Konstrukcija kompilatora

— Alokacija registara —

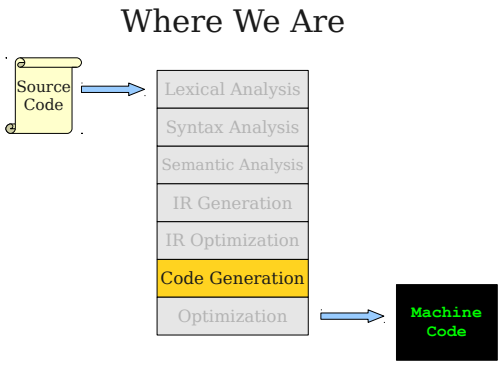
Milena Vujošević Janičić

Matematički fakultet, Univerzitet u Beogradu

Sadržaj

Na slajdovima obavezno pogledati animacije koje su ovde izostavljene!

1 Uvod



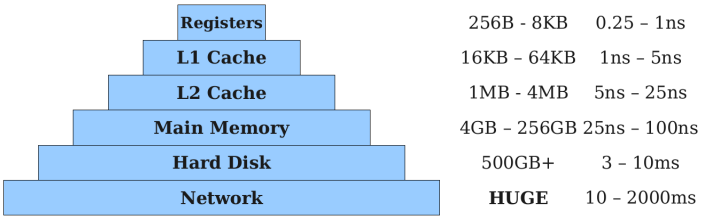
1.1 Izazovi generisanja koda

Generisanje koda

- U ovom trenutku, imamo optimizovan IR kod koji treba da se konvertuje u odgovarajući jezik ciljne mašine
- Ciljevi:
  - Izabrati odgovarajuće mašinske instrukcije za svaku IR instrukciju
  - Podeliti resurse mašine (registri, keš...)
  - Implementirati detalje niskog nivoa izvršnog okruženja
- Mašinski specifične optimizacije se često rade u ovom delu kompilatora

Memorija

- Postoji velika razlika u brzini i veličini memorije
- RAM je brza ali veoma skupa, hard disk je jeftin, ali veoma spor
- Osnovna ideja je dobiti najbolje iz svih svetova korišćenjem različitih vrsta memorije



Izazovi generisanja koda

- Skoro svi programski jezici imaju vrlo grub pogled na memorijsku hijerarhiju: sve promenljive žive u "memoriji" bez zalaženja u detalje na koju se tačno memoriju misli. Jedino se sa diskom i mrežom eksplicitno rukuje drugačije.
- Izazov generisanja koda je da se objekti postave na takav način koji maksimizuje prednosti memorijske hijerarhije
- Dodatno, to je potrebno uraditi potpuno automatski, tj bez pomoći programera

1.2 Registri

Registri

- Većina mašina ima skup registara koji su predviđeni za memorijske lokacije
  - kojima se može jako brzo pristupiti
  - nad kojima se mogu vršiti izračunavanja
  - i koji postoje u malim količinama
- Pametno korišćenje registara je kritičan korak za svaki kompajler
- Dobar alokator registara može da generiše kod koji je nekoliko redova veličine bolji od koda koji generiše loš alokator registara

Alokacija registara

- U TACu postoji neograničen broj promenljivih
- Na fizičkoj mašini postoji samo mali broj registara
  - X86 ima četiri registra opšte namene i još nekoliko specijalizovanih registara
  - MIPS ima 24 registra opšte namene i 8 specijalizovanih registara
- **Alokacija registara je proces dodele varijable registru i upravljanje transferom podataka iz i u registre.**

1.3 Izazovi alokacije registara

Izazovi alokacije registara

- Mali broj registara
  - Obično je neuporedivo manji broj registara nego broj promenljivih koji se koriste u IRu
  - Potrebno je naći način da se registri ponovo koriste uvek kada je to moguće
- Registri su komplikovani

- x86: Svaki registar se sastoji od više manjih registara i oni se ne mogu koristiti istovremeno
- x86: Neke instrukcije moraju svoje rezultate da upišu u neke konkretne registre pa to utiče da je upotreba registara ograničena i izborom instrukcija
- MIPS: Neki registri su rezervisani za neke specijalne namene (npr za operativni sistem)
- Većina arhitektura: Neki registri moraju da sačuvaju svoje vrednosti kroz pozive funkcija

Alokacija registara za MIPS

- Razmotrićemo alokaciju registra za MIPS
- Razmotrićemo tri algoritma
  - Naivni algoritam za alokaciju registara
  - Alokacija registara linearnim skeniranjem
  - Alokacija registara bojenjem grafova

2 Naivni algoritam za alokaciju registara

2.1 Ideja naivnog alokatora

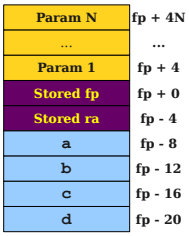
Naivni algoritam za alokaciju registara

- Ideja: Sačuvaj svaku vrednost u memoriji, učitaj je samo onda kada je potrebna
- Generiši kod na sledeći način
  - Generiši instrukciju učitavanja iz memorije u registar
  - Generiši kod da izvršiš izračunavanje nad registrima
  - Generiši instrukciju upisivanja rezultata nazad u memoriju

2.2 Primer naivnog alokatora

Our Register Allocator In Action

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




```
lw  $t0, -12(fp)
lw  $t1, -16(fp)
add $t2, $t0, $t1
sw  $t2,  -8(fp)

lw  $t0,  -8(fp)
sw  $t0, -20(fp)

lw  $t0,  -8(fp)
lw  $t1, -20(fp)
add $t2, $t0, $t1
sw  $t2, -16(fp)
```

2.3 Analiza naivnog alokatora

Analiza ovakvog alokatora

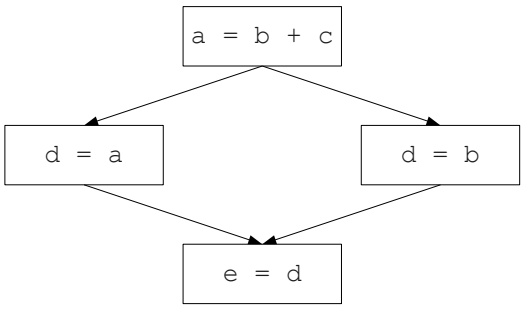
- Mane: jako je neefikasan
  - Ogroman broj učitavanja i upisivanja u memoriju
  - Gubi se vreme i prostor za vrednosti koje bi mogle da budu samo u registrima
  - Veoma lako postaje za red veličine ili dva reda veličine sporiji kod nego što je to potrebno
  - Potpuno neprihvatljiv pristup za kompajlere
- Prednosti: jednostavnost
  - Može da prevede svaki deo IR koda direktno u asembler u jednom prolazu
  - Nikada ne mora da brine da će ostati bez dovoljno registara
  - Ne mora da brine ni o pozivima funkcija ili o registrima specijalne namene
  - Dobar je samo ako nam je potreban prototip kompajlera koji radi

3 Linearno skeniranje

Bolja ideja

- Cilj: Pokušaj da držiš u registrima koliko god promenljivih možeš
- Ta ideja bi trebala da smanji broj čitanja/pisanja u memoriju i uopšte ukupno korišćenje memorije
- Da bi se to ostvarilo, potrebno je da se odgovori na naredna dva pitanja:
  - U koje registre stavljamo promenljive?
  - Šta se dešava kada nemamo dovoljno registara?

Register Consistency



3.1 Konzistentnost registara

Konzistentnost registara

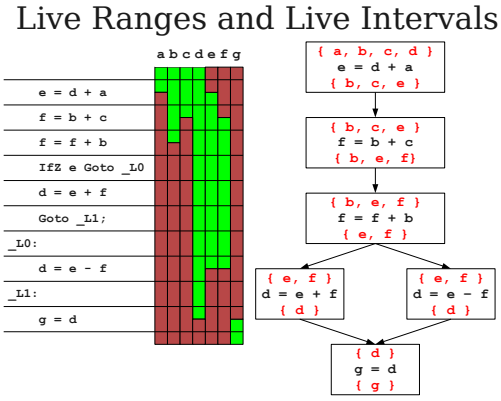
- **U svakoj tački programa svaka varijabla mora da bude na istoj lokaciji**
  - Ovo ne znači da svaka varijabla mora da bude na istoj lokaciji sve vreme, ali bez obzira kojom putanjom smo došli, očekujemo da nam je npr varijabla d u poslednjem bloku prethodne slike u istom registru
- **U svakoj tački programa, svaki registar sadrži najviše jednu živu promenljivu**
  - Ovo znači da možemo da dodelimo više varijabli istom registru ako nikada dve od njih neće biti čitane zajedno (neće biti istovremeno žive)

3.2 Živi opsezi i živi intervali

Živi opsezi (live range) i živi intervali (live interval)

- Promenljiva je živa u nekoj tački u programu ako se njena vrednost čita pre nego što se u nju ponovo nešto upiše
- Živost promenljivih se može pronaći korišćenjem globalne analize živosti
- **Živi opseg promenljive je skup tačaka u programu u kojim je promenljiva živa**

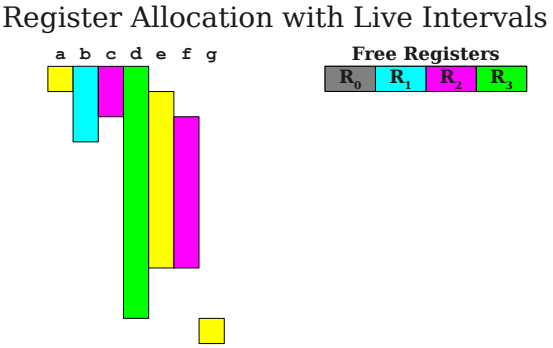
- **Živ interval promenljive je najmanji interval IR koda koji sadrži sve žive opsege promenljive**
- Živ interval je osobina IR koda, ne CFGa, manje je precizna od živih opsega, ali je lakše raditi sa njim

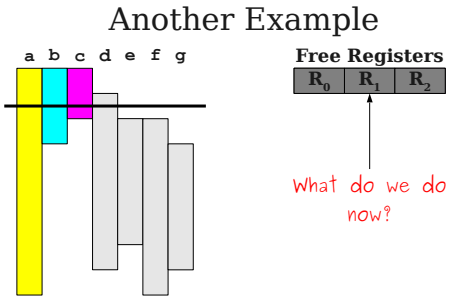


3.3 Alokacija registara korišćenjem živih intervala

Alokacija registara korišćenjem živih intervala

- Ako su dati živi intervali promenljivih u programu, možemo alocirati registre koristeći jednostavan pohlepni algoritam
- Ideja: prati koji registri su slobodni u svakoj tački
- Kada živi interval počne, daj promenljivoj slobodan registar
- Kada se živi interval završi, oslobodi odgovarajući registar
- Problem: šta ako nemamo dovoljno registara...

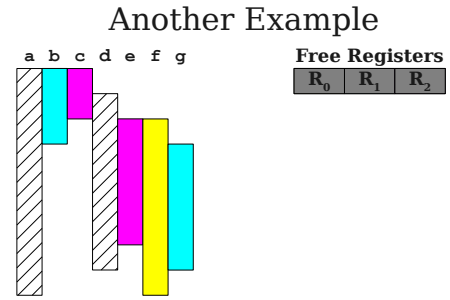




3.4 Register spilling

Razlivanje (prosipanje) registara (engl. register spilling)

- Ako za promenljivu v ne postoji slobodan registar, onda je potrebno njenu vrednost *prosuti*
- Kada je promenljiva prosuta, ona se čuva u memoriji umesto u registru
- Kada nam treba registar za promenljivu koja je bila prosuta potrebno je:
  - da iselimo neki postojeći registar u memoriju
  - da u taj registar učitamo vrednost prosute promenljive
  - kada završimo, potrebno je da sadržaj registra vratimo u memoriju i da u taj registar vratimo originalnu vrednost koja je tu bila
- Razlivanje je sporo, ali ponekad neophodno



3.5 Analiza algoritma

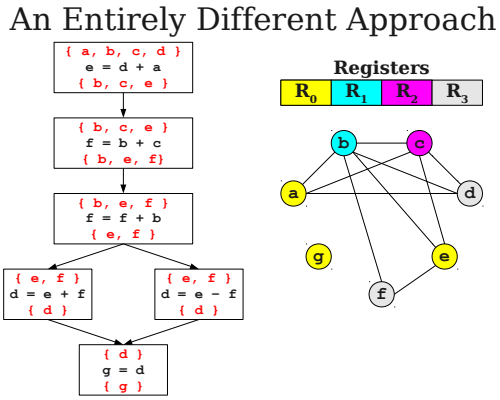
Analiza algoritma

- Prikazan algoritam je alokacija registara linearnim skeniranjem
- Prednosti
  - Veoma je efikasan, nakon računanja intervala živosti, izvršava se u linearnom vremenu
  - Daje dosta kvalitetan kod
  - Alokacija radi u jednom prolazu
  - Često se koristi za JIT kompajlere
- Mane
  - Neprecizan je zbog toga što koristi intervale, a ne opsege
  - Postoje bolje tehnike
- Algoritam se može poboljšati korišćenjem opsega živosti umesto intervala živosti

Skica dokaza ispravnosti

- Ni jedan registar ne sadrži dve žive promenljive u istom trenutku:
  - Živi intervali su konzervativna aproksimacija živih opsega
  - Nikada dve promenljive sa preklapajućim živim intervalima nisu smeštene u isti registar
- U svakoj tački programa svaka promenljiva je na istoj lokaciji jer se svakoj promenljivoj dodeljuje jedinstvena lokacija

4 Bojenje grafova



Graf međusobne zavisnosti registara

- Graf međusobne zavisnosti registara (engl. *Register Interference Graph (RIG))* je neusmeren graf gde
  - Svaki čvor je jedna promenljiva
  - Postoji ivica između dve promenljive koje su žive istovremeno u nekoj tački programa
  - Alokacija registara se svodi na dodelu svakoj promenljivoj različitog registra u odnosu na njegove susede
- Ovde postoji jedan problem...

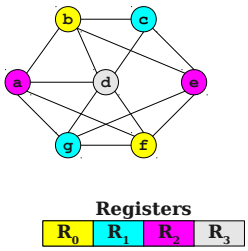
Algoritam bojenja grafova

- Opisani problem je ekvivalentan problemu bojenja grafova, koji je NP-težak ako ima bar tri registra
- Ne postoji algoritam u polinomnom vremenu koji rešava ovaj problem
- Treba da se zadovoljimo sa nekakvom heuristikom koja daje dovoljno dobre rezultate za probleme koji se javljaju u praksi

Čajtinov (Chaitin) algoritam

- Intuicija:
  - Pretpostavimo da pokušavamo da obojimo sa k boja graf. Pronađi čvor sa manje od k ivica
  - Ako obrišemo ovaj čvor iz grafa i obojimo ono što preostane, možemo da nađemo bojenje i za ovaj čvor kada ga dodamo nazad
  - Razlog: sa manje od k suseda, neka boja je sigurno neiskorišćena
- Algoritam
  - Pronađi čvor sa manje od k izlaznih ivica (manje od k suseda)
  - Skloni ga iz grafa
  - Rekurzivno oboj ostatak grafa
  - Vrati čvor nazad
  - Dodeli mu validnu boju

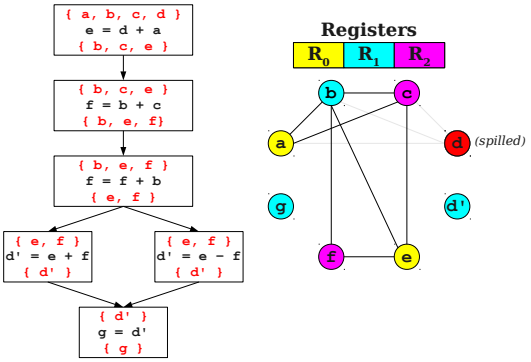
Chaitin's Algorithm



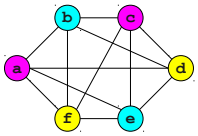
Problem

- Šta se dešava ako ne možemo da pronađemo čvor sa manje od k suseda?
- Izaberi i ukloni proizvoljan čvor, obeleži ga sa "problematičan" (koristi heuristiku da izabereš taj čvor)
- Kada vraćaš taj čvor, možda bude moguće da mu se dodeli ispravna boja.
- U suprotnom, taj čvor će biti prosut u memoriju

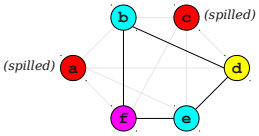
A Smarter Algorithm



Another Example



Another Example



Čajtinov algoritam

- Prednosti:
  - Za mnoge cfg-ove, pronalazi odlične dodele promenljivih registrima
  - Pošto se promenljive razlikuju po korišćenju, proizvede se precizan graf međusobne zavisnosti registara
  - Često se koristi u kompajlerima, npr u GCCu
- Mane:
  - Osnovni pristup se zasniva na NP teškom problemu bojenja grafova
  - Heuristika može da dovede do patološke dodele najgoreg slučaja

Skica dokaza korektnosti

- Nikada dve promenljive koje su žive u istoj tački nisu dodeljene istom registru — to je obezbeđeno kroz bojenje grafova
- U svakoj tački programa promenljiva je uvek na odgovarajućoj lokaciji — ovo se ostvaruje automatski ako se svakoj varijabli dodeli različiti registar, ali zahteva neke dodatne trikove u dokazu ako to nije tako

Poboljšanja algoritma

- Izaberi promenljivu koja će biti prosuta pametno: koristi heuristike (najmanje korišćena promenljiva, najveće poboljšanje ...) da odrediš šta ćeš prosuti
- Rukuj prosipanjem inteligentno: kada odlučiš da prospeš neku promenljivu, ponovo sračunaj RIG zasnovan na ovom prosipanju i koristi novo bojenje da pronađeš registar

Sumiranje algoritama alokacije

- Kritični korak za dobijanje optimizovanog koda
- Algoritam alokacije linearnim skaliranjem koristi žive intervale da pohlepnim algoritmom dodeli promenljive registrima. Koristi se često u JIT kompajlerima zbog efikasnosti
- Čajtinov algoritam koristi graf međusobne zavisnosti registara zasnovan na živim opsezima i bojenje grafova za dodelu registara. Koristi se kao osnovna tehnika u GCCu

5 Sumiranje algoritama i literatura

Sumiranje algoritama alokacije

- Kritični korak za dobijanje optimizovanog koda
- Algoritam alokacije linearnim skaliranjem koristi žive intervale da pohlepnim algoritmom dodeli promenljive registrima. Koristi se često u JIT kompajlerima zbog efikasnosti
- Čajtinov algoritam koristi graf međusobne zavisnosti registara zasnovan na živim opsezima i bojenje grafova za dodelu registara. Koristi se kao osnovna tehnika u GCCu

Literatura

- (The Dragon Book) Compilers: Principles, Techniques, and Tools Alfred V. Aho,Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
- Kompajleri Stanford https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/