



Loop-Invariant Store Hoisting/Elimination

**LISHE**

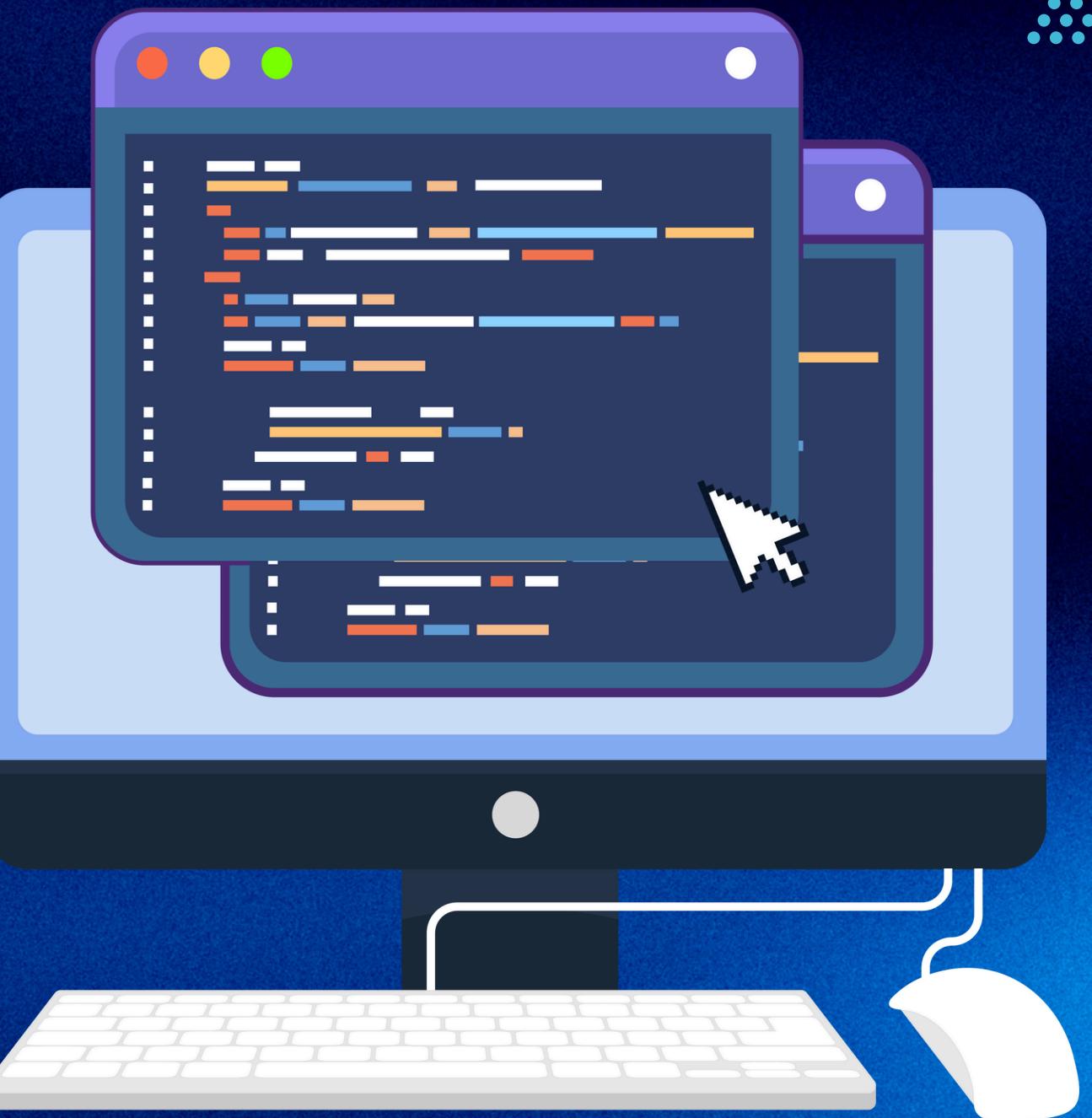
# OPTIMIZACIJE MEĐUKODA U LLVM-U

Zašto optimizujemo međukod?

- Smanjenje broja instrukcija → brže izvršavanje.
- Efikasnije korišćenje memorije i procesorskih resursa.
- Omogućava prevodiocu da ukloni redundantnost i generiše kvalitetniji mašinski kod.

Zašto su bitne optimizacije petlji?

- Petlje se izvršavaju mnogo puta → svaka nepotrebna instrukcija unutra povećava trošak.
- Hoisting invarijantnih operacija van petlje drastično smanjuje broj operacija u runtime-u.
- Posebno značajno za memorijske operacije (load/store).



# LISHE OPTIMIZACIJA

- Fokusirali smo se na store instrukcije unutar petlje.
- Ako adresa i vrednost ne zavise od iteracije → store se može:
  - Hoist-ovati (izvući pre petlje), ili
  - Eliminisati (ako je očigledno mrtav store).

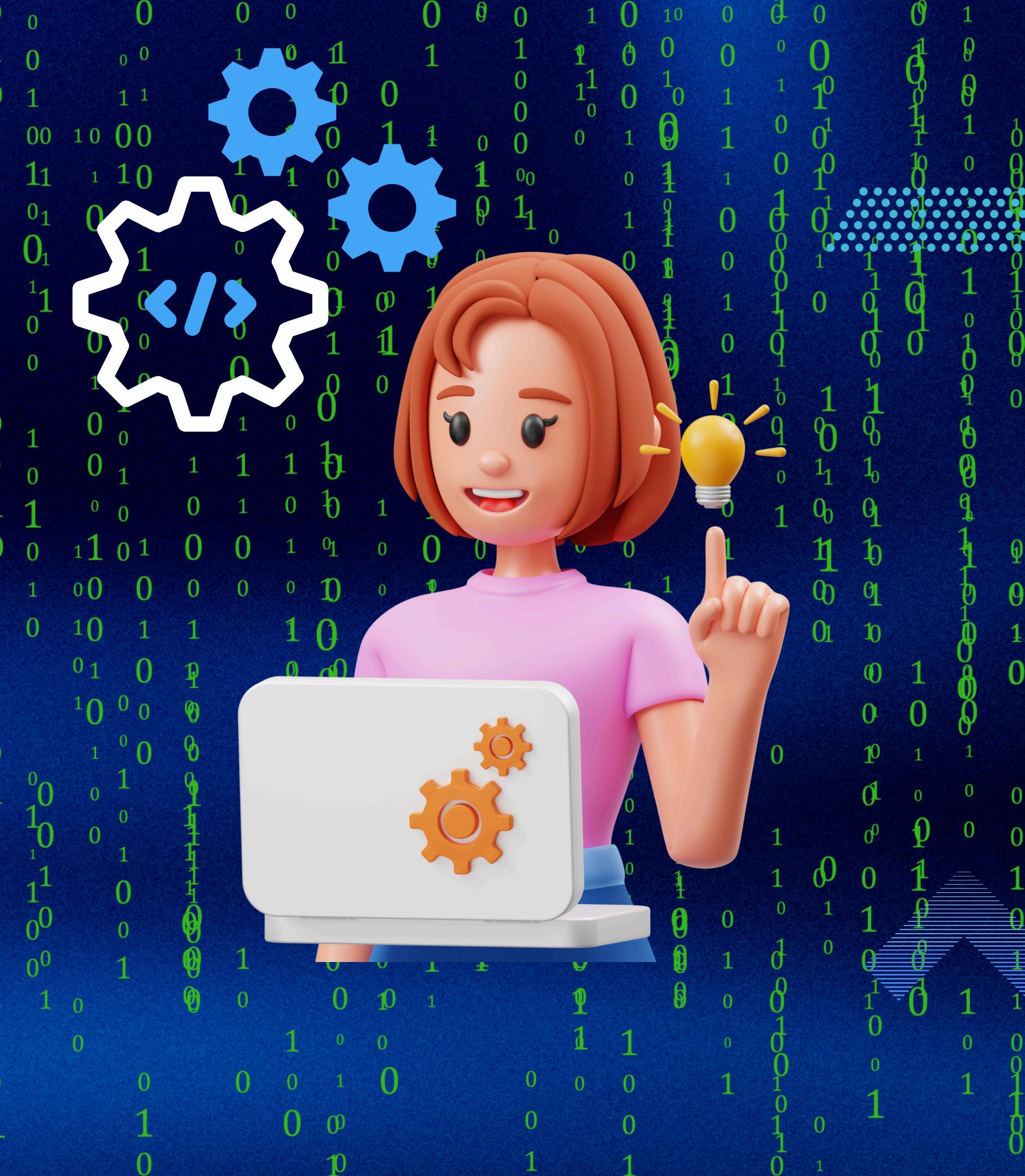
Korišćene analize:

- LoopInfo (struktura petlji).
- AliasAnalysis (provjere aliasinga adresa).
- ScalarEvolution (analiza promenljivih kroz iteracije).



# BITNO U IMPLEMENTACIJI

- Parsiranje LLVM IR i pronalaženje petlji.
- Identifikacija store instrukcija i analiza zavisnosti.
- Provera: da li su pointer i vrednost loop-invariant.
- Upotreba analiza (LoopInfo, AAResults, SE) da osiguramo korektnost.
- Transformacija IR: pomeranje ili brisanje store instrukcija.



# TESTIRANJE LISHE NA PRIMERIMA

BEFORE

```
tests > C 01_basic_hoist.c > basic(int *, int, int)
1 // Candidate for loop-invariant store hoisting
2 // *p = v; - both pointer and value are loop-invariant
3 void basic(int *p, int n, int v)
4 {
5     if (n <= 0)
6         return; // helps ScalarEvolution see that loop executes >= 1
7     for (int i = 0; i < n; ++i)
8     {
9         *p = v; // should be hoisted out of the loop
10    }
11 }
12
```

AFTER

```
25
26 12: ; preds = %19, %11
27     %13 = load i32, ptr %7, align 4
28     %14 = load i32, ptr %5, align 4
29     %15 = icmp slt i32 %13, %14
30     br i1 %15, label %16, label %22
31
32 16: ; preds = %12
33     %17 = load i32, ptr %6, align 4
34     %18 = load ptr, ptr %4, align 8
35     store i32 %17, ptr %18, align 4
36     br label %19
37
38 19: ; preds = %16
39     %20 = load i32, ptr %7, align 4
40     %21 = add nsw i32 %20, 1
41     store i32 %21, ptr %7, align 4
42     br label %12, !llvm.loop !6
43
```

```
14 6: ; preds = %3
15     store i32 %2, ptr %0, align 4
16     br label %7
17
18 7: ; preds = %10, %6
19     %.0 = phi i32 [ 0, %6 ], [ %11, %10 ]
20     %8 = icmp slt i32 %.0, %1
21     br i1 %8, label %9, label %12
22
23 9: ; preds = %7
24     br label %10
25
26 10: ; preds = %9
27     %11 = add nsw i32 %.0, 1
28     br label %7, !llvm.loop !6
```

# TESTIRANJE LISHE NA PRIMERIMA

BEFORE

```
tests > C 06_redundant_same_store.c > ...
1 // Demonstrates redundant store elimination
2 // Two identical stores in the loop; LISHE removes the second one
3 void redundant(int *p, int n, int v)
4 {
5     if (n <= 0)
6         return;
7     for (int i = 0; i < n; ++i)
8     {
9         *p = v; // first store
10        *p = v; // redundant store -> eliminated
11    }
12 }
13
```

AFTER

```
32 16: ; preds = %12
33 %17 = load i32, ptr %6, align 4
34 %18 = load ptr, ptr %4, align 8
35 store i32 %17, ptr %18, align 4
36 %19 = load i32, ptr %6, align 4
37 %20 = load ptr, ptr %4, align 8
38 store i32 %19, ptr %20, align 4
39 br label %21
```

```
14 6: ; preds = %3
15 store i32 %2, ptr %0, align 4
16 br label %7
17
18 7: ; preds = %10, %6
19 %.0 = phi i32 [ 0, %6 ], [ %11, %10 ]
20 %8 = icmp slt i32 %.0, %1
21 br i1 %8, label %9, label %12
22
23 9: ; preds = %7
24 br label %10
25
26 10: ; preds = %9
27 %11 = add nsw i32 %.0, 1
28 br label %7, !llvm.loop !6
29
```

# TESTIRANJE LISHE NA PRIMERIMA

BEFORE

```
tests > C 02_guarded_loop.c > ...
1 // Loop contains a read before the store
2 // Still safe for hoisting because the value being stored is invariant
3 int guarded(int *p, int n, int v)
4 {
5     int sum = 0;
6     if (n <= 0)
7         return 0;
8     for (int i = 0; i < n; ++i)
9     {
10         sum += *p; // reading the memory
11         *p = v;    // can still be hoisted
12     }
13     return sum;
14 }
15 
```

AFTER

```
33
34
35
36 18:                                ; preds = %14
37   %19 = load ptr, ptr %5, align 8
38   %20 = load i32, ptr %19, align 4
39   %21 = load i32, ptr %8, align 4
40   %22 = add nsw i32 %21, %20
41   store i32 %22, ptr %8, align 4
42   %23 = load i32, ptr %7, align 4
43   %24 = load ptr, ptr %5, align 8
44   store i32 %23, ptr %24, align 4
45   br label %25 
```

```
14 6:                                ; preds = %3
15   store i32 %2, ptr %0, align 4
16   br label %7
17
18 7:                                ; preds = %12, %6
19   %.02 = phi i32 [ 0, %6 ], [ %11, %12 ]
20   %.01 = phi i32 [ 0, %6 ], [ %13, %12 ]
21   %8 = icmp slt i32 %.01, %1
22   br i1 %8, label %9, label %14
23
24 9:                                ; preds = %7
25   %10 = load i32, ptr %0, align 4
26   %11 = add nsw i32 %.02, %10
27   br label %12 
```

LISHE pass je namerno pojednostavljen za projekat → hoćemo da pokažemo da može da hoistuje, ali realno to nije 100% bezbedno. Gleda samo da li je pointer p i vrednost v loop-invariant. Ne proverava da li postoji read of the same memory pre store-a. Ali semantički nije uvek bezbedno hoistovati u prisustvu reads pre store-a.



# HVALA NA PAŽNJI!

Jovana Urošević 189/2021  
Lana Matić 143/2021