Konstrukcija kompilatora
— LLVM —

Milena Vujošević Janičić

www.matf.bg.ac.rs/~milena

Matematički fakultet, Univerzitet u Beogradu

Pregled

Najznačajniji C/C++ kompajleri

- GCC
- Clang/LLVM
- ICC
- Microsoft Visual C++ compiler
- Borland C++
- Dev C++
- Embracadero

Pregled

LLVM https://llvm.org/

- Projekat LLVM sastoji se iz biblioteka i alata koji zajedno čine veliku kompajlersku infrastrukturu.
- Započet je kao istraživački projekat na Univerzitetu Ilinois, 2000. godine, kao istraživački rad sa ciljem proučavanja tehnika kompajliranja i kompajlerskih optimizacija
- Ideja: skup modularnih i ponovno iskoristivih kompajlerskih tehnologija, čiji je cilj podrška statičkoj i dinamičkoj kompilaciji proizvoljnih programskih jezika.

LLVM https://llvm.org/

- Osnovna filozofija LLVM-a je da je „svaki deo neka biblioteka" i veliki deo koda je ponovno upotrebljiv.
- First release: Nov 18, 2003 - LLVM 1.0
- Inicijatori projekta su Kris Latner (eng. *Chris Lattner*) i Vikram Adve (eng. *Vikram Adve*).

LLVM https://llvm.org/

- Od tada, LLVM je prerastao u projekat koji se sastoji od velikog broja podprojekata koji se koriste
  - U industriji, na primer Apple, ARM, NVIDIA, Mozilla, Cray...
  - I akademiji:

    LLVM: A compilation **framework** for **lifelong program** analysis & transformation
    C Lattner, V Adve - ... on Code Generation and Optimization, 2004 ..., 2004 - ieeexplore.ieee.org
    We describe **LLVM** (low level virtual machine), a compiler **framework** designed to support transparent, **lifelong program** analysis and transformation for arbitrary **programs**, by providing high-level information to compiler transformations at compile-time, link-time, run ...
    ☆ 99 Cited by 4607 Related articles All 58 versions

- LLVM je dobio 2012. godine nagradu **ACM Software System Award** (nagrada se dodeljuje jednom softverskom sistemu godišnje, počevši od 1983. godine, gcc je dobio nagradu 2015. godine)

  `https://en.wikipedia.org/wiki/ACM_Software_System_Award`

LLVM https://llvm.org/

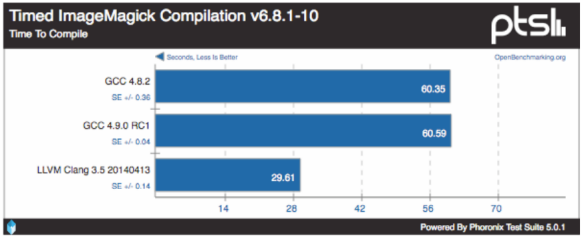
- LLVM je sveobuhvatni naziv za više projekata koji zajedno čine potpun kompajler: prednji deo (eng. *frontend*), središnji deo (eng. *middleend*), zadnji deo (eng. *backend*), optimizatore, asemblere, linkere, libc++ i druge komponente.
- Projekat je napisan u programskom jeziku C++, koristeći prednosti objektno-orijentisane paradigme, generičkog programiranja (upotrebom šablona), a takođe sadrži i svoje implementacije raznih struktura podataka koje se javljaju u standardnim bibliotekama programskog jezika C/C++

LLVM https://llvm.org/

- Clang/Clang++ se često koristi kao sinonim za LLVM kompajler
- Clang/Clang++ ima odlične karakteristike u poređenju sa kompajlerima kao što su gcc i icc.
- Kada se vrše poređenja, ona se vrše na odabranim primerima. U zavisnosti od primera (eng. *benchmarks*), poređenja mogu da daju različite rezultate.

LLVM https://llvm.org/

- Obično Clang/Clang++ ima brže vreme kompilacije u odnosu na pomenute kompajlere.



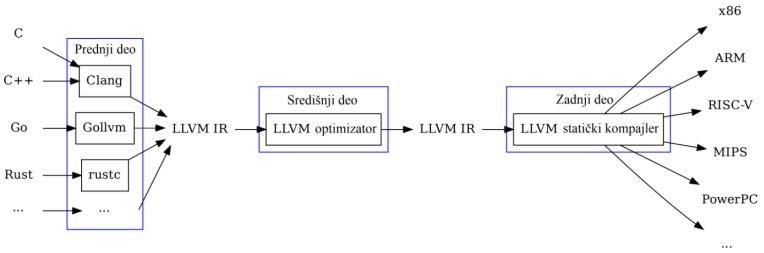
http://www.phoronix.com/scan.php?page=article&item=gcc49_compiler_llvm35&num=2

LLVM https://llvm.org/

- Licenca koda u okviru LLVM projekta je "Apache 2.0 License with LLVM exceptions"
- Licenciranje se menjalo tokom razvoja projekta, ali je uvek bila licenca otvorenog koda (MIT license, the UIUC License (a BSD-like license)...)

LLVM https://llvm.org/

- LLVM implementira kompletan tok kompilacije:
  - Front-end, leksička, sintaksička i semantička analiza — npr alat Clang
  - Middle-end, analize i optimizacije — npr alat opt
  - Back-end, različite arhitekture — npr alat llc



LLVM https://llvm.org/

- Na primer, korisnik može:
  - da kreira svoj front-end, i da ga poveže na LLVM, koji će mu dodati middle-end i back-end
  - za izabran postojeći front-end/back-end da isprobava promene na nivou middle-end-a
  - za novu arhitekturu obezbedi back-end, i da koristi Clang i postojeci middle-end
- Jasno razdvojene celine, čitljiv međukod

LLVM https://llvm.org/

- Za korišćenje biblioteka LLVM-a potrebno je poznavati
  - Moderan C++, posebno generičko programiranje
  - Osnove objektno-orijentisanog programiranja
  - Osnove konstrukcije kompilatora

LLVM https://llvm.org/

- Kompajlerska infrastruktura sa velikim brojem pomoćnih alata, podržanih arhitektura, sa interfejsima ka ostvarivanju različitih analiza i optimizacija https://llvm.org/docs/CommandGuide/index.html

```
$> cd llvm/Debug+Asserts/bin
$> ls
FileCheck          count              llvm-dis           llvm-stress
FileUpdate         diagtool           llvm-dwarfdump     llvm-symbolizer
arcmt-test         fpcmp              llvm-extract       llvm-tblgen
bugpoint           llc                llvm-link          macho-dump
c-arcmt-test       lli                llvm-lit           modularize
c-index-test       lli-child-target   llvm-lto           not
clang              llvm-PerfectSf     llvm-mc            obj2yaml
clang++            llvm-ar            llvm-mcmarkup      opt
llvm-as            llvm-nm            pp-trace           llvm-size
clang-check        llvm-bcanalyzer    llvm-objdump       rm-cstr-calls
clang-format       llvm-c-test        llvm-ranlib        tool-template
clang-modernize    llvm-config        llvm-readobj       yaml2obj
clang-tblgen       llvm-cov           llvm-rtdyld        llvm-diff
clang-tidy
```

LLVM Core libraries

Dva osnovna podprojekta LLVMa su: *LLVM Core libraries* i *Clang*

- *The LLVM Core libraries* obezbeđuju moderni optimizator koji ne zavisi od izvornog koda niti od ciljne arhitekture, kao i podršku za generisanje koda za puno popularnih CPUa (kao i za neke koji se ređe koriste). Ove biblioteke koriste posebnu reprezentaciju koda koja se naziva LLVM međureprezentacija ("LLVM IR").
- *LLVM Core libraries* su dobro dokumentovane, i daju mogućnost da se veoma jednostavno implementira novi jezik koji koristi LLVMov optimizator i generator koda.

Clang

- **Clang** je LLVMov C/C++/Objective-C kompilator, koji ima za cilj
  - efikasnu kompilaciju,
  - veoma detaljnu dijagnostiku (grešake i upozorenja su dobro objašnjene),
  - obezbeđivanje platforme za izgradnju različitih alata koji rade na nivou izvornog koda (analiza i transformacije).
- Clang Static Analyzer i Clang-tidy su alati koji automatski pronalaze greške u kodu, i odlični su primeri alata koji se mogu izgraditi korišćenjem Clang-ovog prednjeg dela kao biblioteke za parsiranje C/C++ koda.

Ostali LLVM projekti

- LLDB — LLVM debager
- libc++ and libc++ ABI — standardna biblioteka
- compiler-rt — podrška za različite run-time izazove
- MLIR — nova međureprezentacija
- OpenMP — podrška za Open Multi-Processing kod
- polly — optimizator petlji
- libclc — standardna biblioteka za OpenCL
- klee — simboličko izvršavanje koda
- lld — LLVM linker

LLDB

- LLDB je debager nove generacije
- On je izgrađen kao skup ponovno upotrebljivih komponenti koje koriste postojeće biblioteke LLVM projekta
- To je debager koji se prirodno koristi za kod koji je preveden sa Clang/LLVM kompajlerom
- Iz Clang-a koristi AST i parser izraza, a iz LLVM-a npr LLVM JIT i LLVM disasembler
- Veoma je vremenski i memorijski efikasan, efikasniji od GDB-a po nekim kriterijumima

libc++ i libc++ ABI

- Projekti **libc++** i **libc++ ABI** pružaju efikasnu i u skladu sa standardom implementaciju C++ standardnih biblioteka, uključujući punu podršku za C++11 i C++14.
- `https://libcxx.llvm.org/`

compiler-rt

- Projekat compiler-rt sastoji se od nekoliko manjih podprojekata, uključujući **builtins**, **sanitizer runtimes**, **profile**
- **builtins** - jednostavna biblioteka koja omgućava softversku implementaciju instrukcija međukoda i drugih runtime komponenti za koje ne postoje odgovarajuće instrukcije u okviru ciljne mašine (tj kada ciljna mašina ne može da neku operaciju međukoda direktno prevede kao kratak niz instrukcija koji odgovaraju ciljnoj arhitekturi)
- Na primer, kada se radi kompilacija za 32-bitnu arhitekturu, konvertovanje double promenljive u 64-bitnu celobrojnu vrednost se kompilira u runtime poziv funkcije `__fixunsdfdi`. Biblioteka **builtins** obezbeđuje optimizovanu implementaciju ove i drugih rutina niskog novoa, bilo u obliku koji je nezavisan od ciljne arhitekture (C kod), bilo u obliku koji je optimizovan asembler

compiler-rt

- **Sanitizer runtimes** obezbeđuje implementaciju run-time biblioteka potrebnih za alate za dinamičko testiranje softvera, ako što su **AddressSanitizer**, **ThreadSanitizer**, **MemorySanitizer**, i **DataFlowSanitizer**.
- Ovi alati omogućavaju da se lakše pronađu greške u programima, u radu sa memorijom, nitima ili u protoku podataka
- **profile** - biblioteka koja se koristi da se pokupe podaci o izvršavanju koda (razne vrste pokrivenosti koda koje koriste profajleri).

MLIR

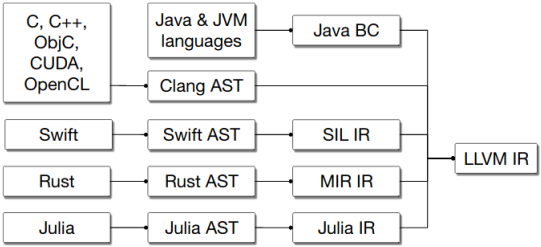
- Multi-Level Intermediate Representation
  `https://arxiv.org/pdf/2002.11054v2.pdf`
- Potprojekat MLIR je novi pristup građenju ponovoiskoristive i proširive kompajlerske infrastrukture
- MLIR ima za cilj da smanji cenu izgradnje novih kompajlera i da pomogne u povezivanju postojećih kompajlera, kao i da unapredi kompilaciju koja ima za cilj heterogen hardver (izvršavanje jedinstvenog softvera na različitim hardverskim komponentama, npr na automobilu)

MLIR

- MLIR omogućava dizajn i implementaciju generatora koda, translatora i optimizatora različitih novoa apstrakcije (tj optimizacije koje se vrše u različitim delovima kompilatora, neke kojima su potrebne informacije visokog nivoa nasuprot optimizacijama kojima su potrebne informacije niskog nivoa), kao i kroz različite domene primene, različite hardvere i različite sredine izvršavanja (engl. *execution environments*).

MLIR

- Neki problemi se bolje modeluju na višem a neki na nižem novou apstrakcije
- Na primer, analiza izvornog C++ koda je veoma teška na nivou LLVM IR



MLIR

- Mnogi jezici, uključujući npr Swift, Rust, Julia, Fortran, razviju svoj sopstveni IR sa ciljem da reše domenski specifične probleme, (npr optimizacije specifične za jezik ili biblioteku jezika, proveravanje tipova koje je osetljivo na kontrolu toka) i da unaprede implementaciju daljeg procesa kompilacije
- Slično, sistemi za mašinsko učenje tipično koriste ML grafove kao domenski specifične apstrakcije
- Iako je razvoj domenski specifičnog IR poznata tehnika, ona je ipak veoma skupa, zahteva vreme i trud, i podložna je greškama

MLIR

- Projekat MLIR ima za cilj da direktno adresira ove izazove — tj da omogući da se veoma jeftino i jednostavno definiše i uvede novi nivo apstrakcije, i da obezbedi infrastrukturu za rešavanje čestih problema koji se javljaju u izgradnji kompajlera
- MLIR to mogućava tako što
  - (1) standardizuje Static Single Assignment (SSA) zasnovane strukture podataka za IR
  - (2) obezbeđuje deklarativni sistem za definisanje IR dijalekata
  - (3) obezbeđuje širok dijapazon potrebne infrastrukture (uključujući dokumentaciju, logiku za parsiranje i štampanje, praćenje lokacija, podršku za višenitnu kompilaciju, upravljanje prolazima i slično)

OpenMP (Open Multi-Processing)

- Projekat **OpenMP** obezbeđuje runtime podršku za korišćenje OpenMP konstrukta u okviru Clang-a
- OpenMP Application Program Interface (OpenMP API) — multi-platformsko višeprocesorsko programiranje sa deljenom memorijom za paralelizam u jezicima C, C++ i Fortran
- `http://www.openmp.org`
- OpenMP održava neprofitni konzorcijum *OpenMP Architecture Review Board* (skraćeno OpenMP ARB) koji uključuje najveće proizvođače hardvera i softvera, uključujući AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, itd.

OpenMP (Open Multi-Processing)

- Primer koda:

```
int main(int argc, char **argv)
{
    int a[100000];
    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }
    return 0;
}
```

polly

- Projekat **polly** je optimizator petlji i upotrebe podataka, kao i infrastruktura za optimizaciju u okviru LLVMa
- U projektu se koristi apstraktna matematička reprezentacija za analizu i optimizaciju pristupa memoriji programa
- Trenutno izvodi klasične transformacije petlji, posebno razdvajanje (engl. *tiling*) i spajanje (engl. *fusion*) petlji.

Loop tiling

- *Loop tiling* — transformacija petlje koja iskorišćava prostornu i vremensku poziciju podataka tj pristupa podacima kod ugnježdenih petlji. Ova transformacija dozvoljava podacima da im se pristupa u blokovima, pri čemu se veličina bloka definiše kao parametar ove transformacije.
- *Loop tiling* može da ima za cilj optimizaciju korišćenja različitih nivoa memorije (uključujući različite nivoe keš memorije) i može da se podešava da se maksimizuje ponovno korišćenje podataka na odgovarajućem nivou memorijske hijerarhije. Transformacija uključuje korišćenje specifičnih veličina blokova koje se mogu fiksirati u fazi kompilacije, a koji se računaju u odnosu na veličine memorije.

Loop tiling

- *Loop tiling* deli iteracije petlje u manje delove tako da se obezbedi da podaci koji se koriste u petlji ostanu u keš memoriji do ponovnog korišćenja. Podela petlje na *prostorne iteracije* vodi deljenju velikih nizova u manje blokove, podešavajući veličinu niza kojem se pristupa veličini keša, i time se poboljšava ponovna iskorišćenost podataka u kešu.

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    for(k=0; k<N; k++)
      C[i][j] = C[i][j] +
                A[i][k]*B[k][j];
```

(A)

```
for(jj=0; jj<N; jj+=Bj)
  for(kk=0; kk<N; kk+=Bk)
    for(i=0; i<N; i++)
      for(j=jj; j<min(jj+Bj,N); j++)
        for(kk=0; kk<min(kk+Bk,N); kk++)
          C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

(B)

Loop fusion

- *Loop fusion* — odnosno spajanje petlji: spaja više petlji u jednu
- *Loop fission* — odnosno razdvajanje petlji: razdvaja jednu petlju u više petlji
- Spajanje petlji može da poveća nivo paralelizma, lokalne upotrebe podataka i da smanji višak koji nastaje kod kontrole petlje (smanjenjem broja petlji smanjuje se upotreba brojača), ali može da poveća pritisak kod alokacije registara

polly

- Polly može da iskoristi OpenMP paralelizam i mogućnosti SIMD instrukcija (single instruction, multiple data)
- Takođe, u okviru projekta se radi i na automatskom generisanju koda za GPU (graphics processing unit)
- `http://polly.llvm.org/`

libclc — standardna biblioteka za OpenCL

- Biblioteka **libclc** je projekat koji implementira standardnu biblioteku za OpenCL
- OpenCL (Open Computing Language) je okvir za pisanje paralelnih programa koji se mogu izvršavati na heterogenim platformama koji se sastoje od procesora (CPU), grafičkih kartica (GPU), procesora za obradu digitalnih signala (DSP), *field-programmable gate arrays* (FPGAs) i slično
- OpenCL definiše programski jezik (zasnovan na C99 i C++11) za programiranje ovih uređaja kao i dogovarajuće interfejse za programiranje (API) koji omogućavaju kontrolu platforme i izvršavanje programa na ovim uređajima.
- OpenCL pruža standardni interfejs za paralelno izračunavanje korišćenjem paralelizma zadataka i paralelizma podataka.

libclc — standardna biblioteka za OpenCL

- OpenCL se koristi u industriji
- Održava ga neprofitni tehnološki konzorcijum *Khronos Group*
- **OpenCL** poboljšava brzinu velikog spektra aplikacija na različitim tržištima uključujući alate za razvoj softvera, naučni i medicinski softver, procesiranje slika, treniranje neuronskih mreža...
  `https://www.khronos.org/opencl/`

klee — automatsko generisanje testova i pronalaženje grešaka

- **Klee** implementira *simboličku virtuelnu mašinu* koja ima za cilj automatsko generisanje testova i pronalaženje grešaka u kodu
- Koristi dokazivač teorema za evaluaciju putanji i uslova ispravnosti
- Važna osobina ovog alata je da za detektovane greške može da proizvede test primere koji prouzrokuju greške
- `http://klee.github.io/`

LLD — linker

- Projekat **LLD** je novi linker.
- To je zamena za sistemske linkere koja bi trebalo da radi brže
- Prihvata iste argumente komandne linije i skripte za linkovanje kao GNU linker
- U okviru FreeBSD projekata, radi se na tome da LLD postane podrazumevani sistemski linker u budućim verzijama operativnog sistema

Ostalo

- Pored zvaničnih projekata koji su potprojekti LLVMa, postoji i veliki broj drugih projekata koji koriste komponente LLVMa za različite zadatke
- Kroz ove eksterne projekte LLVM se može koristiti da se kompajliraju jezici kao što su Ruby, Python, Haskel, Rust, D, PHP, Pure, Lua itd.
- Osnovna snaga LLVMa su svestranost, fleksibilnost i upotrebljivost, zbog čega se koristi za veoma širok izbor različitih zadataka

Pregled

LLVM Prednji deo

- Prednji deo (eng. *frontend*): Prevođenje i analiza izvornog koda, programa napisanih u višim programskim jezicima u LLVM-ovu međureprezentaciju (eng. *intermediate representation*).
- Prevođenje obuhvata leksičku, sintaksičku i semantičku analizu, a završava se fazom generisanja LLVM-ovog međukoda.
- Clang je front-end za C jezike
  Clang: a C Language Family Frontend for LLVM (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript)
- Frontendi za druge programske jezike nisu sistavni deo Clang-a i LLVM-a, ali postoji podrška za veliki broj značajnih jezika, npr Haskell, Fortran, Pascal, Swift, Go, Ruby, JavaScript, Lua...
  `http://llvm.org/ProjectsWithLLVM/`

LLVM Prednji deo

- Prve verzije LLVM-a su kao frontend koristile gcc, tj. za generisanje srednje reprezentacije se koristila modifikovana verzija gcc-a zvana `llvm-gcc`. Da bi se koristio ovaj alat bilo je neophodno skinuti i izgraditi ceo modifikovan gcc. Ovo je bilo vremenski zahtevno i komplikovano.
- Nakon toga, projekat `DragonEgg` je imao za cilj da se izbegne potreba za izgradnjom celog gcc-a. Umesto toga `DragonEgg` predstavlja dodatak koji se uključuje u čist gcc.
- Danas za time više nema potrebe i zvanični prednji deo LLVM-a je Clang koji predstavlja prednji deo za C-olike jezike C\C++\Objective-C

Clang https://clang.llvm.org/

- Clang je prednji deo kompajlera koji prevodi C-olike jezike u srednju reprezentaciju. On je najprepoznatljiviji podprojekat projekta LLVM. Deo je paketnih sistema za skidanje programa na većini BSD (eng. Berkeley Software Distribution) i GNU/Linux distribucija.
- Često se Clang spominje kao kompajler ne uzimajući u obzir da je on samo prednji deo kompajlera i da se ispod njega nalazi moćan optimizator i generator koda.

Clang

- Clang je dizajniran tako da bude memorijski efikasan, da ispisuje jasne, pouzdane i korisne poruke o greškama i upozorenjima.
- Izražajna dijagnostika `https://clang.llvm.org/diagnostics.html`
- Takođe on pruža čist API tako da se on može koristiti u drugim projektima (najčešće kao sintaksni i semantički analizator).

Clang

- Izbor pravog interfejsa za Vašu aplikaciju: `https://clang.llvm.org/docs/Tooling.html`
- LibClang, Clang Plugins, LibTooling

LibClang

- Ceo sistem LLVM-a je organizovan kao skup biblioteka, pa tako imamo i biblioteku `libClang` koja treba da se uključi ukoliko želimo da korisimo API kompajlera Clang.
- LibClang je stabilni C interfejs za Clang. Postoje i drugi interfejsi, ali najčešće je ovo potrebno rešenje.
- LibClang treba koristiti kada koristite Clang iz nekog drugog jezika (tj ukoliko ga ne koristite iz C++-a)
- Takođe, to je pravi izbor ukoliko je potrebno iterirati kroz AST stablo, bez zalaženja u detalje implementacije u okviru Clanga
- Ipak, to nije dobar izbor ukoliko je potrebno imati punu kontrolu ASTa

Clang Plugins

- Clang Plugins omogućavaju pokretanje dodatnih akcija na ASTu kao deo kompilacije. Plugins su dinamičke biblioteke koje se učitavaju u fazi izvršavanja i lako se integrišu u okruženje za izgradnju koda.
- Koristi se za implementaciju novih upozorenja i grešaka
- Koristi se kada je potrebno proizvesti još neke dodatne fajlove prilikom bildovanja
- Koristi se kada je potrebna puna kontrola nad ASTom Clang-a
- `https://clang.llvm.org/docs/ClangPlugins.html`

LibTooling

- LibTooling je C++ interfejs koji ima za cilj da omogući pisanje samostalnih alata
- Koristi se npr za pisanje sintaksnih provera i alata za refaktorisanje
- Koristi se kada se radi analiza jednog fajla ili specifične grupe fajlova, nezavisno od sistema za bildovanje
- Koristi se kada je potrebna puna kontrola nad ASTom Clang-a

Primeri Clang zasnovanih alata

- Clang alati su kolekcija specifičnih alata za razvoj koji su zasnovani na LibTooling infrastrukturi kao deo Clang projekta
- Ovi alati imaju za cilj da automatizuju i unaprede osnovne razvojne aktivnosti C/C++ developera
- Primeri alata
  - Sintaksne provere (clang-check)
  - Automatsko popravljanje grešaka otkrivenih prilikom kompilacije (clang-fixit)
  - Automatsko formatiranje koda (clang-format)
- Clang Static Analyzer `http://clang-analyzer.llvm.org/` — Analiza koda i automatsko pronalaženje grešaka

Clang

- Sve kompajlerske opcije koje postoje u GCC-u su potpuno podržane i u kompajleru Clang.
- Clang podržava različite nivoe optimizacije, npr -O0, -O1, -O2 i -O3.
  Na primer, -O1 sadrži naredne optimizacije (LLVM/Darwin system), koje pripadaju srednjem delu:

  -targetlibinfo -no-aa -tbaa -basicaa -nol -globalopt -ipsccp -deadargelim -instcombine -simplifycfg -basiccg -prune-eh -inline-cost -always-inline -functionvars -sroa -domtree -early-cse -lazy-value-info -jump-threading -correlated-propagation -simplifycfg - instcombine -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -licm -lcssa -loop-unswitch -instcombine -scalar-evolution -loop-simplify - lcssa -indvars -loop-idiom -loop-deletion -loop-unroll -memdep -memcpyopt -sccp - instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree - memdep -dse -adce -simplifycfg -instcombine -strip-dead-prototypes -preverify - domtree -verify

opt (zapravo, srednji deo)

```
$> opt --help
  Optimizations available:
    -adce                       - Aggressive Dead Code Elimination
    -always-inline              - Inliner for always_inline functions
    -break-crit-edges           - Break critical edges in CFG
    -codegenprepare             - Optimize for code generation
    -constmerge                 - Merge Duplicate Global Constants
    -constprop                  - Simple constant propagation
    -correlated-propagation     - Value Propagation
    -dce                        - Dead Code Elimination
    -deadargelim                - Dead Argument Elimination
    -die                        - Dead Instruction Elimination
    -dot-cfg                    - Print CFG of function to 'dot' file
    -dse                        - Dead Store Elimination
    -early-cse                  - Early CSE
    -globaldce                  - Dead Global Elimination
    -globalopt                  - Global Variable Optimizer
    -gvn                        - Global Value Numbering
    -indvars                    - Induction Variable Simplification
    -instcombine                - Combine redundant instructions
    -instsimplify               - Remove redundant instructions
    -ipconstprop                - Interprocedural constant propagation
    -loop-reduce                - Loop Strength Reduction
    -reassociate                - Reassociate expressions
    -reg2mem                    - Demote all values to stack slots
    -sccp                       - Sparse Conditional Constant Propagation
    -scev-aa                    - ScalarEvolution-based Alias Analysis
    -simplifycfg                - Simplify the CFG
    ...
```

Pregled

Središnji deo

- Središnji deo (eng. *middleend*) obuhvata skup mašinski nezavisnih optimizacija koje se izvršavaju nad LLVMovom međureprezentacijom.
- LLVM-ova međureprezentacija predstavlja sponu između prednjeg i zadnjeg dela kompilatora.
- Dizajn LLVM IR je bio od posebne važnosti za uspeh projekta

Središnji deo

- LLVM-ov međukod može biti predstavljen pomoću narednih, međusobno ekvivalentnih formi:
  - **memorijska reprezentacija** (eng. *in-memory*) — implicitna forma međukoda, smeštena u radnoj memoriji računara;
  - **bitkod reprezentacija** (eng. *bitcode*) — prostorno efikasna reprezentacija, smeštena u datotekama ekstenzije *bc*;
  - **pseudo-asemblerska reprezentacija** (eng. *pseudo-assembly*) — lako čitljiva, smeštena u datotekama ekstenzije *ll*.

LLVM-ov međukod http://llvm.org/docs/LangRef.html

- LLVM-ov međukod organizovan je u hijerarhijsku strukturu koja se sastoji od nekoliko važnih entiteta.
- Sadržaj datoteke u kojoj se nalazi međukod definiše **modul**, koji predstavlja najviši entitet hijerarhije. Modul odgovara jednoj jedinici prevođenja (eng. *translation unit*) i sačinjen je od proizvoljnog broja **funkcija** i drugih entiteta (informacije o tipovima podataka ciljne arhitekture, globalnim promenljivama, prototipe eksternih funkcija kao i definicije deklarisanih struktura).

LLVM-ov međukod

- Funkcije konceptualno odgovaraju funkcijama programskog jezika C.
- Naredni entitet u hijerarhiji predstavljaju **osnovni blokovi** (eng. *basic blocks*), od kojih je sačinjena svaka funkcija. Osnovni blok predstavlja najduži niz troadresnih instrukcija koje se izvršavaju sekvencijalno. Važnu karakteristiku osnovnog bloka predstavlja činjenica da se u njega može uskočiti jedino sa početka, a iskočiti jedino sa kraja istog.
- Hijerarhijski najniže entitete predstavljaju same **instrukcije**.

LLVM-ov međukod

- Instrukcijski skup i memorijski model srednje reprezentacije je malo složeniji od asemblera.
- Instrukcije su slične RISC instrukcijama, ali sa ključnim dodatnim informacijama koje omogućavaju efikasne analize.

LLVM-ov međukod

Najvažnije jezičke osobine srednje reprezentacije su:

1. Poseduje svojstvo jednog statičkog dodeljivanja (eng. *Static Single Assignment - SSA*) koje omogućava lakšu i efikasniju optimizaciju koda. Ovo svojstvo označava da se prilikom korišćenja operatora dodele izračunata vrednost sa desne strane uvek dodeljuje novoj promenljivoj.
2. Instrukcije za obradu podataka su troadresne. Imaju dva argumenta i rezultat stavljaju na treću lokaciju.
3. Poseduje neograničen broj registara.

LLVM-ov međukod

- Ove osobine reprezentacije možemo uočiti iz primera

```
%foo = add i32 %0, %1
```

Ovaj primer će sabrati vrednosti lokalnih vrednosti %0 i %1 i smestiti ih u lokalnu promenljivu %foo. Lokalne promenljive su analogne registrima u asembleru i mogu imati bilo koje ime koje počinje sa %.

LLVM-ov međukod

Naredni fragment međukoda predstavlja definiciju funkcije *saberi*, koja prihvata dva celobrojna argumenta širine četiri bajta i vraća celobrojni rezultat iste širine:

```
define i32 @saberi(i32 %a, i32 %b) {
entry:
    %0 = alloca i32, align 4
    %1 = alloca i32, align 4
    store i32 %a, i32* %0, align 4
    store i32 %b, i32* %1, align 4
    %2 = load i32, i32* %0, align 4
    %3 = load i32, i32* %1, align 4
    %4 = add i32 %2, %3
    ret i32 %4
}
```

LLVM-ov međukod

- Lokalni identifikatori imaju prefiks *%*, a globalni prefiks *@*.
- LLVM podržava bogat skup tipova, od kojih su najkorišćeniji:
  - celobrojni tipovi proizvoljne širine — *i1*, *i8*, *i16*, *i32*, *i64*, *i128*
  - tipovi podataka zapisanih u pokretnom zarezu — *float*, *double*
  - vektorski tipovi — vektor koji sadrži deset celobrojnih elemenata širina osam bita predstavlja se kao `<10 x i8>`

LLVM-ov međukod

- Funkcija iz primera sadrži jedan osnovni blok, čiji početak je označen labelom *entry*, a kraj instrukcijom *ret*.
- U primeru se mogu videti naredne instrukcije:
  - `alloca` — rezerviše prostor na stek okviru funkcije,
  - `load` — vrši čitanje iz memorije,
  - `store` — vrši upis u memoriju,
  - `add` — vrši operaciju sabiranja,
  - `ret` — vraća kontrolu toka izvršavanja programa pozivaocu funkcije i (opciono) navedenu vrednost.

LLVM-ov međukod

Primer koji uključuje kontrolu toka

```
#include <stdio.h>

int main()
{
  int a, b, min;
  printf("Unesi dva broja\n");
  scanf("%d%d", &a, &b);
  min = a;
  if(b<min) min = b;
  printf("Najmanji broj je %d\n", min);
  return 0;
}
```

LLVM-ov međukod

- Pogledati primerMedjukoda.pdf

LLVM-ov međukod

- Memorijska reprezentacija međukoda vrlo blisko modeluje prikazanu sintaksu.
- Klase za izgradnju memorijske reprezentacije nalaze se u zaglavlju *include/llvm/IR*, a najvažnije od njih su: *Module*, *Function*, *BasicBlock* i *Instruction*.
- Nazivi klasa jasno ukazuju na svrhu njihove primene prilikom konstruisanja implicitne forme međukoda.

LLVM-ov međukod

Iterating Through Functions, Blocks and Insts

```
for(Function::iterator bb = F.begin(), e = F.end(); bb != e; ++bb) {
  for(BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e; ++i) {
    if(opCounter.find(i->getOpcodeName()) == opCounter.end()) {
      opCounter[i->getOpcodeName()] = 1;
    } else {
      opCounter[i->getOpcodeName()] += 1;
    }
  }
}
```

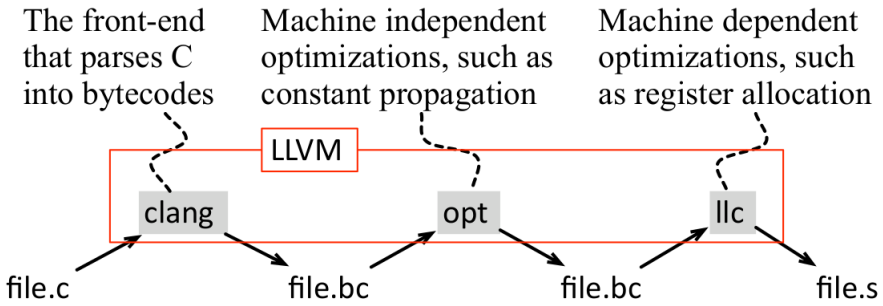
We go over LLVM data structures through iterators.

- An iterator over a Module gives us a list of Functions.
- An iterator over a Function gives us a list of basic blocks.
- An iterator over a Block gives us a list of instructions.
- And we can iterate over the operands of the instruction too.

```
for (Module::iterator F = M.begin(), E = M.end(); F != E; ++F);

for (User::op_iterator O = I.op_begin(), E = I.op_end(); O != E; ++O);
```

opt


The front-end that parses C into bytecodes
Machine independent optimizations, such as constant propagation
Machine dependent optimizations, such as register allocation
LLVM
clang
opt
llc
file.c
file.bc
file.bc
file.s

opt http://llvm.org/docs/CommandGuide/opt.html

- Alat **opt** vrši analizu i optimizaciju koda. Kao ulazni fajl, očekuje LLVM-IR kod (.ll format ili .bc format) i onda pokreće specifičnu optimizaciju ili vrši analizu koda, u zavnosti od ulaznih opcija (ukoliko je zadata ulazna opcija -analyze vrši se analiza koda)
- Ukoliko opcija -analyze nije zadata, opt pokušava da proizvede optimizovan izlazni kod.
- Postoje razne vrste optimizacija koje su dostupne, mogu se videti kroz -help opciju

opt https://llvm.org/docs/Passes.html

```
$> opt --help
  Optimizations available:
    -adce                       - Aggressive Dead Code Elimination
    -always-inline              - Inliner for always_inline functions
    -break-crit-edges           - Break critical edges in CFG
    -codegenprepare             - Optimize for code generation
    -constmerge                 - Merge Duplicate Global Constants
    -constprop                  - Simple constant propagation
    -correlated-propagation     - Value Propagation
    -dce                        - Dead Code Elimination
    -deadargelim                - Dead Argument Elimination
    -die                        - Dead Instruction Elimination
    -dot-cfg                    - Print CFG of function to 'dot' file
    -dse                        - Dead Store Elimination
    -early-cse                  - Early CSE
    -globaldce                  - Dead Global Elimination
    -globalopt                  - Global Variable Optimizer
    -gvn                        - Global Value Numbering
    -indvars                    - Induction Variable Simplification
    -instcombine                - Combine redundant instructions
    -instsimplify               - Remove redundant instructions
    -ipconstprop                - Interprocedural constant propagation
    -loop-reduce                - Loop Strength Reduction
    -reassociate                - Reassociate expressions
    -reg2mem                    - Demote all values to stack slots
    -sccp                       - Sparse Conditional Constant Propagation
    -scev-aa                    - ScalarEvolution-based Alias Analysis
    -simplifycfg                - Simplify the CFG
    ...
```

opt https://llvm.org/docs/Passes.html

- Svaka optimizacija je zadata u vidu prolaza kroz kod i moguće je korišćenjem alata opt pokrenuti izabrane optimizacije u odabranom redosledu
- Da bi se pokrenula neka konkretna optimizacija, navodi se ime odgovarajućeg prolaza -{passname}
- Redosled pojavljivanja opcija u komandnoj liniji određuje redosled kojim će prolazi biti izvršeni
- Neke transformacije zahtevaju da su pre njih izvršene odgovarajuće analize i da su prikupljeni potrebni podaci

opt https://llvm.org/docs/Passes.html

- Prolazi mogu da rade analizu *Analysis Passes* ili optimizaciju *Transformation Passes*
- Postoje i prolazi koji omogućavaju dobijanje raznih važnih informacija, npr informacija koje su bitne za razvoj drugih prolaza (kao npr pregledanje grafa kontrole toka funkcije) ili prolazi koji upsuju modul u odgovarajući bitcode (prebacivanje iz internog zapisa u .bc format). Ovi prolazi se zovu *Utility Passes*.
- Spisak dostupnih prolaza može se videti ovde `https://llvm.org/docs/Passes.html`

LLVM - kako implementirati novi prolaz?

- Da biste napisali neku optimizaciju/prolaz potrebno je da pročitate uputstvo `http://llvm.org/docs/WritingAnLLVMPass.html`
  - Potrebno je da znate šta želite da Vaš prolaz uradi
  - Potrebno je da znate na koji način zavisite od analiza programa koje vrše neki drugi prolazi
  - Potrebno je da znate na koji način utičete na analize programa koje su već izvršene

LLVM prolazi http://llvm.org/docs/WritingAnLLVMPass.html

- Svi prolazi su podklase apstraktne klase **Pass**, obično indirektno
- Pass definiše virtuelne metode koje svi prolazi treba da implementiraju
  - The doInitialization(_ *) method — uključuje razne opšte inicijalizacije, često nevezano od onoga što se konkretno radi
  - The runOnXXX method — suština posla koju prolaz treba da obavi (XXX može da bude runOnModule, runOnFunction...)
  - The doFinalization() method — ukoliko je potrebno još nešto, nakon glavnog posla, uraditi (ređe se koristi)

LLVM prolazi

- Svi prolazi imaju neka svoja pravila šta mogu/ne mogu da rade i kako moraju da se ponašaju
- U zavisnosti od nivoa na kojem operiše, prolaz može da bude
  - ModulePass,
  - CallGraphSCCPass, (strongly connected component, za bottom up prolaske)
  - FunctionPass, (init i finalize rade nad modulom)
  - LoopPass,
  - RegionPass, single entry-single exit regija u funkciji
  - BasicBlockPass (init i finalize rade nad funkcijom)
  - MachineFunctionPass — prolaz koji se ne korsiti u okviru opt-a već za generisanje ciljanog koda
- Registrovanje prolaza RegisterPass template - potrebno da bi prolaz mogao po imenu da se pokrene i isprati njegov rad

Odnos između prolaza

- Prolaz može da menja IR (transformation pass) ili samo da vrši analizu nad IR (analysis pass)
- Prolaz može da zahteva informacije koje obezbeđuju neki drugi prolazi. Zavisnosti između prolaza moraju da budu eksplicitno navedene. Čest obrazac je da transformacija zahteva da je pre toga urađena nekakva analiza koda.
- Metod **getAnalysisUsage** omogućava da se navede skup neophodnih prolaza koji mora da prethodi datom prolazu (implementacija metodom **AddRequired**) i skup prolaza čiji rezultati ostaju validni i nakon samog prolaza (metod **AddPreserved**)

Odnos između prolaza

- **PassManager** ima ulogu da obezbedi ispravno pokretanje prolaza, tj da obezbedi da su ispoštovane njihove međusobne zavisnosti, kao i da optimizuje redosled pozivanja prolaza. Zbog toga svaki prolaz mora da obezbedi informacije koji su prolazi pre njega neophodni i šta je od postojećih informacija narušeno nakon datog prolaza. Na primer, transformacija koda može uticati na neku prethodno urađenu analizu, ali i ne mora.

Odnos između prolaza

- PassManager obezbeđuje deljenje rezultata analize — pokušava da izbegne ponovno računanje analiza uvek kada je to moguće. To znači da on prati koje su analize na raspolaganju, koje su izvršavane ali zbog nekog drugog prolaza više ne važe, a koje tek treba da se urade. PassManager takođe prati koliko dugo treba rezultati da postoje u memoriji i oslobađa memoriju onda kada ti rezultati više nisu potrebni.
- PassManager obezbeđuje efikasno izvršavanje prolaza — npr, da bi se dobilo na efikasnoj upotrebi memorije, PassManager će pokrenuti najpre sve prolaze nad jednom funkcijom, zatim nad drugom i tako redom
- Bitno je da nakon svakog prolaza mora da se ostavi IR u validnom stanju

Pregled

LLVM https://llvm.org/

- Zadnji deo (eng. *backend*) — Generisanje mašinskog koda za navedenu ciljnu arhitekturu, na osnovu prosleđenog LLVM-ovog međukoda, vrši se u zadnjem delu infrastrukture. Takođe, u ovom delu se vrše mnoge mašinski zavisne optimizacije. Neke od trenutno podržanih arhitektura su: MIPS, ARM, x86, SPARC.
- Osnovni alat zadnjeg dela je llc — to je kompajler koji prevodi LLVM bitcode u asemblerski fajl

opt https://llvm.org/docs/CommandGuide/llc.html

```
$> llc --version

  Registered Targets:
    alpha     - Alpha [experimental]
    arm       - ARM
    bfin      - Analog Devices Blackfin
    c         - C backend
    cellspu   - STI CBEA Cell SPU
    cpp       - C++ backend
    mblaze    - MBlaze
    mips      - Mips
    mips64    - Mips64 [experimental]
    mips64el  - Mips64el [experimental]
    mipsel    - Mipsel
    msp430    - MSP430 [experimental]
    ppc32     - PowerPC 32
    ppc64     - PowerPC 64
    ptx32     - PTX (32-bit) [Experimental]
    ptx64     - PTX (64-bit) [Experimental]
    sparc     - Sparc
    sparcv9   - Sparc V9
    systemz   - SystemZ
    thumb     - Thumb
    x86       - 32-bit X86: Pentium-Pro
    x86-64    - 64-bit X86: EM64T and AMD64
    xcore     - XCore
```

opt https://llvm.org/docs/CommandGuide/llc.html

```
$> clang -c -emit-llvm identity.c -o identity.bc

$> opt -mem2reg identity.bc -o identity.opt.bc

$> llc -march=x86 identity.opt.bc -o identity.x86
```

LLVM https://llvm.org/docs/WritingAnLLVMBackend.html

Da bi se ostvarila podrška za novi hardver, potrebno je:

- Poznavati LLVM-IR, jer zadnji deo počinje obradu nad IR-om
- Poznavati tehnike pisanja prolaza
  - **MachineFunctionPass** — deo generatora koda, machine-dependent, vrsta FucntionPass-a, **runOnMachineFunction** — metod koji treba da bude predefinisan
  - Prolazi generatora koda se registruju i inicijalizuju na drugačiji način i ne mogu se pokretati sa opt
  - Ovaj prolaz ne može da menja osnovnu strukturu koda (npr ne može da menja osnovne blokove, module, aliase i slično)

LLVM https://llvm.org/docs/TableGen/index.html

Dalje, da bi se ostvarila podrška za novi hardver, potrebno je:

- Poznavati **td** format.
  - td format precizira način definisanja karakteristika ciljne arhitekture sa ciljem bržeg i jednostavnijeg automatskog generisanja odgovarajućih C++ klasa i metoda
  - Sintaksa liči na generičko programiranje u C++-u
  - Aplikacija TableGen na osnovu ulaznih fajlova u td formatu generise C++ kod koji je potreban za generisanje koda.

LLVM https://llvm.org/docs/CodeGenerator.html

Na kraju, da bi se ostvarila podrška za novi hardver, potrebno je:

- Poznavati algoritme koji se koriste u fazi generisanja koda: izbor instrukcija, redosled instrukcija, SSA-zasnovane optimizacije, alokacija registara, ubacivanje prologa/epiloga funkcija, kasne optimizacije na mašinskom kodu i emitovanje koda
  - Za svaku od prethodnih faza postoji skup klasa koje treba poznavati i predefinisati.
  - Imena klasa najčešće govore o nameni klase, npr klase TargetMachine, DataLayout, TargetRegisterInfo, TargetInstrInfo, TargetFrameLowering, TargetSubtarget, MachineFunction, MachineBasicBlock, MachineInstr...

Pregled

Zaključak

- LLVM je jedan od najznačajnijih projekata otvorenog koda sa višestrukim industrijskim i akademskim primenama
- Postoji veliki broj potprojekata u okviru LLVM infrastrukture, kao i veliki broj spoljnih projekata koji koriste LLVM
- U okviru LLVMa jasno su izdvojeni prednji, središnji i zadnji deo
- Struktura koda je takva da omogućava ponovnu upotrebljivost koda i olakšava građenje novih biblioteka i funkcionalnosti
- Moguće su implementacije dodatnih funkicionalnosti u svakom delu

Literatura

- Uglavnom su uz svaku temu navođeni odgovarajući linkovi koji treba da Vas upute na literaturu. Dodatni linkovi:
  - https://org.computer/dist/pdf/llvm-essentials.pdf
  - https://org.computer/dist/pdf/llvm-cookbook.pdf
  - http://faculty.sist.shanghaitech.edu.cn/faculty/songfu/course/spring2018/CS131/llvm.pdf
  - https://llvm.org/pubs/2002-12-LattnerMSThesis-book.pdf