Projekat na predmetu Konstrukcija kompilatora na Matematičkom fakultetu - Instruction combine pass

Tamara Tomić
mi17122@alas.matf.bg.ac.rs
Đorđe Milošević
mi19221@alas.matf.bg.ac.rs

Jun 2023

Sadržaj

1 Podešavanje virtuelne mašine

Za podešavanje virtuelne mašine ispratiti korake na sledećem linku.

2 Opis pass-a

Instruction combine kombinuje instrukcije tako da se formiraju manje i jednostavnije instrukcije. Primer primene ovog pass-a:

```
%Y = add i32 %X, 1
%Z = add i32 %Y, 1

%Z = add i32 %X, 2
```

Na sledećem linku su opisani koraci koji će u nastavku biti implementirani. Biće implementirani koraci 1, 5 i 6.
Glavna funkcija koja će pozivati funkcije za iteriranje kroz instrukcije i brisanje instrukcija je funkcija **runOnFunction**, koja će biti pozvana svaki put kada se naiđe na funkciju.

```
bool runOnFunction(Function &F) override {

    IterateThroughFunction(F);
    RemoveInstructions();

    return true;
}
```

Funkcija za iteraciju kroz instrukcije prolazi prvo kroz sve basic block-ove funkcije, a zatim za svaku funkciju obilazi sve njene instrukcije. U zavisnosti od instrukcije pozivamo određenu funkciju koja će rukovati njome na traženi način. Takođe za svaku **Load instrukciju** ćemo pamtiti vrednost koju ta promenljiva u load instrukciji pamti u mapi **VariablesMap**.

```
void IterateThroughFunction(Function &F)
{

    for (BasicBlock &BB : F) {
        for (Instruction &Instr : BB) {
            if (LoadInst *Store = dyn_cast<LoadInst>(&Instr)) {
                VariablesMap[Store] = Store->getOperand(0);
            }

            if(IsBinaryOp(&Instr)) {
                HandleBinaryOp(&Instr);
            }

            if (IsCmp(&Instr)) {
                HandleCmp(&Instr);
            }
        }
    }
}
```

2.1 1. korak

Prvi korak nam kaže, da ako binarni operator sadrži operand koji je konstanta, taj operand se pomera na desnu stranu operacije (RHS).

```
void HandleBinaryOp(Instruction *Instr)
{

    BinaryOperator *BO = dyn_cast<BinaryOperator>(Instr);
    IRBuilder Builder(Instr);

    // 1.

    if (Constant *C = dyn_cast<Constant>(Instr->getOperand(0))) {
        switch (BO->getOpcode()) {
            case Instruction::Add:
                Instr->setOperand(0, Instr->getOperand(1));
                Instr->setOperand(1, C);
                break;
            case Instruction::Mul:
                Instr->setOperand(0, Instr->getOperand(1));
                Instr->setOperand(1, C);
                break;
            default:
                break;

        }

    }
```

Ako se konstanta već nalazi na desnoj strani operacije tada nećemo raditi ništa, a u slučaju da se nalazi na levoj, što bi predstavljao **Instr->getOperand(0)**, tada ćemo ga dinamički kastovati u konstantu, i ako je to uspešno izvršeno, ulazimo u if blok.
U **if** bloku ćemo koristiti switch po **BO->getOpcode()** što nam dohvata operaciju instrukcije. U našem slučaju prvo pravilo se može primeniti na sabiranje, množenje kao i na operacije poredjenja.
U svakom od ovih slučajeva ćemo postaviti prvi operand (konstanta) na desnu stranu binarne operacije (**Instr->setOperand(1,C)**), dok ćemo operand sa desne strane smestiti na levu stranu (**Instr->setOperand(0,Instr->getOperand(1))**).

```
void HandleCmp(Instruction *Instr) // 1.
{
    if (ICmpInst *CI = dyn_cast<ICmpInst>(Instr)) {
        Value *op0 = CI->getOperand(0);
        Value *op1 = CI->getOperand(1);
        CmpInst::Predicate pred = CI->getPredicate();

        IRBuilder Builder(Instr);
        Value *newCI = nullptr;

        if (Constant *C = dyn_cast<Constant>(op0)) {

            switch (pred) {
                case ICmpInst::ICMP_EQ:
                    Instr->setOperand(0, op1);
                    Instr->setOperand(1, op0);
                    break;
                case ICmpInst::ICMP_NE:
                    Instr->setOperand(0, op1);
                    Instr->setOperand(1, op0);
                    break;
                case ICmpInst::ICMP_SLT:
                    newCI = Builder.CreateICmpSGT(op1, op0);
                    break;
                case ICmpInst::ICMP_SGT:
                    newCI = Builder.CreateICmpSLT(op1, op0);
                    break;
                case ICmpInst::ICMP_SLE:
                    newCI = Builder.CreateICmpSGE(op1, op0);
                    break;
                case ICmpInst::ICMP_SGE:
                    newCI = Builder.CreateICmpSLE(op1, op0);
                    break;
                default:
                    break;
            }
```

Slično je i za operaciju poređenja, gde sa **CI->getPredicate()** proveravamo po kom uslovu poredimo operande.
Primetimo da kod nekih operacija poređenja kreiramo novu instrukciju poređenja. Moramo paziti na to da kada zamenimo mesta operandima, vrednost izraza ostane ista, tako da mi moramo da kreiramo novu instrukciju. Na kraju ove funkcije ćemo zameniti sve stare instrukcije koje više ne koristimo, novim instrukcijama. Postavlja se pitanje, šta radimo sa starom instrukcijom? Instrukcije koje treba da obrišemo smeštamo u vektor:

```
std::vector<Instruction *> InstructionsToRemove;
```

kroz koji ćemo na kraju proći u jednoj petlji i obrisaćemo iz njenog roditelja, odnosno basic block-a koji je sadrži.

2.2 5. korak

Peti korak nam kaže, da ako imamo izraz koji predstavlja sabiranje 2 iste promenljive, tada ćemo taj izraz predstaviti kao množenje te promenljive brojem 2, odnosno predstavićemo kao šiftovanje u levo za 1.
Pošto je ovaj uslov vezan za binarnu operaciju sabiranja, takođe se nalazi u funkciji gde rukujemo binarnim operacijama.

```
if(BO->getOpcode() == Instruction::Add) {
    if(VariablesMap[BO->getOperand(0)] == VariablesMap[BO->getOperand(1)]) {
        Value *shiftInstr = Builder.CreateShl(BO->getOperand(0), 1);
        Instr->replaceAllUsesWith(shiftInstr);
        InstructionsToRemove.push_back(Instr);
    }

}
```

Prvo proveravamo da li su vrednosti promenljivih, koje smo sačuvali u mapi **VariablesMap** jednake, zatim kreiramo novu instrukciju šiftovanja u levo za 1 i zatim zamenjujemo instrukcije i pamtimo staru instrukciju koju ćemo na kraju obrisati iz roditelja.

2.3 6. korak

Šesti korak predstavlja opšti slučaj petog koraka, gde imamo da se množenje promenljive konstantom, gde je ta konstanta stepen broja 2, predstavlja kao šiftovanje promenljive za taj stepen.
Takođe se i ovaj uslov kao i peti nalazi u funkciji koja rukuje binarnim operacijama.
Uslovi su da je drugi operand celobrojna konstanta, kao i da je vrednost celobrojne konstante stepen dvojke. Ukoliko su ispunjena ta 2 uslova, nalazimo stepen dvojke pomoću:

```
constOp->getValue().logBase2()
```

Zatim kreiramo kao i u 5. koraku instrukciju šiftovanja u levo, samo ovaj put imamo šiftovanje za stepen dvojke. Na kraju opet zamenjujemo instrukcije i stavljamo staru instrukciju u niz za brisanje.

```
if(BO->getOpcode() == Instruction::Mul) {

    Value *operand1 = BO->getOperand(0);
    Value *operand2 = BO->getOperand(1);

    ConstantInt *constOp = dyn_cast<ConstantInt>(operand2);
    if (constOp && constOp->getValue().isPowerOf2()) {
        unsigned int shiftAmount = constOp->getValue().logBase2();
        Value *shiftInstr = Builder.CreateShl(operand1, shiftAmount);
        Instr->replaceAllUsesWith(shiftInstr);
        InstructionsToRemove.push_back(Instr);
    }


}
```

3 Primeri

Za svaki korak ćemo videti na primeru razliku u IR zapisu.

Za prvi korak ćemo napraviti prost .c program koji sa leve strane ima konstante u instrukcijama.

```c
#include <stdio.h>

int main() {
  int a = 1;
  int b = 2;

  int c = 3 + a;
  int d = 5 * c;

  return 0;
}
```

IR zapis bez ikakve obrade će biti:

```
; Function Attrs: noinline
nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1, i32* %2, align 4
  store i32 2, i32* %3, align 4
  %6 = load i32, i32* %2, align 4
  %7 = add nsw i32 3, %6
  store i32 %7, i32* %4, align 4
  %8 = load i32, i32* %4, align 4
  %9 = mul nsw i32 5, %8
  store i32 %9, i32* %5, align 4
  ret i32 0
}
```

primetimo da su zanimljive instrukcije:

```
%7 = add nsw i32 3, %6,
%9 = mul nsw i32 5, %8
```

odnosno na tim instrukcijama očekujemo promenu nakon obrade.

Nakon obrade dobijamo:

```
; Function Attrs: noinline nounwind
optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 1, ptr %2, align 4
  store i32 2, ptr %3, align 4
  %6 = load i32, ptr %2, align 4
  %7 = add nsw i32 %6, 3
  store i32 %7, ptr %4, align 4
  %8 = load i32, ptr %4, align 4
  %9 = mul nsw i32 %8, 5
  store i32 %9, ptr %5, align 4
  ret i32 0
}
```

i primećujemo da kao i što smo napomenuli, došlo je do promene u zapisu odnosno do promene mesta operanada u instrukcijama koje sadrže sabiranje i množenje.
Takođe i na sledećem primeru:

```
int main () {

    int a = 1;
    int b = 2;

    if (2 != a){
        printf("a is not equal 2\n");
    }
    if (3 <= b){
        printf("b is greater or equal 3\n");
    }

    return 0;
}
```

želimo da imamo zapis koji će konstante staviti sa desne strane operacije, sa tim što operacija mora da se adekvatno promeni.

Dobijeni zapis bez obrade je:

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1, i32* %2, align 4
  store i32 2, i32* %3, align 4
  %4 = load i32, i32* %2, align 4
  %5 = icmp ne i32 2, %4
  br i1 %5, label %6, label %8

6:
; preds = %0
  %7 = call i32 (i8*, ...) @printf(i8*
getelementptr inbounds ([18 x i8], [18 x
i8]* @.str, i64 0, i64 0))
  br label %8

8:
; preds = %6, %0
  %9 = load i32, i32* %3, align 4
  %10 = icmp sle i32 3, %9
  br i1 %10, label %11, label %13
```

i ovde vidimo da su linije od interesa

```
%5 = icmp ne i32 2, %4,
%10 = icmp sle i32 3, %9
```

Nakon obrade imamo:

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 1, ptr %2, align 4
  store i32 2, ptr %3, align 4
  %4 = load i32, ptr %2, align 4
  %5 = icmp ne i32 %4, 2
  br i1 %5, label %6, label %8

6:
preds = %0
  %7 = call i32 (ptr, ...) @printf(ptr @.str)
  br label %8

8:
preds = %6, %0
  %9 = load i32, ptr %3, align 4
  %10 = icmp sge i32 %9, 3
  br i1 %10, label %11, label %13
```

gde vidimo da je nakon obrade konstanta sa desne strane operacije, ali su se takođe i operacije promenile(u ovom slučaju $\neq$ je ostalo $\neq$, ali je $\leq$ prešlo u $\geq$).

Za 5. i 6. korak ćemo napraviti .c program:

```c
#include <stdio.h>

int main() {
  int a = 1;
  int b = 2;

  int c = a + a;
  int d = c * 4;
  int e = 4 * d;

  return 0;
}
```

U ovom slučaju imamo 3 instrukcije koje želimo da obradimo. U instrukciji $c = a + a$ želimo da šiftujemo a za jedan u levo, u $d = c * 4$ želimo da šiftujemo c za 2 u levo, i u $e = 4 * d$ želimo da konstantu prebacimo na desnu stranu i šiftujemo d za 2 u levo. IR zapis bez obrade bi bio:

```
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
%4 = alloca i32, align 4
%5 = alloca i32, align 4
%6 = alloca i32, align 4
store i32 0, i32* %1, align 4
store i32 1, i32* %2, align 4
store i32 2, i32* %3, align 4
%7 = load i32, i32* %2, align 4
%8 = load i32, i32* %2, align 4
%9 = add nsw i32 %7, %8
store i32 %9, i32* %4, align 4
%10 = load i32, i32* %4, align 4
%11 = mul nsw i32 %10, 4
store i32 %11, i32* %5, align 4
%12 = load i32, i32* %5, align 4
%13 = mul nsw i32 4, %12
store i32 %13, i32* %6, align 4
ret i32 0
```

Nakon obrade prethodnih slučajeva dobijamo sledeći zapis:

```
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
%4 = alloca i32, align 4
%5 = alloca i32, align 4
%6 = alloca i32, align 4
store i32 0, ptr %1, align 4
store i32 1, ptr %2, align 4
store i32 2, ptr %3, align 4
%7 = load i32, ptr %2, align 4
%8 = load i32, ptr %2, align 4
%9 = shl i32 %7, 1
store i32 %9, ptr %4, align 4
%10 = load i32, ptr %4, align 4
%11 = shl i32 %10, 2
store i32 %11, ptr %5, align 4
%12 = load i32, ptr %5, align 4
%13 = shl i32 %12, 2
store i32 %13, ptr %6, align 4
ret i32 0
```

gde vidimo da smo od instrukcija

```
%9 = add nsw i32 %7, %8
%11 = mul nsw i32 %10, 4
%13 = mul nsw i32 4, %12
```

dobili instrukcije:

```
%9 = shl i32 %7, 1
%11 = shl i32 %10, 2
%13 = shl i32 %12, 2
```