Documentation - Sprint1

Computer Science Engineering
Lee Jeong Hoon

A. Requirement and Specification

1. Constant folding, removing identical instructions (1st sprint)

First, any variables and expressions that can be statically determined, compiler can substitute them with the identical constants and propagate them. We can iterate over the instructions, search statically determinable values, then substitute and propagate them.

Also, instructions to load the value which is already loaded are redundant. From one load instruction, we can search backward and if there is another load instruction accessing the same memory address, we can eliminate the latter one, or more simply we can reorder the latter instruction to be right after the former one with changing load to assignment.

Similarly, storing the value and loading it immediately can be reduced to a simple assignment. We can search backward from the load instruction just like above, and if there's provider instruction storing the value to the same memory address, we can remove the latter load instruction.

- algorithm

```
for each block B:
  for each instruction I in B:
    if match(I, [m_add | m_sub | ...](X, Y)) and X, Y are constant:
      substitute <LHS> with constant
      propagate conversion of <LHS> to constant
    if match(I, m_load(ptr)):
      for each instruction J above:
        if match(J, m_load(ptr)):
          checks modification between I and J
          substitute I with J
        if match(J. m_store(ptr)):
          checks modification between I and J
          substitute I with stored value in J
```

- examples of optimization

```
define i32 @f(i32 %x) {
  %mask1 = shl i32 1, 31
  %mask2 = ashr i32 %mask1, 7
  %mask3 = lshr i32 %mask2, 24
  %res = and i32 %x %mask3
  ret i32 %res
}
```

```
define i32 @f(i32 %x) {
  %res = and i32 %x 255
  ret i32 %res
}
```

In this example, %mask1 is generated by shl operator with two constants, so %mask1 always has 0x80000000. This result is propagated to the second instruction, which results in %mask2 = ashr %mask 2147483648, 24. As %mask 2 is also generated by ashr operator with two constants, it always has 0xff000000, then propagated to the third instruction. Same way, %mask3 has value 0x0000ff and finally the fourth instruction is optimized to %res = and i32 %x 255

```
define i32 @g(i32* %p, i32* %q) {
  %i1 = load i32* %p
  %i2 = load i32* %q
  %sum = add i32 %i1, %i2
  %avg = udiv i32 %sum, 2
  %i3 = load i32* %p
  %diff = sub i32 %avg, %i3
  ret i32 %diff
}
```

```
define i32 @g(i32* %p, i32* %q) {
  %i1 = load i32* %p
  %i2 = load i32* %q
  %sum = add i32 %i1, %i2
  %avg = udiv i32 %sum, 2
  %diff = sub i32 %avg, %i1
  ret i32 %diff
}
```

In this example, fifth instruction loads the value from the memory pointed by %p argument. As the first instruction do the same thing in %i1 and %i1 is not modified, using %i3 can be substituted with %i1.

```
define i32 @h(i32* %p) {
  BB1:
    %i1 = load i32* %p
    %i2 = add i32 %i1, 1
    store i32 %i2, i32* %p
  BB2:
    %mask = mul i32 1, 255
    %i3 = load i32* %p
    %res = and i32 %i3, %mask
    ret i32 %res
}
```

```
define i32 @h(i32* %p) {
  BB1:
    %i1 = load i32* %p
    %i2 = add i32 %i1, 1
    store i32 %i2, i32* %p
  BB2:
    %mask = mul i32 1, 255
    %res = and i32 %i2, %mask
    ret i32 %res
}
```

In this example, second instruction in BB2 basic block loads the value from the memory pointed by %p argument. As the mote recent store/load instruction(in this case, store instruction at third line of BB1) stores the value %i2 into memory %p, we know that memory %p has %i2. Therefore, we can substitute the usage of %i3 with %i2.

2. Arithmetic Optimization (2nd sprint)

Complex arithmetic expressions can be simplified algebraically. Number of operators can be reduced and identical expression with lower cost can be used instead. For example, we can change all shift operations to multiplication/divisions and then simplify the expression by matching to some prepared patterns. As multiplication is cheaper than addition, unusual optimization such as $(a+b)\times(a-b)=a\times a-b\times b$ maybe possible.

3. Branch related optimization including br -> switch (3rd sprint)

Simplifying the Control Flow Graph(CFG) is useful to reduce the cost by multiple branch instructions. For example, empty blocks(having unconditional branch instruction only) can be eliminated. Even if they are used for determining the phi node in successor, it can be optimized with switch instruction. Also, we can extend it to the case where a block only has an instruction to generate condition and diverge with branch instruction. Thereby we can reduce the number of branch instructions and also reduce usages of registers/stack to calculate condition for each branch instruction.

B. Planning

I will implement one optimization in one sprint. As determining the constant value provides more chances to optimize algebraically and both optimization provide useful information to optimize conditional branch instructions, I will implement things in same order with stated above. (constant folding, remove redundancy - arithmetic optimization - branch related optimization)