15-411: Compiler Design — Spring 2026

Recitation 8: Optimizations Solutions — 20 March

For previous labs, we have always introduced a new set of features to the source language that you compile. In L5, you are not adding new features, but optimizing your compiler to output more efficient assembly. We will be discussing some optimizations that you can write and challenges you may face.

Easy Optimizations First

We spend so much time talking about interesting and hard-to-implement optimizations that it's easy to get ahead of oneself and jump straight into implementing something complex like alias analysis. Don't underestimate how much simple things like peephole optimizations and improving instruction selection can impact your performance. Easy-to-implement optimizations such as eliminating self-moves or making better use of x86's parentheses syntax for memory loads could yield drastic performance improvements for relatively little work. We recommend starting L5 by reflecting on your compiler's weak suits and looking for easy improvements to make before moving on to more advanced optimizations.

Peephole Optimizations

- **Constant Folding** is a code transformation that replaces a binary operation with a constant. It works very well in tandem with constant propagation. Be careful to preserve side effects of operations though, e.g. division, modulo, shift.

  Constant folding can also be applied to branches. If the condition of a branch is constant (or can be folded into a constant), the conditional jump can be replaced with an unconditional jump. This may cause entire basic blocks of code to become dead code.

- **Strength Reduction** replaces expensive operations with simpler ones. Arithmetic operations that interact with 0 or 1 can often be replaced with moves, to which constant or copy propagation can be applied.

- **Null Sequences** are sequences of operations that do not have any effects and can be replaced with a single nop. For example, self moves (moves where dest and src are statically the same) are always safe to remove since they do not change the machine state. Moving `a` to `b` and then immediately back from `b` to `a` means that the second move can be replaced with a nop and not change its behavior. Another common example is an unconditional jump to the next instruction.

Checkpoint 0

Identify opportunities for peephole optimizations in the following function.

```
 1 foo:
 2   t1 ← 40 + 2
 3   t2 ← t1 * 1
 4   if 3 < 2 then l1 else l2
 5 l1:
 6   t3 ← t2
 7   goto l3
 8 l2:
 9   t4 ← t1 - 0
10   t3 ← t4
11   goto l3
12 l3:
13   t5 ← phi(t3,t4)
14   t6 ← t5 * 128
15   t5 ← t6 - t6
```

Solution: This is not a comprehensive list.

(a) line 2: evaluate 40 + 2 to 42

(b) line 3: in combo with above, constant prop t1 = 42; strength reduction

(c) line 4: constant fold since 3 < 2 is always false

(d) line 6: variable / constant prop

(e) line 11: null sequence (goto is unnecessary due to fall-through)

Common Subexpression Elimination

When the same operation on the same data is performed multiple times in code, common subexpression elimination replaces the latter instances of the same operation with the result of the earlier operation. This can potentially save repeating the same calculation at runtime. However, in order for this replacement to be correct, we must be sure that the same operation will yield the same result at the point of the replacement, otherwise we risk replacing subexpressions with outdated results.

Doing CSE in SSA form ensures that syntactically equal subexpressions are indeed equal in value, but we still need to make sure that the result being reused is defined along every path from the beginning of the function to the line of reuse. This can be determined by looking at the dominance relation between basic blocks.

Checkpoint 1

Identify opportunities for common subexpression elimination in the following code. Make sure to check the dominance relations.

```
1 l1:
2   c ← b
3   x ← a ⊕ b
4   if a < b then l2 else l3
5 l2:
6   y1 ← a ⊕ b
7   goto l4
8 l3:
9   y2 ← z ⊕ b
10  goto l4
11 l4:
12  y3 ← phi(y1,y2)
13  u ← z ⊕ b
```

Solution: There are 2 pairs of common subexpressions, `a` ⊕ `b` and `z` ⊕ `b`. The former can have CSE applied because the block where it is defined dominates the block we wish to reuse it in. The latter unfortunately does not.

Checkpoint 2

If line 6 above was `y1` ← `a` ⊕ `c` instead, `a` ⊕ `b` would no longer be a syntactic common subexpression. They are clearly semantically equivalent though. How can we identify these semantically but not syntactically equivalent subexpressions to help CSE do better?

Solution: Value Numbering. Essentially on line 2, we would note that c and b must have the same value in the program. Then we would be able to deduce that `a` ⊕ `c` and `a` ⊕ `b` are equivalent too, and apply CSE.

Checkpoint 3

One of the pairs of common subexpressions from checkpoint 1 could not have CSE applied due to the lack of a dominance relation. However, no matter what path we take through the CFG, that subexpression is computed at least once, and possibly twice. How can we move some instructions around (code motion) so that the subexpression in question only gets computed once?

Solution: Partial Redundant Elimination: This is honestly a difficult optimization that we do not expect most teams to implement. Essentially have `t` ← `z` ⊕ `b` to l1, and use replace both y2 and u with t.

Re-motivating SSA

Many optimizations become significantly simpler in SSA form. This is because in SSA, each variable is assigned exactly once, every use has a single, well-defined definition, and definitions dominate uses.

Checkpoint 4

Explain how SSA simplifies:

(a) Constant propagation

(b) Common subexpression elimination

(c) Dead code elimination

Solution: For constant propagation, we no longer need to worry about reassignment and this kills the need for extensive dataflow analysis

For CSE, syntactic equality now implies semantic equality

For DCE, use-def chains are now explicit, so it's now very easy to track users

Conditional Constant Propagation

Many people often implement a simple constant propagation pass in their compilers in the initial labs. However, consider:

```
1  x <- 1
2  if (x < 2) then l1 else l2
3  l1:
4    y <- 5
5  l2:
6    y <- 10
7  return y
```

Standard constant propagation can't eliminate the branch since it doesn't track which branches execute, so it can't prove that the false branch is unreachable.

Conditional Constant Propagation (CPP) tracks which blocks are executable and whether variables are constant, unknown, or varying. It can also help eliminate unreachable branches.

Checkpoint 5

Apply conditional constant propagation (CCP) to the following program. Identify which blocks are exeuctable, as well as the final simplified program after CCP.

```
1  L1:
2    x <- 1
3    y <- 2
4    if x < y then L2 else L3
5
6  L2:
7    z1 <- x + y
8    goto L4
9
10 L3:
11   z2 <- y - x
12   goto L4
13
```

```
14  L4:
15    z3 <- phi(z1, z2)
16    w <- z3 * 2
17    return w
```

Solution: Initially, all blocks are assumed unreachable except entry, and all variables are assumed constant until proven otherwise.

Examining L1: $x = 1$ and $y = 2$, so the condition $x < y$ evaluates to $1 < 2$ which is true.

Side note (unrelated to solution): in lecture, we talked about using a lattice for variables where a variable varies between three states: 1. we have evidence that a variable can hold different values at different times, 2. we have evidence that a variable has been assigned a constant, and 3. a variable is not executed.

Examining the blocks, we can now conclude that L2 is reachable and currently L3 is still unreachable.

Next, we can propagate the values in l2 (we can ignore L3 for now since it's unreachable), to get $z1 = x + y = 3$.

We then continue to L4 and have that the only value assigned to $z3$ is $z1 = 3$, so $w = z3 * 2 = 3 * 2 = 6$.

The final simplified program is thus:

```
1   L1:
2     x <- 1
3     y <- 2
4     goto L2
5   L2:
6     z1 <- 3
7     goto L4
8   L4:
9     w <- 6
10    return 6
```

Tl;dr, CCP removes the unreachable branch l3 and enables further constant folding, dead code elimination, and merging basic blocks.

Aggressive Dead Code Elimination (ADCE)

Standard dead code elimination removes instructions whose results are unused. However, it can miss opportunities because it assumes all code is reachable.

Aggressive Dead Code Elimination (ADCE) takes a more aggressive approach than standard dead code elimination, where we attempt to identify pure statements that are not used previously, where we now assume all instructions are dead unless proven otherwise, and we only keep instuctions that are necessary for observable behavior.

Checkpoint 6

(a) What kinds of instructions must always be considered live initially?

(b) How does liveness propagate backwards through the program?

Solution:

(a) Instructions with side effects must always be live. This includes:

- `return` statements

- memory stores
- function calls (may have side effects, we can use purity analysis to determine that some functions do not have side effects)
- I/O operations

(b) Liveness propagates backwards:

- If an instruction is live, then all instructions that define its operands must also be live
- This follows use-def chains in SSA form
- Additionally, it's useful to compute the control dependence

Checkpoint 7

Apply aggressive dead code elimination (ADCE) to the following program. Which instructions remain live, and what simplified program do we get?

```
1   L1:
2     x <- 5
3     y <- 10
4     if x < y then L2 else L3
5
6   L2:
7     a <- x + y
8     b <- a * 2
9     goto L4
10
11  L3:
12    c <- x - y
13    d <- magic(c)
14    goto L4
15
16  L4:
17    return 0
```

Solution: ADCE starts by assuming everything is dead except instructions that are necessary for observable behavior.

Initially, our worklist consists of `return 0` and `d <- magic(c)`.

`return 0` has no uses.

Considering, `d <- magic(c)`, we find that now `c <- x-y` needs to be live, then `if x < y then L2 else L3`, `x <- 5`, and `y <- 10` must all be live as well.

Hence our final program is:

```
1   L1:
2     x <- 5
3     y <- 10
4     if x < y then L2 else L3
5
6   L2:
7     goto L4
8
9   L3:
10    c <- x - y
11    d <- magic(c)
```

```
12      goto L4
13
14    L4:
15      return 0
```

Optimization Interaction

As a hint, it can often be useful to run optimizations multiple times, in various orders!

Checkpoint 8

Why?

Solution: Certain optimizations can enable other optimizations to make further progress. For example, constant propgation enables dead code elimination, which can then expose more constant propagation, and so forth. Alternatively, loop optimizations (covered in the next recitation) can also expose more simplications for earlier optimizations to eliminate.