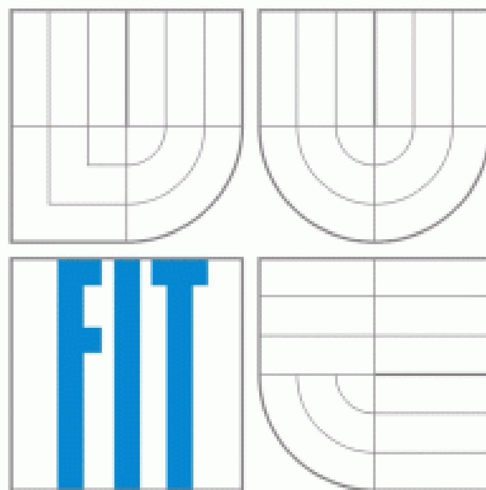
Teorie programovacích jazyků (TJD)

Vybrané metody ladění kódu

Seminární práce



Jakub Křoustek (xkrous00@stud.fit.vutbr.cz)

2009/2010

Abstrakt

Tento text byl vytvořen jako seminární práce v rámci předmětu Teorie programovacích jazyků. Práce prezentuje existující metody ladění kódu, zaměřuje se při tom především na ladění na úrovni zdrojového kódu. Cílem je seznámit čtenáře s touto problematikou a nastínit techniky používané pro vytvoření takového nástroje.

Klíčová slova

Ladění kódu, ladicí nástroj, překladač, linker, programovací jazyk, ladicí informace, COFF, DWARF, bod přerušení, krokování, mapování instrukcí, stack unwinding, tabulka symbolů, optimalizace, VLIW.

OBSAH

Obsah 3
1 Úvod 4
2 Proces překladu kódu 5
3 Ladění kódu pomocí ladicích nástrojů 7
3.1 Princip 7
3.2 Typy ladicích nástrojů 8
3.3 Vlastnosti symbolického ladicího nástroje 8
3.4 Existující ladicí nástroje 10
4 Tvorba ladicího nástroje 11
4.1 Koncept 11
4.2 Formát ladicích informací 12
4.3 Mapování kódu na zdrojové soubory 15
4.4 Krokování programu a body přerušení 16
4.4.1 Typy bodů přerušení 16
4.4.2 Nastavení bodu přerušení 17
4.4.3 Krokování 19
4.5 Zjišťování stavu zásobníku 20
4.6 Zobrazení hodnot proměnných 22
5 Ladění optimalizovaného kódu 23
5.1 Základní typy optimalizací 23
5.2 Architektury zaměřené na vysokou míru paralelismu 25
6 Závěr 27
7 Literatura 28

1 ÚVOD

„Jestliže ladění je procesem odstraňování softwarových chyb, pak programování musí být procesem jejich vytváření.“

Edsger Dijkstra

Společně se vznikem prvních programovacích jazyků se projevila i potřeba ladit programy pomocí nich vytvářené. Tato myšlenka se ukázala být správná, jelikož s rostoucí velikostí programu rapidně stoupá šance na výskyt chyby. Zpočátku bylo ladění kódu realizováno pouze pomocí instrumentace programu (tj. doplnění programu o kontrolní výpisy). Postupně se ale z ladění kódu stal metodický proces hledání a odstraňování chyb v počítačových programech, jenž je v současnosti již nedílnou součástí softwarového inženýrství.

S rostoucí složitostí softwaru i výpočetních systémů vzniklo mnoho metod a nástrojů pro jejich ladění. Jednotlivé druhy ladicích nástrojů můžeme dělit podle úrovní, na které se zaměřují. Můžeme například nalézt nástroje analyzující program staticky (buď jeho binární formu, nebo zdrojový kód), nástroje kontrolující korektní alokaci paměti a přístup k ní a dále tzv. debugger[1]. Jde o nástroj sloužící pro ladění programů za jejich běhu, který dokáže poskytovat informace o aktuálním stavu programu i systému, na kterém je program vykonáván.

Právě posledně jmenovaným nástrojem se zabývá tato práce, konkrétně se jedná o variantu ladění na úrovni zdrojového kódu. V následujících kapitolách budou popsány jednotlivé varianty nástroje a jeho požadované vlastnosti. Detailněji budou probrány principy a algoritmy použité při jeho implementaci a rovněž budou specifikovány podpůrné prostředky pro jeho funkčnost. Na závěr budou diskutovány problémy související s laděním optimalizovaného kódu.

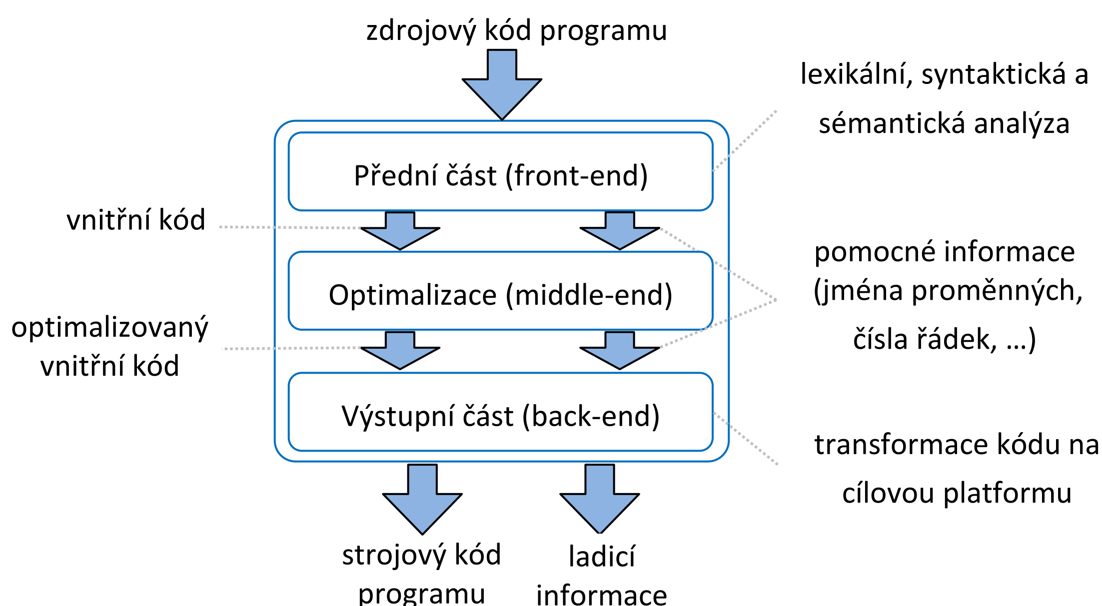
Tato práce je inspirována knihou How Debuggers Work – Algorithms, Data Structures, and Architecture od B. J. Rosenberga [Ros96].

[1] V dalším textu budeme místo slova *debugger* používat jeho český ekvivalent *ladicí nástroj*.

2 PROCES PŘEKLADU KÓDU

Proces ladění kódu úzce souvisí s oblastmi programovacích jazyků a překladačů. Z důvodu pozdějšího odkazování na tyto souvislosti je vhodné na tomto místě uvést některé základní pojmy.

Proces překladu programů z lidsky čitelného kódu ve vyšším programovacím jazyce do binárního, strojově zpracovatelného kódu je poměrně komplexní. Lze ho chápat jako několika úrovňovou transformaci zdrojového kódu do nižších forem reprezentace, kdy při každém kroku transformace dochází ke ztrátě informací o původní podobě kódu. Výsledkem překladu je typicky sekvence instrukcí z instrukční sady cílového procesoru. Tyto instrukce jsou oproti původním příkazům programovacího jazyka značně jednoduché. Z pohledu procesoru tak dochází k abstrakci informace o původně použitém programovacím jazyku, jelikož procesor dokáže vykonávat stále jenom svou instrukční sadu, bez ohledu na původní jazyk. Proces překladu ilustruje [Obrázek 1], detailnější popis je k nalezení v [ALSU06].



OBRÁZEK 1 – OBECNÁ STRUKTURA PŘEKLADAČE

Při lexikální a syntaktické analýze vstupního zdrojového kódu programu jsou o něm překladačem získávány dodatečné informace, jako jsou jména proměnných, čísla řádků s deklaracemi a použitími proměnných a funkcí atd. Sémantická analýza tyto znalosti dále rozšiřuje o informace o datových typech proměnných či argumentech funkcí. Tyto informace mohou být použity při generování **ladicích informací**, jež poté poslouží při ladění programu.

Na generátory ladicích informací jsou kladeny následující požadavky:

1) Generování ladicích informací musí ovlivnit vlastní překlad jenom minimálně.
2) Ladicí informace musí korespondovat s vygenerovaným kódem.
3) Formát ladicích informací musí dodržovat zvolený standard (DWARF, STABS atd.) tak, aby byla zajištěna kompatibilita s existujícími ladicími nástroji.

Části překladače, které kód optimalizují a transformují do strojového kódu, generování ladicích informací naopak komplikují, protože v ladicích informacích pak vznikají nekonzistence, které ne vždy lze vyřešit. To je konflikt s výše uvedeným požadavkem (2). U některých překladačů se při explicitním zapnutí generování ladicích informací nekonzistentní optimalizace vypínají, viz např. [LLVM10]. Vypínání optimalizací by se ale mělo dít pouze v krajních případech, neboť se tím značně ovlivňuje funkčnost překladače, což je ve sporu s požadavkem (1).

Formát a typy ladicích informací budou detailněji popsány v dalších kapitolách.

3 LADĚNÍ KÓDU POMOCÍ LADICÍCH NÁSTROJŮ

3.1 PRINCIP

Základ slova debugger pochází z anglického *bug*, což je možné přeložit jako brouk. Brouk, přesněji řečeno mol, který vletěl do jednoho relé počítače Mark II, byl také první zdokumentovanou chybou při vykonávání programu [Wiki01]. Proces vyčištění relé dal vzniknout slovu **debuggování**. Stalo se tak v roce 1945. Více o historii ladění kódu je k nalezení v [Hay93].

Během let dostal pojem debugger více sofistikovaný význam a v současnosti pod tímto názvem chápeme ladicí nástroj, který slouží k hledání, izolování a odstranění chyb v softwarových programech za jejich běhu. Liší se tak tedy od jiných ladicích technik jako je instrumentace kódu, post-mortem analýzy, profilování či kontroly přístupu k paměti.

Jde o poměrně složitý nástroj, který k ladění programu potřebuje dodatečné (ladicí) informace vzniklé při speciálním způsobu překladu programu. Dále je často silně závislý na operačním systému, na kterém laděný program běží.

Tento ladicí nástroj kontroluje laděný program po celou dobu jeho vykonávání a umožňuje vykonávání na kterémkoliv místě přerušit. To se děje za pomoci tzv. **bodů přerušení** (anglicky **breakpoint**). Vykonávání může být rovněž prováděno v několika módech, například krokováním po příkazech či spuštěním až do dalšího přerušení. Uživateli musejí být při ladění zprostředkovány informace o dynamickém chování laděného programu, jako pozice v kódu, stavy proměnných či paměti. Tyto informace pak slouží jednak k pochopení samotného programu, tak k detekci chyb.

Při ladění a testování kódu je důležité dodržovat tzv. **Heisenbergův princip** [Gra83]. Tato poučka říká, že při ladění uživatelského programu nesmí dojít k ovlivnění jeho chování ladicím nástrojem. V praxi je však Heisenbergův princip často porušován, jelikož není vždy možné zcela izolovat ladicí nástroj od laděného programu. Už jenom fakt, že ladicí nástroj je nahrán do stejné paměti jako analyzovaný program, či že je vykonáván stejným operačním systémem, může způsobit změnu chování programu. Stejně tak každá chyba vyvolaná ladicím nástrojem nebo i jen jeho nepřesnost, může způsobit značný zmatek. Může totiž dojít ke špatné interpretaci chyby a uživatel může nabýt dojmu, že chyba je v laděném programu. Proto by měla být bezchybnost ladicího nástroje zaručena.

3.2 TYPY LADICÍCH NÁSTROJŮ

Existuje několik typů ladicích nástrojů, v závislosti na zvoleném typu úrovně ladění.

Ladění jádra (*kernel debugging*). Tento typ slouží pro ladění jader operačních systémů či ovladačů pro ně. Oproti dále zmíněným typům probíhá ladění na jiném stroji, než na kterém běží laděný operační systém. Hovoříme pak o vzdáleném ladění. Stroj, na kterém běží ladicí nástroj, vzdáleně sleduje stavy laděného systému a ten také pomocí komunikačního protokolu řídí. To vyžaduje speciální verzi operačního systému, kdy jeho výrobce musí vytvořit verzi s ladicími informacemi a pouze omezeně optimalizovanou.

Ladění na úrovni strojového kódu (*machine-level debugging*). Ladění na úrovni strojového kódu je vhodné tam, kde nejsme schopni získat pro laděný program ladicí informace ani původní zdrojový kód. Jsme tak omezení pouze na nejnižší úroveň reprezentace kódu, která je platformě závislá. Zobrazení kódu typicky usnadňuje zabudovaný zpětný assembler (*disassembler*), jenž převede strojový kód do jazyka symbolických instrukcí (více o tomto tématu je k nalezení v [Kro07]). Tím uživatel získá představu o podobě původního kódu, byť na takto nízké úrovni, i když zdrojové kódy nemá k dispozici.

Uživatel tedy musí znát instrukční sadu konkrétního procesoru a jeho zdroje (registry, paměťovou hierarchii, způsob implementace zásobníku atd.). Kvůli absenci ladicích informací není možné zobrazovat údaje o hodnotě proměnných, graf toku řízení ani další užitečné informace. Výhodou je, že na této úrovni jde ladit každá aplikace pro danou platformu. Ladicí nástroje na úrovni strojového kódu jsou také nejčastěji používány v tzv. *reverzním inženýrství*, které může sloužit pro takové zkoumání či transformování programu, které nebylo jeho autorem původně zamýšleno.

Ladění na úrovni zdrojového kódu, či také **symbolické ladění** (*source-level symbolic debugging*). Ladění na úrovni zdrojového kódu je velice intuitivní způsob ladění uživatelských programů. Tento způsob poskytuje model prezentující provádění na úrovni implementačního jazyka. Uživatel tak získává iluzi, že jeho program je prováděn po příkazech z tohoto jazyka a nikoliv jako sekvence instrukcí konkrétního procesoru. Ladění tímto způsobem se ukázalo být jako nejefektivnější a proto je tato technika používána nejčastěji. Touto variantou se bude zabývat následující text.

3.3 VLASTNOSTI SYMBOLICKÉHO LADICÍHO NÁSTROJE

Ladění programu za jeho běhu nemusí být pro uživatele jednoduchý úkol. Při ladění totiž potřebuje mít přehled o aktuálním stavu programu i hostujícího stroje. Z toho důvodu musí ladicí nástroj poskytovat komplexní kontextové informace o programu. Dobrý ladicí nástroj

by se měl řídit mottem, že uživatel potřebuje znát tolik informací, kolik jich je jenom možné získat. Mezi tyto informace patří následující položky:

- **Pozice ve zdrojovém kódu**. Zjištění aktuální lokace v původním zdrojovém programu je nejzákladnější vlastností při symbolickém ladění. Uživatel pohledem do zdrojového kódu získá informaci o aktuální pozici běhu programu. Téměř každý ladicí nástroj dokáže zvýraznit aktuální řádek v rámci zdrojového kódu, ty lepší pak i detailnější části, jako jsou konkrétní výrazy.
- **Stav zásobníku** (*stack backtrace*). Jde o druhou nejdůležitější informaci při ladění na této úrovni. Pomocí pozice ve zdrojovém kódu dokážeme zjistit, v jaké části kódu se aktuálně program nachází. Pomocí stavu zásobníku dokážeme rozpoznat, jak se program do této části dostal. Většina běžných architektur totiž ukládá při volání funkcí návratové adresy na zásobník. Při vícenásobném zanoření tak můžeme sledovat hierarchii volání funkcí zpětným průchodem přes vrchol zásobníku. Krom toho můžeme zjišťovat i s jakými argumenty byly funkce volány (rovněž typicky ukládané na zásobník).
- **Hodnoty proměnných**. Sledování hodnot uživatelských proměnných je užitečná informace například při ladění nekritických chyb (chyba „o jedničku“, špatná inicializace proměnné apod.). Oproti ladění na úrovni strojového kódu získává uživatel přímo hodnoty mapované na původní názvy proměnných a nemusí jejich hodnoty zdlouhavě hledat.
- **Informace o jednotlivých vláknech programu**. U více-vláknových aplikací musí ladicí nástroj udržovat i informace o stavu jednotlivých vláken. Stejně tak u více-procesových aplikací je nutné brát v potaz (kvůli mezi-procesové komunikaci) kontext ostatních procesů, tedy nejenom těch vytvořených laděným programem.
- **Stav procesoru**. Při ladění na úrovni zdrojového kódu je uživatel zpravidla oproštěn od nutnosti sledovat stav hostujícího stroje, ale i tato možnost může být užitečná. Tímto způsobem lze zjistit hodnoty registrů procesoru, obsah paměti či aktuálně zpracovávaný strojový kód. Nejde tedy o nic jiného, než o informace poskytované ladicími nástroji na úrovni strojového kódu. Můžeme tedy říci, že ladění na úrovni zdrojového kódu je jeho nadstavbou.

Velice užitečným rozšířením ladicích nástrojů je i jejich integrace do vývojového prostředí (IDE). Vývojář tak může svůj program ladit ve stejném prostředí, ve kterém ho vyvíjí. Existují samozřejmě i ladicí nástroje pracující v příkazové řádce, ty jsou pro zkušenější uživatele rychleji ovladatelné, na druhou stranu ale ztrácejí možnost interaktivně opravovat chyby ve zdrojovém kódu.

3.4 EXISTUJÍCÍ LADICÍ NÁSTROJE

Ladicí nástroj může být přímo implementován v programovacím prostředí (např. Microsoft Visual Studio) nebo existovat jako separátní aplikace (např. SoftICE).

Mezi nejznámější ladicí nástroje patří:

- **GDB** a **DDD**. *GNU Debugger* a jeho grafická nadstavba *DDD*. GDB samotné pracuje pouze v příkazové řádce, DDD přidává vstřícnější uživatelské rozhraní. GDB pracuje s mnoha programovacími jazyky, např. C, C++, Ada, Fortran, podporuje přes dvě desítky architektur procesorů a jako první debugger podporoval vzdálené ladění (*remote debugging*). Od roku 2009 je podporována funkce zpětného ladění (*reverse debugging*). Jedná se o možnost zpětného krokování již vykonaných instrukcí.
- **Microsoft Visual Studio Debugger**. Tento ladicí nástroj je součástí každé verze vývojového prostředí Visual Studio .NET. Funkčně vychází z aplikace CodeView, což byl textový nástroj v prostředí Microsoft Visual C++. Je stabilní, rychlý a jednoduchý na použití. Záporem je neschopnost ladit jaderné funkce operačního systému.
- **WinDbg**. Další ladicí nástroj od firmy Microsoft. Je určen pouze pro Microsoft Windows. Jedná se o víceúčelový nástroj, který je schopen mimo jiné i ladit ovladače či jaderné služby operačního systému na bázi Windows NT. Využívá se také při zkoumání pádů operačního systému.
- **SoftICE**. Jde o velice účinný ladicí nástroj, který pracuje na nejnižší vrstvě operačního systému. V roce 1987 vytvořila společnost Numera první verzi, vyvíjen byl bezmála 20 let až do roku 2006, kdy byl jeho další vývoj pozastaven. SoftICE pracuje pod operačním systémem Microsoft Windows (dříve také pod Microsoft MS-DOS) a je určen především pro ladění ovladačů hardwarových zařízení, ale díky svým vlastnostem je používán i pro ladění ostatních programů či v reverzním inženýrství.
- **OllyDbg**. OllyDbg je rychlý ladicí nástroj pro operační systém Microsoft Windows, jenž je distribuován jako freeware. Mnohdy se používá jako alternativa k SoftICE. Díky obrovské uživatelské základně pro tento program existuje nepřeberné množství rozšíření a vylepšení.

4 TVORBA LADICÍHO NÁSTROJE

Ačkoliv je využití ladicího nástroje při vývoji softwaru kritické, metodika jeho tvorby není zdaleka tak dobře zdokumentována jako je tomu například u překladačů. Tento fakt je zarážející, jelikož čas strávený laděním programu je mnohonásobně vyšší než čas potřebný pro jeho překlad. Složitost návrhu ladicího nástroje spočívá především v silné závislosti na hostujícím operačním systému, bez jehož pomoci je ladění prakticky nemožné[2].

V této kapitole bude uveden koncept symbolického ladicího nástroje, některé algoritmy pro dosažení požadované funkcionality a dále způsob reprezentace ladicích informací. Popisovaný nástroj bude chápán jako obecný, tedy nebude zaměřen na konkrétní jazyk ani architekturu.

4.1 KONCEPT

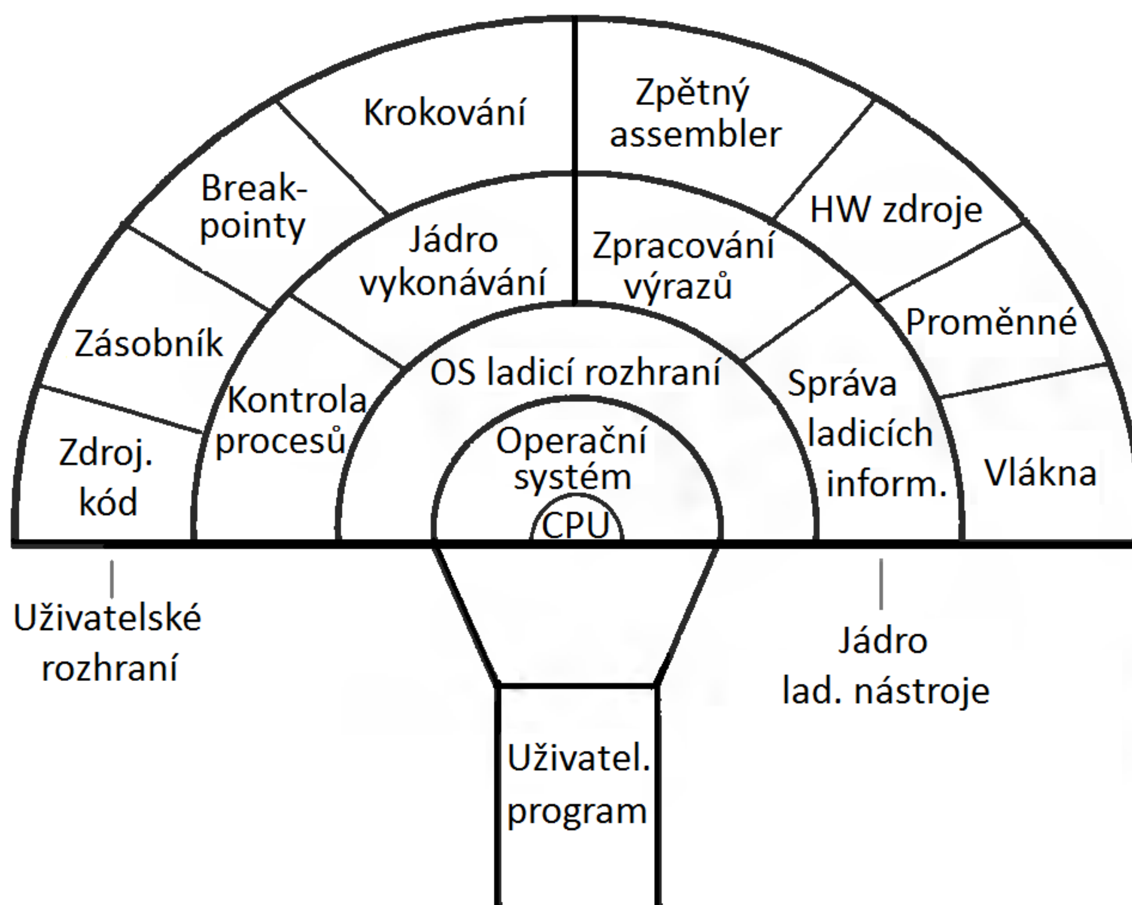
Obecnou strukturu ladicího nástroje zachycuje [Obrázek 2]. Ladicí nástroj může pracovat ve dvou režimech – v režimu příkazové řádky a v režimu s grafickým uživatelským rozhraním (GUI). Vlastní jádro zůstává stejné, u nástrojů s grafickým rozhraním je navíc nutné ošetřit přijímání událostí od operačního systému a jejich zpracování. Ladicí nástroj se pak typicky skládá ze dvou vláken – jedno řídí laděný program, druhé reaguje na události a vykresluje grafické uživatelské rozhraní. Samotné grafické rozhraní slouží pro vizualizaci údajů definovaných v kapitole 3.3 (údaje o stavu programu, pozice ve zdrojovém kódu apod.).

O úroveň níže pod uživatelským rozhraním se nachází vlastní jádro nástroje. Jedná se o část, která se stará o poskytování údajů, které zobrazuje uživatelské rozhraní. Hlavně ale tato vrstva řídí samotné ladění programu. Program, který chceme ladit, je z pohledu operačního systému procesem. Při začátku ladění máme možnost buďto vytvořit nový proces nebo ladit proces již existující (tzv. připojení k procesu). Po skončení ladění musí ladicí nástroj nově vytvořené procesy odstranit.

Jádro je dále zodpovědné za přístup k ladicím informacím, krokování programu a nastavování a kontrolování bodů přerušení. Body přerušení mohou být podmíněné (viz dále), z toho důvodu musí jádro umět vyhodnocovat výrazy v definovaném jazyce. Jde buďto o jednoduchý jazyk k tomu účelu navržený, či o stejný jazyk jakým byl vytvořen laděný program.

[2] Existují ale i metody, jak ladicí nástroj závislosti na hostujícím operačním systému zbavit. Jedním z možných řešení je emulace celé platformy. Laděný program pak běží v rámci platformě nezávislého simulátoru.

Pro přístup k procesu je nutná interakce mezi jádrem ladicího nástroje a operačním systémem. K tomu slouží mezivrstva komunikující s operačním systémem podle standardního rozhraní (API). Dále je mezivrstva používána pro zjišťování stavu procesoru a také se pomocí ní oznamuje stav laděného programu.



OBRÁZEK 2 - TYPICKÁ ARCHITEKTURA LADICÍHO NÁSTROJE NA ÚROVNI ZDROJOVÉHO KÓDU [ROS96].

4.2 FORMÁT LADICÍCH INFORMACÍ

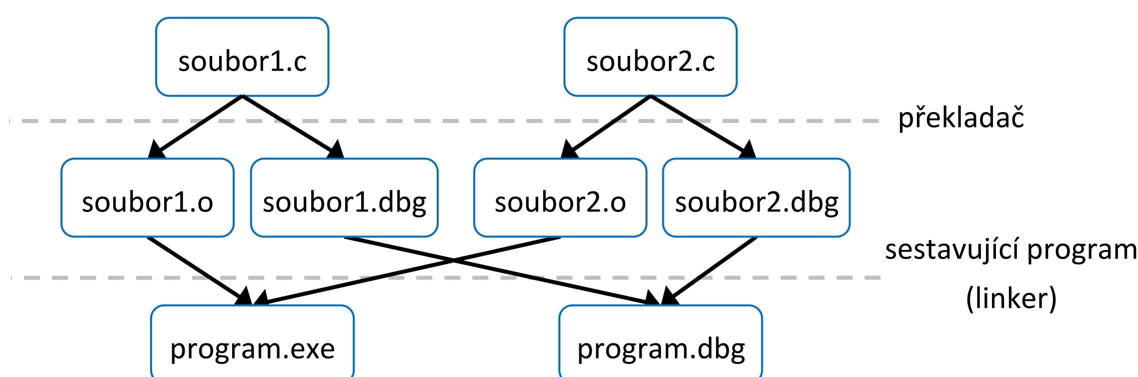
V kapitole 2 byly nastíněny pomocné údaje získané překladačem při generování strojové reprezentace programu. Tyto údaje mohou být uloženy jako ladicí informace a následně pak použity při ladění. Souhrnně jde o tyto údaje:

- Názvy a typy proměnných a jejich umístění v paměti.
- Názvy a typy funkcí a jejich parametrů a jejich umístění v paměti.
- Jména souborů se zdrojovými kódy a čísla řádků odpovídající dílčím úsekům kódu.

- Informace potřebné k vytvoření stavu zásobníku (*stack backtrace*).
- Informace o expandovaných makrech.

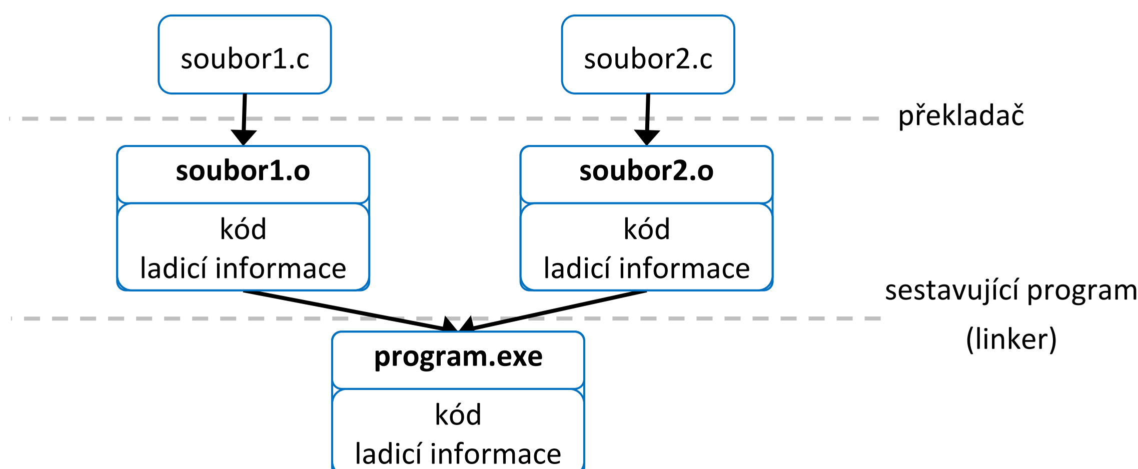
Je úkolem ladicího nástroje tyto informace načíst a při ladění vhodně využít. Formát ani způsob uložení ladicích informací není jednotný a existuje hned několik běžně používaných standardů. Pro způsob uložení existují dvě možnosti – uložení ladicích informací odděleně od programu nebo společně s kódem do jednoho souboru.

První možností je uložení informací **oddělené od programu**, viz [Obrázek 3]. Překladač tedy vytvoří soubor se strojovým kódem programu a navíc i soubor s ladicími informacemi. Vzájemně jsou na sebe vázány jménem a cestou odpovídajícího souboru. Tento styl je používán především společností Microsoft v jejím PDB (dříve DBG) ladicím formátu.



OBRÁZEK 3 – GENEROVÁNÍ LADICÍCH INFORMACÍ ODDĚLENĚ OD PROGRAMU

Ladicí informace jsou uloženy **společně s programem v jednom souboru**, viz [Obrázek 4]. Výhodou je zjednodušení distribuce aplikací, kdy není nutné přikládat soubor s ladicími informacemi. Ty jsou v tomto případě uloženy jako jedna či více speciálních sekcí programu. Toto řešení je nejčastější, používají ho formáty STABS, COFF či DWARF.



OBRÁZEK 4 – GENEROVÁNÍ LADICÍCH INFORMACÍ DO SPOLEČNÉHO SOUBORU

Mezi nejběžněji používané ladicí formáty patří:

- **STABS** (*Symbol TABle Strings*). Pojmenování vychází z konceptu ukládání ladicích informací. Ty jsou kódovány a ukládány jako řetězce v tabulce symbolů. Vzhledem k rostoucímu množství ukládaných informací se toto řešení stalo neefektivním. Formát STABS byl nejprve používaný v UNIXovém *a.out* objektovém formátu, později si tento formát převzaly a modifikovaly některé společnosti (např. Sun Microsystems).

  Nevýhodou je i fakt, že pro tento formát neexistuje žádný standard. Ukázka ladicí informace z tabulky symbolů, která charakterizuje funkci `main`:

  ```
  .stabs "main:F1", 36, 0, 0, _main
  ```

- **COFF** (*Common Object File Format*). COFF je standardem objektového souboru, tedy nejenom ladicích informací v něm obsažených. Díky podpoře pojmenovaných sekcí může takový objektový soubor obsahovat ladicí informace i v jiném než COFF standardu (např. STABS či DWARF), což může být matoucí. Existuje několik verzí tohoto formátu: XCOFF (pro architekturu IBM RS/6000), ECOFF (architektury MIPS a Alpha) či PE-COFF (pro platformu Windows).

  Ladicí informace jsou ukládány jako binární data v sekcích se speciálními jmény. Například ve formátu PE-COFF můžeme nalézt následující sekce s ladicími informacemi:

  `.debug$T` – sekce s informacemi o datových typech
  `.debug$S` – sekce s informacemi o symbolech

  Informace o mapování kódu na řádky zdrojového kódu jsou ve formátu COFF uloženy v globální tabulce, ve formátu PE-COFF je pro jejich uložení využit vlastní formát – *CodeView* (později *CV4*).

- **PDB** (*Program DataBase*). Jde o formát společnosti Microsoft, který je využíván většinou jejích překladačů. Ladicí informace jsou ukládány v externím souboru. Formát PDB je proprietární a zpracování takových souborů je tedy možné jen za pomoci poskytované mezivrstvy nazvané *Debugging Tools for Windows*.
- **DWARF** (*Debugging With Attributed Record Formats*). Historie formátu DWARF sahá až do roku 1980, kdy byl formát navržen pro ladění programů v jazyce C na systému UNIX. První verze specifikace vznikla až v roce 1989, v současnosti je před dokončením její čtvrtá verze [FSG06].

DWARF není závislý na programovacím jazyku, operačním systému ani architektuře procesoru. Nejčastěji je však spojován s objektovým formátem ELF. Je běžně používaným standardem (nativní formát llvm, podpora v gcc a gdb atd.). Formát dokáže mimo jiné uložit informace o datových typech a jménech proměnných, typy a argumenty funkcí, adresy symbolů v paměti, informace o uživatelských datových typech, popis zásobníku, ABI (*Application binary interface*) atd. Oproti ostatním zvládá jemnější určování pozice v souboru, kdy kromě čísla řádku obsahuje i informaci o číslu sloupce [Eag07]. Některé další rozdíly oproti formátu COFF jsou k nalezení v [Dar05].

DWARF ladicí informace jsou ukládány do pojmenovaných sekcí s předdefinovaným významem (prefix „.debug_"). Od verze DWARF 2 se provádí komprese ladicích informací. Děje se tak pomocí používání zkratek (sekce .debug_abbreviations), formátem (U)LEB128 pro uložení čísel na minimálním počtu bitů a pomocí programů v bajtkódu pro generování tabulek informací. Ukázka některých sekcí:

`.debug_info` – sekce s vlastními ladicími informacemi
`.debug_abbrev` – sekce se zkratkami používanými v `.debug_info`
`.debug_line` – sekce s informacemi o číslech řádků
`.debug_ranges` – sekce s informacemi o mapování symbolů do paměti

4.3 MAPOVÁNÍ KÓDU NA ZDROJOVÉ SOUBORY

Jak bylo již řečeno, úkolem překladače je transformovat zdrojový kód do strojového, který bude proveden přímo na hardwarové platformě. Mapování strojového kódu zpět do zdrojového není pro ladicí nástroj triviální úloha. Bez podpory na úrovni překladače (při generování ladicích informací) není možné informace o mapování dodatečně získat.

V nejjednodušší formě si můžeme tyto informace představit jako tabulku se třemi sloupci – fyzickou adresou kódu v programu, názvem souboru se zdrojovým kódem a číslem řádku v souboru. Při nastavení bodu přerušení na některý řádek zdrojového kódu se pak zjistí odpovídající fyzická adresa. Toto je případ ladicího formátu COFF.

S tím, jak překladače optimalizují kód programu, může dojít k posunu instrukcí či k jejich odstranění. Kód pro daný příkaz zdrojového kódu nemusí být uložen sekvenčně, ale proloženě s instrukcemi pro jiné příkazy. Tento jev je typický pro architektury s vysokou úrovní paralelismu v rámci instrukcí (např. VLIW). Proto je vhodné do tabulky s čísly řádků uložit i informaci, zda se nejedná o začátek či konec kódu daného příkazu (tzv. *prolog* a *epilog*). Ladicí nástroj pak nemusí prohledávat celou tabulku a zjišťovat, zda neexistuje ještě

jiná instrukce patřící k aktuálnímu příkazu. Další užitečnou informací je uchovávání čísla sloupce ve zdrojovém kódu. Tato informace dokáže zjemnit určení pozice až na úroveň výrazů či operátorů v rámci složitějších příkazů[3]. Mezi další informace patří kupříkladu identifikace základních bloků či určení instrukční sady instrukce (u architektur s více instrukčními sadami jako je MIPS, ARM či Intel Itanium). Ukázka těchto informací ve formátu DWARF:

adresa	soubor	řádek	sloupec	příkaz	blok	konec	epil.	prol.	ISA
0x0	xyz.c	42	2	ano	ne	ne	ne	ano	ARM
0x9	xyz.c	44	2	ano	ne	ne	ne	ne	ARM
0x2c	abc.c	10	4	ano	ano	ne	ne	ano	Thumb

4.4 KROKOVÁNÍ PROGRAMU A BODY PŘERUŠENÍ

Po úspěšném napojení na proces laděného programu je možno začít ovlivňovat jeho běh. Nejzákladnějším principem je používání bodů přerušení (*breakpoint*). Ty mohou sloužit pro krokování po instrukcích, pro zastavení programu na uživatelem zvoleném místě či pro podmíněné řízení toku programu.

Body přerušení mohou být vkládány na úrovni zdrojového kódu (typicky na řádek ve zvoleném souboru) nebo na adresy strojových instrukcí. Prvně zmíněný typ může znamenat nastavení bodů přerušení na více fyzických adres (např. inline funkce v jazyce C, šablony v C++ atd.). Může ale nastat i opačný případ, kdy se více vysokoúrovňových bodů přerušení namapuje na jednu fyzickou adresu. Vztah mezi nimi je tedy obecně N:M.

Jakmile uživatel nastaví body přerušení, je možno program spustit. Spuštění může proběhnout dvěma způsoby – buďto plnou rychlostí (až do dosažení bodu přerušení) nebo provedení jednoho kroku.

4.4.1 TYPY BODŮ PŘERUŠENÍ

Podle vlastností či způsobu použití rozlišujeme celou řadu bodů přerušení. Výčet těch nejdůležitějších následuje, další typy jsou k dohledání v [Wil10] a [Ros96].

Logické a fyzické. Logický bod přerušení je více abstraktní a určuje, na kterém řádku a v kterém souboru zdrojového kódu (případně symbolické jméno návěští či funkce) se má přerušení provést. Mohou rovněž obsahovat aktivační podmínku, za které se přerušení vykoná (př. počet průchodů, příznak, že je bod aktivní apod.). Struktura pro uložení těchto bo-

[3] Ač jsou ladicí formáty na uložení této informace připraveny, podpora pro její generování na straně současných překladačů je spíše sporadická.

dů je poměrně triviální, obsahuje informace o řádku a zdrojovém souboru, případně adrese zvoleného symbolu z tabulky symbolů v objektovém souboru programu.

Fyzický bod představuje reprezentaci přerušení ve strojovém kódu laděného programu, vytvořenou nejčastěji pomocí substituce některé instrukce za speciální instrukci přerušení. Každý fyzický bod má svou adresu, zálohu originální instrukce před nahrazením a počet logických bodů přerušení, které na něj odkazují. Pro rychlejší kontrolu dosažení přerušení obsahují struktury i seznam všech odpovídajících logických bodů.

Dočasné, trvalé a interní. Dělení závisí na uchování bodu přerušení po jeho dosažení. Dočasné body se po prvním dosažení zruší, trvalé v programu zůstávají i nadále. Někdy je také možné specifikovat, po kolikátém dosažení se přerušení vykoná. To je vhodné například při ladění cyklů. Interní body přerušení jsou důležité pro správnou funkčnost krokování programu (např. krokování po instrukcích či krokování přes funkce), ale pro uživatele jsou skryté. Interní body jsou často zároveň dočasné, což je určeno atributem ve struktuře logického bodu přerušení.

Datové (*data breakpoint*, *watchpoint*). Datové body přerušení jsou používány pro odhalení chyb způsobených neočekávanou hodnotou proměnných. K přerušení dochází pouze v tom případě, kdy je detekován přístup (případně i modifikace) k některým zdrojům procesoru. Může se například jednat o zápis určité hodnoty do paměti na požadované adrese. Výběr zdroje i kontrolované hodnoty je definován uživatelským výrazem. Hardwarová podpora (kontrola adres či stránkování) je téměř nezbytná, protože kontrola na úrovni ladicího nástroje by běh programu výrazně zpomalovala.

Body přerušení lze využít i tak, že při jejich aktivaci se neprovede přerušení, ale obecně jakákoliv uživatelem definovaná akce. Příkladem může být instrumentace kódu, kdy se na zvolených místech vypíší informace o stavu programu, nebo se bude zaznamenávat počet průchodů přes toto místo. V kombinaci s aktivačními podmínkami je možné program odladit bez nutnosti rekompilace, změny se ovšem neuloží do spustitelného souboru.

4.4.2 NASTAVENÍ BODU PŘERUŠENÍ

Při nastavení bodu přerušení se postupuje tak, že podle zvolené lokace bodu přerušení se nejprve vytvoří logický bod, obsahující informace o pozici v souboru. Ten se pak bude odkazovat na fyzický bod přerušení, jenž představuje fyzickou adresu v programu. Při vložení bodu přerušení do programu pak ladicí nástroj získá od operačního systému instrukci na požadované adrese a tu uloží do struktury fyzického bodu přerušení. Následně operačnímu systému místo ní pošle instrukci, která vyvolá zastavení laděného programu a předání řízení ladicímu nástroji. Ten po přijetí řízení obnoví původní instrukci tak, aby v toku řízení bezpro-

středně následovala, a odpovídajícím způsobem dekrementuje programový čítač. Více ilustruje algoritmus nastavení bodu přerušení ve zdrojovém kódu:

Vstup: Jméno souboru a číslo řádky, kde má být běh programu přerušen.
Výstup: Fyzická adresa bodu přerušení nebo indikace chyby.
Postup: Mapování jména souboru a čísla řádky na fyzickou adresu pomocí tabulky symbolů a logických a fyzických bodů přerušení.

1) Získání tabulky symbolů pro zvolený soubor a mapování řádků na instrukce z ladicích informací. Výpis chyby pokud tyto informace nejsou k dispozici.
2) Vytvoření struktury nového logického bodu přerušení, která bude obsahovat informace z části 1).
3) Vytvoření struktury (či inkrementace počítadla) fyzického bodu přerušení, na který se bude odkazovat logický bod z části 2).
4) Získání instrukce na požadované adrese a její záloha ve struktuře fyzického bodu přerušení.
5) Nahrazení instrukce v programu na této adrese za instrukci vyvolávající přerušení.

Operační systém pro nastavení i detekování bodu přerušení potřebuje podporu na úrovni hardwaru. Ta může být realizována několika způsoby:

- **Speciální instrukce**. Tato varianta je nejčastější. V instrukční sadě je vyčleněna speciální instrukce sloužící k vyvolání přerušení operačního systému za účelem ladění. Příkladem je instrukce `INT3` u architektur Intel x86 a AMD64, `BREAK` u architektury MIPS či `BPT` u procesorů Alpha.
- **Neplatná instrukce**. U jiných architektur (např. PowerPC) neexistuje speciální instrukce používaná pro nastavení bodu přerušení, místo toho se použije instrukce s neplatným operačním kódem. Instrukční dekodér procesoru musí s touto variantou počítat.
- **Hardwarové body přerušení**. Jde o jiný přístup než dvě předchozí metody. Procesor v tomto případě obsahuje tzv. *ladicí registry*, jejichž hodnoty nastavuje ladicí nástroj na adresy požadovaných bodů přerušení. V každém cyklu je pak jejich hodnota komparována s programovým čítačem, a pokud hodnota souhlasí, tak je vyvoláno přerušení. Odpadá tak potřeba modifikovat program. Příkladem jsou architektury Intel x86 (registry `DR0`-`DR3` a řídicí registr `DR7`) a AMD64.

4.4.3 KROKOVÁNÍ

Při vyvolání přerušení za běhu programu se ladicí nástroj pokusí podle hodnoty programového čítače najít fyzický bod přerušení. V odpovídající struktuře se zjistí logický bod přerušení a následně se ověří splnění případné podmínky aktivace. První bod přerušení, který má splněny aktivační podmínky, se použije pro zobrazení lokace ve zdrojovém kódu. To se děje pomocí algoritmu aktivace bodu přerušení:

Vstup: Oznámení operačního systému o zastavení laděného programu na určité adrese.

Výstup: Zastavení či pokračování běhu laděného programu v závislosti na postupu níže.

Postup: Mapování fyzických adres na jména souborů a čísla řádků pomocí struktur logických a fyzických bodů přerušení.

1) Prohledání seznamu logických bodů přerušení podle zjištěné adresy přerušení (logických bodů může být i více).
2) Vyhodnocení případných podmínek aktivace u všech logických bodů přerušení zjištěných v části 1).
3) Pokud podmínka aktivace není splněna, pak je tento bod přerušení ignorován. Pokud podmínka splněna je, případně pokud je bod nepodmíněný, pak ho zařaď do finálního seznamu.
4) Jestliže je finální seznam neprázdný, pak bude laděný program zastaven.
5) Jinak bude laděný program pokračovat.

Díky bodům přerušení jsme také schopni provádět vykonávání programu v krocích. Krokování můžeme rozdělit podle způsobu vykonávání funkcí a procedur na dvě varianty: **krokování s vnořením** (*step into*) a **krokování bez vnoření** (*step over*). Krokování s vnořením probíhá včetně krokování v rámci volaných funkcí, kdežto u krokování bez vnoření se tělo funkce provede bez krokování a program je přerušen až po jejím vykonání. Obě funkce navíc mohou být používány jak na úrovni příkazů tak instrukcí.

Funkce krokování s vnořením na úrovni příkazů je ve skutečnosti realizována pomocí ladění na úrovni instrukcí. Je však na ladicím nástroji, aby skutečný stav běhu uživateli abstrahoval do formy zdrojového kódu. To se děje pomocí hledání lokace ve zdrojovém kódu, který bude vykonán jako další. Pokud interní ukazatel není nastaven na aktuální místo zastavení a současně reprezentuje první instrukci nějakého řádku, algoritmus končí a vrátí tento řádek. V opačném případě se ukazatel inkrementuje a pokračuje se další instrukcí (u skokových instrukcí se ukazatel nastaví na adresu cíle). Tento algoritmus se opakuje v cyklu. Na

řádek vrácený tímto algoritmem je poté vložen dočasný bod přerušení a program pokračuje až do jeho vykonání.

Krokování bez zanoření funguje obdobně, pouze s tím rozdílem, že instrukce typu volání (*CALL*) se provedou a přerušení se umístí až za bezprostředně následující instrukci, případně na instrukci ležící na návratové adrese volání. Tedy místo, aby se krokovalo uvnitř volání, se volání vykoná celé a krokování pokračuje až za ním.

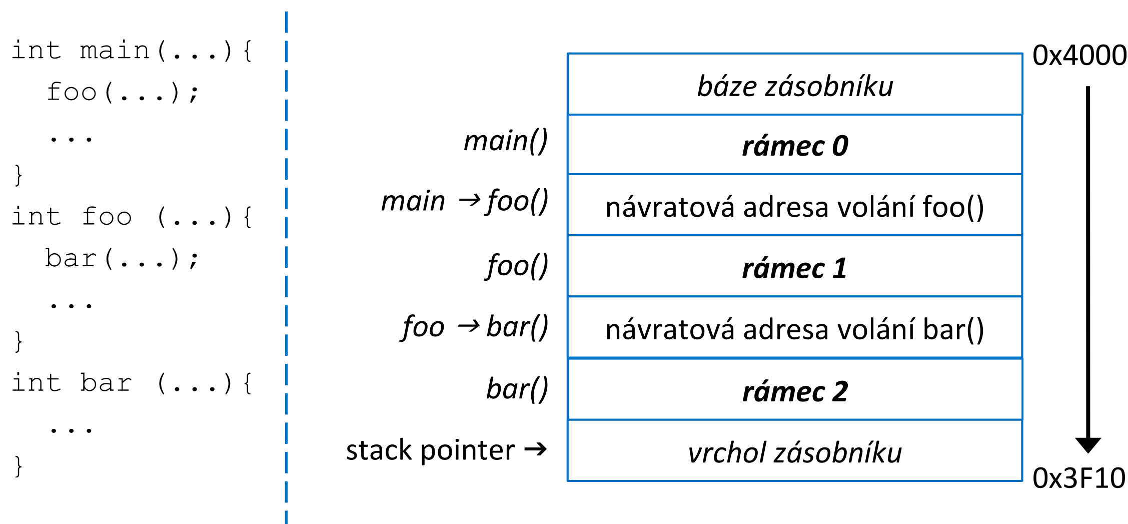
Dalším typem krokování je **běh do konce funkce** (*step out*). Běh pokračuje až na konec právě krokované funkce a zastaví se při návratu této funkce. Pokud je aktuální pozice v hlavní funkci programu, pak program doběhne a ukončí se. Princip vložení přerušení spočívá v načtení návratové adresy ze zásobníku a umístění interního bodu přerušení na tuto adresu.

Krokování po instrukcích je výrazně jednoduší než krokování po příkazech, protože není potřeba znát mapování instrukcí na zdrojový kód. Rovněž vztah mezi fyzickými logickými body přerušení se změní na 1:1 a jejich dělení pak ztrácí smysl. Při vykonávání laděného programu se provede pouze jedna instrukce a okamžitě dojde k předání řízení zpět ladicímu nástroji. Tohoto chování lze docílit buďto opakovaným manuálním vkládáním bodů přerušení na následující instrukci, případně za pomoci hardwarového krokování, kdy tuto činnost vykonává procesor.

4.5 ZJIŠŤOVÁNÍ STAVU ZÁSOBNÍKU

Programový zásobník je struktura podporovaná hardwarem, operačním systémem i překladačem. Procesor obsahuje v registrové sadě ukazatel na zásobník (*stack pointer*) a v instrukční sadě instrukce pro práci se zásobníkem (např. typické instrukce `PUSH` a `POP`). Speciální instrukcí je volání funkce (`CALL`). V tomto případě se na zásobník ukládá návratová adresa a podle volací konvence i funkční argumenty. V rámci funkce se často na zásobníku také ukládají lokální proměnné a kopie hodnot registrů.

V průběhu vykonávání programu dochází k zanořování funkcí a tím pádem k tvorbě **rámců** na zásobníku. Rámec představuje souvislý sled dat na zásobníku, související s voláním konkrétní funkce (tedy návratová adresa, argumenty, lokální proměnné a kopie registrů). Více ilustruje [Obrázek 5].



OBRÁZEK 5 - UKÁZKA TVORBY RÁMCŮ NA ZÁSOBNÍKU (ROSTOUCÍHO SMĚREM DOLŮ)

Jak bylo již uvedeno, při ladění potřebujeme kromě informace o pozici v programu znát i způsob, jakým jsme se do ní dostali, tj. zjištění posloupnosti rámců. K tomu se používá algoritmus **stack unwinding**[4]. Zjednodušená verze algoritmu bez obnovování uložených registrů (předpokládejme, že zásobník roste směrem dolů):

Vstup: Aktuální stav registrů a programového čítače.
Výstup: Seznam rámců na zásobníku.
Postup: Procházení posloupností ukazatelů na rámce na zásobníku.

1) Nechť **Z** a **F** jsou hodnoty ukazatele na zásobník, resp. ukazatele na rámec. Dále nechť **P** je aktuální hodnota programového čítače.
2) Výpis informací o rámci na základě hodnot **Z**, **F** a **P**.
3) Jestliže **F** je větší nebo rovno adrese báze zásobníku, zastav. (*Analyzujeme pouze laděný program, ne podrobnosti o jeho spuštění*).
4) Nechť **S** se rovná hodnotě **F** inkrementované o velikost ukazatele. (*Získání návratové adresy funkce*).
5) Nechť **P** se rovná hodnotě na adrese udané pomocí **S**.
6) Inkrementujme hodnotu **S** o velikost ukazatele. (*Nyní **S** obsahuje adresu následující funkce*).
7) Nahraďme **F** za ukazatel uložený na adrese **F**.
8) Opakování od části 2).

[4] Do češtiny se tento výraz nepřekládá. Přibližný překlad by mohl znít *odvíjení zásobníku*.

4.6 ZOBRAZENÍ HODNOT PROMĚNNÝCH

Pro zobrazení hodnot proměnných laděného programu je nutné znát jejich jména a fyzická umístění. Jména proměnných se zjišťují z tabulky symbolů, což je struktura obsažená v objektovém souboru. Fyzické umístění proměnné může být například v paměti (globální proměnné), na zásobníku (lokální proměnné či argumenty funkcí), případně v registrech (při optimalizaci kódu překladači). Typ fyzického umístění je, podobně jako informace o mapování instrukcí na řádky, umístěn v ladicích informacích.

Při zjišťování jmen proměnných musí ladicí nástroj rozpoznat aktuální kontext programu a jemu se přizpůsobit. Není například možné vypsat hodnoty všech proměnných z tabulky symbolů, protože mnohé z nich jsou lokální a v danou chvíli nemusejí být definované. Stejné tak při rekurzivním volání funkce je nutné zobrazovat více proměnných se stejným jménem odděleně, v závislosti na úrovni zanoření.

Nepřímo se zobrazením hodnot proměnných souvisí i datové body přerušení. Pokud je takový bod nastaven na lokální proměnnou, ladicí nástroj musí zajistit, že program jím nebude přerušen mimo funkci, kde je proměnná definována, například změnou hodnoty jiné proměnné, která je mapována na stejné místo v paměti. Řešením je například deaktivace bodu přerušení při opuštění dané funkce.

5 LADĚNÍ OPTIMALIZOVANÉHO KÓDU

Optimalizace a transformace kódu jsou prováděny překladačem z důvodu snížení času potřebného pro vykonání programu, případně pro snížení jeho velikosti. Mnoho optimalizačních metod je založeno na přeskupení či eliminaci kódu (např. optimalizace „*peephole*“, rozgenerování „*inline*“ funkcí či eliminace mrtvého kódu). Optimalizace není v praxi realizována jako jedna ucelená fáze, ale jedná se o celou sérii transformací, které probíhají ve všech fázích překladu a nezřídka se provádí i v průběhu generování ladicích informací.

Pokud se během optimalizace změní kód a překladač náležitým způsobem nemodifikuje vygenerované ladicí informace, tak dojde ke vzniku nekonzistence s kódem programu. To způsobí při ladění v lepším případě chybu, v horším to povede k chybné interpretaci výsledků ladění. Je tedy nutné, aby překladač ladicí informace opravil, případně alespoň zanechal ladicímu nástroji zprávu, že úsek kódu byl optimalizován.

Může vyvstat otázka, proč se vůbec zabývat laděním optimalizovaného kódu, když tradiční paradigma ladicího procesu nařizuje optimalizace při ladění vypnout. Důvody jsou dva. Prvním je urychlení aplikace i při ladění. Pokud ladíme výpočetně náročný program, je vhodné co nejvíce urychlit části kódu, které nejsou předmětem odlaďování. Výpočetní náročnost je navíc umocněna režií ladicího nástroje. Optimalizace programu je tedy prospěšná pro zkrácení doby vykonávání laděného programu. Druhým důvodem je snaha ověřit funkčnost a výkonnost optimalizovaného programu. Předpokládá se, že finální verze programu bude optimalizována. Uživatel by tedy měl být schopen ladit finální verzi svého programu.

5.1 ZÁKLADNÍ TYPY OPTIMALIZACÍ

V literatuře o překladačích byla oblast optimalizací detailně popsána, proto bude následovat jenom krátký výčet těch metod, které ladění kódu značně komplikují. Informace o dalších metodách je možno najít například v [AlKe01] a [ALSU06].

Zpracování konstant (*Constant folding*). Vyhodnocení výrazu složeného čistě z konstant je provedeno již v době překladu a není tak generován žádný odpovídající kód. Tato optimalizační metoda je prováděna i při nejnižší úrovni optimalizování kódu. Příklad této optimalizace:

Kód před optimalizací

```
1: i = 3 * 4;
2: i = i + 2;
```

Kód po optimalizaci

```
1: i = 14;
```

Uživatel, který by chtěl tento výpočet při ladění krokovat, by byl patrně překvapen tím, že výpočet výrazu se vůbec neprovádí. Podobné chování je možno pozorovat i u metody **propagace proměnných**.

Přesunutí invariantů cyklů (*Loop-invariant Code Motion*). Jde o optimalizaci smyček, která se snaží přesunout mimo smyčku takové výpočty, které produkují stejné výsledky v každém cyklu (tj. jsou vůči cyklu invariantní). Viz následující příklad:

Kód před optimalizací

```
1: i = 0;
2: while (i < 500) {
3:   j = x * y;
4:   a[i] = i + j;
5:   i++;
6: }
```

Kód po optimalizaci

```
1: i = 0;
2: j = x * y;
3: while (i < 500) {
4:   a[i] = i + j;
5:   i++;
6: }
```

Při ladění tohoto kódu může dojít ke zmatení, jelikož hodnota proměnné `j` je přiřazena ještě před vstupem do smyčky. Obdobným způsobem pracuje i metoda **transformace indukčních proměnných**.

Eliminace nepotřebného kódu. Dochází zde k odstranění částí kódu, které jsou sice prováděny, ale nemají žádný význam. Kupříkladu přiřazení proměnné sama sobě:

```
1: x = x;
```

Takovýto kód je překladačem eliminován a není možné ho tak namapovat na žádnou strojovou instrukci, což může být problém při nastavování bodu přerušení.

Registrové proměnné. Jde o další optimalizaci, u které může dojít k problémům při ladění. Uvažujme například následující kód:

```
1: int i;
2: for (i = 0; i < 10000; i++) {
3:   a[i] = i * 2;
4: }
```

Pokud bude kód neoptimalizovaný, tak bude proměnná `i` uložena v paměti (či na zásobníku). Ladicí nástroj tak nebude mít problémy se zobrazováním její hodnoty, protože se bude nalézat stále na stejném místě, a to i v průběhu cyklu. V něm bude hodnota proměnné `i` vždy načtena z paměti, provede se s ní porovnání, bude inkrementována a následně se opět uloží do paměti.

Pokud však překladač provede optimalizace, může po dobu provádění cyklu proměnnou `i` uložit do registru (což bude výrazně rychlejší) a hodnota v paměti nemusí být po celou dobu provádění cyklu aktualizována. Jestliže informace o dynamické změně umístění proměnné není obsažena v ladicích informacích, pak ladicí nástroj bude ukazovat chybné hodnoty.

5.2 ARCHITEKTURY ZAMĚŘENÉ NA VYSOKOU MÍRU PARALELISMU

Paralelní vykonávání kódu mnoha nezávislými výpočetními jednotkami urychluje běh programu. Architektury založené na paralelismu mohou mít různé podoby v závislosti na vlastnostech výpočetních jednotek a jejich propojení. Můžeme se například setkat se superskalárními procesory, více-jádrovými procesory či multiprocesorovými systémy. Z pohledu ladicího nástroje se vždy jedná o nárůst množství informací, které musí zjišťovat a spravovat.

Do jisté míry speciálním typem architektury je architektury VLIW (*Very Large Instruction Word*). Ta je oproti ostatním zaměřena na dosažení vysoké míry paralelismu na úrovni instrukcí. Architektura se skládá z mnoha nezávislých funkčních jednotek, které jsou řízeny paralelně. Procesor v každém taktu vydává dlouhé instrukce, které jsou složeny z mnoha (typicky 4-16) operací pro funkční jednotky. Každá operace má podobu běžné instrukce pro architekturu RISC.

Míra paralelismus vykonávání kódu na architektuře VLIW je omezena pouze tím, jak dobře dokáže překladač naplánovat instrukce. Pro efektivní plánování se do jedné dlouhé instrukce vkládají operace realizující různé příkazy zdrojového kódu současně. Dochází tak k rozbití pořadí provádění příkazů. Ladění neoptimalizovaného kódu pro VLIW architektury je tak v mnohém horší než ladění optimalizovaného kódu jiných architektur. Vzhledem k jednoduchosti hardwaru VLIW architektur je jejich efektivita závislá na kvalitě překladače [FFY05]. Z toho důvodu byla vytvořena celá řada optimalizací speciálně pro ni (např. *software pipelining*, *modulo scheduling*, *trace scheduling* atd.). Při použití optimalizací se problém ladění kódu dále stupňuje.

Metoda mapování řádků zdrojového kódu na instrukce se tak stává nedostatečnou, jelikož není schopna zachytit současné rozpracování několika příkazů současně. Tento problém je v literatuře zmiňován pouze minimálně. Jedním z řešení, podle publikace [Coo92], by mohla být transformace tabulky s informacemi o řádcích do podoby tabulky s informacemi o instrukcích. Nebyly by v ní tedy uloženy informace o mapování příkazů na jednotlivé instrukce, ale naopak by obsahovala údaje o tom, které příkazy jsou prováděny danou instrukcí. To by vyžadovalo podporu na straně překladače, který by krom jiného mu-

sel generovat i informace o sloupcích, aby bylo možno uživateli vyobrazit, která část příkazu se aktuálně provádí. Následuje ukázka tohoto konceptu (převzato z [Coo92]):

Kód v jazyce C:

```
// 12345678901234567890 – čísla sloupců
1: int i1, i2, i3;
2: float f1, f2, f3;
3:
4: i1 = i2 + i3;
5: f1 = (f2 / f3) / i1;
6: return;
```

Odpovídající kód v jazyce symbolických instrukcí pro VLIW architekturu se 4 sloty:

```
0: load r1,i2    load r2,i3     load fr1,f2    load fr2,f3
1: add r1,r2     nop            fdiv fr1,fr2   nop
2: store r1,i1   nop            nop            nop
3: nop           nop            load fr2,i1    nop
4: nop           nop            fdiv fr1,fr2   nop
5: nop           nop            store fr1,f1   nop
6: ret           nop            nop            nop
```

Zjednodušená podoba tabulky s informacemi o mapování řádků a instrukcí:

Adresa instrukce	**Pozice ve zdrojovém kódu (řádek:sloupec)**	**Komentář**
0x0	4:6; 4:11; 5:7; 5:12	načtení i2, i3, f2 a f3
0x1	4:9; 5:10	i2 + i3, f2 / f3
0x2	4:4	přiřazení výsledku
0x3	5:18	načtení i1
0x4	5:16	/ i1
0x5	5:4	přiřazení výsledku
0x6	6:1	return

6 ZÁVĚR

Proces ladění kódu je v současnosti již nezbytný pro vytvoření kvalitního softwaru. V této práci jsme se zaměřili na jednu z metod ladění – ladění kódu na úrovni zdrojového kódu. Tento přístup je pro uživatele intuitivní, jelikož svůj program odlaďují na stejné úrovni, na které ho vytvářeli. Jsou tak zbaveni nutnosti znát architekturu, na které ladění probíhá, jako je tomu při ladění na úrovni strojového kódu. Na druhou stranu je proces návrhu a implementace takového nástroje obtížnější.

V prvé řadě je nutné získat podporu v překladači při generování ladicích informací, které jsou pro ladicí nástroj kritické. Jejich formát není univerzální a často se liší překladač od překladače. Ladicí nástroj se pak musí specializovat na práci s některým ze standardů, jakými jsou v současnosti formáty DWARF či Microsoft PDB.

Ladicí informace jsou pak využity k mnoha účelům. Jedním z nich je zjišťování fyzického umístění proměnných, které se použije při zobrazování jejich hodnot za běhu programu. Druhým možným využitím je mapování čísel řádek ve zdrojovém kódu (tj. úroveň příkazů programovacího jazyka) na strojové instrukce. Díky tomuto jsme schopni uživateli navodit iluzi, že program je prováděn po příkazech, i když je ve skutečnosti prováděn po instrukcích. Program pak můžeme po příkazech i krokovat, což je činnost ovlivňující jeho rychlost i způsob vykonávání.

(Nejenom) pro krokování se využívají body přerušení. Způsobů uplatnění je ale celá řada, což jenom podtrhuje jejich nezbytnost v procesu ladění. Tento princip musí být podpořen na hardwarové úrovni, což se děje na různých architekturách různým způsobem. Ladicí nástroj je pak buďto platformě závislý nebo musí počítat s odlišnými podmínkami na různých platformách.

Samotnou kapitolou je pak ladění optimalizovaného kódu a kódu pro architektury zaměřené na paralelismus. Tato témata jsou v současnosti pro výrobce ladicích nástrojů poměrně palčivá, proto by se měl další výzkum ubírat především touto cestou.

Práce měla spíše informativní charakter, kdy nebylo cílem popsat všechny metody a techniky používané při ladění na úrovni zdrojového kódu, ale především představit ty nejdůležitější.

7 LITERATURA

[AlKe01] Allen, R., Kennedy, R.: *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*, Morgan Kaufmann, 2001.

[ALSU06] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Addison Wesley, Aug. 2006

[Coo92] Cool, L. E.: *Debugging VLIW Code After Instruction Scheduling*, Master's Thesis, Oregon Graduate Institute of Science & Technology, 1992.

[Dar05] Darling, D.: *The Impact of DWARF on TI Object Files*, Texas Instruments, 2005.

[Eag07] Eager, M. J.: *Introduction to the DWARF Debugging Format*, 2007.

[FFY05] Fisher, J. A., Faraboschi, P., Young, C.: *Embedded Computing – A VLIW Approach to Architecture, Compilers, and Tools*. Morgan-Kaufmann Elsevier Publishers, 2005.

[FSG06] Free Standards Group: *DWARF Version 3 Standard Released*, 2006.

[Gra83] Gramlich, W. C.: *Debugging Methodology*, Session Summary for ACM Workshop on Debugging, 1983.

[Hay93] Hayes, B.: *The Information Age: The Discovery of Debugging*, The Sciences, Vol. 33, No. 4, pp. 10-13, 1993.

[Krou07] Křoustek, J.: *Analýza a transformace kódů*, bakalářská práce, Brno, FIT VUT v Brně, 2007.

[LLVM10] Lattner, C.: *Source Level Debugging with LLVM*, LLVM Documentation, www: <http://llvm.org/docs/SourceLevelDebugging.html>, 2010.

[Ros96] Rosenberg, B. J.: *How Debuggers Work – Algorithms, Data Structures, and Architecture*, Wiley Computer Publishing, 1996.

[Wiki01] *Computer Bug*, www: <http://en.wikipedia.org/wiki/Computer_bug>, 2010.

[Wil10] Wilczák, M.: *Ladicí nástroj generických simulátorů procesorů*, diplomová práce, Brno, FIT VUT v Brně, 2010