Optimization Techniques

• The most complex component of modern compilers
• Must always be sound, i.e., semantics-preserving
  • Need to pay attention to exception cases as well
  • Use a conservative approach: risk missing out optimization rather than changing semantics

• Reduce runtime resource requirements (most of the time)
  • Usually, runtime, but there are memory optimizations as well
  • Runtime optimizations focus on frequently executed code
    • How to determine what parts are frequently executed?
      • Assume: loops are executed frequently
      • Alternative: profile-based optimizations
  • Some optimizations involve trade-offs, e.g., more memory for faster execution

• Cost-effective, i.e., benefits of optimization must be worth the effort of its implementation
Code Optimizations

• High-level optimizations
  • Operate at a level close to that of source-code
  • Often language-dependent

• Intermediate code optimizations
  • Most optimizations fall here
  • Typically, language-independent

• Low-level optimizations
  • Usually specific to each architecture
High-level optimizations

- **Inlining**
  - Replace function call with the function body

- **Partial evaluation**
  - Statically evaluate those components of a program that can be evaluated

- **Tail recursion elimination**
- **Loop reordering**
- **Array alignment, padding, layout**
Intermediate code optimizations

• Common subexpression elimination
• Constant propagation
• Jump-threading
• Loop-invariant code motion
• Dead-code elimination
• Strength reduction
Constant Propagation

- Identify expressions that can be evaluated at compile time, and replace them with their values.

- \( x = 5; \)  \( \Rightarrow \)  \( x = 5; \)  \( \Rightarrow \)  \( x = 5; \)
- \( y = 2; \)  \( \Rightarrow \)  \( y = 2; \)  \( \Rightarrow \)  \( y = 2; \)
- \( v = u + y; \)  \( \Rightarrow \)  \( v = u + y; \)  \( \Rightarrow \)  \( v = u + 2; \)
- \( z = x * y; \)  \( \Rightarrow \)  \( z = x * y; \)  \( \Rightarrow \)  \( z = 10; \)
- \( w = v + z + 2; \)  \( \Rightarrow \)  \( w = v + z + 2; \)  \( \Rightarrow \)  \( w = v + 12; \)
- ...  \( \Rightarrow \)  ...  \( \Rightarrow \)  ...

...
Strength Reduction

• Replace expensive operations with equivalent cheaper (more efficient) ones.
  \[ y = 2; \quad \Rightarrow \quad y = 2; \]
  \[ z = x^y; \quad \Rightarrow \quad z = x^x; \]
  \[ \ldots \quad \ldots \]

• The underlying architecture may determine which operations are cheaper and which ones are more expensive.
Loop-Invariant Code Motion

• Move code whose effect is independent of the loop's iteration outside the loop.

  for (i=0; i<N; i++) {
      for (j=0; j<N; i++) {
          ... a[i][j] ...
      }
  }

  for (i=0; i<N; i++) {
      base = a + (i * dim1);
      for (j=0; j<N; i++) {
          ... (base + j) ...
      }
  }
Low-level Optimizations

- Register allocation
- Instruction Scheduling for pipelined machines.
- Loop unrolling
- Instruction reordering
- Delay slot filling
- Utilizing features of specialized components, e.g., floating-point units.
- Branch Prediction
Peephole Optimization

- Optimizations that examine small code sections at a time, and transform them
- Peephole: a small, moving window in the target program
- Much simpler to implement than global optimizations
- Typically applied at machine code, and some times at intermediate code level as well
- Any optimization can be a peephole optimization, provided it operates on the code within the peephole.
- Redundant instruction elimination
- Flow-of control optimizations
- Algebraic simplifications
Profile-based Optimization

• A compiler has difficulty in predicting:
  • likely outcome of branches
  • functions and/or loops that are most frequently executed
  • sizes of arrays
  • or more generally, any thing that depends on dynamic program behavior.

• Runtime profiles can provide this missing information, making it easier for compilers to decide when certain
Example Program: Quicksort

```c
void quicksort(m, n)
    int m, n;
{
    int i, j;
    int 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;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* fragment ends here */
    quicksort(m, j); quicksort(i+1, n);
}
```

- Most optimizations opportunities arise in intermediate code
  - Several aspects of execution (e.g., address calculation for array access) aren’t exposed in source code
- Explicit representations provide most opportunities for optimization
- It is best for programmers to focus on writing readable code, leaving simple optimizations to a compiler
3-address code for Quicksort

(1) \( i := m - 1 \)
(2) \( j := n \)
(3) \( t_1 := 4*n \)
(4) \( v := a[t_1] \)
(5) \( i := i + 1 \)
(6) \( t_2 := 4*i \)
(7) \( t_3 := a[t_2] \)
(8) if \( t_3 < v \) goto (5)
(9) \( j := j - 1 \)
(10) \( t_4 := 4*j \)
(11) \( t_5 := a[t_4] \)
(12) if \( t_5 > v \) goto (9)
(13) if \( i >= j \) goto (23)
(14) \( t_6 := 4*i \)
(15) \( x := a[t_6] \)
(16) \( t_7 := 4*i \)
(17) \( t_8 := 4*j \)
(18) \( t_9 := a[t_8] \)
(19) \( a[t_7] := t_9 \)
(20) \( t_{10} := 4*j \)
(21) \( a[t_{10}] := x \)
(22) goto (5)
(23) \( t_{11} := 4*l \)
(24) \( x := a[t_{11}] \)
(25) \( t_{12} := 4*i \)
(26) \( t_{13} := 4*n \)
(27) \( t_{14} := a[t_{13}] \)
(28) \( a[t_{12}] := t_{14} \)
(29) \( t_{15} := 4*n \)
(30) \( a[t_{15}] := x \)
Organization of Optimizer

- Code optimizer
  - Control flow analysis
  - Data flow analysis
  - Transformations
- Code generator

Front end
Flow Graph for Quicksort

- B1,…,B6 are basic blocks
  - sequence of statements where control enters at beginning, with no branches in the middle

- Possible optimizations
  - Common subexpression elimination (CSE)
  - Copy propagation
    - Generalization of constant folding to handle assignments of the form x = y
  - Dead code elimination
  - Loop optimizations
    - Code motion
    - Strength reduction
    - Induction variable elimination
Common Subexpression Elimination

- Expression previously computed
- Values of all variables in expression have not changed.
- Based on available expressions analysis
Copy Propagation

Consider

\[ x = y; \]
\[ z = x \cdot u; \]
\[ w = y \cdot u; \]

Clearly, we can replace assignment on \( w \) by
\[ w = z \]

This requires recognition of cases where multiple variables have same value (i.e., they are copies of each other)

One optimization may expose opportunities for another

- Even the simplest optimizations can pay off
- Need to iterate optimizations a few times
Dead Code Elimination

- **Dead variable**: a variable whose value is no longer used.
- **Live variable**: opposite of dead variable.
- **Dead code**: a statement that assigns to a dead variable.
- **Copy propagation** turns copy statement into dead code.
Induction Vars, Strength Reduction and IV Elimination

• Induction Var: a variable whose value changes in lock-step with a loop index

• If expensive operations are used for computing IV values, they can be replaced by less expensive operations

• When there are multiple IVs, some can be eliminated
Strength Reduction on IVs

(a) Before

\[
i := m - 1 \\
j := n \\
t_1 := 4 \times n \\
v := a[t_1]
\]

\[
j := j - 1 \\
t_4 := 4 \times j \\
t_5 := a[t_4] \\
\text{if } t_5 > v \text{ goto } B_3
\]

\[
\text{if } i \geq j \text{ goto } B_6
\]

(b) After

\[
i := m - 1 \\
j := n \\
t_1 := 4 \times n \\
v := a[t_1]
\]

\[
t_4 := 4 \times j \\
\]

\[
j := j - 1 \\
t_4 := t_4 - 4 \\
t_5 := a[t_4] \\
\text{if } t_5 > v \text{ goto } B_3
\]

\[
\text{if } i \geq j \text{ goto } B_6
\]
After IV Elimination ...

```

\[
\begin{align*}
  i &:= m - 1 \\
  j &:= n \\
  t_1 &:= 4 \times n \\
  v &:= a[t_1] \\
  t_2 &:= 4 \times i \\
  t_4 &:= 4 \times j \\
  t_2 &:= t_2 + 4 \\
  t_3 &:= a[t_2] \\
  \text{if } t_3 < v \text{ goto } B_2 \\
  t_4 &:= t_4 - 4 \\
  t_5 &:= a[t_4] \\
  \text{if } t_5 > v \text{ goto } B_3 \\
  \text{if } t_2 \geq t_4 \text{ goto } B_6 \\
  a[t_2] &:= t_5 \\
  a[t_4] &:= t_3 \\
  \text{goto } B_2 \\
  t_{14} &:= a[t_1] \\
  a[t_2] &:= t_{14} \\
  a[t_1] &:= t_3 \\
  \text{goto } B_2
\end{align*}
\]
```
Program Analysis

- Optimization is usually expressed as a program transformation
  \[ C_1 \iff C_2 \] when property \( P \) holds
- Whether property \( P \) holds is determined by a program analysis
- Most program properties are undecidable in general
  - Solution: Relax the problem so that the answer is an “yes” or “don’t know”
Applications of Program Analysis

• Compiler optimization
• Debugging/Bug-finding
  • “Enhanced” type checking
    • Use before assign
    • Null pointer dereference
    • Returning pointer to stack-allocated data
• Vulnerability analysis/mitigation
  • Information flow analysis
    • Detect propagation of sensitive data, e.g., passwords
    • Detect use of untrustworthy data in security-critical context
  • Find potential buffer overflows
• Testing – automatic generation of test cases
• Verification: Show that program satisfies a specified property, e.g., no deadlocks
  • model-checking
Dataflow Analysis

• Answers questions relating to how data flows through a program
  • What can be asserted about the value of a variable (or more generally, an expression) at a program point

• Examples
  • Reaching definitions: which assignments reach a program statement
  • Available expressions
  • Live variables
  • Dead code
  • ...

Dataflow Analysis

- Equations typically of the form
  \[ \text{out}[S] = \text{gen}[S] \cup (\text{in}[S] – \text{kill}[S]) \]
  where the definitions of out, gen, in and kill differ for different analysis

- When statements have multiple predecessors, the equations have to be modified accordingly

- Procedure calls, pointers and arrays require careful treatment
Points and Paths

\(d_1: i := m - 1\)
\(d_2: j := n\)
\(d_3: a := u1\)

\(d_4: i := i + 1\)

\(d_5: j := j - 1\)

possibly reaching

d_1

definitely reaching

\(d_6: a := u2\)
A *definition* of a variable \( x \) is a statement that assigns to \( x \).

- **Ambiguous definition:** In the presence of aliasing, a statement may define a variable, but it may be impossible to determine this for sure.

A definition \( d \) reaches a point \( p \) provided:
- There is a path from \( d \) to \( p \), and this definition is not “killed” along \( p \)
  - “Kill” means an unambiguous redefinition

**Ambiguity \( \Rightarrow \) approximation**
- Need to ensure that approximation is in the right direction, so that the analysis will be *sound*
DFA of Structured Programs

- \( S \rightarrow \text{id} := E \)
  - \( S;S \)
  - \( \text{if } E \text{ then } S \text{ else } S \)
  - \( \text{do } S \text{ while } E \)

- \( E \rightarrow E + E \)
  - \( \text{id} \)
DF Equations for Reaching Defns

\[
\begin{align*}
gen[S] &= \{d\} \\
kill[S] &= D_a - \{d\} \\
out[S] &= \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])
\end{align*}
\]

\[
\begin{align*}
\text{gen}[S] &= \text{gen}[S_2] \cup (\text{gen}[S_1] - \text{kill}[S_2]) \\
\text{kill}[S] &= \text{kill}[S_2] \cup (\text{kill}[S_1] - \text{gen}[S_2]) \\
\text{in}[S_1] &= \text{in}[S] \\
\text{in}[S_2] &= \text{out}[S_1] \\
\text{out}[S] &= \text{out}[S_2]
\end{align*}
\]
DF Equations for Reaching Defns

\[
\begin{align*}
\text{gen}[S] &= \text{gen}[S_1] \cup \text{gen}[S_2] \\
\text{kill}[S] &= \text{kill}[S_1] \cap \text{kill}[S_2] \\
in[S_1] &= in[S] \\
in[S_2] &= in[S] \\
out[S] &= out[S_1] \cup out[S_2]
\end{align*}
\]

\[
\begin{align*}
\text{gen}[S] &= \text{gen}[S_1] \\
\text{kill}[S] &= \text{kill}[S_1] \\
in[S_1] &= in[S] \cup \text{gen}[S_1] \\
out[S] &= out[S_1]
\end{align*}
\]
Direction of Approximation

- Actual *kill* is a superset of the set computed by the dataflow equations
- Actual *gen* is a subset of the set computed by these equations
- Are other choices possible?
  - Subset approximation of kill, superset approximation of gen
  - Subset approximation of both
  - Superset approximation of both
- Which approximation is suitable depends on the intended use of analysis results
Solving Dataflow Equations

- Dataflow equations are recursive
- Need to compute so-called fixpoints, to solve these equations
- Fixpoint computations uses an interative procedure
  - \( \text{out}^0 = \phi \)
  - \( \text{out}^i \) is computed using the equations by substituting \( \text{out}^{i-1} \) for occurrences of \( \text{out} \) on the rhs
  - Fixpoint is a solution, i.e., \( \text{out}^i = \text{out}^{i-1} \)
Computing Fixpoints: Equation for Loop

Rewrite equations using more compact notation, with:
- \( J \) standing for in\([S]\) and
- \( I, G, K, \) and \( O \) for in\([S1]\), gen\([S1]\), kill\([S1]\) and out\([S1]\):
  \[
  I = J \cup O,
  \]
  \[
  O = G \cup (I - K)
  \]

Letting \( I^0 = O^0 = \emptyset \), we have:

\[
I^1 = J
\]
\[
I^2 = J \cup O^1 = J \cup G
\]
\[
I^3 = J \cup O^2 = J \cup G \cup (J - K)
\]
\[
O^1 = G \cup (I^0 - K) = G
\]
\[
O^2 = G \cup (I^1 - K) = G \cup (J - K)
\]
\[
O^3 = G \cup (I^2 - K) = G \cup (J - K) = O^2
\]

(Note that for all sets \( A \) and \( B \), \( A \cup (A - B) = A \), and

for all sets \( A, B \) and \( C \), \( A \cup (A \cup C - B) = A \cup (C - B) \).)

Thus, we have a fixpoint.

\[
I = J \cup G
\]

\[
I = G_1 \cup G_2 \cup G_3 - K_4
\]
Use-Definition Chains

- Convenient way to represent reaching definition information
- ud-chain for a variable links each use of the variable to its reaching definitions
  - One list for each use of a variable

\[ x = y + z \]

\[ x = y - z \]
Available Expressions

• An expression \( e \) is available at point \( p \) if
  - every path to \( p \) evaluates \( e \)
  - none of the variables in \( e \) are assigned after last computation of \( e \)

• A block *kills* \( e \) if it assigns to some variable in \( e \) and does not recompute \( e \).

• A block *generates* \( e \) if it computes \( e \) and doesn’t subsequently assign to variables in \( e \)

• **Exercise:** Set up data-flow equations for available expressions. Give an example use for which your equations are sound, and another example for which they aren’t
Available expressions -- Example

\[ a := b + c \]

\[ b := a - d \]

\[ c := b + c \]

\[ d := a - d \]
Live Variable Analysis

• A variable $x$ is *live* at a program point $p$ if the value of $x$ is used in some path from $p$.
• Otherwise, $x$ is *dead*.
• Storage allocated for dead variables can be freed or reused for other purposes.
• $\text{in}[B] = \text{use}[B] \cup (\text{out}[B] – \text{def}[B])$
• $\text{out}[B] = \bigcup \text{in}[S]$, for $S$ a successor of $B$.
• Equation similar to reaching definitions, but the role of in and out are interchanged.
Def-Use Chains

- du-chain links the definition of a variable with all its uses
  - Use of a definition of a variable $x$ at a point $p$ implies that there is a path from this definition to $p$ in which there are no assignments to $x$
- du-chains can be computed using a dataflow analysis similar to that for live variables
Optimizations and Related Analyses

- Common subexpression elimination
  - Available expressions

- Copy propagation
  - In every path that reaches a program point $p$, the variables $x$ and $y$ have identical values

- Detection of loop-invariant computation
  - Any assignment $x := e$ where the definition of every variable in $e$ occurs outside the loop.

- Code reordering: A statement $x := e$ can be moved
  - earlier before statements that (a) do not use $x$, (b) do not assign to variables in $e$
  - later after statements that (a) do not use $x$, (b) do not assign to variables in $e$
Difficulties in Analysis

- Procedure calls
- Aliasing
Difficulties in Analysis

• Procedure calls
  • may modify global variables
    • potentially kill all available expressions involving global variables
    • modify reaching definitions on global variables

• Aliasing
  • Create ambiguous definitions
  • \texttt{a[i] = a[j]} --- here, \(i\) and \(j\) may have same value, so assignment to \(a[i]\) can potentially kill \(a[j]\)
  • \texttt{*p = q + r} --- here, \(p\) could potentially point to \(q, r\) or any other variable
    • creates ambiguous redefinition for all variables in the program!