



LLVM OPTIMIZACIJE

Viktor Šević 91/2022
Aleksa Vukadinović 103/2022
Maša Lazić 97/2022

Sadržaj



Loop Unrolling



Loop Invariant
Code Motion



Merge functions

Loop unrolling

Cilj ove optimizacije je smanjivanje broja iteracija petlje, tako što se telo petlje duplira više puta. Na ovaj način se smanjuje broj skokova i grananja, a povećava iskorišćenost procesorskog keša

```
#include <stdio.h>

int main() {
    int sum = 0;

    for (int i = 0; i < 4; i++) {
        sum += i;
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

```
#include <stdio.h>

int main() {
    int sum = 0;

    sum += 0;
    sum += 1;
    sum += 2;
    sum += 3;

    printf("Sum = %d\n", sum);
    return 0;
}
```

```
define i32 @main() {
entry:
    %sum = alloca i32, align 4
    store i32 0, i32* %sum, align 4
    %i = alloca i32, align 4
    store i32 0, i32* %i, align 4
    br label %loop

loop:
    %i_val = load i32, i32* %i, align 4
    %cmp = icmp slt i32 %i_val, 4
    br i1 %cmp, label %loop_body, label %loop_end

loop_body:
    %i_val2 = load i32, i32* %i, align 4
    %sum_val = load i32, i32* %sum, align 4
    %add = add i32 %sum_val, %i_val2
    store i32 %add, i32* %sum, align 4
    %i_next = add i32 %i_val2, 1
    store i32 %i_next, i32* %i, align 4
    br label %loop

loop_end:
    %sum_final = load i32, i32* %sum, align 4
    ; poziv printf
    ret i32 0
}
```

```
define i32 @main() {
entry:
    %sum = alloca i32, align 4
    store i32 0, i32* %sum, align 4

    %sum0 = load i32, i32* %sum, align 4
    %add0 = add i32 %sum0, 0
    store i32 %add0, i32* %sum, align 4

    %sum1 = load i32, i32* %sum, align 4
    %add1 = add i32 %sum1, 1
    store i32 %add1, i32* %sum, align 4

    %sum2 = load i32, i32* %sum, align 4
    %add2 = add i32 %sum2, 2
    store i32 %add2, i32* %sum, align 4

    %sum3 = load i32, i32* %sum, align 4
    %add3 = add i32 %sum3, 3
    store i32 %add3, i32* %sum, align 4

    ret i32 0
}
```

```
void MapVariables(Loop *L)
{
    Function *F = L->getHeader()->getParent();
    for (BasicBlock &BB : *F) {
        for (Instruction &I : BB) {
            if (isa<LoadInst>(&I)) {
                VariablesMap[&I] = I.getOperand(0);
            }
        }
    }
}
```



```
void findLoopCounterAndBound(Loop *L)
{
    for (Instruction &I : *L->getHeader()) {
        if (isa<ICmpInst>(&I)) {
            LoopCounter = VariablesMap[I.getOperand(0)];
            if (ConstantInt *ConstInt = dyn_cast<ConstantInt>(I.getOperand(1))) {
                isLoopBoundConst = true;
                BoundValue = ConstInt->getSExtValue();
            }
        }
    }
}
```

- Granica konstanta: full unrolling
 - Petlja se potpuno eliminiše
- Granica nije konstanta: partial unrolling sa fiksnim faktorom
 - Prave se dodatne kopije tela petlje samo nekoliko puta

Prednosti i mane

- Manje grananja, skokova i provera - brže izvršavanje
- Veća iskorišćenost CPU pipeline-a (mogućnost paralelnog izvršavanja instrukcija)
- Povećava veličinu koda, naročito kod full unrollinga
- Smanjuje keš lokalnost ako telo petlje postane preveliko
- Bez garancije za poboljšanje performansi (ako je petlja mala ili se izvršava malo puta)

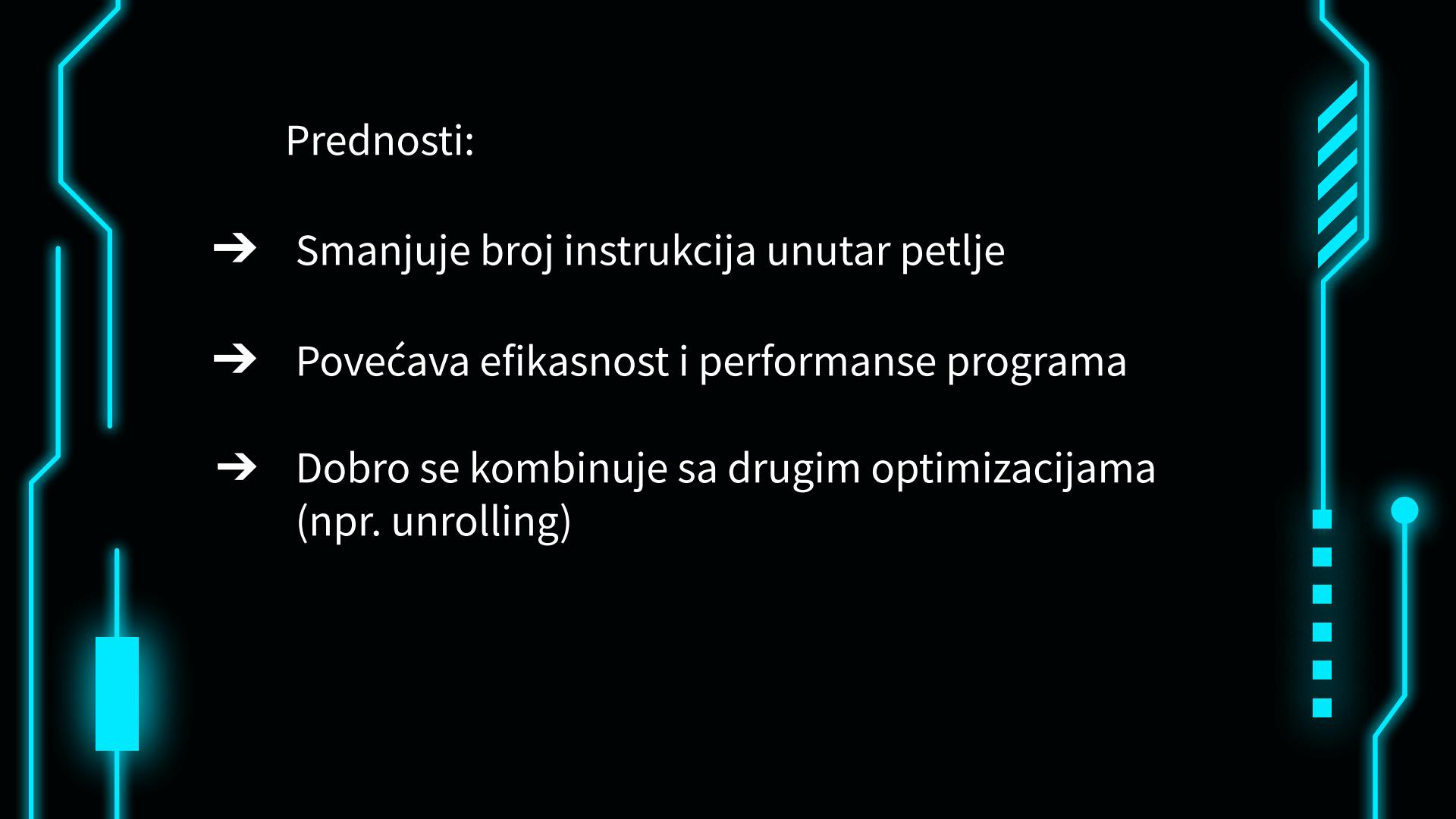


Loop Invariant Code Motion

LICM je optimizacija koja izdvaja izraze koji ne zavise od iteracija petlje i pomera ih izvan petlje (najčešće u preheader blok).

Ideja:

Ako se neka instrukcija u petlji uvek računa isto (nema zavisnosti od promenljivih koje se menjaju u petlji) nema potrebe da se ponavlja svaka iteracija.



Prednosti:

- Smanjuje broj instrukcija unutar petlje
- Povećava efikasnost i performanse programa
- Dobro se kombinuje sa drugim optimizacijama
(npr. unrolling)

```
void test(int *A, int n) {  
    for (int i = 0; i < 10; i++) {  
        int x = n * 2;  
        *A = x;  
    }  
}
```



```
void test(int *A, int n) {  
    int x = n * 2;  
    for (int i = 0; i < 10; i++) {  
        *A = x;  
    }  
}
```

```
define void @test(i32* %A, i32 %n) {
entry:
    br label %loop.preheader

loop.preheader:           ; Preheader for LICM
    br label %loop

loop:
    %i = phi i32 [0, %loop.preheader], [%i.next, %loop]
    %x = mul i32 %n, 2          ; loop-invariant, should be hoisted
    store i32 %x, i32* %A
    %i.next = add i32 %i, 1
    %cmp = icmp slt i32 %i.next, 10
    br i1 %cmp, label %loop, label %exit

exit:
    ret void
}
```

```
define void @test(i32* %A, i32 %n) {
entry:
    br label %loop.preheader

loop.preheader:
    %x = mul i32 %n, 2          ; <--- HOISTED from loop
    br label %loop

loop:
    %i = phi i32 [0, %loop.preheader], [%i.next, %loop]
    store i32 %x, i32* %A
    %i.next = add i32 %i, 1
    %cmp = icmp slt i32 %i.next, 10
    br il %cmp, label %loop, label %exit

exit:
    ret void
}
```

Implementacija

Prvi korak u runLICM funkciji je pronalaženje preheadera petlje. U tom bloku se kasnije smeštaju instrukcije koje su izvučene iz tela petlje

Zatim se prolazi kroz sve blokove u okviru petlje i svaka instrukcija se proverava pomoću `isLoopInvariant` i `isSafeToHoist`.

Ako su oba uslova zadovoljena, instrukcija se pomera pre terminadora preheader bloka, tj. Izvršava se samo jednom pre ulaska u petlju.

Ova implementacija koristi postojeće LLVM analize:
`DominatorTreeWrapperPass` za proveru dominacije,
`AAResultsWrapperPass` za proveru nezavisnosti pristupa memoriji

Merge functions

Glavna ideja ovog LLVM pasa je da automatski identificuje i spoji funkcije koje su strukturalno identične unutar jednog modula. U velikim projektima često se dešava da se iste funkcije pišu više puta ili generišu automatski sa malim razlikama, što povećava veličinu binarnog koda i otežava optimizaciju. Ovaj pass omogućava detekciju takvih funkcija i njihovo spajanje u jednu jedinstvenu funkciju, čime se smanjuje redundancija.

Implementacija

Pass se implementira kao ModulePass i koristi jednostavan sistem hashiranja funkcija po njihovoj strukturi. Svaka funkcija se prolazi po BasicBlock-ovima i instrukcijama, a na osnovu operacija i broja operanada generiše se hash. Funkcije sa istim hash-om se generišu u "bucket-e", a zatim se porede detaljnije kako bi se utvrdilo da li su zaista identične.

Prolazak kroz sve funkcije u modulu

Heširanje funkcija na osnovu instrukcija

Grupisanje po heš vrednostima

Poredjenje unutar svakog bucket-a

jesu iste

nisu iste

- Zadrži jednu
- Zameni pojavljivanja jedne sa drugom
- Dodaj suvisnu za brisanje

```
#include <stdio.h>

float f() {
    int x = 10;
    return 10.0;
}

int g() {
    int x = 10;
    return 10;
}

float h() {
    int y = 10;
    return 10.0;
}

int main() {
    f();
    g();
    h();
    return 0;
}
```

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
    %1 = alloca i32, align 4
    store i32 0, ptr %1, align 4
    %2 = call float @f()
    %3 = call i32 @g()
    %4 = call float @h()
    ret i32 0
}
```

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
    %1 = alloca i32, align 4
    store i32 0, ptr %1, align 4
    %2 = call float @f()
    %3 = call i32 @g()
    %4 = call float @f()
    ret i32 0
}
```

Hvala na
pažnji

