GLOBAL OPTIMIZATION

Baishakhi Ray

Fall 2020

These slides are motivated from Prof. Alex Aiken and Prof. Calvin Lin
Other Global Optimization:

- Constant Propagation
- Dead-code elimination
- Liveness analysis
- Common subexpression elimination
- Loop optimization
Local Optimization

- Recall the simple basic-block optimizations
  - Constant propagation
  - Dead code elimination

\[
\begin{align*}
X &= 3 \\
Y &= Z \times W \\
Q &= X + Y
\end{align*}
\]

\[
\begin{align*}
X &= 3 \\
Y &= Z \times W \\
Q &= 3 + Y
\end{align*}
\]

\[
\begin{align*}
Y &= Z \times W \\
Q &= 3 + Y
\end{align*}
\]
Global Optimization

- These optimizations can be extended to an entire control-flow graph.
These optimizations can be extended to an entire control-flow graph.
Correctness

- How do we know it is OK to globally propagate constants?
- There are situations where it is incorrect:

\[
\begin{align*}
X &:= 3 \\
B &> 0 \\
Y &:= Z + W \\
X &:= 4 \\
A &:= 2 \times X \\
Y &:= 0
\end{align*}
\]
Correctness (cont..)

To replace a use of \( x \) by a constant \( k \) we must know that:

On every path to the use of \( x \), the last assignment to \( x \) is

\[
x := k
\]

- The correctness condition is not trivial to check
- “All paths” includes paths around loops and through branches of conditionals
- Checking the condition requires global analysis
  - An analysis of the entire control-flow graph
Global Analysis

- Global optimization tasks share several traits:
  - The optimization depends on knowing a property $X$ at a particular point in program execution
  - Proving $X$ at any point requires knowledge of the entire program
  - It is OK to be conservative. If the optimization requires $X$ to be true, then want to know either
    - $X$ is definitely true
    - Don’t know if $X$ is true
    - It is always safe to say “don’t know”
Global Analysis (cont..)

- Global dataflow analysis is a standard technique for solving problems with these characteristics.

- Global constant propagation is one example of an optimization that requires global dataflow analysis.
To make the problem precise, we associate one of the following values with X at every program point:

<table>
<thead>
<tr>
<th>value</th>
<th>interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥ (&quot;bottom&quot;)</td>
<td>This statement never executes</td>
</tr>
<tr>
<td>c</td>
<td>X = constant c</td>
</tr>
<tr>
<td>T (&quot;top&quot;)</td>
<td>X is not a constant</td>
</tr>
</tbody>
</table>
Example

\[
\begin{align*}
    Y & := Z + W \\
    X & := 4
\end{align*}
\]

\[
\begin{align*}
    A & := 2 \times X
\end{align*}
\]

\[
\begin{align*}
    X & := 3 \\
    B & > 0
\end{align*}
\]

\[
\begin{align*}
    Y & := 0
\end{align*}
\]

\[
\begin{align*}
    X & := 4
\end{align*}
\]
Using the Information

- Given global constant information, it is easy to perform the optimization
  - Simply inspect the $x = ?$ associated with a statement using $x$
  - If $x$ is constant at that point replace that use of $x$ by the constant

- But how do we compute the properties $x = ?$
Using the Information

- The idea is to “push” or “transfer” information from one statement to the next.

- For each statement $s$, we compute information about the value of $x$ immediately before and after $s$.

  \[
  C(s,x,\text{in}) = \text{value of } x \text{ before } s \\
  C(s,x,\text{out}) = \text{value of } x \text{ after } s
  \]
Define a transfer function that transfers information one statement to another

In the following rules, let statement $s$ have immediate predecessor statements $p_1, \ldots, p_n$
Rule 1

if $C(p_i, x, \text{out}) = T$ for any $i$, then $C(s, x, \text{in}) = T$
Rule 2

\[ C(p_i, x, \text{out}) = c \land C(p_j, x, \text{out}) = d \land d \neq c \]

then \[ C(s, x, \text{in}) = T \]
Rule 3

If \( C(p_i, x, \text{out}) = c \) or \( \perp \) for all \( i \),
then \( C(s, x, \text{in}) = c \)
Rule 4

if $C(p_i, x, \text{out}) = \bot$ for all $i$, then $C(s, x, \text{in}) = \bot$
- Rules 1-4 relate the out of one statement to the in of the next statement
- Now we need rules relating the in of a statement to the out of the same statement
Rule 5

\[
C(s, x, \text{out}) = \bot \\
\text{if } C(s, x, \text{in}) = \bot
\]
Rule 6

C(x := c, x, out) = c if c is a constant
Rule 7

\[
\begin{align*}
\text{C}(x := f(...), x, \text{out}) &= T \\
x &= ? \\
x &= T
\end{align*}
\]
Rule 8

\[ y := \ldots \]

\[ x = a \]

\[ C(y := \ldots, x, \text{out}) = C(y := \ldots, x, \text{in}) \text{ if } x \leftrightarrow y \]
Algorithm

1. For every entry $s$ to the program, set $C(s, x, \text{in}) = T$

2. Set $C(s, x, \text{in}) = C(s, x, \text{out}) = \perp$ everywhere else

3. Repeat until all points satisfy 1-8:
   - Pick $s$ not satisfying 1-8 and update using the appropriate rule

**Ordering:**
- We can simplify the presentation of the analysis by ordering the values $\perp < c < T$
Common subexpression elimination

- Example:
  
  \[
  \begin{align*}
  a &:= b + c \\
  c &:= b + c \\
  d &:= b + c \\
  \end{align*}
  \Rightarrow
  \begin{align*}
  a &:= b + c \\
  c &:= a \\
  d &:= b + c \\
  \end{align*}
  \]

- Example in array index calculations
  
  - \( c[i+1] := a[i+1] + b[i+1] \)
  - During address computation, \( i+1 \) should be reused
  - Not visible in high level code, but in intermediate code
Code Elimination

- **Unreachable code elimination**
  - Construct the control flow graph
  - Unreachable code block will not have an incoming edge
  - After constant propagation/folding, unreachable branches can be eliminated

- **Dead code elimination**
  - Ineffective statements
    - $x := y + 1$ (immediately redefined, eliminate!)
    - $y := 5 \quad \Rightarrow \quad y := 5$
    - $x := 2 \ast z$  $x := 2 \ast z$
  - A variable is dead if it is never used after last definition
    - Eliminate assignments to dead variables
  - Need to do data flow analysis to find dead variables
Function Optimization

- **Function inlining**
  - Replace a function call with the body of the function
  - Save a lot of copying of the parameters, return address, etc.

- **Function cloning**
  - Create specialized code for a function for different calling parameters
Loop Optimization

- Loop optimization
  - Consumes 90% of the execution time
    ⇒ a larger payoff to optimize the code within a loop

- Techniques
  - Loop invariant detection and code motion
  - Induction variable elimination
  - Strength reduction in loops
  - Loop unrolling
  - Loop peeling
  - Loop fusion
Loop Optimization

- Loop invariant detection
  - If the result of a statement or expression does not change within a loop, and it has no external side-effect
  - Computation can be moved to outside of the loop
  - Example
    
    ```
    for (i=0; i<n; i++) a[i] := a[i] + x/y;
    ```
  
    - Three address code
      
      ```
      for (i=0; i<n; i++) { c := x/y; a[i] := a[i] + c; }
      ⇒ c := x/y;
      for (i=0; i<n; i++) a[i] := a[i] + c;
      ```
Loop Optimization

- **Code Motion**
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```c
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
```

```c
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```
Loop Optimization

- **Strength reduction in loops**
  - Replace costly operation with simpler one
  - Shift, add instead of multiply or divide
    \[ 16 \times x \rightarrow x \ll 4 \]
    - Depends on cost of multiply or divide instruction
  - Recognize sequence of products

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

```c
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```
Loop Optimization

- **Strength reduction in loops**
  - Replace costly operation with simpler one
  - Shift, add instead of multiply or divide
    
    \[ 16 \times x \rightarrow x \ll 4 \]
    
    - Depends on cost of multiply or divide instruction
  - Recognize sequence of products

```c
s := 0;
for (i=0; i<n; i++)
{
  v := 4 * i;
  s := s + v;
}
```

```c
s := 0;
for (i=0; i<n; i++)
{
  v := v + 4;
  s := s + v;
}
```
Loop Optimization

- **Induction variable elimination**
  - If there are multiple induction variables in a loop, can eliminate the ones which are used only in the test condition.
  - Example
    
    ```
    s := 0;  for (i=0; i<n; i++) { s := 4 * i; … }   -- i is not referenced in loop
    ⇒ s := 0;  e := 4*n; while (s < e) { s := s + 4; }
    ```

```
  s := 0;  for (i=0; i<n; i++)
            { s := 4 * i; … }   -- i is not referenced in loop
```
Code Optimization Techniques

- **Loop unrolling**
  - Execute loop body multiple times at each iteration
  - Get rid of the conditional branches, if possible
  - Allow optimization to cross multiple iterations of the loop
    - Especially for parallel instruction execution
  - Space time tradeoff
    - Increase in code size, reduce some instructions

- **Loop peeling**
  - Similar to unrolling
  - But unroll the first and/or last few iterations
Loop Optimization

- **Loop fusion**
  
  - Example
    
    ```
    for i=1 to N do
        A[i] = B[i] + 1
    endfor
    
    for i=1 to N do
        C[i] = A[i] / 2
    endfor
    
    for i=1 to N do
        D[i] = 1 / C[i+1]
    endfor
    ```

  Before Loop Fusion

```python
for i=1 to N do
    A[i] = B[i] + 1
    C[i] = A[i] / 2
    D[i] = 1 / C[i+1]
endfor
```
Loop Optimization

- **Loop fusion**
  - **Example**
    ```
    for i = 1 to N do
      A[i] = B[i] + 1
    endfor
    for i = 1 to N do
      C[i] = A[i] / 2
    endfor
    for i = 1 to N do
      D[i] = 1 / C[i+1]
    endfor
    ```

    Before Loop Fusion

    Is this correct? Actually, cannot fuse the third loop
Limitations of Compiler Optimization

- Operate Under Fundamental Constraint
  - Must not cause any change in program behavior under any possible condition
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.

- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., data ranges may be more limited than variable types suggest

- Most analysis is performed only within procedures
  - whole-program analysis is too expensive in most cases

- Most analysis is based only on static information
  - compiler has difficulty anticipating run-time inputs

- When in doubt, the compiler must be conservative