

MEM2REG

SEPTEMBAR 2025.

Zeljko Milovanovic
Luka Arambasic
Milica Sopalovic

SADRZAJ

01 Uvod

02 Motivacija za promenu

03 Zasto bas mem2reg?

04 Motivacija za promenu

05 simple mem2reg

06 Kljucne funkcije

07 Testovi

08 Zakljucak

UVOD

Početna tačka u LLVM-u

Kada se program prevede u LLVM IR na najnižem nivou optimizacije, sve lokalne promenljive se čuvaju u memoriji preko:

- alloca (alocira prostor na steku),
- store (upisuje vrednost u memoriju),
- load (čita vrednost iz memorije).

Problem sa ovim pristupom

- Takav kod je spor i nepotrebno komplikovan.
- Svaka promenljiva je kao "kutija u memoriji" u koju stalno pišemo i čitamo, iako bi bilo jednostavnije držati vrednosti direktno u registrima.
- Analize kompjajlera (npr. propagacija konstanti, eliminacija mrtvog koda) teško rade kad se sve skriva iza load/store

MOTIVACIJA ZA PROMENU

- Umesto da se stalno piše u memoriju, promene se prenose kroz SSA registre.
- Svaka varijabla ima jednu definiciju u SSA formi – lakše praćenje zavisnosti.
- Optimizacije mogu jasno da vide: “ovde se vrednost dodeljuje, ovde se koristi”.

Šta je SSA (Single Static Assignment)

- Forma IR-a u kojoj svaka promenljiva dobija vrednost tačno jednom.
- Kada se granaju programi (if/else), koristi se φ -čvor da spoji vrednosti iz različitih putanja.
- Na primer: ako je u then grani $x=1$, a u else grani $x=2$, na spajanju dobijamo $x = \varphi(1,2)$.

ZAŠTO BAŠ MEM2REG PAS

- To je najpoznatiji i najčešći pas u LLVM-u – praktično svi kompjajleri ga koriste.
- On je osnova za skoro sve druge optimizacije.
- Kada jednom pretvorimo kod u SSA, sledeći pass (npr. dead code elimination, constant folding) postaje mnogo efikasniji.

SIMPLE MEM2REG

Osnovna ideja projekta

Implementirali smo sopstvenu verziju mem2reg passa u LLVM-u pod imenom MatfSimpleMem2Reg.

Cilj je bio da pokažemo osnovne slučajeve promocije iz memorije u SSA registre.

Koraci koje radi naš pas:

1. Skuplja sve alloca instrukcije u funkciji.
2. Analizira njihove korisnike (store/load) i proverava da li su čisti kandidati (nema volatile, nema čudnih pokazivača).
3. Primeni jedno od pravila:
 - Ako nema load - briše sve store i samu alloca (mrtva varijabla).
 - Ako nema store - svaki load se zamenjuje vrednošću undef.
 - Ako postoji samo jedan store - svi load-ovi se zamenjuju tom vrednošću, store i alloca se brišu.
 - Ako postoje dva store-a u if/else - naš pas (uz flag -matf-phi) ubacuje φ -čvor u merge bloku.

SIMPLE MEM2REG

- **Ključne komponente implementacije:**
 - collectUses – prolazi kroz korisnike i nalazi sve store i load instrukcije za jednu alloca.
 - eraseLifetimesFor – briše nepotrebne lifetime_start i lifetime_end intrinzične pozive.
 - findTwoPredMerge – pronalazi blok sa tačno dva prethodnika (tipični if/else obrazac) i tu umeće φ-čvor.
 - DominatorTree – koristi se da proveri da li jedan store dominira sve load-ove (sigurnost transformacije).
- **Specifičnosti našeg projekta:**
 - Dodali smo mogućnost verbose ispisa za lakše testiranje (-matf-verbose).
 - Testirali smo na malim primerima (ex1.ll, ex2.ll) i pratili promene before/after.

KLJUČNE FUNKCIJE U PROJEKTU

- **collectUses**
 - Glavna rutina koja prolazi kroz sve korisnike jedne alloca instrukcije.
 - Razdvaja ih u dve liste: store instrukcije i load instrukcije.
 - Proverava da li ima nešto što nije dozvoljeno (npr. volatile operacije ili upotreba adrese).
 - Ako sve prođe proveru - ta promenljiva može biti kandidat za promociju u SSA.
- **eraseLifetimesFor**
 - Briše posebne pozive (llvm.lifetime.start i llvm.lifetime.end).
 - Ti pozivi služe samo da pomognu optimizacijama, ali posle mem2reg postaju nepotrebni.
 - Uklanjanjem njih čistimo kod i uklanjamo "šum".

KLJUČNE FUNKCIJE U PROJEKTU

- **findTwoPredMerge**
 - Traži blok koda (merge blok) koji ima tačno dva prethodnika.
 - Tipična situacija: if/else struktura gde obe grane vode u isti kraj.
 - Ako takav blok postoji, naš pas umeće φ -čvor baš na njegovom početku.
 - To omogućava da se vrednosti iz then i else spoje u jednu SSA promenljivu.
- **DominatorTree korišćenje**
 - LLVM-ov alat za proveru dominacije između blokova.
 - Nama treba da proverimo: da li store dominira sve load-ove?
 - Ako dominira, bezbedno je zameniti sve load-ove direktno vrednošću iz tog store-a.
 - Bez ove provere moglo bi da dođe do greške u semantici programa.

TEST 1

```
1  int f(int a, int b) {  
2      int x;  
3      x = a;  
4      int r = x + b;  
5      r += x;  
6      if (b > 0) {  
7          r += x;  
8      }  
9      return r;  
10 }
```

TEST 1 - BEFORE

```
8    entry:
9      %a.addr = alloca i32, align 4
10     %b.addr = alloca i32, align 4
11     %x = alloca i32, align 4
12     %r = alloca i32, align 4
13     store i32 %a, i32* %a.addr, align 4
14     store i32 %b, i32* %b.addr, align 4
15     %0 = load i32, i32* %a.addr, align 4
16     store i32 %0, i32* %x, align 4
17     %1 = load i32, i32* %x, align 4
18     %2 = load i32, i32* %b.addr, align 4
19     %add = add nsw i32 %1, %2
20     store i32 %add, i32* %r, align 4
21     %3 = load i32, i32* %x, align 4
22     %4 = load i32, i32* %r, align 4
23     %add1 = add nsw i32 %4, %3
24     store i32 %add1, i32* %r, align 4
25     %5 = load i32, i32* %b.addr, align 4
26     %cmp = icmp sgt i32 %5, 0
27     br i1 %cmp, label %if.then, label %if.end
28
29 if.then:                                ; preds = %entry
30   %6 = load i32, i32* %x, align 4
31   %7 = load i32, i32* %r, align 4
32   %add2 = add nsw i32 %7, %6
33   store i32 %add2, i32* %r, align 4
34   br label %if.end
35
36 if.end:                                  ; preds = %if.then, %entry
37   %8 = load i32, i32* %r, align 4
38   ret i32 %8
39 }
```

TEST 1 - AFTER

```
8   entry:  
9     %r = alloca i32, align 4  
10    %add = add nsw i32 %a, %b  
11    store i32 %add, ptr %r, align 4  
12    %o = load i32, ptr %r, align 4  
13    %add1 = add nsw i32 %o, %a  
14    store i32 %add1, ptr %r, align 4  
15    %cmp = icmp sgt i32 %b, 0  
16    br i1 %cmp, label %if.then, label %if.end  
17  
18  if.then:                                ; preds = %entry  
19    %1 = load i32, ptr %r, align 4  
20    %add2 = add nsw i32 %1, %a  
21    store i32 %add2, ptr %r, align 4  
22    br label %if.end  
23  
24  if.end:                                ; preds = %if.then, %entry  
25    %2 = load i32, ptr %r, align 4  
26    ret i32 %2  
27 }
```

TEST 2

```
1  int sel(int c, int a, int b) {  
2      int x;  
3      if (c) {  
4          x = a;  
5      } else {  
6          x = b;  
7      }  
8  
9      return x + 1;  
10 }
```

TEST 2 - BEFORE

```
8    entry:
9        %c.addr = alloca i32, align 4
10       %a.addr = alloca i32, align 4
11       %b.addr = alloca i32, align 4
12       %x = alloca i32, align 4
13       store i32 %c, i32* %c.addr, align 4
14       store i32 %a, i32* %a.addr, align 4
15       store i32 %b, i32* %b.addr, align 4
16       %0 = load i32, i32* %c.addr, align 4
17       %tobool = icmp ne i32 %0, 0
18       br i1 %tobool, label %if.then, label %if.else
19
20   if.then:
21       %1 = load i32, i32* %a.addr, align 4
22       store i32 %1, i32* %x, align 4
23       br label %if.end
24
25   if.else:
26       %2 = load i32, i32* %b.addr, align 4
27       store i32 %2, i32* %x, align 4
28       br label %if.end
29
30   if.end:
31       %3 = load i32, i32* %x, align 4
32       %add = add nsw i32 %3, 1
33       ret i32 %add
34 }
```

TEST 2 - AFTER

```
8    entry:
9        %tobool = icmp ne i32 %c, 0
10       br i1 %tobool, label %if.then, label %if.else
11
12      if.then:                                ; preds = %entry
13          br label %if.end
14
15      if.else:                                ; preds = %entry
16          br label %if.end
17
18      if.end:                                 ; preds = %if.else, %if.then
19          %x.phi = phi i32 [ %b, %if.else ], [ %a, %if.then ]
20          %add = add nsw i32 %x.phi, 1
21          ret i32 %add
22      }
```

TEST 3

```
1 int mix(int a) {  
2     int dead;  
3     dead = a;  
4  
5     int u; // u is never initialized  
6     int r = 0;  
7     r += u;  
8     return r;  
9 }
```

TEST 3 - BEFORE

```
8    entry:  
9        %a.addr = alloca i32, align 4  
10       %dead = alloca i32, align 4  
11       %u = alloca i32, align 4  
12       %r = alloca i32, align 4  
13       store i32 %a, i32* %a.addr, align 4  
14       %0 = load i32, i32* %a.addr, align 4  
15       store i32 %0, i32* %dead, align 4  
16       store i32 0, i32* %r, align 4  
17       %1 = load i32, i32* %u, align 4  
18       %2 = load i32, i32* %r, align 4  
19       %add = add nsw i32 %2, %1  
20       store i32 %add, i32* %r, align 4  
21       %3 = load i32, i32* %r, align 4  
22       ret i32 %3  
23   }
```

TEST 3 - AFTER

```
8     entry:  
9         %r = alloca i32, align 4  
10        store i32 0, ptr %r, align 4  
11        %0 = load i32, ptr %r, align 4  
12        %add = add nsw i32 %0, undef  
13        store i32 %add, ptr %r, align 4  
14        %1 = load i32, ptr %r, align 4  
15        ret i32 %1  
16    }  
17
```

TEST 4

```
1 int casted(int a) {  
2     int x;  
3     void *p = &x;  
4     int *q = (int*)p;  
5  
6     *q = a;  
7     return *q;  
8 }
```

TEST 4 - BEFORE

```
8    entry:  
9      %a.addr = alloca i32, align 4  
10     %x = alloca i32, align 4  
11     %p = alloca i8*, align 8  
12     %q = alloca i32*, align 8  
13     store i32 %a, i32* %a.addr, align 4  
14     %0 = bitcast i32* %x to i8*  
15     store i8* %0, i8** %p, align 8  
16     %1 = load i8*, i8** %p, align 8  
17     %2 = bitcast i8* %1 to i32*  
18     store i32* %2, i32** %q, align 8  
19     %3 = load i32, i32* %a.addr, align 4  
20     %4 = load i32*, i32** %q, align 8  
21     store i32 %3, i32* %4, align 4  
22     %5 = load i32*, i32** %q, align 8  
23     %6 = load i32, i32* %5, align 4  
24     ret i32 %6  
25 }
```

TEST 4 - AFTER

```
7      define dso_local i32 @casted(i32 noundef %a) #0 {
8          entry:
9              %x = alloca i32, align 4
10             %0 = bitcast ptr %x to ptr
11             %1 = bitcast ptr %0 to ptr
12             store i32 %a, ptr %1, align 4
13             %2 = load i32, ptr %1, align 4
14             ret i32 %2
15 }
```