Data Structures and Algorithms in Compiler Optimization

Comp314 Lecture
Dave Peixotto
What is a compiler

* Compilers translate between program representations
* Interpreters evaluate their input to produce a result
* Writing a compiler makes large use of different data structures such as graphs, trees, and sets
High Level Organization

Source Program → Front End → IR → Optimizer → IR → Back End → Target Program
Data Structures

* Front End
  * Abstract Syntax Tree
    * Created during parsing and usually replaced by another IR that is better for analysis
  * IR
    * Control Flow Graph
      * Usually remains throughout the life of the compiler
* Back End
  * Data flow analysis Sets
    * Created and destroyed during compiler optimization. Attached to nodes of the CFG to hold facts about the program at that point in the graph
Intermediate Representations

* Graph Based - the program is represented as a graph.
  * Example: AST, DAG

* Linear - the program is represented as a straight line sequence of instructions
  * Example: assembly code with jumps and branches for an abstract machine

* Hybrid - the program is represented as a combination of linear and graph structures.
original code

\[
 q = a + d; \\
\text{if false then} \\
 e = b + 8; \\
\text{else} \\
 e = a + 7; \\
v = a + d;
\]

AST IR

Parsing
Linear IR

original code

q = a + d;
if false then
  e = b + 8;
else
  e = a + 7;
v = a + d;

Translation

ADD q a d
LDi c 1
BReq A B c
A:
ADDi e b 8
JMP C
B:
ADDi e a 7
JMP C
C:
ADD v a d
Hybrid IR
(Control Flow Graph)

original code

\[ q = a + d \]

if false then
\[ e = b + 8 \]
else
\[ e = a + 7 \]
\[ v = a + d \]

Parsing + Translation

ADD q a d
LDi c 1
ADDi e b 8
ADDi e a 7
ADD v a d

JMP
c = 0
c = 1
Control Flow Graph

Terminology

* Nodes are basic blocks
* Edges represent control flow instructions
* Basic block - maximal sequence of straight line instructions
* If one instruction in the basic block executes they all execute

Nodes:
- ADD q a d
- ADD b a d
- ADD v a d
- ADDi e b 8
- ADDi e a 7
Extended Basic Blocks

Sequence of basic blocks such that each block (except the first) has a single predecessor

Forms a tree rooted at the entry to the EBB
Extended Basic Blocks

- Sequence of basic blocks such that each block (except the first) has a single predecessor
- Forms a tree rooted at the entry to the EBB
SSA Form
(Static Single Assignment)

A particular way of choosing variable names in the intermediate representation

\[
\begin{align*}
q &= a + d \\
q &= g + h \\
e &= b + 8 \\
e &= a + 7 \\
v &= e + d \\
\end{align*}
\]

\[
\begin{align*}
q_0 &= a_0 + d_0 \\
q_1 &= g_0 + h_0 \\
e_0 &= b_0 + 8 \\
e_1 &= a_0 + 7 \\
e_2 &= \phi(e_0, e_1) \\
v_0 &= e_2 + d_0 \\
\end{align*}
\]
SSA Form

* Two invariants that characterize SSA
  * Each variable is defined (assigned a value) exactly once
  * Every use refers to exactly one definition

* Encodes information about control flow and data flow into the variable names in the program
  * Phi-Nodes indicate join points in the CFG
  * Variable names show where a particular definition is used
Why Use SSA?

* Simplifies compiler optimizations by providing strong links between definitions and uses

* Single transformation can be used for analysis in many data flow problems

* Makes detecting certain errors trivial (e.g. variable used before it is initialized)
Compiler Optimization

* Optimization is a misused word, we are really talking about transformations of the IR

* Scope
  * Local - within a single basic block
  * Superlocal - within an extended block
  * Global - within an entire procedure
  * Interprocedural - between procedures
Example Optimization: Value Numbering

* This optimization finds and eliminates redundant computations (common subexpression elimination)

* Instead of recomputing an answer we save the value and reuse it in a later computation

* This is a standard optimization performed by many compilers
Local Value Numbering

Original Code

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

Value Numbered

\[
\begin{align*}
a^3 &= b^1 + c^2 \\
b^5 &= a^3 - d^4 \\
d^5 &= a^3 - d^4
\end{align*}
\]

Transformed Code

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

Rewrite
Value Numbering

* Technique originally designed for linear IRs, graphical IRs would use a DAG for this optimization instead

* Each expression is assigned a value number

* Value number is computed as a hash of value numbers in the expression and the operands

* Hashtable maps expressions to value
for each instruction
(assume instruction is of the form $x := y \text{ op } z$)
look up the value numbers of $y$ and $z$
build the hash key "$y_{vn} \text{ op } z_{vn}$"
lookup key in the hash
if key in hash
    replace the instruction with a copy operation
    record value number for $x$
else
    add key to hash with a new value number
Local Value Numbering
(a problem)

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Value Numbered</th>
<th>Transformed Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a := b + c )</td>
<td>( a^3 := b^1 + c^2 )</td>
<td>( a := b + c )</td>
</tr>
<tr>
<td>( b := a - d )</td>
<td>( b^5 := a^3 - d^4 )</td>
<td>( b := a - d )</td>
</tr>
<tr>
<td>( b := d + a )</td>
<td>( b^6 := a^3 + d^4 )</td>
<td>( b := a + d )</td>
</tr>
<tr>
<td>( d := a - d )</td>
<td>( d^5 := a^3 - d^4 )</td>
<td>( d := ??)</td>
</tr>
</tbody>
</table>
Local Value Numbering
With SSA

Original Code

\[ a_0 := b_0 + c_0 \]
\[ b_1 := a_0 - d_0 \]
\[ b_2 := d_0 + a_0 \]
\[ d_1 := a_0 - d_0 \]

Value Numbered

\[ a_0^3 := b_0^1 + c_0^2 \]
\[ b_1^5 := a_0^3 - d_0^4 \]
\[ b_2^6 := a_0^3 + d_0^4 \]
\[ d_1^5 := a_0^3 - d_0^4 \]

Transformed Code

\[ a_0 := b_0 + c_0 \]
\[ b_1 := a_0 - d_0 \]
\[ b_2 := d_0 + a_0 \]
\[ d_1 := b_1 \]
Superlocal Value Numbering

- Operates over extended basic blocks (EBBs)
- Allows us to capture more redundant computations
- Treat individual paths through an EBB as if it were a single basic block
Treat each path through the EBB as a basic block
Treat each path through the EBB as a basic block
Value Numbering Over EBBs

- Treat each path through the EBB as a single basic block
- Initialize the hash table from the previous basic block in the path
- Remove the entries from hash table for the basic block when recursing up the path
Value Numbering Example

\[ m_0 = a_0 + b_0 \]
\[ n_0 = a_0 + b_0 \]
\[ p_0 = c_0 + d_0 \]
\[ r_0 = d_0 + c_0 \]

\[ q_0 = a_0 + b_0 \]
\[ r_1 = c_0 + d_0 \]

\[ e_0 = b_0 + 18 \]
\[ s_0 = a_0 + b_0 \]
\[ v_0 = e_0 + f_0 \]

\[ e_1 = a_0 + 17 \]
\[ f_0 = c_0 + d_0 \]
\[ v_1 = e_1 + f_0 \]

\[ e_2 = \phi(e_0, e_1) \]
\[ v_2 = \phi(v_0, v_1) \]
\[ v_0 = a_0 + b_0 \]
\[ w_0 = e_0 + d_0 \]
\[ x_0 = e_2 + f_0 \]

\[ r_2 = \phi(r_0, r_1) \]
\[ y_0 = a_0 + b_0 \]
\[ z_0 = c_0 + d_0 \]
Value Numbering Example

\[ m_0 = a_0 + b_0 \]
\[ n_0 = a_0 + b_0 \]
\[ p_0 = c_0 + d_0 \]
\[ r_0 = c_0 + d_0 \]

\[ e_0 = b_0 + 18 \]
\[ s_0 = a_0 + b_0 \]
\[ u_0 = e_0 + f_0 \]

\[ q_0 = a_0 + b_0 \]
\[ r_1 = c_0 + d_0 \]

\[ e_1 = a_0 + 17 \]
\[ t_0 = c_0 + d_0 \]
\[ u_1 = e_1 + f_0 \]

\[ e_2 = \phi(e_0, e_1) \]
\[ u_2 = \phi(u_0, u_1) \]
\[ v_0 = a_0 + b_0 \]
\[ w_0 = c_0 + d_0 \]
\[ x_0 = e_2 + f_0 \]

\[ r_2 = \phi(r_0, r_1) \]
\[ y_0 = a_0 + b_0 \]
\[ z_0 = c_0 + d_0 \]

Local
Value Numbering Example

\[ m_0 = a_0 + b_0 \]
\[ n_0 = a_0 + b_0 \]

\[ p_0 = c_0 + d_0 \]
\[ r_0 = c_0 + d_0 \]

\[ q_0 = a_0 + b_0 \]
\[ r_1 = c_0 + d_0 \]

\[ e_0 = b_0 + 18 \]
\[ s_0 = a_0 + b_0 \]
\[ u_0 = e_0 + f_0 \]

\[ e_1 = a_0 + 17 \]
\[ f_0 = c_0 + d_0 \]
\[ u_1 = e_1 + f_0 \]

\[ e_2 = \phi(e_0, e_1) \]
\[ u_2 = \phi(u_0, u_1) \]
\[ v_0 = a_0 + b_0 \]
\[ w_0 = c_0 + d_0 \]
\[ x_0 = e_2 + f_0 \]

\[ r_2 = \phi(r_0, r_1) \]
\[ y_0 = a_0 + b_0 \]
\[ z_0 = c_0 + d_0 \]
Room for improvement

- Still miss some opportunities because we must discard the value table each time we reach a node with multiple predecessors.
- Would like to keep some information about values we have already seen.
- There is another technique we can use to find more opportunities for optimization.
A node \( m \) dominates a node \( n \) if every path from the start node to \( n \) goes through \( m \).

By definition a node dominates itself
Dominator Based Value Numbering

* Preorder traversal of the dominator tree
* Initialize the value table with the blocks immediate dominator in the tree
* Remove the entries from the table when returning from the block
Can we do better?

- Yes, using global value numbering.
- Technique uses data flow analysis to compute which expressions are available at any point in the program.
- Take comp 512 for all the data flow analysis you could ever want.
Resources

* Engineering a Compiler
* Cooper and Torczon
* The Dragon Book
* Aho, Sethi and Ullman
* Comp 412