1], [2]).

Ladicí informace v tomto formátu jsou zpravidla ukládána do sekcí se jmény začínajícími na „.debug_“, uvedeny jsou vybrané případy bez této předpony:

- abbrev – zde jsou uloženy zkratky
- info – sekce obsahující DWARF záznamy
- line – obsahuje takzvaný „Line Number Program“
- ranges – rozsahy adres odkazovaných DWARF záznamy
- str – skládá se z řetězců používaných sekcí .debug_info

Dále budou stručně probrány některé vybrané sekce formátu DWARF. Podrobnější informace lze získat v [2].

3.2.2 Informace o řádcích

Pro ladicí informace poskytující mapování adresy programového čítače na řádky ve zdrojovém kódu a umožňující tak získat v něm aktuální pozici existuje speciální sekce s názvem „*.debug_line*“. Na tyto informace se odkazuje ze záznamu příslušné kompilační jednotky.

Vlastní uložení těchto informací by teoreticky bylo možno uložit jako matici přiřazení čísel řádků ve zdrojovém souboru (a dalších informací, jako například pozice na řádku) a instrukce. Jelikož se ale při tvorbě standardu DWARF uvažovaly mimo jiné i paměťové nároky, v zájmu jejich minimalizace jsou informace o řádcích uloženy ve formě speciálního bajtkódu. Požaduje-li potom uživatel tyto informace, zmíněný bajtkód se zpracuje pomocí stavového automatu, čímž dojde k expanzi na původně zmiňovanou matici.

3.2.3 Vyhledávání

Formát DWARF umožňuje vyhledávání ve svých záznamech. To typicky požaduje debugger, například potřebujeme-li vyhledat daný objekt. Aby toto hledání proběhlo co nejrychleji, existují DWARF sekce *.debug_pubnames* a *.debug_arrages*. Ty po řadě zajišťují vyhledávání podle jména a podle adresy. Každá z těchto sekcí obsahuje tabulku, v případě první z nich její záznam obsahuje informace o kompilační jednotce, ve které jsou objekty uloženy, následované dvojicemi tvořenými jménem objektu a jeho pozicí v kompilační jednotce. V případě sekce *.debug_arranges* je tabulka tvořena záznamy, které obsahují popis části adresového prostoru, ve kterém se nachází kompilační jednotka. Za tímto popisem v záznamu následují dvojice složené z počáteční adresy a délky adresového rozsahu, kde se nachází objekt.

3.2.4 Seznam lokací

Existuje-li objekt, jehož umístění se během jeho existence může měnit, je tato skutečnost evidována v sekci *.debug_loc*. Odkazuje se sem z kompilační jednotky a veškeré adresy jsou uvedeny jako relativní k bázové adrese kompilační jednotky. Záznam je tvořen počáteční a koncovou adresou, dále pak popisem umístění. Na jeden objekt může připadat více záznamů.

3.2.5 Popis rámce

DWARF poskytuje nástroj, který umožňuje získat informace o tom, co se děje s registry za běhu programu, resp. jak s nimi nakládají procedury. Informace lze reprezentovat jako tabulku, kde pro každý řádek kódu existuje záznam skládající se z položek označených číslem registru a obsahujících pravidlo, které určuje, jak získat informaci daného registru, a dalších informací.

Jelikož by matice v této podobě byla příliš objemná, je speciálně uložena v sekci *.debug_frame*. Zde jsou informace uspořádány do dvou typů záznamů, a to Common Information Entry (CIE) a Frame Description Entry (FDE).

4 Projekt Lissom

Projekt se obecně řečeno zabývá vývojem jazyka pro popis architektur mikroprocesorů a tvorbou pokročilých nástrojů založených na základě těchto popisů. Jedná se mimo jiné o oblast hardware/software co-designu, jde o vývoj prostředků, které umožní korektní a hlavně časově šetrný (oproti separovanému vývoji) návrh procesorů a aplikací pro ně. Projekt byl založen roku 2003 na Fakultě informačních technologií Vysokého učení technického v Brně.

K dosažení těchto cílů v projektu Lissom již existují prostředky, které prochází neustálým vývojem. Jde především o jazyk ISAC [24] (specializuje se k popisu modelu procesoru) a sadu knihoven, které umožňují řízení jednotlivých nástrojů, jejich generování a simulace.

Projekt Lissom disponuje dále prostředky nezávislými na cílové architektuře. Jde o vývojové prostředí, které ve zjednodušené podobě může zastoupit i klient příkazové řádky, middleware, nástroje pro překlad jazyka ISAC a související, například generátory nástrojů závisejících na cílové platformě. Ty se při překladu vytvářejí pro daný model (jednotlivé navrhované procesory se většinou liší svou architekturou či instrukční sadou a je potřeba tuto skutečnost brát na zřetel), jedná se zejména o překladače jazyka C a jazyka symbolických instrukcí, simulátor, disassembler a ladicí nástroj.

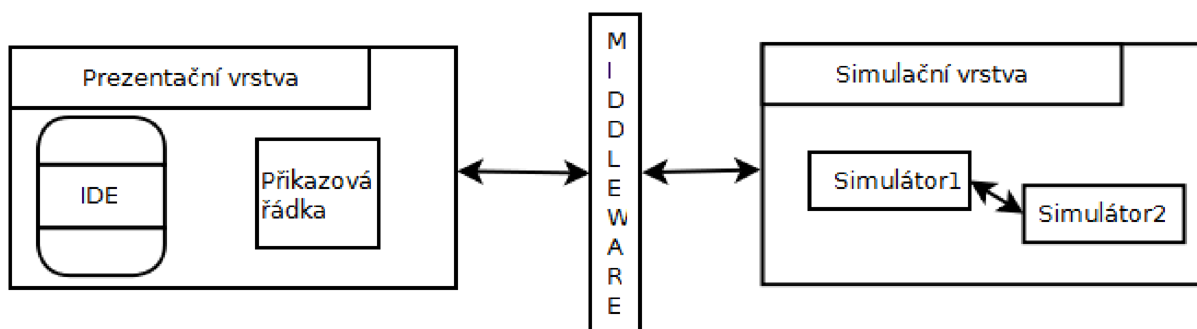
V následujících kapitolách proběhne krátké seznámení s některými prostředky projektu Lissom, objasnění aktuálního stavu a návaznost této práce.

4.1 Jazyk ISAC

Jedná se o jazyk, který slouží k popisu architektury procesoru a jeho instrukční sady a vychází z jazyka LISA. Z hlediska architektury můžeme popisovat hardwarové prvky procesoru, jako jsou paměti, jednotlivé registry, pipeline či ALU procesoru a jejich propojení. Co se týče instrukční sady, zde lze popisovat jednotlivé operace, syntaktický předpis instrukční sady nebo kódování instrukcí a jejich chování [21].

4.2 Architektura projektu

Z povahy projektu lissom je zřejmé, že k zajištění fungování všech nástrojů jako harmonického celku je potřeba něčeho, co by dalo stávajícím prostředkům nějaký řád a hierarchii. Hovoříme tedy o architektuře, která je v projektu Lissom přítomna. Tu lze znázornit následujícím schématem.



Obrázek 3. Znázorňuje komunikaci v architektuře Lissom. Šipky představují vzájemnou komunikaci.

Skládá se ze tří základních vrstev, kterými jsou prezentační vrstva, střední vrstva a simulační vrstva. Mezi nimi pak probíhá komunikace, která zajišťuje správný chod zúčastněných nástrojů.

Jako první je vhodné zmínit střední vrstvu (middleware). Jelikož nástroje používané při vývoji aplikací pro procesory jsou, jak bylo dříve zmíněno, závislé na cílové architektuře, potřebujeme tyto nástroje pro každý procesor generovat. Rovněž se musí vzít v potaz, že následný běh aplikací, simulace, nemůže probíhat svévolně a je třeba jej nějakým způsobem řídit. Mimo jiné tyto úkoly má na starosti právě střední vrstva. Stará se navíc i o překlad modelu procesoru, pro který následně může generovat nástroje, zajišťuje komunikaci mezi uživatelem a generovanými nástroji v simulační vrstvě.

Další součástí architekturu je prezentační vrstva, která uživateli umožňuje komunikovat se střední vrstvou. Typickými představiteli jsou klient příkazové řádky, a vývojové prostředí Eclissom. To je založeno na programu Eclipse Galileo.

Poslední je simulační vrstva, která vzniká díky middleware. Ta může být spuštěna na stejném i jiném počítači, než kde je spuštěna střední vrstva. Po instalaci a spuštění simulátoru je možné řídit jeho běh, zejména jej zastavovat, což je výhodné, provádíme-li zrovna ladění simulované aplikace.

4.2.1 Komunikační protokol

V rámci architektury projektu Lissom je vhodné zmínit se také o komunikačním protokolu, pomocí kterého komunikuje middleware s prezentační a simulační vrstvou. Tato komunikace probíhá oběma směry – prezentační vrstva zasílá požadavky middleware a ten vrací zpět odpověď. Střední vrstva poté může předávat zprávy simulační vrstvě, které návazně reaguje.

Komunikační protokol svým formátem velmi připomíná XML. Na rozdíl od něj ale postrádá kořenový element a hlavičku. Navíc je svázán i některými omezeními, aby bylo možné zprávy v tomto formátu snadno a co nejrychleji zpracovávat. Detailnější popis protokolu lze nalézt ve zdroji [23].

4.3 LLVM

Nejedná se o nástroj vyvinutý projektem Lissom, nicméně tvoří jeho zásadní část a proto je zde zmíněn. Jde o překladač, který lze považovat za alternativu ke známému gcc. Jeho zřejmou výhodou a přínosem pro projekt Lissom je fakt, že jej lze použít jako platformu pro generování kódu nezávislého na cílové architektuře.

Jako přední část používá gcc front-end, na middle-endu je zajímavé to, že se specializuje optimalizace v době linkování aplikace a na meziprocedurální optimalizace. Stěžejní je ale především back-end tohoto kompilátoru, který se stará o reprezentaci a následné generování kódu. Více informací je dostupných v [14].

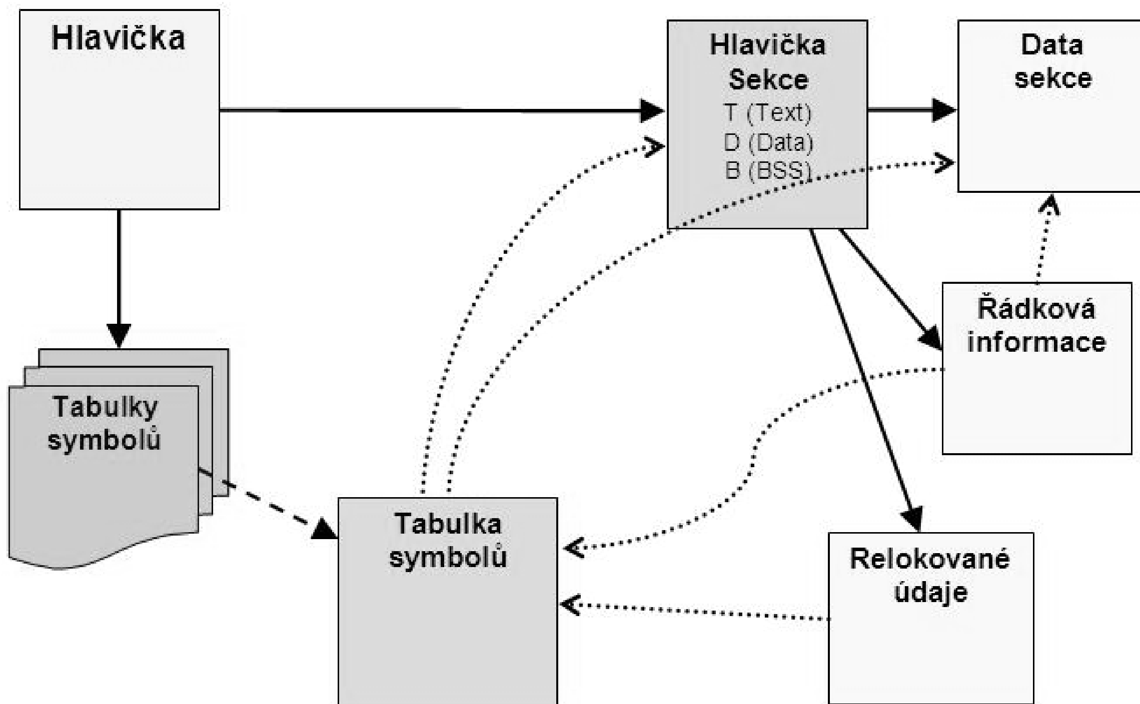
4.4 Formát objektového souboru

Požadavek vyvíjet aplikace pro různé druhy procesorů může být zdánlivě neškodný. Zvážíme-li jej, musíme ale nevyhnutelně dospět k závěru, že kdybychom vyvíjeli aplikace pro každý procesor zvlášť, tento proces by se z časového hlediska prodloužil a celkově prodražil. To je jeden z důvodů, proč je vhodné použít specializovaný jednotný typ souboru pro všechny platformy.

Projekt Lissom disponuje vlastním formátem objektového souboru, jehož původní návrh byl vytvořen v únoru roku 2004. Od té doby prošel několika změnami, nebude zde detailně popsána jeho struktura, zaměříme se pouze na některé klíčové aspekty tohoto formátu.

Na formát byly v době jeho tvorby kladeny požadavky, jako jsou nezávislost na délce slova instrukce, endianitě, charakteru instrukční sady a způsobu uložení do paměti. Ve srovnání s ostatními formáty objektových souborů by měl být snáze čitelný a interpretovatelný. Následující schéma nám nyní přiblíží organizaci objektového souboru typu LOFF [22] (tak se v současnosti jmenuje formát objektového souboru v projektu Lissom).

Základní charakteristika formátu LOFF je takováto. Jedná se o textový formát skládající se ze záznamů, kde je každý ukončen unixovým znakem konce řádku (0x0A). Obsahuje dva typy informací, textové a číselné, číselné navíc ve dvou formátech, pro vlastní data se používá binární soustava, pro dílčí informace (například velikost sekce) desítková. Textové informace jsou předcházeny záznamem určujícím jejich velikost. Pro ilustraci následuje přibližné schéma souboru typu LOFF (přibližné proto, že formát prochází stále vývojem):



Obrázek 4. Převzat ze zdroje [22]. Ukazuje přibližnou vnitřní organizaci formátu LOFF.

4.5 Současný stav ladicího nástroje

V současnosti v projektu Lissom existuje řada funkčních nástrojů, které middleware generuje pro konkrétní procesory. Z nich je pro tuto práci nejpodstatnější ladicí nástroj. Ten na začátku této práce fungoval jako debugger na úrovni strojového kódu a nebylo tedy možné ladit na vyšší úrovni abstrakce. Ladicí nástroj šlo používat přes klienta příkazové řádky i pomocí vývojového prostředí Eclissom. Pro účely práce bude stručně popsáno ladicí rozhraní klienta příkazové řádky a poté ladicí knihovna. Následující informace lze získat v podrobnější podobě z [5].

4.5.1 Příkazová řádka

Implicitně běží v základním režimu. Z něj je možné zejména překládat modely procesorů, aplikace pro ně, lze generovat nástroje a spouštět je. V tomto režimu není možné aplikace ladit, proto se nadále budeme práce zabývat ladicím režimem.

Do něj se lze přepnout příkazem *gdb* a opouští se příkazem *exit*. Ladicí režim podporuje řadu příkazů, mezi které patří například výběr simulátoru (*simulator*), řízení běhu simulace (*run, continue, step*), zobrazování informací o registrech nebo bodech přerušení (*info registers* nebo *info breakpoints*). Dále příkazová řádka v tomto režimu umožňuje vložení bodu přerušení (*break*).

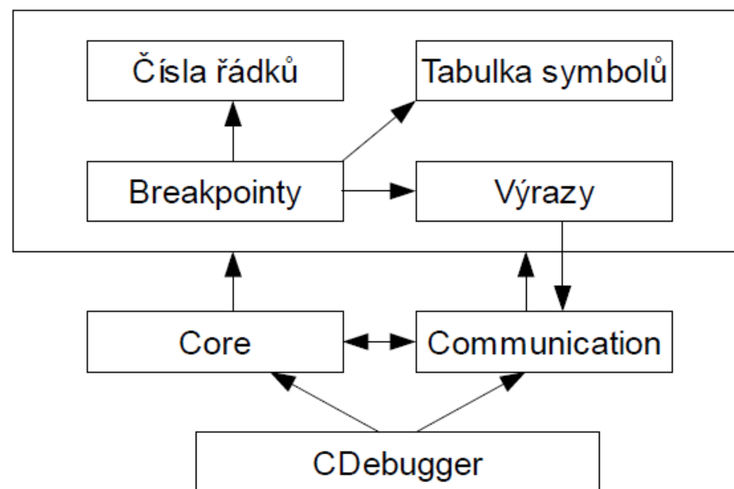
Vložení breakpointu lze provést dvěma základními způsoby. Prvním je vložení přímo na adresu. V takovémto případě se použije číslo specifikující adresu následované hvězdičkou, která

informuje o tomto způsobu vložení. Druhou možností, jak vložit bod přerušení, je uvést jméno souboru se strojovým programem následované dvojtečkou a číslem řádku v tomto souboru. Při vynechání jména s dvojtečkou se použije implicitní soubor. Po vytvoření breakpointu je k němu přiřazeno číslo, s jehož pomocí můžeme tento bod přerušení v případě potřeby ovlivňovat.

Breakpoint lze smazat příkazem *delete* následovaným jeho číslem. Kromě jeho mazání lze body přerušení rovněž aktivovat nebo deaktivovat (*enable* nebo *disable*), přičemž jsou následovány číslem příslušného breakpointu.

4.5.2 Ladicí knihovna

Tento nástroj se typicky zavádí po spuštění simulátoru a inicializaci procesoru, přičemž získává přístup k prostředkům, jak jsou například paměť, programový čítač a zdroje. Současná ladicí knihovna se skládá z několika částí. Základní ladicí funkce poskytuje jádro debuggeru, třída `CDebuggerCore`, která zapouzdřuje třídy jako například `CDbgExpr`, `CDbgSymbolTable`, `CDbgBreakpoints` nebo `CDbgLineNumbers`, na které pak jádro předává požadavky spojené s laděním. Komunikaci s okolím potom zajišťuje třída `CDebuggerCommunication`. Pro přiblížení je dále přiloženo schéma ladicí knihovny. Více se lze dozvědět v [5].



Obrázek 5. Pochází ze zdroje [5]. Znázorňuje základní rozdělení ladicí knihovny.

5 Vlastní řešení

Základem práce je najít vhodný způsob, jak využít formát ladicích informací DWARF pro ladění simulátorů na vysoké úrovni abstrakce, to jest na úrovni zdrojového kódu. Důležitou fází potřebnou k dosažení tohoto cíle je analýza a určení způsobu, jakým je možno formát ladicích informací zpracovat. Implementace vlastního nástroje sloužícího k interpretaci DWARFu připadá totiž v úvahu pouze v případě krajní nouze, kdy by nebylo možné použít žádné existující techniky. Přestože je totiž formát DWARF flexibilní a přehledný, stále je nesmírně složitý. Práce se tedy nyní zaměří na tuto problematiku.

5.1 Zpracování formátu DWARF

Zpracování tohoto formátu je možné provést několika způsoby. Jako hlavní kritéria bereme především nezávislost přístupu na architektuře procesoru a formátu objektového souboru. Možnosti jsou (podrobněji v [1], [7]) :

- užití nástroje readelf [9] (umožňuje tisknout záznamy ve formátu DWARF a jejich atributy jako text)
- GDB (debugger schopný přímo číst a interpretovat požadované ladicí informace. Lze jej také použít jako back-end)
- použití programu dwarfdump (umožňuje interpretovat ladicí informace formátu DWARF jako textové řetězce)
- kombinace libelf a libdwarf (poskytují interní reprezentaci DWARF informací)
- kombinace nástrojů dwarfdump a libdwarf (zatímco libdwarf je schopen získat ladicí informace z objektového souboru, dwarfdump tyto získané informace uspořádá a umožní je zobrazit jako textový řetězec).

Všechny zmíněné nástroje existují a jsou soustavně opravovány a vyvíjeny. I přes přínos programů readelf a dwarfdump je třeba tyto možnosti zamítnout, jelikož vizualizace ladicích informací, přestože jejich čtení oproti původní formě usnadňuje, neřeší nastolený problém, tedy potřebu ladicí informace nejen zachytit, ale i zpracovat.

V tomto ohledu se může jevit jako lepší řešení využití ladicího nástroje GDB. Tento program umí snadno přečíst a interpretovat ladicí informace ve formátu DWARF, které načte přímo z objektových souborů. Navíc jej lze použít jako back-end. Bohužel, zde pro změnu nastává ten problém, že GDB je poměrně komplexní nástroj a díky tomu svou velikostí neúnosně převyšuje očekávanou velikost nástroje, který by měl pouze zpracovat ladicí informace. Navíc je tento program licencován podmínkami GPL, které nelze pro oblast dalšího použití v projektu Lissom akceptovat.

Dále existuje nástroj libdwarf. Jedná se o knihovnu, která poskytuje rozhraní, jehož pomocí je možno detekovat a zpracovat ladicí informace DWARF. Teoreticky lze tuto knihovnu použít s libovolným formátem objektového souboru, ovšem za předpokladu, že k této knihovně doimplementován nástroj, který pro její potřeby tento objektový soubor zpracuje. Implicitně tento libdwarf spolupracuje s knihovnou libelf, které pro něj zpracuje objektový soubor typu ELF [8].

Jelikož ani poslední zmíněná možnost nevyhovuje ze všech pohledů kladeným požadavkům (ačkoliv libdwarf umožňuje zpracovat ladicí informace, jejich interpretace není na dostatečně vysoké úrovni), bylo třeba najít nebo implementovat nástroj, s jehož pomocí je možné převést tyto informace do vyšší úrovně abstrakce. Navíc se zde vyskytl ten problém, že je potřeba dodat podporu pro objektový formát projektu Lissom. První ze zmíněných nesnází řeší knihovna pro podporu ladění SymtabAPI.

5.1.1 Využití knihovny SymtabAPI

SymtabAPI vznikla jako původně pevná (neoddělitelná) součást nástroje DyninstAPI, což je knihovna umožňující různé operace (hlavně analýzu a úpravu) nad objektovými soubory. Díky své komplexnosti a vnitřním závislostem se musel přenášet nebo používat celý DyninstAPI, proto bylo rozhodnuto celý koncept pozměnit a separovat jednotlivé funkcionality. Zde tedy vzniká mimo jiné SymtabAPI.

Knihovna poskytuje rozhraní s množstvím pro ladění již předpřipravených metod, lze získávat mapování instrukcí na řádky zdrojového kódu, informace o proměnných a jejich rozsahu platnosti, v neposlední řadě je třeba zmínit možnost náročnějšího vyhledávání (více v [11]).

SymtabAPI tedy umožňuje zisk ladicích informací a používání rozhraní pro práci s nimi na vysoké úrovni abstrakce. Poskytuje nezávislost na platformě díky tomu, že data získaná z objektových formátů převádí do své jednotné interní reprezentace. Knihovna podporuje pouze pevně stanovené kombinace formátů objektových souborů, ladicích informací a platformy (např. PE-COFF a PDB, COFF a STABS, ELF a DWARF. Pochopitelně tato knihovna nepodporuje objektový soubor typu LOFF a jelikož se ukázalo, že zisk DWARF informací je závislý na předchozím zpracování objektového souboru, bude předmětem další diskuze otázka, která zůstala před touto podkapitolou nezodpovězena, totiž jak zajistit podporu formátu LOFF pro potřeby k zisku ladicích informací.

Obecně existuje několik postupů, kterými se lze vydat, aby bylo možno používat knihovnu SymtabAPI pro interpretaci ladicích informací. Společné je pro ně to, že pokaždé zde bude snaha přidat minimální potřebnou podporu pro formát LOFF, která by zajistila funkčnost extrakce ladicích informací DWARF. Možnosti jsou následující:

- Implementace vlastního nástroje pro zpracování ladicích informací DWARF v kombinaci s formátem objektového souboru LOFF

- Konverze spustitelného souboru na některý z podporovaných formátů knihovnou SymtabAPI, ze kterého je potom možno DWARF informace získat
- Využití existující podpory formátu DWARF za předpokladu úpravy části již zpracovávající ladicí informace s některým z podporovaných typů objektových souborů.

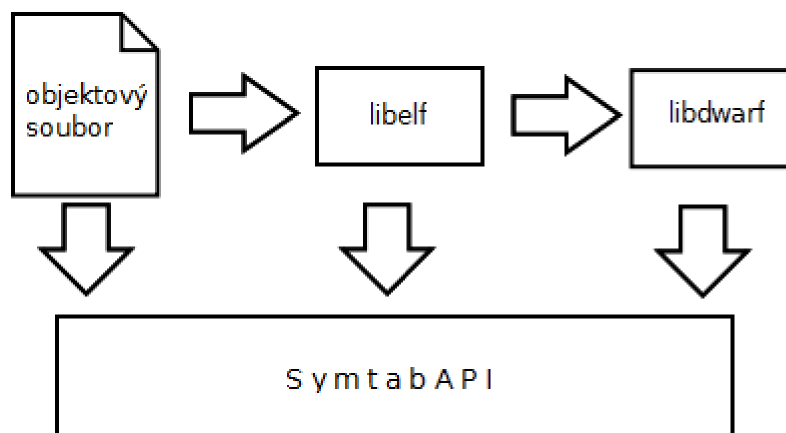
První zmiňovaná možnost se sice může jevit jako nejčistší možný postup, nicméně, jak již bylo zmíněno dříve zpracování samotných DWARF informací je složitý proces a jeho časová náročnost je neúměrná.

Zbývající dva postupy jsou založeny na použití již existující podpory zpracování ladicích informací DWARF. Knihovna SymtabAPI dovoluje zpracovat DWARF pouze za přítomnosti objektového souboru typu ELF. Zbývající postupy se tudíž musí pokusit tuto skutečnost využít.

Prvním z nich, konverze objektového souboru na formát ELF, je metoda přípustná. Má sice za následek tvorbu nadbytečných dat v podobě tohoto převedeného souboru, nicméně při jejich malé velikosti lze tuto skutečnost ignorovat.

Druhý postup, úprava nástrojů zpracovávajících objektový soubor ve formátu ELF tak, aby byla zajištěna minimální potřebná funkčnost pro extrakci ladicích informací DWARF, se tedy ukazuje jako řešení neprodukující nadbytečné soubory a může se zdát, že je nejvýhodnější použít právě jej.

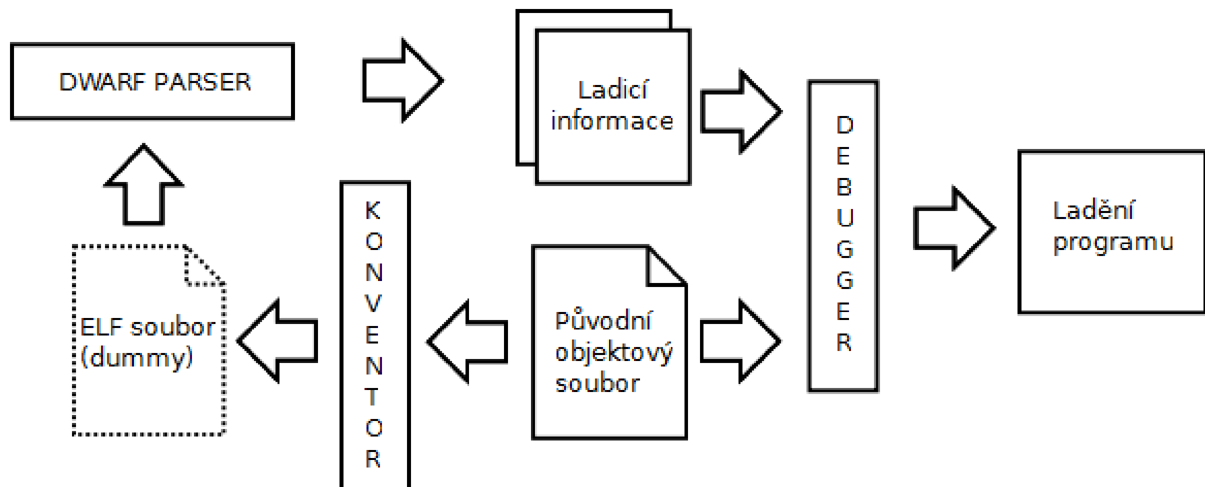
Při realizaci tohoto řešení ovšem vyšlo najevo, že provázanost knihovny s objektovým formátem ELF (obrázek) je natolik hluboká, že výsledkem tohoto postupu by nakonec stejně byl konvertor formátů objektových souborů. SymtabAPI pracuje v režimu získávání dat z kombinace formátů ELF a DWARF následovně. Nejprve získá základní informace pomocí knihovny Libelf, která zpracuje objektový soubor (ve formátu ELF), získá z něj například informace o architektuře, umístění tabulky symbolů aj. Dále používá knihovnu Libdwarf ke zisku a interpretaci ladicích informací. V neposlední řadě přistupuje SymtabAPI přímo k objektovému souboru, aby takto například prošel některé sekce (zobrazeno na následujícím schématu).



Obrázek 6. Popisuje hierarchii mezi jednotlivými zdroji informací. Informace jsou produkovány/získány z počátku šipky na žádost objektu na konci šipky.

5.2 Jak debugger pracuje

Po důkladném zvážení nastolených možností bylo rozhodnuto vydat se cestou konverze objektového souboru typu LOFF na formát ELF. Aby bylo zaručeno, že užitou konverzí nebudou do procesu ladění zaneseny případné chyby způsobené změnou formátu (obecně mohou různé typy objektových souborů obsahovat různé informace, pro které jinde nemusí být místo), výsledek převodu slouží pouze ke zisku ladicích informací. K vykonávání laděného programu se použije původní objektový soubor. Fungování debuggeru pomůže ilustrovat následující schéma.



Obrázek 7. Schéma debuggeru s využitím konvertoru. Původní objektový soubor představuje LOFF, který požadujeme ladit. Dummy je výsledek konverze sloužící k extrakci DWARF informací. DWARF parser je kombinací nástrojů libelf, libdwarf a SymtabAPI.

5.3 Implementace

Základní přidaná funkčnost, kterou se tato práce zabývá, je umožnění ladění na úrovni zdrojového kódu, přičemž schopnost ladit strojový kód musí zůstat zachována.

5.3.1 Podpora DWARFu

Nejprve bude zmíněna podpora debuggeru pro ladicí informace ve formátu DWARF. Zabývá se jím třída `CDbgDwarfExtractor`, která má za úkol zapouzdřit rozhraní knihovny SymtabAPI a poskytnout jeho výstupy jádru debuggeru. V metodě `ProcessFile` proběhne postupně konverze objektového souboru a výsledek je poté načten knihovnou SymtabAPI, kde dojde k jeho zpracování. Metoda `ProcessLineInfo` má za úkol získat z načtených informací údaje o řádcích a jejich

přiřazení k adrese programového čítače. Tyto informace budou sloužit k určení aktuální pozice ve zdrojovém kódu a případně k jeho vzájemnému mapování na kód v jazyce symbolických instrukcí.

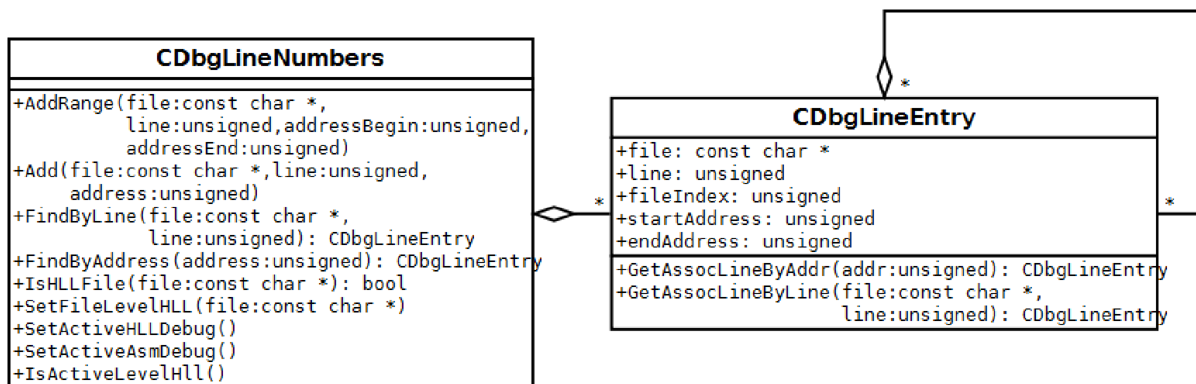
Aby bylo možno ladění s informacemi DWARF snadno odlišit od ladění bez nich (pro potřeby debuggeru), nadřazená třída `CDebuggerCore` má možnost detekovat přítomnost těchto informací pomocí metody `CheckDwarfContained`. Ta projde sekce objektového souboru a zjišťuje, zda některá z nich nenese jméno „*.debug_info*“. Její přítomnost signalizuje obsažení ladicích informací DWARF a nelze je bez ní používat. K výsledku této metody lze potom kdykoliv přistoupit metodou `HasDwarf`.

5.3.2 Ladicí informace

Třída `CDbgLineNumbers` slouží pro ukládání informací o vzájemném přiřazení čísel řádků a aktuální pozice v prováděném programu. Zahrnuje dvě hašovací tabulky, které slouží k vyhledávání podle umístění řádku ve zdrojovém souboru nebo podle hodnoty programového čítače. Původně tato třída obsahovala seznam všech souborů, ve kterých byla evidována pozice. Ten byl rozšířen na seznam struktur, kde se k danému souboru přidává informace, zda jde o zdrojový soubor s kódem zapsaným v jazyce C nebo v jazyce symbolických instrukcí. Jméno souboru společně s číslem řádku slouží při nastavování breakpointů, jeho dosažení a případně krokování. Třída dále nově obsahuje metody `SetActiveHLLDebug` a `SetActiveAsmDebug`, které umožňují přepínání mezi laděním na úrovni

`CDbgLineNumbers` na začátku této práce pracovala jako seznam informací o řádcích. Aby bylo možno zachovat co nejvíce stávající funkcionality, záznamy tohoto seznamu získaly schopnost odkazovat se na jiné záznamy. To je vyžadováno skutečností, že na jednu adresu programového čítače se zpravidla mapuje řádek ve zdrojovém souboru v jazyce C a několik řádků ze souboru v jazyce symbolických instrukcí.

Vyhledáme-li aktuální pozici podle adresy programového čítače nebo podle čísla řádku v konkrétním zdrojovém souboru (metody `FindByAddress`, `FindByLine`), vždy získáme jeden záznam seznamu, jehož typem je struktura `CDbgLineEntry`. Ta kromě určení pozice ve zdrojovém kódu, počáteční a koncové adresy programového čítače (rozsah je patrný pro jazyk C, pro asembler se tyto dvě adresy rovnají) obsahuje také seznam dalších záznamů, které připadají na stejný adresový rozsah. K těm lze přistupovat metodami této struktury `GetAssocLineByAddr` nebo `GetAssocLineByLine` podle požadované adresy nebo čísla řádku a zdrojového souboru. Ty slouží zejména při výběru správné informace pro krokování v závislosti na ladicí úrovni. Uspořádání zmíněných tříd dokreslí následující schéma.



Obrázek 8. Diagram tříd popisující hierarchii zpracování informací o číslech řádků.

5.3.3 Komunikace

Tyto metody struktury `CDbgLineEntry` se také používají při zasílání zpráv. Jelikož ladicí nástroj pracuje na simulační vrstvě, v případě dosažení bodu přerušení nebo provedení kroku musí informovat middleware a ten pak spraví klienta na prezentační vrstvě. Původně se zasílala pouze pozice v aktuálním zdrojovém souboru, s přidáním podpory pro ladění na úrovni zdrojového kódu přibyla potřeba zasílat polohu pro všechny aktivní zdrojové soubory, jelikož například ve vývojovém prostředí bude nutné umožnit přepínání mezi ladicími úrovněmi.

Protokol je dále obohacen příkazem *level*, který s parametry *c* nebo *asm* přepíná ladicí úroveň a jinak informuje o aktuálním ladicím stavu.

5.3.4 Další vývoj

Dosud zmíněné úpravy byly implementovány zároveň s jejich vizualizací v klientu příkazové řádky na prezentační vrstvě. V plánu projektu Lissom je implementace zobrazování těchto změn i v rozhraní vývojového prostředí Eclissom.

Možnosti ladění na úrovni zdrojového kódu zdaleka nekončí s určením pozice v kódu. V blízké budoucnosti je nejvhodnější přidat podporu pro zisk informací o proměnných a potažmo o dalších symbolech. V současnosti existuje v debuggeru jednoúrovňová tabulka symbolů, pro použití s vyššími programovacími jazyky je vhodné tuto tabulku symbolů přepracovat (více v [5]). Za uvážení rovněž stojí integrace tabulky symbolů používané v SymtabAPI. Knihovna disponuje rozhraním pro zisk informací, jako jsou lokální proměnné, argumenty funkcí a jiné. Při zkoumání využití SymtabAPI byl zjištěn problém při získávání informací o lokálních proměnných. Pokud totiž při detekci jejich umístění v paměti budou figurovat registry, knihovna neposkytuje rozhraní, pomocí kterého k nim lze přistoupit. Možným řešením může být doimplementování této funkcionality do knihovny SymtabAPI nebo přímo do debuggeru, obojí nejlépe s pomocí knihovny libdwarf.

6 Závěr

Hlavním úkolem této práce bylo prozkoumat možnost použití formátu ladicích informací DWARF při tvorbě multiplatformního debuggeru v rámci projektu Lissom. Využití tohoto formátu spočívá v poskytnutí podpory pro ladění na úrovni zdrojového kódu. To tedy představuje stěžejní rozšíření existujícího ladicího nástroje, který dosud pracoval pouze na úrovni strojového kódu.

Čtenář byl postupně seznámen se základy problematiky ladění a debuggerů, dále byly předvedeny některé formáty ladicích informací, zběžně proběhlo seznámení s formátem DWARF, práce popsala využití tohoto formátu a zdůvodnila jeho výběr. Jelikož je práce uváděna jako vzniklá v rámci projektu Lissom, čtenářovi byly poskytnuty některé základní informace o tomto projektu. Práce dále seznámila s aktuálním stavem debuggeru, nabídla možná řešení problému zpracování ladicích informací formátu DWARF a naznačila postup při integraci použitých nástrojů. Dále byl čtenář seznámen s provedenými úpravami ladicí knihovny. Proběhla také krátká diskuze o možném dalším vývoji debuggeru.

Jelikož doba, kdy bude člověk psát bezchybné programy bez asistence jakýchkoliv podpůrných nástrojů, je v nedohlednu a rostoucí složitost výsledných aplikací ji navíc s přehledem oddaluje, používání ladicích nástrojů bude pravděpodobně vždy součástí vývojového cyklu softwaru, a to samozřejmě i v oblasti paralelního návrhu hardware a jeho softwarového vybavení. Postupné zdokonalování prostředků podporujících vývoj, mezi které řadíme i debuggery, je žádoucí, pokud ne z důvodu, že zajistíme bezchybnost výsledného produktu, tak alespoň proto, že efektivnější nástroje jejich uživatelům pomohou lépe uspořit zdroje, zejména čas. Z tohoto důvodu se domnívám, že další snahy v navázání na výsledky této práce jsou bezpochyby žádoucí.

Literatura

[1] Eager, M. J.: Introduction to the DWARF Debugging Format, 2007. Dostupné na URL: <http://www.dwarfstd.org/doc/Debugging%20using%20DWARF.pdf>

[2] Free Standards Group: DWARF Version 3 Standard Released, 2006. Dostupné na URL: <http://www.dwarfstd.org/doc/Dwarf3.pdf>

[3] Menapace, J., Kingdon, J., MacKenzie, D.: The „stabs“ debug format [online]. c1992-1993, aktualizováno 1993 [cit. 2011-1-20]. Dostupné na URL: <http://docs.freebsd.org/info/stabs/stabs.pdf>

[4] Křoustek, J.: Vybrané metody ladění kódu, seminární práce, Brno, FIT VUT v Brně, 2009/2010.

[5] Wilczák, M.: Ladicí nástroj generických simulátorů procesorů, diplomová práce, Brno, FIT VUT v Brně, 2010

[6] Rosenberg, B. J.: How Debuggers Work – Algorithms, Data Structures, and Architecture, Wiley Computer Publishing, 1996.

[7] DWARF FAQ [online]. aktualizováno 2010-10-13 [cit. 2011-1-20]. Dostupné na URL: <http://wiki.dwarfstd.org/index.php?title=DWARF_FAQ>

[8] Kolektiv autorů TIS Comitee: Executable and Linkable Format (ELF). 1995. Dostupné na URL: <http://refspecs.freestandards.org/elf/elf.pdf>

[9] Haas, J.: Linux / Unix Command: readelf. Dostupné na URL: <http://linux.about.com/library/cmd/blcmdl1_readelf.htm>

[10] Meduna, A., Lukáš, R.: Formální jazyky a překladače. aktualizováno 2011-01-14. Dostupné na URL: <https://www.fit.vutbr.cz/study/courses/IFJ/private/prednesy/Ifj-all-pdf.zip>

[11] Paradyn project: SymtabAPI Programmer’s Guide. 2009, Dostupné na URL: <ftp://ftp.cs.wisc.edu/paradyn/releases/release6.1/doc/dyninstProgGuide.v61.pdf>

[12] Quenelle, C.: Stabs versus dwarf [online]. 2005-6-7, Dostupné na URL: <http://blogs.sun.com/quenelle/entry/stabs_versus_dwarf>

[13] Meduna, A., Lukáš, R.: Formální jazyky a překladače, studijní opora. Brno, FIT VUT v Brně, 2010. Dostupné na URL: <https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IFJ-IT/texts/OporaIFJ_rev2010-11-14.pdf>

[14] The LLVM Compiler Infrastructure Project [online]. 2003 [cit. 2011-05-12]. Dostupné z WWW: <http://llvm.org/>.

[15] Microsoft: PDB Files [online], [cit. 2011-5-8]. Dostupné na URL: <http://msdn.microsoft.com/en-us/library/yd4f8bd1(v=vs.71).aspx>

[16] Microsoft: /Z7, /Zi, /ZI, (Debug Information format) [online], [cit. 2011-5-8]. Dostupné na URL: <http://msdn.microsoft.com/en-us/library/958x11bc%28v=vs.80%29.aspx>

[17] Gray, R., Mulchandani, D.: Dr.Dobb's Journal [online]. 1997-05-01 [cit. 2011-05-08]. Object File Formats. Dostupné z WWW: <http://drdobbs.com/architecture-and-design/184410191>.

[18] Ramsey, N., Hanson, D. R.: A Retargetable Debugger. In Conference on Programming Language Design and Implementation [online]. San Francisco : [s.n.], 1992 [cit. 2011-05-09].

[19] Siero, M.: Blob.perl.org [online]. 98-10-07 [cit. 2011-05-10]. LDB Logic Debugger. Dostupné z WWW: <http://blob.perl.org/tpc/1998/User_Applications/LDB%20Logic%20Debugger/ldb_paper.html>.

[20] GNU Operating System [online]. Version 7.2. 1999, 2011-04-03 [cit. 2011-05-12]. GDB: The GNU Project Debugger. Dostupné z WWW: <http://www.gnu.org/software/gdb/>.

[21] Husár, A.: Implementace obecného assembleru. Brno, 2007, diplomová práce, FIT VUT v Brně.

[22] Kolář, D.: Návrh výstupního formátu pro assembler a linker [online]. [s.l.] : [s.n.], 5.2. 2004, 14.1. 2011 [cit. 2011-05-10]. Dostupné z CVS: <cvs://lissom.apsbrno.cz/cvs/lissom/doc/tool/vystupni format-0.00.5.doc>. Interní materiál.

[23] Masařík, K., Ilčík, O., Přykryl, Z.: Integrované vývojové prostředí [online]. [s.l.] : [s.n.], 2010-06-07, 2010-10-22 [cit. 2011-05-10]. Dostupné z CVS: <cvs://lissom.apsbrno.cz/cvs/lissom/binapp/ide_impl.doc>. Interní materiál.

[24] Novotný, T.: Jazyk ISAC - Příručka [online]. [s.l.] : [s.n.], 2006-08-16, 2008-01-10 [cit. 2011-05-10]. Dostupné z CVS: <cvs://lissom.apsbrno.cz/cvs/lissom/doc/Lisa/ISAC_manual.doc>. Interní materiál.