Kompilatorutförda optimeringsstrategier GCC och LLVM

Lars Sundman

6 april 2016

Sammanfattning

I avhandlingen diskuterar jag modern modulär kompilatordesign och optimeringsstrategier med utgångspunkt i de två stora kompilatorprojekten GCC och LLVM. Både GCC och LLVM är modulära kompilatorramverk i den bemärkelsen att kompilatorernas framsidor och baksidor är löst kopplade. Genom att kompilatorerna använder sig av en intern representation som är passligt abstrakt har man i båda projekten lyckats skapa en struktur där framsidan och baksidan kan bytas ut enligt vilken typ av källkod som behandlas och för vilken typ av dator som programmeraren vill kompilera programmet.

GCC och LLVM åstadkommer ändå detta på olika sätt. Vilket leder till olika resultat och i viss mån till olika användningsområden för den maskinkod som kompilatorerna producerar. En viktig skillnad mellan de två är hur de valt att göra optimeringen av programmet. LLVM har gått in för en så kallad ”lifelong compilation model” där programmet inte bara optimeras under själva kompileringssteget, medan GCC använder sig av en mera traditionell modell [1, 2, 3, 4]. Detta betyder ändå inte att LLVM nödvändigtvis ur alla synvinklar skulle vara mera effektiv eller producera snabbare maskinkod än GCC utan detta är något som varierat från version till version. I den här avhandlingen är mitt mål att undersöka hur de olika tillvägagångssätten för att sköta optimering av program skiljer sig från varandra samt att i viss mån diskutera hur detta påverkar de praktiska användningsområdena för de två kompilatorerna.

Innehåll

1 Inledning

En kompilator har, på ett mycket allmänt plan, i uppgift att översätta källkod till maskinkod. Kompilatorer har länge setts som en nödvändig del av ett komplett operativsystem. Redan i slutet på 1950-talet hade man börjat uttrycka så gott som alla komplicerade koncept i något programmeringsspråk på en högre abstraktionsnivå än processorspecifik maskinkod [5, 6].

I grund och botten exekveras förstås all programkod i någon form på processorn, oberoende av abstraktionsnivå eller språktyp, men det visade sig snabbt att de konstruktioner man med hjälp högnivåspråk kunde utnyttja helt enkelt i de flesta fall var så arbetsdryga att implementera att det inte skulle ha varit lönsamt att göra det för hand i maskinkod, fastän en handgjord version av programmet kanske skulle ha varit effektivare. Dessutom visade det sig nu att högnivåspråken, mycket på grund av deras mer naturliga syntax och struktur, inte bara gjorde programmerarna mer produktiva, men också sporrade dem till att uppfinna sådana program och användningsområden för datorer som man helt enkelt inte hade kommit att tänka på tidigare. På det här sättet har framsteg inom den tillämpade datavetenskapen i många fall gått hand i hand med framsteg inom kompilatordesign och kompilatorkonstruktion [5, 6].

Den här trenden är något som fortsatt ännu i modern tid. Ett exempel på den här effekten är den populära webbsidan GitHub:s Electron ramverk. Electron är ett ramverk som låter en utvecklare skriva program för flera operativsystem och processorer på samma gång, genom att det låter en utvecklare skriva skrivbordsprogram, alltså program som körs på en persondator och inte en server, i Javascript och HTML. Tidigare har det varit otänkbart att skriva hela skrivbordsprogram i Javascript, då tolkarna för språket helt enkelt inte kunde exekvera programmen tillräckligt effektivt.

Javascript är alltså ursprungligen ett tolkat språk som kräver att en tolk körs i bakgrunden och översätter Javascript-koden till maskinkod under exekveringen. Electron är i grunden baserat på Node.js som är ett ramverk för webbapplikationer vars Javascript-funktionalitet baserar sig på Googles V8 Javascriptmotor, som i sin tur klarar av att kompilera Javascript-kod till maskinkod, alltså på förhand före exekveringen, och på så sätt låta processorn exekvera programmet mycket snabbare eftersom Javascript-tolken inte måste fungera som mellanhand. Hur kompilatorn fungerar inverkar alltså inte bara på hur effektiva de kompilerade programmen blir, men också på vilka program man

överhuvudtaget kan bygga.

GNU Compiler Collection[1] (GCC) och LLVM[2] är båda stora kompilatorprojekt vars programvara är utgiven under licenser som gör källkoden öppen och fri för alla att förändra och bidra till. Båda projekten utvecklas över Internet av stora gemenskaper utvecklare.

GCC ges ut under licensen GNU General Public License version 3 och är alltså öppen källkod. GCC har sin början i slutet på 1980-talet som ett resultat av Richard Stallmans arbete med att skapa en kompilator för operativsystemprojektet GNU. Kompilatorn blev snabbt populär, mycket på grund av att den var ett gratis alternativ till kommersiella C-kompilatorer, som inte heller i många fall fungerade bättre än GCC. GCC är idag känd för att vara kompilatorn som Linux och många andra projekt som består av öppen källkod kompileras med.

LLVM ges ut under licensen University of Illinois Open Source License och är alltså också öppen källkod, men på grund av skillnader i licenserna kan man till exempel utveckla system där LLVM ingår utan att hela systemet behöver använda sig av samma licens som LLVM, medan det motsatta gäller för den mer restriktiva licensen som GCC licenseras under. LLVM-projektets skapare och huvudsakliga ledare är Chris Lattner som presenterade ramverket i hans magistersavhanding år 2002 [4]. Nuförtiden används LLVM på många håll, bland annat som en integrerad del i många olika projekt, till exempel GPU-programmeringsspråket OpenCL [7].

Både GCC och LLVM är stora och komplicerade projekt. För att förstå hur de fungerar måste man först förstå hur kompilatorer i allmänhet fungerar, här följer en kort redogörelse.

2 Kompilatordesign

På grund av kompilatorns uppgift att översätta källkod till maskinkod ställs också vissa krav på den. Den måste känna igen vilka program som är korrekta, alltså att deras logiska konstruktioner är vettiga och kan utföras på processorn, och vilka som inte är det. Den måste generera kod som är meningsfull och

[1] Operativsystemprojektet GNUs kompilatorsamling.

[2] Tidigare en förkortning för Low-Level Virtual Machine, en virtuell maskin på låg nivå, men numera är det egentliga namnet LLVM.

motsvarar det som programmeraren vill göra och den måste känna till den omgivning den fungerar i för att se till att maskinkoden den producerar är korrekt och kan ges över till operativsystemet att köra [8, 9].

Utöver detta är kompilatorn också ett verktyg vars uppgift är att förbättra det program den får in [9]. Detta kan helt enkelt ses som att den genererade maskinkoden är en förbättring i sig, eftersom programmet nu går att ge åt en dator att utföra, men i många fall har man också byggt in ett steg i kompilatorn som förbättrar koden genom att till exempel förenkla den så att det resulterande programmet tar mindre minnesutrymme än om programmet skulle ha implementerats som sådant.

Inom mjukvaruutveckling brukar man ofta dela upp större system enligt deras ansvarsområden [9]. Till exempel i kärnan på de flesta operativsystem har man olika moduler som ansvarar för olika sorters hårdvara, så kallade drivrutiner. Inom kompilatordesign tänker man ofta på kompilatorn som bestående av flera delar. Delarna varierar lite enligt upphovsmakarens tycke, men den vanligaste modellen är att beskriva kompilatorn som indelad i en framsida och en baksida som sinsemellan är löst kopplade. De talar alltså ett gemensamt språk men de känner inte till varandras interna funktioner [9, 8].

2.1 Kompilatorns framsida

Framsidan är den del av kompilatorn som programmeraren arbetar med. Den ansvarar för att ta emot programkoden av programmeraren och rapporterar eventuella fel i syntaxen och semantiken. Utöver jämförelsen med de tillåtna tecknen och orden så granskar den alltså också att den logiska helheten är kongruent. I de flesta kompilatorprojekten består framsidan i alla fall av en skanner och en parser.

Skannern går igenom den teckenström som programmeraren matar in och delar in tecknen och orden i grupper som till exempel operander, nyckelord och variabelnamn. Där skannern har i uppgift att läsa av texten har parsern som mål att förstå det programmeraren matat in. Parsern skapar med hjälp av symbolerna och deras kategorier ett binärt sökträd där varje löv representerar en variabel och varje nod en operation på två löv. Till exempel skulle uttrycket $x = 2 + y$ representeras som ett träd där den högsta noden är resultatet av tilldelningsoperatorn, till exempel med namnet x, med ett högerled som består av noden som representerar $+$ med två löv, 2 och y, som barn.

Det här sökträdet kallas för ett abstrakt syntaxträd. Efter att parsern gjort sitt, och om man traverserar trädet på rätt sätt, kommer trädet förutom operanderna också innehålla den matematiska precedens vi är ute efter, till exempel att multiplikation utförs före addition och att inre funktionsanrop utförs före yttre. Med hjälp av syntaxträdet kan också kompilatorns framsida rapportera semantiska fel i koden. Varje nod kan bara ha två barn och operander är alltid löv medan operationer kräver operander att operera på och så vidare. Om något tecken eller ord faller utanför trädet har det ingen logisk plats i helheten.

Eftersom de två delarna, framsidan och baksidan, är löst kopplade måste de ha något sätt att utbyta information. Det överenskomna sättet att överföra källkodens mening och struktur från framsidan till baksidan kallas för kompilatorns interna representation (IR). Typen av IR varierar från kompilator till kompilator, men till exempel det abstrakta sökträdet kan fungera helt eller delvis som en sådan representation.

2.2 Kompilatorns baksida

Kompilatorns andra steg, baksidan, används för att generera maskinkod som kan köras på processorn. För det syntaxträd jag diskuterade ovan kan den här processen illustreras som postfix traversering av trädet där löven 2 och `y` översätts till instruktionerna (`load a 2`) och (`load b y`) och noden med operatorn $+$ blir till instruktionen (`add c a b`) och tilldelningen $=$ till (`mov v1 c`). Instruktionerna radas efter varandra och på grund av trädets struktur kommer de i rätt ordning för att kunna evalueras. Resultatet efter tilldelningen finns i register `v1` vilket i det här fallet används som platsen för variabeln `x` i minnet.

De tidigare stegen är i praktiken lösta problem. Datavetenskapen känner till de mest optimala metoderna för att bygga en framsida oberoende av vilka parametrar man ger för konstruktionen. Baksidan har däremot flera delar som ger upphov till problem som är NP-kompletta, det vill säga problem som inte kan lösas allmänt, utan bara för begränsade delmängder av parametrarna. Till exempel när kompilatorn ska välja rätt maskininstruktioner för en viss del av det abstrakta syntaxträdet kan det i praktiken existera ett antal instruktioner som alla i slutändan skulle ge mer eller mindre samma resultat, vilket inte skulle vara ett problem om resultatet är det viktigaste, men den optimala

lösningen går inte att hitta för det allmänna fallet. Kompilatorn måste därför hitta en tillräckligt bra lösning genom att utnyttja diverse heuristiska algoritmer [9, 8].

2.3 Optimeraren

Den maskinkod som producerats enligt modellen diskuterad tidigare borde vara korrekt. Den borde alltså uttrycka samma sak som den ursprungliga källkoden och dessutom borde man kunna köra den på den önskade processorn. En kompilator som lyckas producera korrekt maskinkod är redan användbar, bland annat gör den programmerarens arbete på många sätt mycket lättare, men det finns ändå ett antal kompilatorer som skapar mera än bara en översättning. Kompilatorer har i själva verket från första början konstruerats för att förbättra de program som de behandlar. När högnivåspråket Fortran utvecklades i slutet på 1950-talet var det speciellt viktigt att maskinkoden som kompilatorn producerade skulle vara så effektiv som möjligt, bl.a. på grund av de ringa resurserna i form av processorkraft och minne. Fortran kompilerades därför från första början med hjälp av en kompilator som var utvecklad för att inte bara översätta utan också optimera källkoden [5]. Många av dagens kompilatorer är av samma sort som Fortran kompilatorn, alltså optimerande kompilatorer.

Det vanliga sättet att planera optimerande kompilatorer är lägga till ett steg mellan framsidan och baksidan, ett steg som kan kallas mellansteget eller optimeraren. Optimerarens uppgift är som sagt att analysera den interna representationen och transformera den enligt behov till en mera effektiv form. Effektivitet kan förstås betyda flera olika saker, till exempel hur snabbt kompilatorn arbetar, hur snabbt det färdiga programmet exekveras och hur mycket utrymme binärfilerna upptar. Det är upp till programmeraren att från projekt till projekt avgöra vilka apekter som är viktiga. Ofta är optimeringssteget uppgjort av många korta och enkla transformationer som kan sättas ihop på olika sätt för att uppnå den önskade effektiviteten [9]. I vilken ordning och hur dessa skall användas är inte trivialt, kompilatorutvecklaren måste se till att transformationerna i sitt sammanhang inte påverkar programmets mening, men på samma gång måste han eller hon komma ihåg att konstruera kompilatorn så att transformationerna används i ett sammanhang där de gör så stor nytta som möjligt.

Meningen med den här avhandlingen är som sagt att undersöka hur kompilatorerna GCC och LLVM utför optimeringen av de program de tar in. Både GCC och LLVM är stora projekt med stöd för många olika språk och processorarkitekturer, men på grund av att båda projekten använder en vettig IR mellan framsidan, optimeraren och baksidan har man i båda fallen lyckats skapa ett arbetsflöde som gör det möjligt att återanvända optimeraren för alla de språk som framsidorna stöder. Valet av IR är mycket viktigt för hur optimeringsprocessens framgång [10].

3 GCC

GCC är alltså ett optimerade och modulärt kompilatorramverk bestående av framsidor som stöder allt som allt sju olika programmeringsspråk och en samling baksidor som tillsammans stöder ett flertal olika processorarkitekturer. De delar alla en gemensam optimeringssprocess. [2].

Fördelen med det här upplägget är inte egentligen teknisk, i teorin skulle ett antal olika kompilatorer lika bra klara av de uppgifter som GCC uppfyller som ett enhetligt system, men den mänskliga och organisatoriska fördelen av detta planeringssätt är stor. Detta upplägg gör att arbetet som görs för att förbättra optimeringssteget kommer att leda till bättre program i alla de språk som projektet stöder. På samma sätt kommer förbättringar till en enskild arkitekturs kodgenerering också förbättra alla de program som kompileras för den, oberoende av det ursprungliga programmeringsspråket. Fördelningen mellan framsida, optimerare och baksida gör också att personer med olika färdigheter och bakgrund kan bidra till den del av projektet som de är mest lämpade för [7].

3.1 GCCs interna representation

Både GCC och LLVM använder sig av IR i SSA (Static Single Assignment) form, det vill säga en representation av instruktioner där varje variabel bara blir tilldelad en gång [10]. Alla språk där en enskild variabel inte kan få ett nytt värde efter den ursprungliga tilldelningen är i en viss mening i SSA-form, till exempel en del funktionella programmeringsspråk kan anses vara i SSA-form [11], men termen SSA används oftast bara när man pratar om kompilatorer

och deras interna representationer.

GCC använder sig av tre olika IR, GENERIC, GIMPLE och RTL (Register Transfer Language). GENERIC är ett samlande namn för de datastrukturer som framsidorna använder sig av för att representera källkod med hjälp av syntaxträd. Idén med GENERIC är att träden skapas på samma sätt för alla programmeringsspråk och på det sättet hjälpa framsidorna med att utnyttja optimeraren genom att erbjuda ett enhetligt gränssnitt mot mellansteget. I GCC generar alltså framsidorna syntaxträd i GENERIC form som sedan skickas vidare till optimeraren för att behandlas vidare. Programmet som GENERIC är ännu inte i SSA form [1].

GIMPLE är en IR som introducerades samtidigt som GENERIC. GIMPLE representerar koden som träd i så kallad three-address code (TAC), det vill säga programkod i en form där varje uttryck har högst tre operander. Till exempel uttrycket `a = 2 * (b ** 2)` utvecklas till uttrycken `b1 = b ** 2` och `a = 2 * b1` när det omvandlas till TAC. GCC konverterar GENERIC till GIMPLE som ett steg i optimeringsprocessen. GENERIC i sig fungerar egentligen mest som ett gränsnitt mot optimeraren, några få transformationer kan appliceras redan i det skedet, men de flesta sker först när koden är i GIMPLE format. GIMPLE existerar i mellansteget både som TAC och som TAC i SSA form, beroende på vilka transformationer som används [1].

GENERIC och GIMPLE liknar i sin syntax ganska långt C. Den tredje formen av IR som GCC använder kallas Register Transfer Language, alltså RTL, och liknar mest någon form av assembler med lite extra. Man kan tänka sig RTL som ett sorts universell assembler. På samma sätt som GENERIC kan representera alla högnivåspråk är idén att RTL skall kunna representera assembler för alla processorer som GCC stöder. Tranformationerna som görs på RTL-kod är relativt få jämfört med dem som kan utföras på koden i GIMPLE format eftersom RTL är på en så låg nivå att det inte går att effektivt göra sådana transformationer som kräver en större kontext, till exempel när det kommer till loopar i programmet [9, 1].

3.2 GCCs optimeringspass

Som tidigare diskuterat finns de flera olika aspekter en programmerare kan vilja att kompilatorn ska effektivera som en del av optimeringsprocessen. GCC tar emot några parametrar som argument för att ange vilken nivå av optime-

ring som programmeraren är ute efter. DDessa alternativ är `O0, O1, O2, O3, Os, Ofast, Og` och anger vilka transformationer som GCC skall använda sig av [1].

- `O0` ger samma resultat som om GCC används utan ett argument och ger direktiv åt kompilatorn att kompilera så snabbt som möjligt och bibehålla den usrprungliga strukturen så långt som möjligt för att till exempel göra det lättare att avlusa programmet i samband med utvecklingsarbetet
- `O1,O` instruerar GCC att skapa en så utrymmessnål och snabb binär som möjligt, utan att optimeringssteget avsevärt förlänger kompileringstiden
- `O2` instruerar GCC att agera som med `O1` men utan att ta hänsyn till kompileringstiden
- `O3` instruerar GCC att köra alla optimeringspass för att göra den resulterande binärfilen så snabb som bara möjligt, med den sidoeffekten att filen kan ta upp mera hårddiskutrymme än med `O2`
- `Os` instruerar GCC att arbeta som med `O2` samt göra det resulterande programmet så litet som möjligt
- `Ofast` instruerar GCC att använda samma pass som i `O3` samt de pass som inte kan användas på program som är behöver förlita sig på IEEE och ISO standarder för matematiska funktioner
- `Og` låter GCC arbeta som med `O1` men ser till att information som kan vara nyttig för avlusning inte försvinner i transformationerna

4 LLVM

LLVM liknar i sin struktur GCC i den bemärkelsen att de båda består av ett antal framsidor som delar en gemensam optimerande mittendel som i sin tur skickar vidare den optimerade koden till de processorspecifika baksidorna. Strukturen på LLVMs IR och hur den hanteras i ramverket är däremot annorlunda än GCC. Det här syns speciellt i arbetet som krävs för att utveckla nya framsidor, LLVM är på flera sätt mera modulär och kräver mindre kunskaper om hela systemet än GCC [7].

4.1 LLVMs interna representation

4.2 LLVMs optimeringspass

5 Slutsatser

Referenser

[1] R. M. Stallman and the GCC Developer Community, "Gnu compiler collection internals (6.0.0)," 2016.

[2] R. M. Stallman and the GCC Developer Community, "Using the gnu compiler collection (5.3.0)," 2016.

[3] *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation*, no. 0-7695-2102-9, Proceedings of the International Symposium on Code Generation and Optimization, 2004.

[4] C. Lattner, "Llvm: An infrastructure for multi-stage optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[5] R. W. Sebesta, *Concepts of Programming Languages*. Addison-Wesley, 2012.

[6] D. E. Knuth and L. T. Pardo, "The early development of programming languages," *A history of computing in the twentieth century*, 1980.

[7] C. Lattner, "Llvm," in *The Architecture of Open Source Applications* (G. W. Amy Brown, ed.), vol. 1 of *The Architecture of Open Source Applications*, Amy Brown,Greg Wilson, 2014.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, second edition ed., 2006.

[9] K. Cooper and L. Torczon, *Engineering a Compiler*. Morgan Kaufmann, 2 ed., 2011.

[10] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[11] A. W. Appel, "Ssa is functional programming," *ACM SIGPLAN Notices*, 1998.