Konstrukcija kompilatora

SROA i MergeFunc optimizacije

Jovana Brkljač Marina Vračarić

Matematički fakultet
Univerzitet u Beogradu

Beograd, 2025.

Pregled

SROA – Uvod

- Cilj optimizacije:
  - rastavljanje složenih tipova (struktura) u pojedinačne skalare
  - olakšavanje daljih optimizacija
- SROA je poznata LLVM transformacija koja:
  - smanjuje pristupe memoriji
  - povećava iskorišćenost SSA forme
  - ubrzava izvršavanje koda
- U projektu je implementirana uprošćena verzija ove optimizacije za osnovne slučajeve.

SROA – Ideja i motivacija

- U LLVM IR-u, strukture se obično alociraju u memoriji pomoću `alloca`.
- Svaki pristup polju strukture ide kroz `getelementptr` (GEP), `load` i `store`.
- Takav pristup:
  - otežava optimizacije (npr. dead code elimination, constant propagation)
  - povećava broj instrukcija i memorijskih operacija
- **Ideja:** svako polje strukture izdvojiti kao poseban skalarni `alloca`.
- Dobijamo nezavisne promenljive koje LLVM može automatski promovisati u SSA registre.

SROA – Implementacija

- Glavni koraci:
  1. Pronalaženje `alloca` instrukcija za strukture.
  2. Kreiranje novih `alloca` za svako polje.
  3. Zamena svih `GEP`, `load`, `store` instrukcija novim skalarnim verzijama.
  4. Brisanje starih instrukcija i pokretanje `mem2reg` passa.

LLVM IR pre optimizacije

Originalni IR:

```
%p = alloca %struct.Pair, align 4
%p_x = getelementptr %struct.Pair, %struct.Pair* %p, i32 0, i32 0
store i32 %x, i32* %p_x
%p_y = getelementptr %struct.Pair, %struct.Pair* %p, i32 0, i32 1
store i32 %y, i32* %p_y
%tmp1 = load i32, i32* %p_x
%tmp2 = load i32, i32* %p_y
%sum = add i32 %tmp1, %tmp2
ret i32 %sum
```

Karakteristike:

- Jedan `alloca` za celu strukturu `Pair`.
- Pristup poljima putem `getelementptr`.
- Višestruki `load/store` pristupi memoriji.

LLVM IR posle MySROA optimizacije

Optimizovani IR:

```
%p_x = alloca i32, align 4
%p_y = alloca i32, align 4
store i32 %x, i32* %p_x
store i32 %y, i32* %p_y
%tmp1 = load i32, i32* %p_x
%tmp2 = load i32, i32* %p_y
%sum = add i32 %tmp1, %tmp2
ret i32 %sum
```

Razlike u odnosu na original:

- Uklonjen `getelementptr`.
- Svako polje (`x`, `y`) izdvojeno kao poseban `alloca`.
- Struktura više ne postoji — koristi se rad nad skalarima.

MergeFunc – Uvod

- Kompajleri često generišu više funkcija koje su **strukturno identične** ili imaju istu implementaciju zbog procesa linkovanja, šablona i makroa.
- Takve funkcije bespotrebno zauzimaju memoriju i povećavaju veličinu binarnog fajla.
- **MergeFunc** je LLVM optimizacioni pass koji:
  - pronalazi funkcije sa istim telom,
  - spaja ih u jednu zajedničku implementaciju,
  - zamenjuje sve njihove pozive jednom funkcijom.
- Na taj način se smanjuje duplirani kod i optimizuje binarna veličina modula.

MergeFunc – Ideja i motivacija

- Dve funkcije su **ekvivalentne** ako su iste po svojoj strukturi i ponašanju.
- LLVM IR omogućava da se svaka funkcija posmatra kao niz:
  - osnovnih blokova (`BasicBlock`),
  - instrukcija unutar blokova i njihovih operanada.
- **Ideja:** ako dve funkcije imaju isti broj blokova, isti redosled i tipove instrukcija, one su kandidati za spajanje.
- MergeFunc zatim proverava detaljno podudaranje:
  - tip povratne vrednosti i argumenata,
  - redosled i strukturu IR instrukcija,
  - vrednosti konstanti i operanada (uz podršku za komutativnost).
- Ako se sve poklapa $\rightarrow$ funkcije se **spajaju u jednu**.

MergeFunc – Implementacija

- Naivno rešenje: porediti svaku funkciju sa svakom — složenost $O(n^2 \cdot k)$

```
}
bool runOnModule(Module &M) override {
  for (auto F1 = M.begin(); F1 != M.end(); F1++){
        for (auto F2 = std::next(F1); F2 != M.end();){
            if(sameFunctionBody(&*F1,&*F2)){
                (&*F2)->replaceAllUsesWith(&*F1);
                auto toErase = &*F2;
                ++F2;
                toErase->eraseFromParent();
            }
            else{
                ++F2;
            }
        }
  }
  return false;
}
```

Vremenska složenost: $O(n^2 \cdot k)$

Strukturalno heširanje funkcija

- Naivno poredjenje svake funkcije sa svakom drugom ima složenost: $\mathbf{O(n^2 \cdot k)}$, gde je n broj funkcija, a k prosečan broj instrukcija.
- Takav pristup je neefikasan za veće module.

- **Rešenje: strukturalno heširanje.**
  - Svakoj funkciji se dodeljuje heš vrednost zasnovana na njenoj strukturi.
  - Porede se samo funkcije sa istim hešom — smanjuje se broj poredjenja.
  - Unutar svake grupe ("bucket") proverava se potpuna strukturna jednakost.

- **Složenost:**
  - Izračunavanje heša: $O(n \cdot k)$
  - Grupisanje funkcija po hešu: $O(n)$
  - Provere unutar bucket-a: prosečno $O(n \cdot k)$
- Ukupno: $\mathbf{O(n \cdot k)}$ — značajno ubrzanje u odnosu na naivno rešenje.

MergeFunc optimizacija – pre i posle

Pre optimizacije:

```
define i32 @add1(i32 %x) {
  %1 = alloca i32
  store i32 %x, ptr %1
  %2 = load i32, ptr %1
  %3 = add nsw i32 %2, 1
  ret i32 %3
}

define i32 @add2(i32 %x) {
  %1 = alloca i32
  store i32 %x, ptr %1
  %2 = load i32, ptr %1
  %3 = add nsw i32 %2, 1
  ret i32 %3
}
```

Posle optimizacije:

```
define i32 @add1(i32 %x) {
  %1 = alloca i32
  store i32 %x, ptr %1
  %2 = load i32, ptr %1
  %3 = add nsw i32 %2, 1
  ret i32 %3
}

; add2 je uklonjena { koristi add1
```

MergeFunc – prepoznata komutativnost

Pre optimizacije:

```
define i32 @add3(i32 %x) {
  %1 = alloca i32
  store i32 %x, ptr %1
  %2 = load i32, ptr %1
  %3 = add nsw i32 1, %2
  store i32 %3, ptr %1
  %4 = load i32, ptr %1
  ret i32 %4
}

define i32 @add4(i32 %x) {
  %1 = alloca i32
  store i32 %x, ptr %1
  %2 = load i32, ptr %1
  %3 = add nsw i32 %2, 1
  store i32 %3, ptr %1
  %4 = load i32, ptr %1
  ret i32 %4
}
```

Posle optimizacije:

```
define i32 @add3(i32 %x) {
  %1 = alloca i32
  store i32 %x, ptr %1
  %2 = load i32, ptr %1
  %3 = add nsw i32 %2, 1
  store i32 %3, ptr %1
  %4 = load i32, ptr %1
  ret i32 %4
}

; add4 je uklonjena ;  spojena sa add3
; komutativnost: 1 + x == x + 1
```

Zaključak

- **SROA** i **MergeFunc** predstavljaju dve strane iste ideje – kako program učiniti jednostavnijim.
- Jedna optimizacija rastavlja složeno u jednostavno, druga spaja ponovljeno u jedinstveno.
- Zajedno, doprinose jasnijem, efikasnijem LLVM kodu.