Compiler Optimization and Code Generation

Professor: Sc.D., Professor
Vazgen Melikyan
Course Overview

- Introduction: Overview of Optimizations
  - 1 lecture
- Intermediate-Code Generation
  - 2 lectures
- Machine-Independent Optimizations
  - 3 lectures
- Code Generation
  - 2 lectures
Machine-Independent Optimizations
Causes of Redundancy

- Redundancy is available at the source level due to recalculations while one calculation is necessary.
- Redundancies in address calculations
  - Redundancy is a side effect of having written the program in a high-level language where referrals to elements of an array or fields in a structure is done through accesses like A[i][j] or X -> f1.

As a program is compiled, each of these high-level data-structure accesses expands into a number of low-level arithmetic operations, such as the computation of the location of the [i, j]-th element of a matrix A. Accesses to the same data structure often share many common low-level operations.
A Running Example: Quicksort

void quicksort (int m, int n)
    /* recursively sorts a[ m ] through a[ n ] */
    {
        int i, j, v, x;
        if (n <= m) return;
        /* fragment begins here */
        i = m - 1; j = n; v = a[ n ];
        while (1) {
            do i = i + 1; while (a[ i ] < v);
            do j = j - 1; while (a[ j ] > v);
            if ( i >= j ) break;
            x = a[ i ]; a[ i ] = a[ j ]; a[ j ] = x;  /* swap a[ i ], a[ j ]*/
        }
        x = a[ i ]; a[ i ] = a[ n ]; a[ n ] = x;  /* swap a[ i ], a[ n ]*/
        /* fragment ends here */
        quicksort ( m, j ); quicksort ( i + 1, n );
    }
Flow Graph for The Quicksort Fragment

B1: \( i = m - 1 \)
\( j = n \)
\( t1 = 4*n \)
\( v = a[t1] \)

B2: \( i = i + 1 \)
\( t2 = 4*i \)
\( t3 = a[t2] \)
if \( t3 < v \) goto B2

B3: \( j = j - 1 \)
\( t4 = 4*j \)
\( t5 = a[t4] \)
if \( t5 > v \) goto B3

B4: if \( i \geq j \) goto B6

B5: \( t6 = 4*i \)
\( x = a[t6] \)
\( t7 = 4*i \)
\( t8 = 4*j \)
\( t9 = a[t8] \)
\( a[t7] = t9 \)
\( t10 = 4*j \)
\( a[t10] = x \)
goto B2

B6: \( t11 = 4*i \)
\( x = a[t11] \)
\( t12 = 4*i \)
\( t13 = 4*n \)
\( t14 = a[t13] \)
\( a[t12] = t14 \)
\( t15 = 4*n \)
\( a[t15] = x \)
Semantics-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

- Common-subexpression elimination
- Copy propagation
- Dead-code elimination
- Constant folding
- Code motion
- Induction-variable elimination
Common-Subexpression Elimination

- An occurrence of an expression E is called a common subexpression if E was previously computed and the values of the variables in E have not changed since the previous computation.

Avoid recomputing E if can be used its previously computed value; that is, the variable x to which the previous computation of E was assigned has not changed in the interim.
Flow Graph After C.S. Elimination

B1: \( i = m - 1 \)  
   \( j = n \)  
   \( t1 = 4*n \)  
   \( v = a[ t1 ] \)

B2: \( i = i + 1 \)  
   \( t2 = 4*i \)  
   \( t3 = a[ t2 ] \)  
   if \( t3 < v \) goto B2

B3: \( j = j - 1 \)  
   \( t4 = 4*j \)  
   \( t5 = a[ t4 ] \)  
   if \( t5 > v \) goto B3

B4: if \( i \geq j \) goto B6

B5: \( x = t3 \)  
   \( a[ t2 ] = t5 \)  
   \( a[ t4 ] = x \)  
   goto B2

B6: \( x = t3 \)  
   \( t14 = a[ t1 ] \)  
   \( a[ t2 ] = t14 \)  
   \( a[ t1 ] = x \)
Copy Propagation (1)

- This optimization concerns assignments of the form $u = v$ called copy statements.
- The idea behind the copy-propagation transformation is to use $v$ for $u$, wherever possible after the copy statement $u = v$.
- Copy propagation work example:

```
a = d + e
b = d + e
c = d + e
t = d + e
a = t
t = d + e
b = t
c = t
```
Copy Propagation (2)

- The assignment $x = t3$ in block B5 is a copy. Here is the result of copy propagation applied to B5.

```
B5:  x = t3
    a[ t2 ] = t5
    a[ t4 ] = x
    goto B2
```

```
B5:  x = t3
    a[ t2 ] = t5
    a[ t4 ] = t3
    goto B2
```

- This change may not appear to be an improvement, but it gives the opportunity to eliminate the assignment to $x$.
- One advantage of copy propagation is that it often turns the copy statement into dead code.
Dead-code Elimination

- Code that is **unreachable** or that does not affect the program (e.g. **dead stores**) can be eliminated.

While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

- Deducing at compile time that the value of an expression is a constant and using the constant instead is known as **constant folding**.
Dead-code Elimination: Example

- In the example below, the value assigned to i is never used, and the dead store can be eliminated. The first assignment to global is dead, and the third assignment to global is unreachable; both can be eliminated.

```c
int global;
void f ()
{
    int i;
    i = 1; /* dead store */
    global = 1; /* dead store */
    global = 2;
    return;
    global = 3; /* unreachable */
}
```

Before and After Dead Code Elimination

```c
int global;
void f ()
{
    global = 2;
    return;
}
```
Code Motion

- Code motion decreases the amount of code in a loop. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and evaluates the expression before the loop.

  Evaluation of limit - 2 is a loop-invariant computation in the following while-statement:

  ```c
  while ( i <= limit-2) /* statement does not change limit */
  ```

- Code motion will result in the equivalent code:

  ```c
  t = limit-2;
  while ( i <= t ) /* statement does not change limit or t */
  ```
Induction-Variable (IV) Elimination

- Variable \( x \) is said to be an "induction variable" if there is a positive or negative constant \( c \) such that each time \( x \) is assigned, its value increases by \( c \).
- For instance, \( i \) and \( t2 \) are induction variables in the loop containing B2 of QuickSort example.
- Induction variables can be computed with a single increment (addition or subtraction) per loop iteration. The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as strength reduction.
Flow Graph After IV Elimination

B1: \[ i = m - 1 \]
\[ j = n \]
\[ t1 = 4n \]
\[ v = a[t1] \]
\[ t2 = 4l \]
\[ t4 = 4j \]

B2: \[ t2 = t2 + 4 \]
\[ t3 = a[t2] \]
if \[ t3 < v \] goto B2

B3: \[ t4 = t4 - 4 \]
\[ t5 = a[t4] \]
if \[ t5 > v \] goto B3

B4: if \[ t2 > t4 \] goto B6

B5: \[ a[t7] = t5 \]
\[ a[t10] = t3 \]
goto B2

B6: \[ t14 = a[t1] \]
\[ a[t2] = t14 \]
\[ a[t1] = t3 \]
Flow Analysis

- **Flow analysis** is a fundamental prerequisite for many important types of code improvement.

- Generally control flow analysis precedes data flow analysis.

- **Control flow analysis** (CFA) represents flow of control usually in form of graphs. CFA constructs:
  - Control flow graph
  - Call graph

- **Data flow analysis** (DFA) is the process of asserting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.
Classification of Flow Analysis

- Two orthogonal classifications of flow analysis:
  - Control flow analysis
  - Data flow analysis

- Interprocedural optimizations usually require a call graph.
- In a call graph each node represents a procedure and an edge from one node to another indicates that one procedure may directly call another.
"Data-flow analysis" refers to a body of techniques that derive information about the flow of data along program execution paths.

All the previous optimizations depend on data-flow analysis.

- Example 1: One way to implement global common subexpression elimination requires us to determine whether two textually identical expressions evaluate to the same value along any possible execution path of the program.

- Example 2: If the result of an assignment is not used along any subsequent execution path, then the assignment can be eliminated as dead code.
Data Flow Problems

- **Reaching definitions**
  - Determine the set of variable definitions that can reach a CFG node, i.e. the definition occurs at least on one path prior to that node (forward problem).

- **Available expressions**
  - Determine the set of expressions that are available at a CFG node, i.e. the expression is evaluated on all paths prior to that node (forward problem).

- **Live/dead variables**
  - Determine the set of variables that are live at a CFG node, i.e. the variable is used after control passes that node at least on one path; if the variable is not used, it is called dead (backward problem).
Introduction to Data-Flow Analysis (2)

- **Local analysis** (e.g. value numbering)
  - Analyze effect of each instruction
  - Compose effects of instructions to derive information from beginning of basic block to each instruction

- **Data flow analysis**
  - Analyze effect of each basic block
  - Compose effects of basic blocks to derive information at basic block boundaries
  - From basic block boundaries, apply local technique to generate information on instructions
The Data-Flow Abstraction

- The execution of a program is viewed as a series of transformations of the program state, which consists of the values of all the variables in the program.
- Each execution of an intermediate-code statement transforms an input state to a new output state.
  The input state is associated with the program point before the statement and the output state is associated with the program point after the statement.
- While analyzing the behavior of a program, all the possible sequences of program points ("paths") through a flow graph must be considered. For the particular data-flow analysis problem an extraction from the possible program states at each point is used.
**Static Program vs. Dynamic Execution**

- **Statically**: Finite program
- **Dynamically**: Can have infinitely many possible execution paths
- Data flow analysis abstraction:
  - For each point in the program information of all the instances of the same program point are combined.

```
B1: a = 10
B2: if input () -> exit
B3: b = a
    a = a + 1
```
The Execution Path

- Execution path (or just path) from point p1 to point p2, is a sequence of points $p_1, p_2, p_3, \ldots p_n$ such that for each $i = 1, 2, \ldots n - 1$ either
  - $p_i$ is the point immediately preceding a statement and $p_i + 1$ is the point immediately following that same statement, or
  - $p_i$ is the end of some block and $p_i + 1$ is the beginning of a successor block.

- In general, there is an infinite number of possible execution paths through a program, and there is no finite upper bound on the length of an execution path.

- Program analyses summarize all the possible program states that can occur at a point in the program with a finite set of facts.
The Data-Flow Analysis Scheme (1)

- In each application of data-flow analysis, every program point is associated with a value that represents an abstraction of the set of all possible program states that can be observed for that point. The set of possible data-flow values is the domain for this application.

- A particular data-flow value is a set of definitions. The choice of abstraction depends on the goal of the analysis.

- Data-flow values are denoted before and after each statement s by $IN[s]$ and $OUT[s]$ respectively. The data-flow problem is to find a solution to a set of constraints on the $IN[s]$s and $OUT[s']$s, for all statements s. There are two sets of constraints: those based on the semantics of the statements and those based on the flow of control.
The Data-Flow Analysis Scheme (2)

- Build a flow graph (nodes = basic blocks, edges = control flow)
- Set up a set of equations between \( \text{in}[b] \) and \( \text{out}[b] \) for all basic blocks \( b \)
  - Effect of code in basic block:
    - Transfer function \( f_b \) relates \( \text{in}[b] \) and \( \text{out}[b] \) for same \( b \)
  - Effect of flow of control:
    - Relates \( \text{out}[b_1], \text{in}[b_2] \) if \( b_1 \) and \( b_2 \) are adjacent
Effects of a Basic Block

- Effect of a statement: \( a = b + c \)
  - Uses variables (\( b, c \))
  - Kills an old definition (old definition of \( a \))
  - New definition (\( a \))

- Compose effects of statements - effect of a basic block
  - Any definition of a data item in the basic block kills all definitions of the same data item reaching the basic block.
  - A locally available definition = last definition of data item in basic block
Reaching Definitions

- A definition $d$ reaches a point $p$ if there exists a path from the point immediately following $d$ to $p$ such that $d$ is not killed (overwritten) along that path.

- Problem statement
  - For each point in the program, determine if each definition in the program reaches the point
  - A bit vector per program point, vector length = $\# \text{def}$

```plaintext
entry

| d1: $i = m - 1$ | gen (b1) = \{ d1, d2, d3 \} |
| d2: $j = n$ | kill (b1) = \{ d4, d5 \} |
| d3: $a = u1$ |

| d4: $i = i + 1$ | gen (b1) = \{ d4, d5 \} |
| d5: $j = j - 1$ | kill (b1) = \{ d1, d2 \} |

exit
```
Effects of a Statement

- $f_s$: A transfer function of a statement
  - Abstracts the execution with respect to the problem of interest
- For a statement $s(d: x = y + z)$
  \[
  \text{out}[s] = f_s(\text{in}[s]) = \text{Gen}[s] \cup (\text{in}[s] - \text{Kill}[s])
  \]
  - Gen[s]: definitions generated: $\text{Gen}[s] = \{d\}$
  - Propagated definitions: $\text{in}[s] - \text{Kill}[s]$
    where $\text{Kill}[s] = \text{set of all other defs to } x$ in the rest of program

\[
\begin{array}{c}
\text{in B0} \\
\downarrow \\
\text{out B0}
\end{array}
\]

- $d0: y = 3$ \hspace{1cm} $fd0$
- $d1: x = 10$ \hspace{1cm} $fd1$
- $d2: y = 11$ \hspace{1cm} $fd2$
Effects of a Basic Block

- Transfer function of a statement s:
  - \( \text{out}[s] = f_s(\text{in}[s]) = \text{Gen}[s] \cup (\text{in}[s] - \text{Kill}[s]) \)
- Transfer function of a basic block B:
  - Composition of transfer functions of statements in B
  - \( \text{out}[B] = f_B(\text{in}[B]) = f_{d_2}f_{d_1}f_{d_0}(\text{in}[B]) \)
  - \( = \text{Gen}[d_2] \cup (\text{Gen}[d_1] \cup (\text{Gen}[d_0] \cup (\text{in}[B] - \text{Kill}[d_0])) - \text{Kill}[d_1])) - \text{Kill}[d_2] \)
  - \( = \text{Gen}[B] \cup (\text{in}[B] - \text{Kill}[B]) \)
  - Gen[B]: locally exposed definitions (available at end of b.b.)
  - Kill[B]: set of definitions killed by B

\[
\begin{align*}
\text{in B0} & : \\
\text{d0: y = 3} & \quad f_{d_0} \\
\text{d1: x = 10} & \quad f_{d_1} \\
\text{d2: y = 11} & \quad f_{d_2} \\
\text{out B0} & :
\end{align*}
\]

\[
f_B = f_{d_2}f_{d_1}f_{d_0}
\]
Effects of a Basic Block: Example

A transfer function $f_b$ of a basic block $b$:

$$out[b] = f(in[b])$$

incoming reaching definitions $\rightarrow$ outgoing reaching definitions

A basic block $b$

- Generates definitions: $Gen[b]$,
  - Set of locally available definitions in $b$
- Kills definitions: $in[b] - Kill[b]$,
  - Where $Kill[b]$ = set of defs (in rest of program) killed by defs in $b$

$$out[b] = Gen[b] \cup (in[b] - Kill[b])$$
Effects of Edges: Acyclic

- $\text{out}[b] = f_b (\text{in}[b])$
- Join node: a node with multiple predecessors
- Meet operator:
  - $\text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup ... \cup \text{out}[p_n]$, where $p_1, p_2, ..., p_n$ are all predecessors of $b$

<table>
<thead>
<tr>
<th>$f$</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1,2}</td>
<td>{3,4,6}</td>
</tr>
<tr>
<td>2</td>
<td>{3,4}</td>
<td>{1,2,6}</td>
</tr>
<tr>
<td>3</td>
<td>{5,6}</td>
<td>{1,3}</td>
</tr>
</tbody>
</table>
Cyclic Graphs

- Equations still hold
  - $\text{out}[b] = f_b(\text{in}[b])$
  - $\text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup \ldots \cup \text{out}[p_n]$}
- Find: fixed point solution
Iterative Algorithm to Reach Definitions

Input: control flow graph \( \text{CFG} = (N, E, \text{Entry}, \text{Exit}) \)

// Boundary condition
\( \text{out}[\text{Entry}] = \emptyset \)

// Initialization for iterative algorithm
For each basic block \( B \) other than \( \text{Entry} \)
\( \text{out}[B] = \emptyset \)

// Iterate
While (Changes to any \( \text{out}[] \) occur) {
    For each basic block \( B \) other than \( \text{Entry} \) {
        \( \text{in}[B] = \bigcup (\text{out}[p]) \), for all predecessors \( p \) of \( B \)
        \( \text{out}[B] = \text{fb}(\text{in}[B]) \) // \( \text{out}[B] = \text{gen}[B] \bigcup (\text{in}[B]-\text{kill}[B]) \)
    }
}

Reaching Definitions: Worklist Algorithm

Input: control flow graph CFG = (N, E, Entry, Exit)
// Initialize
out[Entry] = ∅ // can set out[Entry] to special def
// if reaching then undefined use

For all nodes i
out[i] = ∅ // can optimize by out[i]=gen[i]
ChangedNodes = N

// Iterate
While ChangedNodes ≠ ∅ {
  Remove i from Changed Nodes
  in[i] = U (out[p]), for all predecessors p of i
  oldout = out[i]
  out[i] = fi(in[i]) // out[i]=gen[i] U (in[i]-kill[i])
  if (oldout ≠ out[i]) {
    for all successors s of i
      add s to Changed Nodes
  }
}
Reaching Definitions: Example (1)

entry

\[ d1: i = m - 1 \]
\[ d2: j = n \]
\[ d3: a = u1 \]

\[ d4: i = i + 1 \]
\[ d5: j = j - 1 \]

\[ d6: a = u2 \]

B1

\[ d7: i = u3 \]

B2

B3

B4

exit

\[ \text{gen b1} = \{ d1, d2, d3 \} \]
\[ \text{kill b1} = \{ d4, d5, d6, d7 \} \]

\[ \text{gen b2} = \{ d4, d5 \} \]
\[ \text{kill b2} = \{ d1, d2, d7 \} \]

\[ \text{gen b3} = \{ d6 \} \]
\[ \text{kill b3} = \{ d3 \} \]

\[ \text{gen b4} = \{ d7 \} \]
\[ \text{kill b4} = \{ d1, d4 \} \]
Reaching Definitions: Example (2)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>000 0000</td>
<td>000 0000</td>
<td>111 0000</td>
<td>000 0000</td>
<td>111 0000</td>
</tr>
<tr>
<td>B2</td>
<td>000 0000</td>
<td>111 0000</td>
<td>001 1100</td>
<td>111 0111</td>
<td>001 1110</td>
</tr>
<tr>
<td>B3</td>
<td>000 0000</td>
<td>001 1100</td>
<td>000 1110</td>
<td>001 1110</td>
<td>000 1110</td>
</tr>
<tr>
<td>B4</td>
<td>000 0000</td>
<td>001 1110</td>
<td>001 0111</td>
<td>001 1110</td>
<td>001 0111</td>
</tr>
<tr>
<td>EXIT</td>
<td>000 0000</td>
<td>001 0111</td>
<td>001 0111</td>
<td>001 0111</td>
<td>001 0111</td>
</tr>
</tbody>
</table>

Computation of IN and OUT
Live Variable Analysis (1)

- **Definition**
  - A variable \( v \) is live at point \( p \) if the value of \( v \) is used along some path in the flow graph starting at \( p \)
  - Otherwise, the variable is **dead**.

- **Motivation**
  - E.g. register allocation
    ```
    for i = 0 to n
      ... i ...
    for i = 0 to n
      ... i ...
    ```

- **Problem statement**
  - For each basic block determine if each variable is live in each basic block
  - Size of bit vector: one bit for each variable
Live Variable Analysis (2)

control flow

\[ \text{IN}[b] = f_b(\text{OUT}[b]) \]

\[ \text{b} \quad f_b \]

\[ \text{OUT}[b] \]

- **A basic block b can**
  - Generate live variables: \( \text{Use}[b] \)
    - Set of locally exposed uses in b
  - Propagate incoming live variables: \( \text{OUT}[b] - \text{Def}[b] \),
    - where \( \text{def}[b] = \) set of variables defined in basic block

- **Transfer function for block b:**
  \[ \text{in}[b] = \text{Use}[b] \cup (\text{out}(b) - \text{Def}[b]) \]
Live Variable Analysis: Flow Graph

- \( \text{IN}[b] = f_b(\text{OUT}[b]) \)
- Join node: a node with multiple successors
- **Meet operator:**
  - \( \text{out}[b] = \text{in}[s_1] \cup \text{in}[s_2] \cup \ldots \cup \text{in}[s_n] \), where \( s_1, \ldots, s_n \) are all successors of \( b \)

<table>
<thead>
<tr>
<th>f</th>
<th>Use</th>
<th>Def</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{ e }</td>
<td>{a, b}</td>
</tr>
<tr>
<td>2</td>
<td>{ }</td>
<td>{a, b}</td>
</tr>
<tr>
<td>3</td>
<td>{ a }</td>
<td>{a, c}</td>
</tr>
</tbody>
</table>
Live Variable Analysis: Iterative Algorithm

input: control flow graph CFG = (N, E, Entry, Exit)
// Boundary condition
in[Exit] = ∅

// Initialization for iterative algorithm
For each basic block B other than Exit
in[B] = ∅

// Iterate
While (Changes to any in[] occur) {
  For each basic block B other than Exit {
    out[B] = \( \bigcup \) (in[s]), for all successors s of B
    in[B] = fb(out[B])  // in[B]=Use[B] \( \bigcup \) (out[B]-Def[B])
  }
}

## Summary of Two Data-flow Problems

<table>
<thead>
<tr>
<th></th>
<th>Reaching Definitions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td>Sets of definitions</td>
<td>Sets of variables</td>
</tr>
<tr>
<td><strong>Direction</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>forward:</td>
<td>backward:</td>
</tr>
<tr>
<td></td>
<td>( \text{out}[b] = f_b(\text{in}[b]) )</td>
<td>( \text{in}[b] = f_b(\text{out}[b]) )</td>
</tr>
<tr>
<td></td>
<td>( \text{in}[b] = \land \text{out}[\text{pred}(b)] )</td>
<td>( \text{out}[b] = \land \text{in}[\text{succ}(b)] )</td>
</tr>
<tr>
<td><strong>Transfer function</strong></td>
<td>( f_b(x) = \text{Gen}_b \cup (x - \text{Kill}_b) )</td>
<td>( f_b(x) = \text{Use}_b \cup (x - \text{Def}_b) )</td>
</tr>
<tr>
<td><strong>Meet Operation (\land)</strong></td>
<td>( \cup )</td>
<td>( \cup )</td>
</tr>
<tr>
<td><strong>Boundary Condition</strong></td>
<td>( \text{out}[\text{entry}] = \emptyset )</td>
<td>( \text{in}[\text{exit}] = \emptyset )</td>
</tr>
<tr>
<td><strong>Initial interior points</strong></td>
<td>( \text{out}[b] = \emptyset )</td>
<td>( \text{in}[b] = \emptyset )</td>
</tr>
</tbody>
</table>
Predictable Success