15-411: Compiler Design — Spring 2024

Recitation 8: Optimizations Solutions — 22 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 like SSA. 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 $\texttt{t1} = 42$; 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 unecessary 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 add `t ← z ⊕ b` to l1, and replace both y2 and u with t.

Dataflow Analysis on Basic Blocks

Recall the liveness rules in set format presented during recitation 2:

$$\text{LiveIn}(\ell) = \text{Uses}(\ell) \cup (\text{LiveOut}(\ell) - \text{Defs}(\ell))$$
$$\text{LiveOut}(\ell) = \bigcup_{s \in \text{succ}(\ell)} \text{LiveIn}(s)$$

To solve liveness, simply apply these rules repeatedly until the fixpoint is reached. This simple dataflow analysis algorithm operates across individual instructions. All programs, however, have more individual instructions than basic blocks, so dataflow across basic blocks would be a lot more efficient. First, let's just take the rules above and simply replace all instances of lines/instructions with basic blocks:

$$\text{LiveIn}(b) = \text{Uses}(b) \cup (\text{LiveOut}(b) - \text{Defs}(b))$$
$$\text{LiveOut}(b) = \bigcup_{s \in \text{succ}(b)} \text{LiveIn}(b)$$

The challenge arises in figuring out what the sets $\text{Uses}(b)$ and $\text{Defs}(b)$ should be. An intuitive guess is to simply take the union the uses and defs respectively for all lines of the block. While that would actually work for $\text{Defs}(b)$, $\text{Uses}(b)$ is a little more subtle. A variable that is only used after being defined in the same block is not being used *overall* by the whole block. This motivates the per-block sets[1]

$\text{Uses}(b) :=$ all variables v in b such that v is used before its first definition
$\text{Defs}(b) :=$ all variables defined in b

To actually keep dataflow over basic blocks fast, these sets should be computed for each basic block during preprocessing. There are several different ways, but one of them is a simple forward pass:

```
1  Uses(b) = {}
2  Defs(b) = {}
3  for line/instruction ℓ in b going forward:
4    Uses(b) = (Uses(b) ∪ Uses(ℓ)) − Defs(b)
5    Defs(b) = Defs(b) ∪ Defs(ℓ)
```

Once these sets are computed, we can apply the 2 liveness rules above, but for blocks instead of instructions, until fixpoint is reached. This will yield LiveIn and LiveOut sets for each basic block. While we have verbally explained the (Kildall's) algorithm before, here it is in pseudocode:

```
1  for basic block b in program:
2    LiveOut(b) = {}
3
4  changed = true
5  while (changed):
6    changed = false
7
8    for basic block b in program:
9      LiveOut(b) = ⋃_{s∈succ(b)} LiveIn(b)
10     oldLiveIn = LiveIn(b)
11     LiveIn(b) = Uses(b) ∪ (LiveOut(b) − Defs(b))
12
13     if oldLiveIn ≠ LiveIn(b):
14       change = true
```

[1] In literature, Uses and Defs are often respectively called Gen and Kill instead. They also often have slightly different definitions but would still converge to the same liveness solution in the end.

A known optimization to make this specific dataflow converge faster is to always iterate over the basic blocks (in the inner for loop) in postorder (DFS finish order) on the CFG.

Once we have the LiveOut sets for every basic block, we can easily compute LiveOut sets for every instruction by applying the original 2 rules:

$$\text{LiveIn}(\ell) = \text{Uses}(\ell) \cup (\text{LiveOut}(\ell) - \text{Defs}(\ell))$$
$$\text{LiveOut}(\ell) = \bigcup_{s \in \text{succ}(\ell)} \text{LiveIn}(s)$$

We simply set the LiveOut set of the last instruction of a basic block to the LiveOut set of the whole block, and do a backwards pass to compute the LiveOut sets of each instruction in the basic block.

Checkpoint 4

On written homework 1, you were asked to do liveness analysis on the following program.

```
label .entry
L00: t0 <- 42 // "input"
L01: t1 <- 6
L02: t2 <- t0 * t1
L03: t3 <- 2
L04: t4 <- t2 - t3
L05: t5 <- 1
L06: t6 <- 0
L07: t7 <- 1

label .loop
L08: t4 <- t4 >> t5
L09: t6 <- t6 + t7
L10: branch t4 .loop .exit

label .exit
L11: ret t6
```

Do it again, but using dataflow over basic blocks instead.

Solution: Preprocessing:

- entry
  Uses(entry): –
  Defs(entry): t0, t1, t2, t3, t4, t5, t6, t7
- loop
  Uses(loop): t4, t5, t6, t7
  Defs(loop): t4, t6
- exit
  Uses(exit): t6
  Defs(exit): –

Dataflow on Blocks:

- entry
  LiveIn(entry): –
  LiveOut(entry): t4, t5, t6, t7
- loop
  LiveIn(loop): t4, t5, t6, t7
  LiveOut(loop): t4, t5, t6, t7
- exit
  LiveIn(exit): t6
  LiveOut(exit): –

LiveOuts:

- L0: t0
- L1: t0, t1
- L2: t2
- L3: t2, t3
- L4: t4
- L5: t4, t5
- L6: t4, t5, t6
- L7: t4, t5, t6, t7
- L8: t4, t5, t6, t7
- L9: t4, t5, t6, t7
- L10: t4, t5, t6, t7
- L11: –