Machine Independent Compiler Optimization

- What is classical machine independent optimization?
- Control flow graph, basic blocks, local opts
- Control flow abstractions: loops, dominators
- Four classical dataflow problems
  - Reaching definitions
  - Live variables
  - Available expressions
  - Very busy expressions

Phases of Compilation

Optimization is a semantics-preserving transformation
Example

• To define classical optimizations using an example loop from Fortran scientific code
• Opportunities for these optimizations result from table-driven code generation

```fortran
sum = 0
do 10 i = 1, n
  10 sum = sum + a(i) * a(i)
...```

Three Address Code

1. sum = 0; initialize loop counter
2. i = 1
3. if i > n goto 15; loop test, check for limit
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6; a[i] * a[i]
11. t8 = sum + t7; increment sum
12. sum = t8
13. i = i + 1; increment loop counter
14. goto 3
15.```
Control Flow Graph (CFG)

1. sum = 0
2. i = 1
3. if i > n goto 15
   
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. T

Local Common Subexpression Elimination (CSE)

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
10a t7 = t3 * t3
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
11a sum = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. Blue code eliminated; Red code added
Invariant Code Motion

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15.

Reduction in Strength

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
3. if i > n goto 15
4. t2 = i * 4
5. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
13. i = i + 1
14. goto 3
15.
Test Elision and Induction Variable Elimination

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
2b t2 = i * 4
3. if i > n goto 15
6. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
13. i = i + 1
14. goto 3
15

Constant Propagation and Dead Code Elimination

1. sum = 0
2. i = 1
2a t1 = addr(a) - 4
2b t2 = i * 4
2c t9 = 4 * n
3a if t2 > t9 goto 15
6. t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
11b t2 = t2 + 4
13. i = i + 1
14. goto 3a
15
New Control Flow Graph

1. \( \text{sum} = 0 \)
2. \( t1 = \text{addr}[a] - 4 \)
3. \( t2 = 4 \)
4. \( t9 = 4 \times n \)

5. if \( t2 > t9 \) goto 11

6. \( t3 = t1[t2] \)
7. \( t7 = t3 \times t3 \)
8. \( \text{sum} = \text{sum} + t7 \)
9. \( t2 = t2 + 4 \)
10. \( \text{goto 5} \)

11.

How to build CFG?

- Need to find basic blocks and possible branches between them
- Basic block leader statements
  - First program statement
  - Targets of conditional or unconditional goto’s
  - Any statement following a conditional goto
- For each leader \( s \), construct basic block \( B_s \), as all statements \( t \) reachable from \( s \) through straight-line code
- Eventually, any statements not included in some basic block are unreachable from program entry

dead code
Leader Statements

1. sum = 0
2. i = 1
3. if i > n goto 15
4. t1 = addr(a) - 4
5. t2 = i * 4
6. t3 = t1[t2]
7. t4 = addr(a) - 4
8. t5 = i * 4
9. t6 = t4[t5]
10. t7 = t3 * t6
11. t8 = sum + t7
12. sum = t8
13. i = i + 1
14. goto 3
15. branch
target

Local CSE

• Accomplished while translating into three address code
• For each statement, form expression DAGs (for operand sharing)
  – Operands are children of operator nodes
  – Operand nodes can be used by more than one operator node
  – Intermediate results that must be stored cause creation of compiler temporaries
  – Multiple labels on same node mean CSE
Expression DAG construction

\[ t1 = \text{addr}[a] - 4 \]
\[ t2 = i \times 4 \]
\[ t3 = t1[t2] \]

\[ \text{addr}[a] \quad i \quad 4 \]

Expression DAG construction

\[ t1 = \text{addr}[a] - 4 \]
\[ t2 = i \times 4 \]
\[ t3 = t1[t2] \]
\[ t4 = \text{addr}[a] - 4 \]
\[ t5 = i \times 4 \]
\[ t6 = t4[t5] \]
\[ t7 = t3 \times t6 \]
\[ t8 = \text{sum} + t7 \]
\[ \text{sum} = t8 \]
\[ i = i + 1 \]
DAG construction

• How to add a subexpression into a partially constructed DAG? \( A = B + C \)
• Is there a node already for \( B + C \)?
  – If so, add A to its list of labels
  – If not,
    • Is there a node labeled B already? If not, create a leaf labeled B
    • Is there a node labeled C already? If not, create a leaf labeled C
  – Create a node labeled A for + with left child B and right child C

Flow of Control Abstractions

• Dominator A node \( x \) dominates a node \( y \) if and only if all paths from the control flow graph (CFG) entry node to \( y \) pass through \( x \).
• (Natural) Loop Let \((y, x)\) be a CFG edge such that \( x \) dominates \( y \). Then all nodes on paths from \( x \) to \( y \) are in the loop defined by \((y, x)\).
  – \((y, x)\) is called a back edge
  – For reducible graphs, the set of back edges is unique
  – CFG is reducible if each loop can be entered through a single node
  – Irreducible means contains a subgraph
**Loops**

Back edges: (5,3), (4,3), (6,2)  
Loop (5,3) = {3, 4, 5}  
Loop (4,3) = {3, 4}  
Loop (6,2) = {2, 3, 4, 5, 6}  
combined

---

**General Step in Strength Reduction**

transforms an repeated expensive operation into a less expensive one
General Code Motion

n := 1; k := 0; m := 3; read x;
while n ≤ 10 do
    if 2 + x < 5 then k := 5;
    if 3 + k = 3 then m := m + 2;
    n := n + k + m;
endwhile;

definitions within loop are barriers to code motion

General Code Motion

n := 1; k := 0; m := 3; read x;
if 2 + x < 5 then k := 5;//move first
    t1 := 3 + k = 3 //move second
while n ≤ 10 do
    if 2 + x < 5 then k := 5;
    if 3 + k = 3 then m := m + 2;
    if t1 then m := m + 2;
    n := n + k + m;
endwhile;

Why can’t we move any more code out of the loop?
Program Analysis

• Performed at compile-time, deriving something about semantics of program
• Termed flow analysis
  – Control flow analysis reveals possible execution paths
    • Cannot tell actual feasibility of path. Why not?
  – Dataflow analysis determines information about modification, preservation, and use of data entities in a program

Four Classical Data Flow Problems

• Reaching definitions, Live uses of variables, Available expressions, Very Busy Expressions
• Def-use and Use-def chains, built from Reach and Live, used for many optimizations
• Avail enables global common subexpression elimination
• VeryB was used for conservative code motion
Reaching Definitions

- **Definition** A statement which may change the value of a variable
- A definition of a variable \( x \) at node \( k \) *reaches* node \( n \) if there is a definition-clear path from \( k \) to \( n \).

\[
\begin{array}{c}
\text{k} \quad x = \ldots \\
\downarrow \\
\text{n} \\
\ldots = x
\end{array}
\]

Live Uses of Variables

- **Use** Appearance of a variable as an operand of a 3 address statement
- A use of a variable \( x \) at node \( n \) is *live on exit* from node \( k \) if there is a definition-clear path for \( x \) from \( k \) to \( n \).

\[
\begin{array}{c}
\text{k} \quad x = \ldots \\
\downarrow \\
\text{n} \\
\ldots = x
\end{array}
\]
Def-use Relations

- Use-def chain links an use to a definition that reaches that use
- Def-use chain links a definition to an use that it reaches

```
x = k
```

Optimizations Enabled by Def-use

- Dead code elimination (Def-use)
- Code motion (Use-def)
- Strength reduction (Use-def)
- Test elision (Use-def)
- Constant propagation (Use-def)
- Copy propagation (Def-use)
**Dead Code Elimination**

```plaintext
sum = 0
i = 1

if i >= n goto 15

T

F

t1 = addr(a) - 4
t2 = i * 4
i = i + 1

After strength reduction, test elision, constant propagation the def-use links for i=1 disappear and it becomes dead code.
```

**Constant Propagation**

- same constant
- different constants

```plaintext
p: i*2
q: 5*i+3 = 8
```
Classical Dataflow Problems

- How to formulate analysis from CFG to dataflow equations?
- *Forward* and *backward* dataflow problems
- *May* and *must* dataflow problems

Reaching Definitions

Reach(m1) Reach(m2) Reach(m3)

\[
\begin{align*}
\text{Reach(m1)} & \quad \text{Reach(m2)} & \quad \text{Reach(m3)} \\
\text{m1} & \quad \text{m2} & \quad \text{m3} \\
\downarrow & \quad \downarrow & \quad \downarrow \\
\text{j} & \quad \text{Reach(j)} \\
\end{align*}
\]

forward, may dataflow problem
**Reaching Definitions Equations**

\[ \text{Reach}(j) = \bigcup \{ \text{Reach}(m) \cap \text{pres}(m) \cup \text{dgen}(m) \} \]

where:
- \( \text{pres}(m) \) is the set of defs preserved through node \( m \)
- \( \text{dgen}(m) \) is the set of defs generated at node \( m \)
- \( \text{Pred}(j) \) is the set of immediate predecessors of node \( j \)

**Live Uses of Variables**

Diagram showing live uses of variables with nodes labeled \( j \), \( m_1 \), \( m_2 \), and \( m_3 \). The diagram illustrates the backward, may dataflow problem with arrows indicating live uses.
**Live Uses Equations**

\[ \text{Live}(j) = \bigcup \{ \text{Live}(m) \cap \text{pres}(m) \cup \text{ugen}(m) \} \quad m \in \text{Succ}(j) \]

where

- \( \text{pres}(m) \) is the set of uses preserved through node \( m \)
  (these will correspond to variables whosedefs are preserved)
- \( \text{ugen}(m) \) is the set of uses generated at node \( m \)
- \( \text{succ}(j) \) is the set of immediate successors of node \( j \)

**Available Expressions**

- An expression \( X \text{ op } Y \) is *available* at program point \( n \) if EVERY path from program entry to \( n \) evaluates \( X \text{ op } Y \), and after every evaluation prior to reaching \( n \), there are NO subsequent assignments to \( X \) or \( Y \).
Global Common Subexpressions

Global CSE

Cannot be eliminated because does not have \(a\times b\) available on all paths
Available Expressions

\[
\text{Avail} (j) = \bigcap \{ \text{Avail}(m) \cap \text{epres}(m) \cup \text{egen}(m) \} \\
\text{m} \in \text{Pred}(j)
\]

where:
- \text{epres}(m) is the set of expressions preserved through node \text{m}
- \text{egen}(m) is the set of (downwards exposed) expressions generated at node \text{m}
- \text{pred}(j) is the set of immediate predecessors of node \text{j}
Very Busy Expressions

• An expression $X \text{ op } Y$ is very busy at program point $n$, if along EVERY path from $n$, we come to a computation of $X \text{ op } Y$ BEFORE any redefinition of $X$ or $Y$.

```
\[ \begin{align*}
  X &= t_1 = X \text{ op } Y \\
  Y &= X \text{ op } Y \\
\end{align*} \]
```

Code Hoisting

• SAFETY: Assume $X \text{ op } Y$ is in VeryB($n$) and $n$ dominates all expression calculations that are hoisting candidates $p$.

• For every $X \text{ op } Y$ at program point $p$, trace backwards from $p$ to $n$ to ensure there is a path from $n \rightarrow p$ without any definitions of $X$, $Y$, $X \text{ op } Y$

• Hoist (Calculate $t=X \text{ op } Y$) at exit of node $n$; change candidate calculations from $s = X \text{ op } Y$ to $s = t$.

• PROFITABILITY: Check that copy propagation can eliminate all copies introduced in the previous step. If not, undo the hoist.
Very Busy Expressions

\[
\text{VeryB}(j) = \bigcap \{ \text{VeryB}(m) \cap \text{epres}(m) \cup \text{vgen}(m) \} \\
m \in \text{Succ}(j)
\]

where:
- \text{epres}(m) is the set of expressions preserved through node \( m \)
- \text{vgen}(m) is the set of (upwards exposed) expressions generated at node \( m \)
- \text{succ}(j) is the set of immediate successors of node \( j \)
## Dataflow Problems

<table>
<thead>
<tr>
<th></th>
<th>May Problems</th>
<th>Must Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward Problems</strong></td>
<td>Reaching Defs</td>
<td>Available Exprs</td>
</tr>
<tr>
<td><strong>Backward Problems</strong></td>
<td>Live Uses of</td>
<td>Very Busy</td>
</tr>
<tr>
<td></td>
<td>Variables</td>
<td>Expressions</td>
</tr>
</tbody>
</table>

Machinelnomp. 1, Sp06 © B. Ryder