Optimalizace kódu produkčních překladačů

2. ledna 2013

Obsah

1 Úvod

Nebude to o celých překladačích, moc rozsáhle a provázané. Dělí se na frontend a backend. Frontend je v podstatě parser jazyka. Hlavní problém je aby šly rychle a házely dobré chybové hlášky.

Z toho leze AST (abstraktní syntaktický strom). Poté se z toho udělá reprezentace v mezijazyce (průchodem do hloubky). To je seznam funkcí, v tom seznam basic-blocků, každý má nějaký seznam příkazů (většinou trojice – typ operace, cíl, operandy).

Potom to žere backend. Ten optimalizuje a potom generuje kód, alokace registrů a nakonec z toho leze assembler.

To mezitím se nazývá middle-end. Frontendy a backendy se mění (podle jazyka, podle platformy). Na nový kus je potřeba celou tu danou část v zásadě přepsat. Middle end je to, co „přežívá“ – je dostatečně nezávislé jak na jazyce, tak na architektuře (něco o nich ví, ale není to moc do hloubky).

Tato přednáška bude o middle-endech.

1.1 Existující projekty

1.1.1 GCC

Klíčový z projektů GNU, nějaká 80 léta, už v roce 1987 podporoval skoro celé ansi-C. V té době uměl alfy.

Kolem verze 2.0 (92) převzal Richard Kenner, důležitý vývojář do dnes.

V roce 1995 se rozhodl (2.7.0), že vydá, až bude hotové C++, vzniklo mnoho forků. Odrazovalo to nové vývojáře (byrokracie), vývoj ve firmě cygnus byl taky fork, zaměstnávali mnoho vývojářů.

1997 vydal cygnus egcs, fork, mnoho problémů, nakonec se stalo hlavním GCC – 2.95, vzniklo steering comitee, nové otevření.

2001 verze 3.0, podpora C++, poté se šlo do architektury, 2005 verze 4.0, má middle end.

Asi nejvíce podporovaných platforem, mezijazyk RTL je jednoduchý na připisování nových backendů.

Mnoho systémů ho používá.

1.1.2 LLVM

Původně dizertačka Chris Lattnera. Původně založen na frontendech GCC, je to Low-Level Virtual Machine, jednoduchý mezijazyk. Snaží se ho dotáhnout

Apple, zatím se to úplně nedaří (nedodělané).

Má podporu x86-64, mips, ppc, nvidia GPU.

Hodně modulární (phase manager, který řeší, kdy co pouštět), zjednodušuje vývoj.

Výsledný kód není úplně nejlepší.

1.1.3 Open64

Původně se jmenoval MipsPro, má dlouhou historii, generoval rychlý kód pro tehdejší architekturu. Poté to mířilo na itánium, což SGI zkrachovalo, těsně předtím překladač uvolněn (2000), ne kompletní (něco patřilo někomu jinému), vzniklo nahrazením C++ frontendu od GCC, zahodili backend pro itanium.

Vývojáři nesměli pracovat na stejném projektu (měli to ve smlouvě), problémy (nikdo tomu nerozuměl, nespravoval).

Mezijazyk WHIRL (Wery High Level Intermediate Language). Má několik mezistavů, optimalizace na správné úrovni. Znalo ho několik univerzit v té době.

Vzniklo několik klonů:

- ORC – Open Research Compiler, měl to být kvalitní překladač pro itanium, zaniklo 2003.
- PathScale – x86-64. Špatný bussiness model.

Umí generovat docela rychlý kód.

2 Mezijazyky

Překladače mají různé úrovně optimalizací (high-level – např. inlinování funkcí, middle-level, low-level – např. alokace registrů).

Na každou úroveň se obvykle používají různé mezijazyky. Např. gcc má:

- Generic
- Gimple
- RTL

Teprve nedávno to začalo používat ty „vyšší“.

Při návrhu je několik problémů:

2.1 Reprezentace typů

Také, od nejvyšších, po nižší – nějaké jak je to naparsované, přes to, co umí daná architektura (integery, floaty, skládání do struktur/polí), až po ty nejnižší, co umí jen procesor.

Gimple gcc dělí na druhy:

- Integer
- Real
- Pointer
- Complex
- Vector
- Bool
- Enum
- Array
- Record
- Union
- Kvalifikovaný union (to je z ADY)
- Funkce
- Metoda

Každé to nese mnoho dalších informací (rozsahy, jak přetýká, etc).

V RTL už je těch informací mnohem méně, jen 16-bitová informace, co je to zač (liší se v zásadě jen v délce a jestli je to int, float a nebo flagy procesoru).

Potom tam bude spousta přetypování mezi velmi podobnými typy. Např. v C++ je 60% instrukcí přetypování, nepomáhá to optimalizacím (i když to negeneruje kód, překáží to optimalizerům, když se to snaží umísťovat tak, aby minimalizoval operace). Na druhou stranu lze nad tím něco studovat (přeuspořádávat položky struktury).

Potom se řeší nějaké deklarace – kus paměti, který má nějaký typ a tak. Ty nižší mají spíše registry, než deklarace.

Dále je problém, jak popsat control-flow. Ta vysoká úroveň má nějaké podmínky, cykly, etc, na střední nějaký control flow graph, na konci už jsou jen podmíněné skoky.

Potom je potřeba reprezentovat vlastní instrukce. Jedno je abstraktní syntaktický strom (vypadne nějaká deklarace, přiřadítko, ...). Druhý extrém je nějaké instrukce. V tom prostředku se to snaží být ten syntaktický strom nějak zploštělý do nějakých n-tic, co se má s čím dít.

2.1.1 RTL

To se snaží popisovat assemblerové instrukce cílového procesoru. Jsou zase nějaké n-tice, které říkají, co se má provádět, s čím, a tak. Jsou v tom nějaké basic-bloky, kam to patří, odkaz na následující, na předchozí, debug info.

2.1.2 Přístupy do paměti

Jeden extrém je reprezentovat to podle toho, jak je to napsané (popsané idnexování polí, lezení do struktur), až k tomu, kdy se to zapisuje jako matematický výraz s ukazateli. Občas je potřeba řešit překlady do dědičnosti a podobně.

Je potřeba řešit aliasování (že něco může ukazovat na stejné/různé místo). Je vhodné tohle vyndat z typů a uložit do těch přiřazení.

2.2 Debugovací informace

Je vhodné, aby šlo debugovat i to, co je zoptimalizované. Ukládá se to do formátu dwarf (poměrně rozsáhlý standard). Je to programovací jazyk – může to i počítat spoustu věcí, kde co je a tak (například kvůli stacku) – umí např. revertovat převod for cyklu do pointerové aritmetiky.

V překladači je potřeba to táhnout „s sebou“. Některé věci (interní) nemají debug info. Každý kus si nese něco zvané „locus“ – info, kde se to stalo (řádek, zdroják).

Většina věcí si pamatuje, odkud vzniklo (registry, proměnné...). Takže je možné mít víc instancí (v kódu) proměnných.

Jsou tam nějaké instrukce typu „tady by se nastavilo tohle, kdyby se to nevyoptimalizovalo“.

2.3 Exception handling

Jsou dvě strategie – setjump a longjump, druhé jsou unwind zásobníku.

To první funguje, jenže je to pomalé (setjumpy jsou pomalé).

Navíc musí být kompatibilní mezi jazyky, mezi verzemi, překladači a podobně. Používá se norma pro C++ a itanium (nezávislá na platformě a

jazyce).

Unwindování stacku se dělá pomocí runtimu, který používá dwarf popisy, opravuje to zásobník, volá handlery. Jednak tam jsou tedy ty dwarfové popisy zásobníků, jednak action chainy (každá instrukce dělá jinou výjimku, vyhledává se, co se s ní dělá – každá může být jiná).

Ve vysokých jazycích jsou různé regiony:

- Cleanup (na konci bloku, na uvolňování/volání destruktorů).
- Try (tam kde napíše uživatel).
- Must not throw (když to člověk oflaguje, že nehlásí, když vyhodí, tak to skončí). Nafukuje program. Taky kolem cleanupu.
- Allowed exceptions (něco jako must not throw, ale s nějakými povolenými).

Jsou gcc instrukce eh_dispatch, resX (chytání, pokračování unwindování).

2.4 Control flow graph

Orientovaný graf, každý vrchol je basic block (kus kódu, kde se neskáče). Hrany jsou možné přechody. Vnitřek basic blocku bývá spoják. Většina překladačů má explicitně reprezentované hrany, obvykle přímo za basic blockem.

Z basic blocku obvykle vede max. 2 hrany.

Jsou různé druhy hran:

- Skoky.
- Exception handling (problémové).
- Abnormální hrany (longjump $\rightarrow$ setjump (do všech, neví se který)). Navíc tyhle nejdou přesměrovávat.

Je to na každou funkci zvlášť, kvůli tomu, že se to na „celek“ nehodí.

Pak je ještě callgraph, je to multigraf, mezi funkcema.

Je více druhů callgraphů (některým optimalizacím vadí, když by se uprostřed basic blocku volalo exit nebo tak).

3 Propagace konstant

Základní implementace je lokální – v rámci jednoho basic-blocku a dosazuje, pokud je to konstanta, zkusí vypočítá dál, propagovat. Lze to zlepšit, pokud jsou nějaké basic blocky za sebou a hrany jen odchází, pak to jde propagovat i skrz. Lze vykoukat i nějaké konstanty z podmínek (např. když je podmínka, že něco je rovno konstantě).

Extended basic block je strom, který se jen dělí (neslučuje). V rámci toho lze propagovat také.

Můžeme také přeskakovat smyčky (pokud se v ní nic nemění, pak to může jít skrz).

Hyperblock je acyklická část control flow grafu. Například toto obsahuje i if-then-else. Lze si pamatovat hodnoty „s podmínkou“. Lze toho využívat. Na to se používá if-konverze, přidá to ty hyperbloky, pak je zase rozebírá.

4 Data-flow

Máme svaz hodnot, control flow graph, přechodové funkce a meet operator.

To nesmí užírat konstantní hodnoty (protože by to zhoršovalo) a ten svaz musí mít konečnou hloubku.

Zabírá to hodně paměti, počet iterací je omezen velkým číslem, může trvat dlouho.

Umí se to dobře řešit při acyklických, ty se smáčknou, smyčky se řeší zvlášť.

Jsou dopředný a dozadný.

Dělá se taky něco jako sparse reprezentace – u každého výskytu se odkazuje na místa, kde to mohlo vzniknout.

4.1 SSA forma

Single-Static Assignment. Říká, že každá proměnná se definuje jednou. Bohužel tím nejde vytvořit for-cyklus. Přidává se φ operátor, na začátku basic-blocku, jeho parametry odpovídají vstupním hranám, je tam několik výrazů, on vybere ten správný výraz.

A ***dominuje*** B právě když na každé cestě do B je A.

Jednak, B dominuje samo sebe. Dále je to tranzitivní. Pokud dva dominují třetího, tak se taky dominují navzájem (právě jedním směrem).

Tohle definuje strom. Dá se spočítat v $O(n \cdot \alpha(n))$. Existuje mnohem jednodušší algoritmus $O(n \cdot \log n)$.

5 Alias analýza

Řeší, jestli náhodou něco nemůže ukazovat na stejné věci a dělat tedy problémy. Vrací to, jestli ano, ne, nebo možná. To samozřejmě nejde na jistotu rozhodnout (obecně). Navíc chceme, aby se nikdy nespletla.

Lze vykoukat, že pokud máme statickou proměnnou, kde když se nevezme adresa, tak nikdy nemůže aliasovat. Dále je (obvykle) zakázané, aby se aliasovaly různé typy.

Dá se to také stopovat. Je problém, jak to reprezentovat. Můžeme zaznamenat všechna „pravdivá tvrzení“, ale má to docela velikou velikost.

Pozor, tohle nemusí být ekvivalence. Takže reprezentovat třídami ekvivalencemi moc nefunguje.

Také je možné ukládat množinky a hrany mezi nimi, co může ukazovat na co.

Je výhodné občas rozlišit, co daný jeden ukazatel znamená na různých místech. Na to se hodí například SSA forma.

Lze to kromě lokální analýzy dělat i globálně, brát v úvahu volání funkcí externích knihoven a podobně.

Dá se pomoct i v programu, nějakým constem (u funkce), nebo restrictem.

6 Meziprocedurální optimalizace

Basic blocky mohou brát v úvahu nějaké počty, pravděpodobnosti výstupů a podobně (třeba z profilování). Pro to je potřeba obsadit hrany, dává se jen na některé (ne na kostru, ta jde dopočítat). Některé překladače dělají jen počítání basic-blocků, GCC i hrany (je to dražší). Daly by se generovat i různé cesty, ne jen hrany.

Toto je problém při vícevláknech.

Když toto není, tak se to snaží hádat. Například se generuje statický profil, heuristika na to, že funkce „error“ se spíš nezavolá a podobně.

Smyčky se hledají tarjanovým algoritmem na 2-souvislost. Přirozené smyčky se hledají průchodem do hloubky, najde to strom vnořených smyček.

GCC má dvě sady heuristiky, používá first-match (vezme se první, co řekne, že ví).

Jsou loop exit (končící smyčku spíš ne), test dvou proměnných na stejnost asi nebude, odhady počtu iterací, většina pointerů nebývá null, goto se asi neprovede. Kopie první podmínky for cyklu asi bude pravdivá, etc.

Pokud tohle nevyjde, tak se dělají nějaké další odhadovací počítání z tvaru

toho grafu a podobně. Na to se počítá hitrate a coverage (měření, jak často a jak dobře).

Dříve byly že vždy jedna funkce, teď obvykle unit at time (otázka, co je unit). U některých to umí spojovat dohromady. Také se dají dělat linkové optimalizace (až nad assemblerem). Možné je i na celý program, lze předpokládat, že už se nebude nic přidávat.

Potom to má k dispozici callgraph, viditelnost, dostupnost.

Nejdůležitější je asi inlining, constant propagace, alias analýza, mazání mrtvého kódu, změna datových struktur, devirtualizace, specializace, parametry funkcí.

6.1 Inlining

Inlinování má tu výhodu, že se okolo dají prohazovat příkazy, optimalizovat „skrz“, ale zase to zvětšuje program (disk, cache), zvětšuje nároky na kompilaci (ne všechno je lineární s velikostí kusu kódu, ukládání struktur).

Je potřeba porozhodovat, co inlinovat. Jednak do toho může mluvit programátor, jednak podle toho, kolikrát je funkce volaná, podle její váhy, je potřeba váhy při inlinu něčeho přepočítávat. Chceme spíš inlinovat něco, co je volané často, ale zase ne něco, co je velké. Hodí se vyhodit všechna volání, když už se inlinuje (aby se jí dalo zbavit).

Hodí se, aby už byly předoptimalizované, aby se lépe počítaly odhady. Na druhou stranu, inlining je dobrý k mnoha dalším věcem.

Občas se dělá tzv. early inlining – jen lokální měření, jak je velká a podobně. Inlinuje se, když je menší nebo srovnatelná s vlastním voláním.

Potom se občas dělá částečné inlinování – např. když je tam nějaká podmínka na začátku.

6.2 Informace o funkcích

Často se hodí o funkcích vědět nějaké dodatečné informace, jako například že nic nikde nemění, že nemá side-effecty, že závisí pouze na parametrech a podobně. To jednak umožní optimalizace okolí volání, jednak i třeba volání občas vyhodit nebo zkonstant-propagovat.

Je třeba taky propagovat konstanty skrz funkce. Občas se dělá také forward substituce a partial inlining (inlinovat jen kus funkce, vložit počítání parametru až někam dovnitř).

Také se dá dělat třeba reorganizace datových struktur.

Type escape analýza – kontrola, že nikam neutíká datový typ.

U inlinování se používá buď top-down nebo bottom-up (jestli se ve stromu leze z vrchu nebo zespodu).

6.3 Přerovnávání funkcí

Je lepší, když fce, co se volají, jsou pohromadě a skoky pokud možno dopředu.

7 Globální optimalizace

To optimalizuje vždy celou funkci.

7.1 PRE

To je Partial Redundancy Elimination. Například pokud se jeden výraz počítá vícekrát, tak se ukládá. Jde na hodně rozšířených basic-blocích, ale ne úplně na všem.

Globální verze funguje tak, že hledá, pokud to může být stejné ve více věcích, tak to propaguje skrz to a hledá, kde je to stejné. Napřed se pomocí data-flow zjišťuje, jestli ho něco zabíjí a kde všude je to vidět.

Toto je potřeba iterovat, protože po zpropagování mohou vzniknout nové duplicity.

Tohle může být problematické, pokud to nedojde do konce – je hromada proměnných, překážejí register-alokátoru, který se snaží znovu z toho občas dělat výrazy.

Občas je potřeba nějakou větev „doplnit“ o výpočet (jiné větve počítají, některé ne), umístit na hranu.

Jde to i opačně, mít šipky „nahoru“ – a říkat, že čekám tenhle výraz. Občas je možno/nutno něco nepočítat.

Taky se chce, aby těch výpočtů bylo na konci méně, než na začátku. Strká se to nejvíc nahoru, jak jen to jde.

Je potřeba zařídit, že se nesmí počítat navíc nic, co by ten program shodilo. Také nesmím ignorovat žádné zabití.

Dá se dokázat, že přes to strkání na hranici je to nejlepší.

Tohle má ale tendenci stěhovat všechno co nejvíc nahoru, tím ucpává registry. Také to škodí ve smyčkách (předpočítá invarianty, ale když jich je hodně...).

7.2 GVN-PRE

GVN – třídy výrazů, které jsou stejné, se vyhodnotí stejně.

Postaví se SSA forma nad proměnnýma i nad výrazama.

Má dvě věci – instrukce a výrazy. Máme množinu všech takových dostupných na začátku basic blocku. Umíme vypočítat, co bude k dispozici na konci.

Pak se to kanonizuje (vyháže, co je tam vícekrát).

8 Loop unrolling

Chceme detekovat smyčky a indukční proměnné. Ta dělá v SSA grafu silně souvislou komponentu. Podle indukční proměnné umím určit počet iterací. Z toho se dělá i loop unrolling – něco se nakopíruje vícekrát, snažíme se zbavit výstupní podmínku. Lze buď vědět počet, nebo dodat kód, který to spočítá na začátku a potom to „dožene“, aby zbyl násobek správného čísla. Ale nevyplatí se rozbalit cokoliv (mnoho jich proběhne málokrát). Může se taky hodit vykopírovat pár kopií dopředu, pro případ, že je hodně krátkých cyklů.

Indukční proměnné se často přepíšou do adresního módu.

Stride redukce – když je více indexačních/iteračních proměnných, možná se dají odvodit z jedné. Stejně se většina dá spočítat během adresace.

Občas se vyplatí otáčet smyčky (třeba díky instrukci loop, ta umí do nuly) – ale jen někdy a ne vždy se vyplatí (třeba prefetching). Je potřeba dependency testing. Na to se používá iterační vektor (vektor, kolikátá iterace je to uvnitř které vnořené smyčky).

GCD test – zjišťuje závislosti velikosti smyček.

Omega test (má to knihovnu). Dělá normalizaci (když vše je dělitelné něčím, tak to tím vydělí), eliminuje rovnosti, řeší nerovnosti.

Dělá se otáčení, prohazování smyček. Je potřeba něco, čemu se říká „perfect nest“ (příkazy jen uvnitř té vnitřní).

Polyhedrální model – rozebere se, nad tím jsou operace a potom se to zase složí. Má kusy, které mají jen jeden vstup a výstup. Ten se dá prohlásit jako jedna instrukce. V prostoru jsou mohoúhelníky, ve kterých jsou instrukce, potom nějak udělá pořadí, jak se jeho celočíselné body projdou. Ne vše jde převést do tohodle. Proto se dějí hacky, jako přidávají podmínky, přidávají jiné kusy na jednu hranu, etc.

Také se optimalizuje na cache procesoru – prohazování smyček, scewing (naklonění polyhedru), slučování smyček, které jsou za sebou, či občas naopak (kvůli přístupům ke cache, či kvůli chybějícím registrům), distribuce, pre-

fetching (přednačítání) – pozor, pere se s HW prefetchingem. Jsou občas instrukce, které nepoužívají cache – reuse analýza (důležité je spíše neprefetchovat, než opačně, už kvůli paralelizmu, ucpávání, ...).

8.1 Vektorizace

Je to jen pattern-matching, dependency testing (nesmí ty smyčky na sebe záviset). Problém je alignování, pakování – spíše speciální případy, než teorie.

8.2 Software pipelining

Když má něco latenci, tak se vyplatí promíchat iterace za sebou, aby se tím nacpaly všechny jednotky (na začátku hora čtení, potom hora dalšího...). Obvykle se udělá unroll, vytáhne prostředek, z toho prostředka (kde je to promíchané) udělá zase smyčku. Může používat i třeba registrová okna.

8.3 Predictive commoning

Občas programátor to minulé už napíše sám.