Konstrukcija kompilatora

always-inline, inst-combine, LICM

Marko Cvijetinović, Jovana Lazić, Matija Radulović

always-inline

- **Garantovan inlining:** Ključna reč *inline* u jeziku C++ je samo sugestija kompajleru, on na osnovu heuristika odlučuje da li će stvarno da inline-uje. Always-Inline uvek ispoštuje direktivu.
- **Podešavanje i optimizacija performansi:** Programer može imati specifično znanje koje sugeriše ugradnju određene funkcije, čak i ako heuristika kompajlera možda neće automatski ugraditi tu funkciju. Ovo se može uraditi kako bi se eliminisalo opterećenje poziva funkcija, omogućile dalje optimizacije ili fino podesile performanse za kritične putanje koda.
- **Eksperimentalna upotreba i analiza:** Istraživači ili programeri koji sprovode eksperimente na transformacijama programa ili karakteristikama performansi mogu koristiti ovaj prolaz da analiziraju uticaj agresivnog ugrađivanja na veličinu koda, brzinu izvršavanja ili druge metrike.

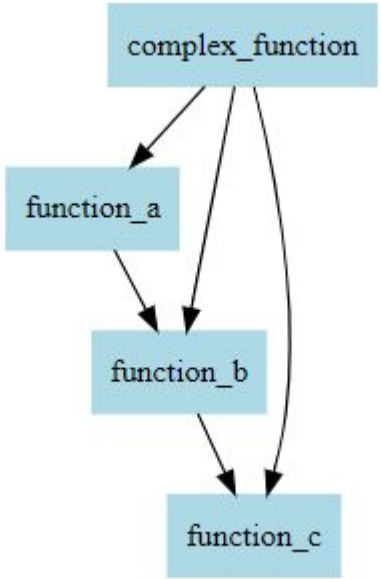
always-inline

- Najjednostavnija varijanta pass-a
- U svakom bloku trazimo funkcije označene sa *always-inline* i pozivamo ugrađene funkciju za inline-ovanje
- Pritom preskačemo eskterne funkcije i deklaracije

```
if (F.isDeclaration()) continue;

for (BasicBlock &BB : F) {
  for (auto It = BB.begin(); It != BB.end(); ) {
    Instruction &I = *It++;
    auto *CB = dyn_cast<CallBase>(&I);
    if (!CB) continue;

    Function *Callee = CB->getCalledFunction();
    if (!Callee) continue;
    if (Callee->isDeclaration()) continue;

    if (!Callee->hasFnAttribute(Attribute::AlwaysInline)) continue;

    InlineFunctionInfo IFI;
    InlineFunction(*CB, IFI);
  }
}
```

always-inline

- **Problem:** Rekurzija pravi beskonačan kod
- Postoje razna rešenja: pretraga callgraph-a, ograničenje dubine…
- Ovde smo se odlučili da se oslonimo sa function-attrs pass
- Kreirana je skripta za pokretanje koja brine o tome da se prvo pokrene function-attrs
- Changed služi da zamaptimo da smo menjali kod za kasnije




```
if (!Callee->hasFnAttribute(llvm::Attribute::NoRecurse)) continue;

InlineFunctionInfo IFI;
auto Ret = InlineFunction(*CB, IFI);
if (Ret.isSuccess()) Changed = true;
```

always-inline

- Ostalo je da sredimo kod jer sadrži informacije za obrisane funkcije
- Tokom obrade blokova dodajemo funkcije koje se ne koriste u vector, kasnije brišemo njih i atribute koje je napravio function-attrs pass
- Posle ovoga kod izgleda isto kao što izbacuje llvm-om always-inline

```
// Ovaj blok koda je unutar petlje koji obradjuje blok
// ostatala dva nisu
if (Callee->use_empty() && !Callee->isDeclaration() &&
  (Callee->hasLocalLinkage() || Callee->hasLinkOnceODRLinkage()))
    ToErase.push_back(Callee);

for (Function *Fn : ToErase)
  if (Fn && Fn->use_empty())
    Fn->eraseFromParent();

if (Changed)
  for (Function &F : M)
    F.setAttributes(AttributeList());
return Changed;
```

inst-combine

- inst-combine (Instruction Combining) je jedan od najvažnijih i najčešće korišćenih optimization pass-ova u LLVM-u
- Njegova uloga je da prepozna jednostavne airtmetičke obrasce i zameni ih instrukcijama koje proizvode isti rezultat ali sa manje operacija
- Ovaj pass ne sprovodi velike transformacije nad celim programom, već gleda lokalne obrasce od nekoliko instrukcija u okviru funkcije
- Oslanja se na poznate aritmetičke i logičke identitete, a nekad i na samu arhitekturu procesora (npr. množenje i deljenje stepenima dvojke)

inst-combine

- Kanonizacija
- Bitovska kanonizacija
- Redukcija poređenja
- Bulovska redukcija poređenja
- X + X -> X << 1
- X * 2^k -> X << k
- X / 2^k -> X << k
- Algebarski identiteti
- Identiteti iskazne logike
- Spajanje konstanti
- (X + C1) + C2 -> X + (C1 + C2)
- Prepoznavanje negacije

```
static bool applyOptimizations(Instruction &I)
{

  auto Opts = {
      moveConstToRHS,
      orderBitwiseWithConst,
      foldRelICmpToEqNe,
      foldICmpOnBool,
      foldAddXX,
      foldMulPow2ToShl,
      foldDivPow2ToShr,
      foldSimpleArith,
      foldLogicBasics,
      foldConstOp,
      reassocAddConst,
      foldNegations,
  };

  for(auto &Opt : Opts)
  {
    if (Opt(I))
      return true;
  }

  return false;
}
```

inst-combine

Kanonizacija odnosno premešatanje svih konstanti na desnu stranu komutativnog binarnog operatora. Primer:

5 + x -> x + 5

8 * x -> x * 8

```
static bool moveConstToRHS(Instruction &I)
{
  auto *BO = dyn_cast<BinaryOperator>(&I);
  if (!BO) return false;

  unsigned Op = BO->getOpcode();
  if (!Instruction::isCommutative(Op)) return false;

  Value *L = BO->getOperand(0);
  Value *R = BO->getOperand(1);

  if (isa<Constant>(L) && !isa<Constant>(R)) {
    BO->setOperand(0, R);
    BO->setOperand(1, L);
    return true;
  }

  return false;
}
```

inst-combine

Addition identities:
X + 0 -> X
Subtraction identity:
 X - 0 -> X
Multiplication identities:
X * 1 -> X
X * 0 -> 0
Bitwise identities:
X && T -> X
X || F -> X
X ^ F -> X
Self-operations:
 X ^ X = 0
 X - X = 0

```
static bool foldSimpleArith(Instruction &I)
{
  auto *BO = dyn_cast<BinaryOperator>(&I);
  if (!BO) return false;
  Value *X = nullptr;

  if (match(BO, m_Add(m_Value(X), m_Zero())) || match(BO, m_Add(m_Zero(), m_Value(X))))
    return replaceInst(I, X);

  if (match(BO, m_Sub(m_Value(X), m_Zero())))
    return replaceInst(I, X);

  if (match(BO, m_Mul(m_Value(X), m_One())) || match(BO, m_Mul(m_One(), m_Value(X))))
    return replaceInst(I, X);

  if (match(BO, m_Mul(m_Value(X), m_Zero())) || match(BO, m_Mul(m_Zero(), m_Value(X))))
    return replaceInst(I, Constant::getNullValue(I.getType()));

  if (match(BO, m_And(m_Value(X), m_AllOnes())) || match(BO, m_And(m_AllOnes(), m_Value(X))))
    return replaceInst(I, X);

  if (match(BO, m_Or(m_Value(X), m_Zero())) || match(BO, m_Or(m_Zero(), m_Value(X))))
    return replaceInst(I, X);

  if (match(BO, m_Xor(m_Value(X), m_Zero())) || match(BO, m_Xor(m_Zero(), m_Value(X))))
    return replaceInst(I, X);

  if (match(BO, m_Xor(m_Value(X), m_Specific(X))) || match(BO, m_Sub(m_Value(X), m_Specific(X))))
    return replaceInst(I, Constant::getNullValue(I.getType()));

  return false;
}
```

LICM

- LICM (Loop Invariant Code Motion) je optimizacioni prolaz koji izmešta izraze koji daju isti rezultat pri svakoj iteraciji petlje izvan same petlje, u njen preheader blok

Ako neka instrukcija u telu petlje:

- ne zavisi od vrednosti koje se menjaju unutar petlje, i
- ne menja stanje memorije na koje druga instrukcija u petlji zavisi,

onda se ta instrukcija može hoistovati (premestiti iz petlje).

- Smanjuje broj instrukcija koje se izvršavaju u svakoj iteraciji.
- Poboljšava iskorišćenost keša i prediktivnost grananja.

LICM

Potrebni koraci:

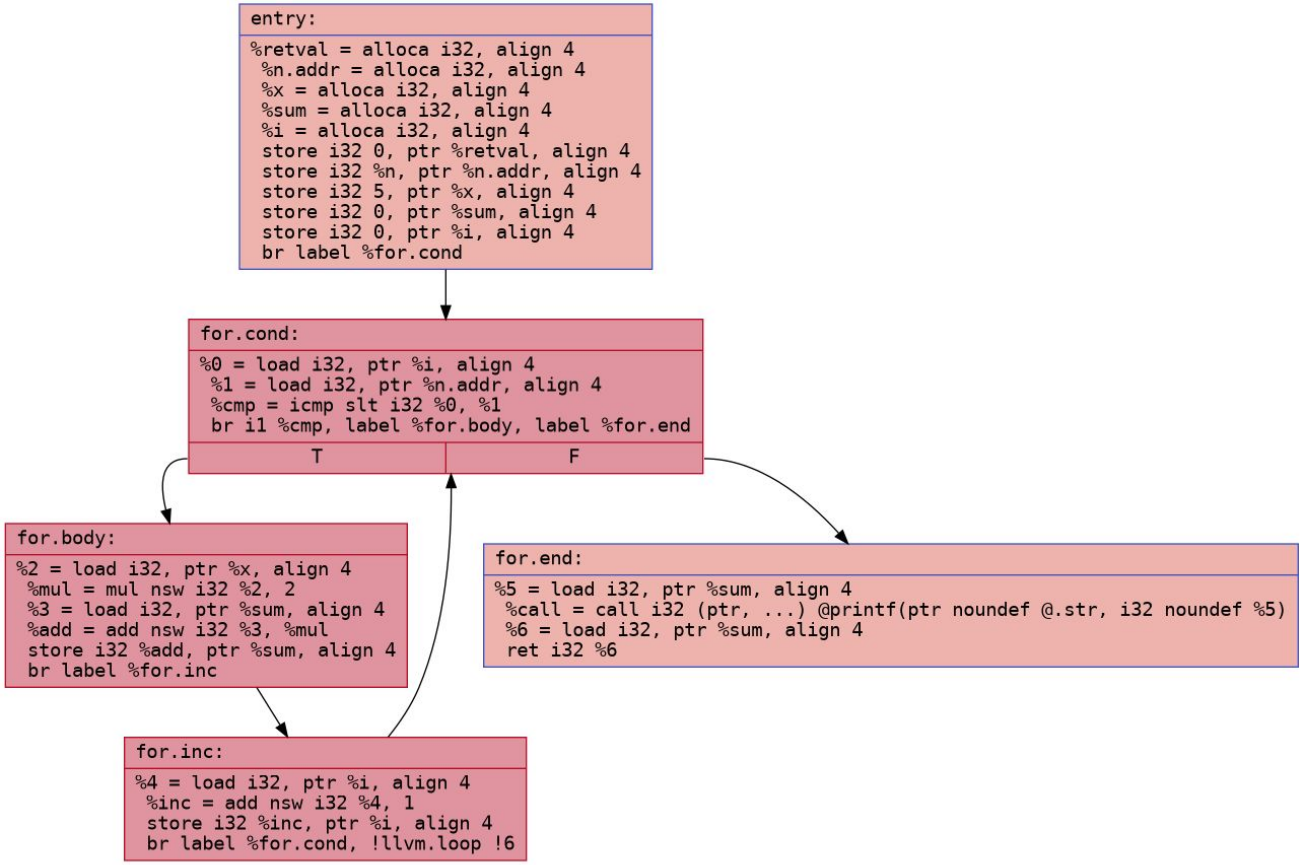
- Analiza dominatora - utvrditi da li je bezbedno izdignuti instrukciju u zavisnosti od toga dal preheader predhodi (dominira) svim korišćenjima instrukcije (u svakom toku izvrašavanja se do instrukcije prolazi kroz header)
- Landing pad - ukoliko se petlja ne izvršava ni jednom izdignuće instrukcije narušiće semantiku programa
- Analiza invarijantnih instrukcija - ukoliko su svi operandi konstantni ili definisani izvan petlje, kod jednostavnih instrukcija
- Analiza alijasa - posebna pažnja je potrebna za load i store zbog mogućih aliasa

LICM

```c
int main(int n) {
    int x = 5;
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += x * 2; // x*2 is loop-invariant
    }
    printf("%d\n",sum);
    return sum;
}
```

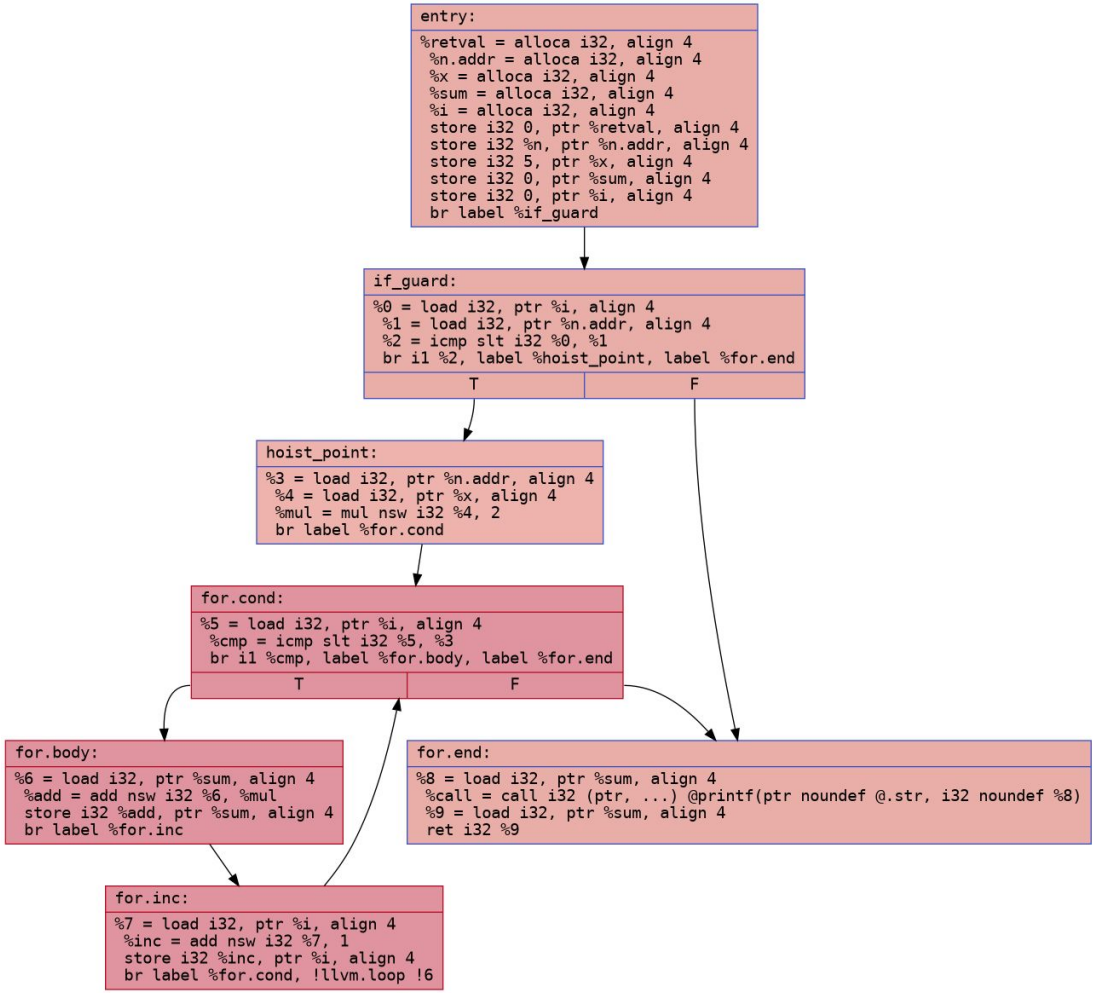
```c
int main(int n) {
    int x = 5;
    int sum = 0;
    int z = x * 2; //hoisted
    for (int i = 0; i < n; i++) {
        sum += z;
    }
    printf("%d\n",sum);
    return sum;
}
```

LICM



CFG for 'main' function

LICM



CFG for 'main' function