Pre-class Puzzle

For each of these Control Flow Graphs (CFGs), what is a C program that corresponds to it?

A

B

C

D

Take some time to try to figure these out---I’ll ask for volunteers to share their answers around 10:05
CS443: Compiler Construction
Lecture 15: Loop Optimization
Stefan Muller
Based on material by Steve Zdancewic, Stephen Chong and Greg Morrisett
Announcements

• Project 4 due 10/20 (1 week from today)
  • Start early!
  • Partners may be helpful

• Project 3 will be graded next week

• OH Today
  • Online (obviously), usual Monday Zoom link
  • 3-4 pm instead of 2-3 pm
Loop optimizations are especially important!

• Programs spend most of the time in loops
• Lots of loop optimizations:
  • Loop invariant removal
  • Induction variable elimination
  • Loop unrolling
  • Loop fusion
  • Loop fission
  • Loop peeling
  • Loop interchange
  • Loop tiling
  • Loop parallelization
  • Software pipelining
Invarient removal: Don’t recompute things in a loop

l0:
  %i = bitcast i32 0 to i32
  br %l1
l1:
  %i = add i32 %i 1
  %t = add i32 %a %b
  %el = getelementptr i32, i32* %arr, i32 %i
  store i32 %t, i32* %el
  %lt = icmp lt i32 %i %N
  br i1 %lt, label %l1, label %l2
l2:
  ret %t
Invariant removal: Don’t recompute things in a loop

l0:
  %i = bitcast i32 0 to i32
  %t = add i32 %a %b
  br %l1
l1:
  %i = add i32 %i 1
  %el = getelementptr i32, i32* %arr, i32 %i
  store i32 %t, i32* %el
  %lt = icmp lt i32 %i %N
  br i1 %lt, label %l1, label %l2
l2:
  ret %t
Loop induction variable: Avoid recomputation based on loop induction variables

l0:
  %i = bitcast i32 0 to i32
l1:
  \boxed{%t1 = mul i32 %i 4}  
  %t2 = add i32 %a %t1  
  %s = add i32 %s %t2  
  %i = add i32 %i 1  
  %lt = icmp lt %i 100  
  br i1 %lt, label %l1, label %l2
l2: ...  

\textbf{t1 is always equal to i * 4}
Loop induction variable: Avoid recomputation based on loop induction variables

l0:
    %i = bitcast i32 0 to i32
    %t1 = bitcast i32 0 to i32
l1:
    %t1 = add i32 %t1 4
    %t2 = add i32 %a %t1
    %s = add i32 %s %t2
    %i = add i32 %i 1
    %lt = icmp lt %i 100
    br i1 %lt, label %l1, label %l2
l2: ...
Loop induction variable: Avoid recomputation based on loop induction variables

l0:
  \%i = bitcast i32 0 to i32
  \%t1 = bitcast i32 0 to i32
  \%t2 = bitcast i32 \%a to i32
l1:
  \%t1 = add i32 \%t1 4
  \%t2 = add i32 \%t2 \%4
  \%s = add i32 \%s \%t2
  \%i = add i32 \%i 1
  \%lt = icmp lt \%i 100
  br i1 \%lt, label \%l1, label \%l2
l2: ...

Can eliminate \%t1!
Loop induction variable: Avoid recomputation based on loop induction variables

l0:
   %i = bitcast i32 0 to i32
   %t2 = bitcast i32 %a to i32
l1:  %t2 = add i32 %t2 %4
    %s = add i32 %s %t2
    %i = add i32 %i 1
    %lt = icmp lt %i 100
    br i1 %lt, label %l1, label %l2
l2:  ...
Loop induction variable: Avoid recomputation based on loop induction variables

l0:
  %i = bitcast i32 0 to i32
  %t2 = bitcast i32 %a to i32
  %endt2 = add i32 %a 400
l1:
  %t2 = add i32 %t2 %4
  %s = add i32 %s %t2
  %lt = icmp lt %t2 %endt2
  br i1 %lt, label %l1, label %l2
l2: ...
Before we can optimize loops, we have to find them

- In C: easy!
- In LLVM: surprisingly hard!

**Definition** (loop):
- Subset S of nodes in CFG
- Designated "header" node h
- S is strongly connected
- No edge from outside S to S \ {h}
Loops can be nested
Non-example: (there can’t be a header)

- (But this can’t arise in C/C++ without goto)
A node $d$ dominates $n$ if every path (from start) to $n$ must go through $d$
We can define loops based on dominators

- A **back edge** is an edge from \( n \) to a dominator \( d \)
- If there’s a back edge \( n \rightarrow d \), there is a **loop** consisting of the set of nodes \( x \) such that \( d \) dominates \( x \) and there is a path from \( x \) to \( n \) not including \( d \)
Example

- For each node, what nodes does it dominate? Back edges?
  - a dominates a, b, c, d, e, f, g, h
  - b dominates b, c, d, e, f, g, h
  - c dominates c, e
  - d dominates d
  - f dominates f, g, h
  - g dominates g, h
  - h dominates h,
  - Back edges: g -> b, h -> a
We can calculate dominators with a dataflow analysis!

• $\text{out}[n] =$ set of nodes that dominate $n$
  • $\text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']$
  • $\text{out}[n] := \text{in}[n] \cup \{n\}$

• Forward must analysis: initialize $\text{out}[n]$, $\text{in}[n]$ to all nodes
We can represent dominators with a “dominator tree”

- Edge to n from its “immediate dominator” (dominator other than n that is dominated by other dominators other than n)
Identifying loop invariants

• An instruction $\%x = \text{opc op1, op2, \ldots, opN}$ represented by a node $n$ is invariant for a loop if for each operand $\text{op}_i$:
  • $\text{op}_i$ is constant, or
  • all definitions of $\text{op}_i$ that reach $n$ are outside the loop, or
  • only one definition reaches $\text{op}_i$ and it is a loop invariant
Loop invariant example from before

l0:
  %i = bitcast i32 0 to i32
  br %l1
l1:
  %i = add i32 %i 1
  %t = add i32 %a %b
  %el = getelementptr i32, i32* %arr, i32 %i
  store i32 %t, i32* %el
  %lt = icmp lt i32 %i %N
  br i1 %lt, label %l1, label %l2
l2:
  ret %t
Actually moving (\textit{hoisting}) invariants out of the loop is pretty tricky

- Move to a “pre-header” (CFG node before header)
Need to make sure hoisting wouldn’t interfere with other uses!

l0:
    %i = bitcast i32 0 to i32
    br %l1
l1:
    %i = add i32 %i 1
    %el = getelementptr i32, i32* %arr, i32 %i
    store i32 %t, i32* %el
    %t = add i32 %a %b
    %lt = icmp lt i32 %i %N
    br i1 %lt, label %l1, label %l2
l2:
    ret %t
Need to make sure hoisting wouldn’t interfere with other uses!

- $n := \%x = \text{opc\ op1, op2, ...}, \text{opN}$ is safe to hoist if:
  - $n$ dominates all loop exits at which $\%x$ is live, and
  - there is only one definition of $x$ in the loop, and
  - $x$ is not live at the pre-header

```c
for (i = 0; i < 100; i += 2) {
    t = a + b;
    a[i] = t;
    t = a - b;
    a[i + 1] = t;
}
```

```c
t = 0;
while(1) {
    break;
    t = a + b;
    a[i] = t;
}
return t;
```
Loop Unrolling: Copy over the body of a loop

```
for (int i = 0; i < n; i++) {
    a[i] = i;
}

//Handle the first few in case n not a multiple of 3
for (int i = 0; i < n % 3; ++i) a[i] = i;
for (; i < n; i+=3) {
    a[i] = i;
    a[i + 1] = i + 1;
    a[i + 2] = i + 2;
}
```

Why is this an optimization?
Loop unrolling: costs and benefits

• Benefits:
  • Amortize tests, jumps over more instructions

• Costs:
  • Program size increases (why is this a problem?)
Loop Peeling: “Peel off” the first or last N iterations of a loop

```java
for (int i = 0; i < N; i++) {
    if (i <= 1) {
        a[i] = i;
    } else {
        a[i] = a[i - 1] + a[i - 2];
    }
}
```

```java
a[0] = 0;
a[1] = 1;
for (int i = 2; i < N; i++) {
    a[i] = a[i - 1] + a[i - 2];
}
```
Loop Interchange: Swap order of nested loops

```java
for (int i = 0; i < w; i++) {
    for (int j = 0; j < h; j++) {
        sum += a[j][i];
    }
}
```

```java
for (int j = 0; j < h; j++) {
    for (int i = 0; i < w; i++) {
        sum += a[j][i];
    }
}
```

Why is this an optimization?