

Code Optimization

Code Optimization

- Two steps:
 1. Analysis (to uncover optimization opportunities)
 2. Optimizing transformation
- Optimization:
 - must be semantically correct.
 - shall *improve* program according to some criterion (program rarely *optimal* under any measure).

Note: This demand sounds trivial, but it is not. It is difficult to show that an optimization always improves e.g. performance.

- Question:
 - Should it be allowed to introduce new statements on some path?
 - What about optimizations which may produce worse results for some inputs?

Taxonomy of Optimizations (Orthogonal)

163

Scope

- Local
 - Basic block: sequential execution, usual simple
 - Example: common subexpression elimination
- Intraprocedural
 - whole procedure: CFG
 - classical data flow analysis
- Interprocedural
 - whole program: call graph

Machine dependent/independent

- independent: e.g. dead code elimination, code motion
- dependent: e.g. latency hiding; managing bounded machine resources like registers, functional units, caches.

Compiler Design, WS 2005/2006

Taxonomy of Optimizations

164

Which kind of information?

- static properties of program
- runtime information: feedback directed optimization or profile guided optimization

Time when optimizer executes

- compile time: classical
 - » early optimizations
 - constant folding, algebraic simplification
 - » late optimizations
 - peephole optimizations
- installation time: adaptive optimizations
- runtime: dynamic optimizations

Compiler Design, WS 2005/2006

Flow Analysis

165

- **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 ascertaining 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.

Compiler Design, WS 2005/2006

Data Flow Analysis (Datenflussanalyse)

166

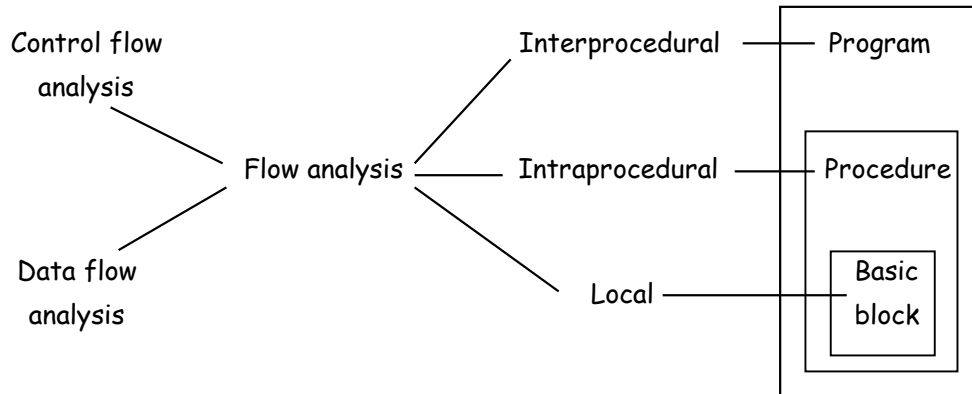
- DFA solves **data flow problems** by propagating data flow information along the paths of a flow graph.
- **Data flow information** is the set of information which is relevant for a data flow problem.
- Data flow information is propagated through a CFG by setting up and solving **data flow equations**.
- (Global) DFA collects data flow information across jump boundaries (considerably more difficult than local analysis).
- DFA problems can be classified as
 - **forward problems**: information propagated in the direction of the control flow
 - **backward problems**: propagation in the opposite direction of the control flow.

Compiler Design, WS 2005/2006

Classification of Flow Analysis

167

Two orthogonal classifications of flow analysis:



Interprocedural optimizations usually require a **call graph**.

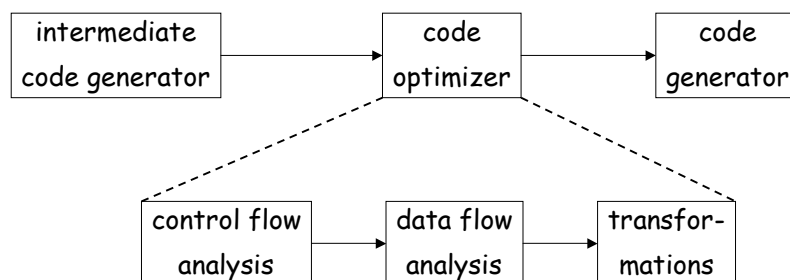
Def. : In a **call graph** each node represents a procedure and an edge from one node to another indicates that one procedure may directly call another.

Compiler Design, WS 2005/2006

Example of an Optimizer

168

Consider the following Figure:



Compiler Design, WS 2005/2006

Data Flow Problems

169

- **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).

Compiler Design, WS 2005/2006

Optimizations based on DFA

170

- **Common subexpression elimination** (based on available expression information).
 - If an expression occurs several times, compute it once and reuse the result.
- **(Partial) redundancy elimination (PRE).**
 - Perform code motion to eliminate partially redundant code, i.e. code that is at least on one path redundant.
- **(Partial) dead code elimination (PDE).**
 - Perform code motion to eliminate partially dead code, i.e. code that is at least on one path dead.
- **Loop-invariant code motion.**
 - Move loop-invariant code out of loops (can be covered by combining PRE and PDE).
- **Constant propagation.**
 - Determine the value of a variable, if it is a constant.

Compiler Design, WS 2005/2006