CSE 5317

Lecture 20: Optimization
6 April 2010

Nate Nystrom
UTA
Optimization
Constant folding

- Very simple optimization
- Replace 1+2 with 3
- Often done on the AST
Constant folding implementation

Node visit(Add n) {
    Node e1 = n.left.accept(this);
    Node e2 = n.right.accept(this);
    if (e1 instanceof IntLit && e1 instanceof IntLit) {
        int v1 = ((IntLit) e1).value;
        int v2 = ((IntLit) e2).value;
        return new IntLit(v1 + v2);
    }
    return new Add(v1, v2);
}
Constant propagation

Replace temporaries with constants

\[ x = 3 \]
\[ y = 4 \]
\[ z = x \times y \]

\[ x = 3 \]
\[ y = 4 \]
\[ z = 12 \]
Constant propagation dataflow

• Similar to copy propagation analysis:
  • forward, must (all paths)
  • does variable $x$ contain constant $k$ on all paths to this point?

• But different lattice:
  • map every variable to a constant:
  • sets of (variable, constant) pairs

• What’s the ordering?
Constant propagation lattice

Map each variable to a member of the lattice:
Constant propagation analysis

• Initialization:
  • Every variable maps to unknown

• Meet operation:
  • Do a meet on the lattice for each variable:
    • \texttt{unknown} \cap 0 = 0
    • 1 \cap 1 = 1
    • 2 \cap 3 = \texttt{not\_a\_constant}
    • 4 \cap \texttt{not\_a\_constant} = \texttt{not\_a\_constant}
Incorporating folding

- Transfer function should incorporate constant folding

\{
  x = 1, \ y = 2
\}

\[
z = x + y
\]

\{
  x = 1, \ y = 2, \ z = 3
\}
Constant propagation transformation

• Do analysis.

• Traverse IR:
  • if variable maps to a constant value, substitute constant in the instruction:

\[
\begin{align*}
\{ & \ y = \text{NOT\_CONSTANT}, \ z = 3 \ \\
& \ x = y + z \\
\} \\
\rightarrow \\
& \ x = y + 3
\end{align*}
\]
Profitable?

• Almost always!
  • Removes temporaries – reducing register pressure
  • Shortens live rangers – reducing register pressure

• Creates dead, unreachable code:
  • if (false) ...
  • while (false) ...
Constant propagation for loops

\[
x = 0;
while (x > 10) S;
\]

\[
->
\]

\[
while \ (false) \ S;
\]

• Dead code elimination can now remove the loop.
Dead code elimination

• Do liveness analysis

• If variable not live, remove its definition
• This removes uses of variables on the rhs:
  • ex: removing \( x = y + z \), removes uses of \( y \) and \( z \)
• Repeat!
  • ex: this may remove definitions of \( y \) and \( z \) also

• Remove unreachable code
Interprocedural copy propagation

Copy propagation very valuable when combined with inlining

```c
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n * fact(n-1);
}
```

```c
x = fact(3);  
if (n <= 1)  x = 1;
else x = n * fact(n-1);
```

```c
x = 3 * fact(2);
if (3 <= 1)  x = 1;
else x = 3 * fact(2);
```

```c
x = 3 * fact(2);
x = 3 * 2 * fact (1);
```

```c
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n * fact(n-1);
}
```

```c
x = fact(3);  
if (n <= 1)  x = 1;
else x = n * fact(n-1);
```

```c
x = 3 * fact(2);
if (3 <= 1)  x = 1;
else x = 3 * fact(2);
```

```c
x = 3 * fact(2);
x = 3 * 2 * fact (1);
```

```c
x = 6 * fact(1);
x = 6 * fact(1);
x = 6 * 1;
x = 6;
```

```c
x = 6 * fact(1);
x = 6 * 1;
x = 6;
```
Reaching definitions

• Every assignment is a definition.

• A definition $d$ reaches a point $p$ if there exists a path from the point immediately after $d$ to $p$ where $d$ is not killed.
Reaching definitions

• Dataflow:
  • lattice: sets of definitions
  • meet = union
  • transfer function: out[i] = (in[i] - kill[i]) U gen[i]
  • gen[i] = {i}
  • kill[i] = { d | target(d) = target(i) }
Chains

• Many optimizations use the following data structures:

  • ud chain
  • maps a use of variable $x$ to a list of all reaching definitions of $x$

  • du chain
  • maps a def of variable $x$ to a list of all uses of that def

• Can build directly from reaching definitions

• ud chains are the central data structure of many analyses and optimizations
Constant propagation with ud chains

- For each use u, compute $\text{const}(u)$:
  - $\text{const}(u) = \bigcap \{ \text{constDef}(d) \mid d \text{ in ud-chain}(u) \}$
  - $\text{constDef}(d) = c$ if $d$ is “$x = c$”
  - $\text{constDef}(d) = c$ if $d$ is “$x = y + z$” and $c = \text{const}(y) + \text{const}(z)$

- where + on lattice elements is defined to propagate 
  not_a_constant and unknown:
  ex: $c + \text{not_a_constant} = \text{not_a_constant}$

- other instructions are similar

- Replace $u$ with $\text{const}(u)$ if $\text{const}(u)$ is constant
Loop optimizations

• Most time in programs spent in loops – duh!

• Important to make loop bodies as fast as possible

• Variety of optimizations to do this:
  • code hoisting (e.g., partial redundancy elimination–last time)
  • loop transformations (next time)
  • strength reduction
  • induction variable elimination
Loop structure

To identify loops:
DFS on the CFG to find strongly connected components.

- **preheader**: predecessor of the header
- **header**: target of the back edge
- **back edge**: exit of the loop
Loop optimizations

• Today:
  • strength reduction
  • induction variable elimination

• Both require identifying **induction variables**
Induction variables

- Variables whose values form a **linear progression**:
  - 0, 4, 8, 12, 16, ...

- Useful for:
  - strength reduction
  - induction variable elimination
  - loop transformations

- Simple approach:
  - Look for instructions $i = i + c$ where there are no other assignments to $i$ in the loop
  - This doesn’t catch all IVs. Why not?
Induction variable identification

• Basic induction variables (e.g., loop index)
  • defined once in a loop by statement of the form:
    \[ i = i + c, \text{ where } c \text{ is a constant integer} \]

• Derived induction variables
  • defined once in a loop as a \textbf{linear function} of another induction variable:
    \[ k = j + c, \text{ or} \]
    \[ k = d \times j, \text{ where } c \text{ and } d \text{ are loop invariant} \]
Induction variables

```c
for (i = 0; i < n; i++)
    a[i] = 0
```

In IR:

```c
i = 0

// top: if i >= n goto end

t = 4 * i

// p = a + t

[p] = 0

// i = i + 1

i = i + 1

goto top

end:
```
Induction variables

for (i = 0; i < n; i++)
    a[i] = 0

In IR:

i = 0

\text{top: if i} \geq n \text{ goto end}
\text{t} = 4 \ast i
\text{p} = a + t
\text{[p]} = 0
i = i + 1 \quad \leftarrow \text{i is an induction variable}
goto \text{top}
end:
Induction variables

for (i = 0; i < n; i++)
    a[i] = 0

In IR:

i = 0

\textbf{top:} if i >= n goto end

$\text{t} = 4 * \text{i}$ \hspace{2cm} \text{t is an induction variable, } t==4*\text{i}

$p = a + t$

$p] = 0$

$i = i + 1$ \hspace{2cm} \text{i is an induction variable}

goto top

end:
Induction variables

for (i = 0; i < n; i++)
    a[i] = 0

In IR:

i = 0

\textbf{top: if } i \geq n \textbf{ goto end}

\begin{align*}
t & = 4 \times i \\
p & = a + t \\
[p] & = 0 \\
i & = i + 1
\end{align*}

\textbf{goto top}

\textbf{end:}

\textit{t} is an induction variable, \textit{t}==4*i
\textit{p} is an induction variable, \textit{p}==a+4*i
\textit{i} is an induction variable
Induction variable analysis

• Maps each variable $k$ to a term: $c_1 + c_2*i$

• $i$ is a basic induction variable
• $c_1$ and $c_2$ are constants such that $k = c_1 + c_2*i$ when $k$ defined
• “$k$ belongs to the family of $i$”

• Term for a basic induction variable $i$:
  • $0 + 1 * i$
Algorithm

• Given a loop $L$, ud-chains, and loop invariant information

• For each instruction $s$ in $L$:
  • if $s$ is $i = i + c$, create $i \rightarrow 0+c*i$

• For each instruction $s$ in $L$:
  • if $s$ is $k = j + c$ or $k = j * d$
    and $j \rightarrow a+b*i$
    and $c$ and $d$ are loop invariant
    and $k$ defined only once in $L$
    and if $j$ is derived ($i! = j$), then
  • only def of $j$ that reaches $k$ must be in $L$
  • no def of $i$ occurs on any path from def of $j$ to $k$
  • then, create $k \rightarrow (a+c)+b*i$ or $k \rightarrow (a*d)+(b*d)*i$
Algorithm

\begin{align*}
i &= 0 \\
top: & \text{ if } i \geq n \text{ goto end} \\
t &= 4 \times i \\
p &= a + t \\
[p] &= 0 \\
i &= i + 1 \\
goto & \text{ top} \\
end:
\end{align*}

Computes:

\begin{align*}
i &\rightarrow 0 + 1 \times i \\
t &\rightarrow 0 + 4 \times i \\
p &\rightarrow a + 4 \times i
\end{align*}
Strength reduction

• For each derived IV $j \rightarrow a+b*i$ in loop $L$
  where $i \rightarrow 0+c*i$
  
• note: $j$ is incremented by $b*c$ each iteration
• create new IV $k$
• put $t = b*c$ in loop preheader
• put $k = a+b*i$ at end of loop preheader
• after each def $i = i+c$ in $L$, add $k = k+t$
• replace def of $j$ with $j = k$

• Note: multiplication has been moved out of the loop
i = 0

top: if i >= n goto end

i = i + 1
goto top

end:

i = 0
t = 0
p = a

[p] = 0

i = i + 1
t = t + 4
p = p + 4

goto top

dend:
Dead code elimination

\[
i = 0 \\
top: \text{ if } i \geq n \text{ goto end} \\
t = 4 \times i \\
p = a + t \\
[p] = 0 \\
i = i + 1 \\
goto \text{ top} \\
end: \\
\]

\[
i = 0 \\
p = a \\
top: \text{ if } i \geq n \text{ goto end} \\
[p] = 0 \\
i = i + 1 \\
p = p + 4 \\
goto \text{ top} \\
end: \\
\]
Profitable?

- Can introduce more temporaries
  - increasing register pressure

- Can eliminate expensive operations
  - on x86:
    - latency of multiply instruction is 3-14 cycles
    - good to replace multiply with shifts and adds

  - `imul` writes to both `%eax` and `%edx`
  - removing an `imul` instruction can put less constraints on the register allocator
Shifts and adds

• If strength reduction cannot eliminate multiply, can replace multiply with shifts and adds:

• To optimize:

\[ x = y \times k \]

if \[ k = 2^{31}a_{31} + \ldots + 2^{2}a_{2} + 2^{1}a_{1} + 2^{0}a_{0} \]

where \( a_{i} \) is 0 or 1

• generate:

\[ x = a_{31}(y \ll 31) + \ldots + a_{2}(y \ll 2) + a_{1}(y \ll 1) + a_{0}y \]

\[ t_{1} = y \ll 4 \]
\[ t_{2} = y \ll 2 \]
\[ x = t_{1} + t_{2} \]
IV elimination

• For each basic IV i
  • if only uses are to compute other IVs and in conditional branches
    • choose \( j \rightarrow c+d*i \) – preferably with \( c = 0 \)
    • modify conditional involving \( i \) to use \( j \)
      \[
      \text{if } i < x \text{ goto } L \quad \rightarrow \quad \text{if } j < c+d*x \text{ goto } L
      \]
  • delete all assignments to \( i \)

• Apply copy propagation and dead code elimination to clean up

• Remove any IV def where the IV is only used and defined within that definition
i = 0
p = a
top: if i >= n goto end
[p] = 0
i = i + 1
p = p + 4
goto top
der:

top: if p >= N goto end
[p] = 0
i = i + 1
p = p + 4
goto top
der:
i = 0
p = a
N = a + 4*n
top: if p >= N goto end
[p] = 0
i = i + 1
p = p + 4
goto top
depth:
p = a
N = a + 4*n
top: if p >= N goto end
[p] = 0
p = p + 4
goto top
depth: