Konstrukcija Kompilatora

– Loop Strength Reduction Pass –

Marko Veljović Mateja Janić

Matematički fakultet
Univerzitet u Beogradu

Beograd, 2025.

Literatura

- Zasnovano na snimcima časova iz predmeta Konstrukcija Kompilatora na I smeru Matematičkog fakulteta, Univerziteta u Beogradu, kao i na zvaničnoj LLVM dokumentaciji koja se može pronaći na sledećem linku, ali i na dokumentacijama raznih svetskih univerziteta na ovu temu. Jedna od njih je dostupna ovde.

Pregled

Cilj

- Šta želimo da postignemo?
  - Želeli bismo da, gde god je to moguće, zamenimo neke komplikovane operacije jednostavnijima. Konkretno, u našem primeru, želimo da prepoznamo odredjene tipove množenja i zamenimo ih sabiranjem.
- Zbog čega?
  - Ova vrsta optimizacije može značajno poboljšati vreme izvršavanja programa, pogotovo u velikim projektima.

Ideja

- Traženje specifičnih promenljivih
  - Želimo da detektujemo one promenljive koje se kroz petlju menjaju sa konstantnim korakom, a učestvuju u množenjima u petlji - najjednostavniji primer ovakve promenljive je sam brojač petlje, čiji slučaj je našom implementacijom i pokriven
- ***Ideja***
  - Ukoliko pretpostavimo da naša nadjena promenljiva figuriše u petlji u izrazu oblika $a \cdot i + b$, kao i da se njena vrednost svakom iteracijom petlje menja za neki pomeraj k, onda jednostavno možemo sva njena pojavljivanja zameniti novom promenljivom, idx, inicijalizovanom na $a \cdot i_{init} + b$, čiju ćemo vrednost svakom iteracijom petlje menjati za pomeraj $a \cdot k$.

Implementacija

- Najpre, pogledajmo šta se dešava u IR-u kada naidjemo na ovakvu instrukciju:

```
%12 = load i32, ptr %4, align 4
%13 = load i32, ptr %6, align 4
%14 = mul nsw i32 %12, %13
%15 = load i32, ptr %5, align 4
%16 = add nsw i32 %14, %15
```

Implementacija

- Možemo primetiti da se instrukcija razlaže na 5 komponenti. Najpre učitavamo naše *i* i *a*. Zatim izvršavamo množenje i smeštamo rezultat u privremenu promenljivu, a nakon toga se isti postupak ponavlja za *b* i operaciju sabiranja.
- Dakle, možemo zaključiti da pronalazak instrukcije množenja, nakon kojeg se dešavaju učitavanje i sabiranje, može da bude mesto na kojem ćemo tražiti potencijalne kandidate za optimizaciju.

Rezultat

Preheader

```
%iv.init = load i32, ptr %6, align 4
%idx.init = mul i32 %iv.init, %a.init
%idx.offset = add i32 %idx.init, %b.init
store i32 %idx.offset, ptr %Idx, align 4
```

Body

```
%idx.curr = load i32, ptr %Idx, align 4
store i32 %idx.curr, ptr %2, align 4
```

Latch

```
%idx.old = load i32, ptr %Idx, align 4
%idx.next = add i32 %idx.old, %a.init
store i32 %idx.next, ptr %Idx, align 4
```

Primena na nizove

- Sada kada razumemo samu optimizaciju, treba da predstavimo njenu mnogo važniju primenu. Ona se odnosi na nizove, konkretno, na indeksiranje u nizovima, sa čime se u praksi neopisivo češće srećemo nego sa samim množenjem.
- Šta se dešava kada napišemo `array[i]`?
  - Najpre učitavamo pokazivač na početak niza, a zatim i puta pomeramo pokazivač unapred, ne bi li smo stigli do željenog elementa. U pozadini, traženje željenog elementa u nizu iziskuje računanje pomeraja **množenjem** vrednosti promenljive i sa veličinom tipa elemenata u nizu $\rightarrow$ pogodno za optimizaciju.

Ideja

- Slično kao i u zameni običnog množenja sabiranjem, ovde želimo da zapamtimo pokazivač na početak niza, a da ga svakom iteracijom petlje pomeramo unapred za onaj pomeraj za koji se menja i naš brojač. Medjutim, postavlja se pitanje: Kako da pronadjemo instrukciju koju treba optimizovati?
- Ukoliko pogledamo naš IR (sledeći slajd), možemo primetiti da su se prilikom instrukcije `niz[i] = 4;` desile sledeće dve operacije: `getelementpointer` i `store`. Ovo je upravo ono što želimo da pronadjemo i ispitamo da li prilikom pristupanja elementu niza koristimo baš naš brojač. Ukoliko da, menjamo instrukciju na gore opisan način.

Ideja

Pre optimizacije

```
%13 = load i32, ptr %7, align 4
%14 = sext i32 %13 to i64
%15 = getelementptr inbounds [100 x i32], ptr %2, i64 0, i64 %14
store i32 4, ptr %15, align 4
```

- U kodu ispod možemo videti da smo sada potpuno izbacili operaciju množenja kod računanja pokazivača.

Rezultat

Preheader

```
%p.init = getelementptr [100 x i32], ptr %2, i32 0, i32 0
store ptr %p.init, ptr %p, align 8
```

Body

```
%p.cur = load ptr, ptr %p, align 8
store i32 4, ptr %p.cur, align 4
```

Latch

```
%p.old = load ptr, ptr %p, align 8
%p.next = getelementptr i32, ptr %p.old, i32 1
store ptr %p.next, ptr %p, align 8
```

Kombinacija

- Zamislimo sada još malo kompleksniju situaciju. Šta ako bismo u isto vreme imali i indeksiranje i linearnu kombinaciju po našem brojaču?
- Kada smo implementirali prethodna dva slučaja optimizacije, intuitivno je pomisliti da njih dve mogu i zajedno da se iskoriste, a ne samo nezavisno jedna od druge. Kako to uraditi?

Implementacija

- Zapravo, mi smo implementacijom prethodna dva slučaja skoro potpuno utabali put za primenu optimizacije i na ovaj trenutni kompleksni slučaj. Jedino što je preostalo da uradimo jeste da za našu promenljivu kojom pristupamo elementima niza proverimo da li je ona upravo oblika $a \cdot i + b$, a ukoliko jeste, prvo ćemo primeniti prvi deo optimizacije, a odmah nakon i drugi.

Rezultat

Preheader

```
%iv.init = load i32, ptr %7, align 4
%idx.init = mul i32 %iv.init, %a.init
%idx.offset = add i32 %idx.init, %b.init
store i32 %idx.offset, ptr %Idx, align 4
%p.init = getelementptr [100 x i32], ptr %2, i32 %idx.offset
store ptr %p.init, ptr %p, align 8
```

Body

```
%idx.curr = load i32, ptr %Idx, align 4
%p.cur = load ptr, ptr %p, align 8
store i32 4, ptr %p.cur, align 4
```

Latch

```
%idx.old = load i32, ptr %Idx, align 4
%idx.next = add i32 %idx.old, %a.init
store i32 %idx.next, ptr %Idx, align 4
%p.old = load ptr, ptr %p, align 8
%p.next = getelementptr i32, ptr %p.old, i32 %a.init
store ptr %p.next, ptr %p, align 8
```

Zaključak

- Iako je koncept Loop Strength Reduction (LSR) optimizacije jednostavan za razumevanje, veoma često se može implementirati i, kao što je već rečeno, značajno ubrzati vreme izvršavanja programa.
- Za sam kraj, želeli bismo da spomenemo (iako je očigledno) da je ova implementacija samo mali deo svega onoga što LSR ume da prepozna i ubrza. Zvanična implementacija poseduje napredne tehnike prepoznavanja koda koji se može optimizovati i baš zato ima više od 7000 linija koda!