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.
- \[ x = 5; \Rightarrow x = 5; \Rightarrow x = 5; \]
  \[ y = 2; \quad y = 2; \quad y = 2; \]
  \[ v = u + y; \quad v = u + y; \quad v = u + 2; \]
  \[ z = x \times y; \quad z = x \times y; \quad z = 10; \]
  \[ w = v + z + 2; \quad w = v + z + 2; \quad w = v + 12; \]

Strength Reduction

- Replace expensive operations with equivalent cheaper (more efficient) ones.
  \[ y = 2; \Rightarrow y = 2; \]
  \[ z = x^y; \Rightarrow z = x \times x; \]
- 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.
  \[ \text{for (i=0; i<N; i++)} \quad \Rightarrow \quad \text{for (i=0; i<N; i++)} \]
  \[ \text{for (j=0; j<N; j++)} \quad \text{base = a + (i * dim1)}; \]
  \[ \quad \text{... a[i][j] ...} \quad \text{for (j=0; j<N; j++)} \]
  \[ \quad \text{... (base + j) ...} \]

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, anything that depends on dynamic program behavior.
- Runtime profiles can provide this missing information, making it easier for compilers to decide when certain optimizations can be applied.

Example Program: Quicksort

```c
void quicksort(int n)
{
    int i, j;
    int v, x;
    if ( n <= 1 ) return;
    /* fragment begins here */
    i = n-1; j = n; v = a[n];
    while ( ) {
        x = a[i]; a[i] = a[j]; a[j] = x;
        x = a[i]; a[i] = a[n]; a[n] = x;
        /* fragment ends here */
        quicksort(n, j); quicksort(i+1, n);
    }
}
```

3-address code for Quicksort

```
1) i := n-1
2) j := n
3) t1 := 4*n
4) v := a[t1]
5) i := i+1
6) t2 := 4*i
7) t3 := a[t2]
8) if t3 < v goto (5)
9) j := j-1
10) t4 := 4*j
11) t5 := a[t4]
12) if t5 > v goto (9)
13) if i >= j goto (23)
14) t6 := 4*i
15) x := a[t6]
16) t7 := 4*i
17) t8 := 4*j
18) a[t8] := a[t7]
19) a[t7] := x
20) t9 := 4*j
21) a[t9] := x
22) goto (5)
23) t10 := 4*i
24) x := a[t10]
25) t11 := 4*i
26) t12 := 4*i
27) a[t12] := a[t11]
28) a[t11] := x
29) t13 := 4*n
30) a[t13] := x
```
Organization of Optimizer

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 * u;
  - w = y * 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

After IV Elimination …
Program Analysis

- Optimization is usually expressed as a program transformation $C_1 \leftrightarrow C_2$ 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
  $$out[S] = gen[S] \cup (in[S] – kill[S])$$
  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

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

- $S \rightarrow id := E$
  - $S;S$
  - if $E$ then $S$ else $S$
  - do $S$ while $E$
- $E \rightarrow E + E$
  - id

DF Equations for Reaching Defns

- $gen[S] = \{d\}$
  - $kill[S] = D_a - \{d\}$
  - $out[S] = gen[S] \cup (in[S] - kill[S])$

- $gen[S] = gen[S_2] \cup (gen[S_1] - kill[S_2])$
  - $kill[S] = kill[S_2] \cup (kill[S_1] - gen[S_2])$
  - $in[S_1] = in[S]$
  - $in[S_2] = out[S_1]$
  - $out[S] = out[S_2]$
DF Equations for Reaching Defns

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

Direction of Approximation

- Actual \textit{kill} is a superset of the set computed by the dataflow equations.
- Actual \textit{gen} is a subset of the set computed by these equations.
- Are other choices possible?
  - Subset approximation of \textit{kill}, superset approximation of \textit{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 \textit{fixpoints}, to solve these equations.
- Fixpoint computations uses an iterative procedure.
  - \(out^0 = \phi\)
  - \(out^i\) is computed using the equations by substituting \(out^{i-1}\) for occurrences of \(out\) on the rhs.
- Fixpoint is a solution, i.e., \(out^i = out^{i-1}\).

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 \(out[S1]\):
  \[\begin{align*}
  I &= J \cup O, \\
  O &= G \cup (I - K)
  \end{align*}\]
- Letting \(I^0 = O^0 = \phi\), we have:
  \[\begin{align*}
  I^1 &= J \\
  O^1 &= G \cup (I^0 - K) = G \\
  I^2 &= J \cup O^1 = J \cup G \\
  O^2 &= G \cup (I^1 - K) = G \cup (J - K)
  \end{align*}\]
- \(I^p = J \cup G \cup (J - K) = O^p\)
  (Note that for all sets \(A\) and \(B\), \(A \cup (A \cup B) = A\), and for all sets \(A, B\) and \(C\), \(A \cup (A \cup C \cup B) = A \cup (C \cup B)\).)
- Thus, we have a fixpoint.
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

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
    - $a[i] = a[j]$ --- here, $i$ and $j$ may have same value, so assignment to $a[i]$ can potentially kill $a[j]$
    - $*p = q + r$ --- here, $p$ could potentially point to $q$, $r$ or any other variable
      - creates ambiguous redefinition for all variables in the program!