

Goals of Program Optimization (1 of 2)

Goal: Improve program performance within some constraints

Ask Three Key Questions for Every Optimization

1. Is it legal?
2. Is it profitable?
3. Is it compile-time cost justified?

(1) Is it legal?

- Must preserve the semantics of the program
- It is sufficient to preserve *externally observable results*
- This is a *language-dependent property*
 - E.g., exceptions in C vs. exceptions in Java
- May need even more flexibility
 - Reordering floating point operations

Goals of Program Optimization (2 of 2)

(2) Is it profitable?

- Improve performance of average or common case
- Limit negative impact in cases where performance is reduced
- *Predicting* performance impact is often non-trivial
- Choosing profitable optimization sequences is a major challenge

(3) Is its compile-time cost justified?

- The list of possible optimizations is *huge*
- It is easy to go overboard: try everything
- Must be justified by performance gain
- E.g., whole-program (interprocedural) optimizations usually O4

Classifying Optimizations (1 of 2)

We can classify optimizations along 3 axes

(1) By scope

- *Local* $\equiv$ within a single basic block
- *Peephole* $\equiv$ on a window of instructions
- *Loop-level* $\equiv$ on one or more loops or loop nests
- *Global* $\equiv$ for an entire procedure
- *Interprocedural* $\equiv$ across multiple procedures or whole program

(2) By machine information used

- *Machine-independent* $\equiv$ uses no machine-specific information
- *Machine-dependent* $\equiv$ otherwise

Classifying Optimizations (2 of 2)

(3) By effect on structure of program:

- *Algebraic transformations* $\equiv$ uses algebraic properties
 - E.g., identities, commutativity, constant folding ...
- *Code simplification transformations* $\equiv$ simplify complex code sequences
 - *Control-flow simplification* $\equiv$ simplify branch structure
 - *Computation simplification* $\equiv$ replace expensive instructions with cheaper ones (e.g., constant propagation)
- *Code elimination transformations* $\equiv$ eliminates unnecessary computations
 - DCE, Unreachable code elimination
- *Redundancy elimination transformations* $\equiv$ eliminate repetition
 - Local or global CSE, LICM, Value Numbering, PRE
- *Reordering transformations* $\equiv$ changes the order of computations
 - *Loop transformations* $\equiv$ change loop structure
 - *Code scheduling* $\equiv$ reorder machine instructions

Topics in Program Optimization

1. A catalog of local and peephole optimizations

focus on *What*, not *How*

2. Control flow graph and loop structure

3. Global dataflow analysis

2 example dataflow problems

- (a) reaching definitions
- (b) available expressions
- (c) live variables
- (d) def-use and use-def chains

4. Some key global optimizations

Sparse Conditional Constant Propagation (SCCP)

Loop-Invariant Code Motion (LICM)

Global Common Subexpression Elimination (GCSE)

Local and Peephole Optimizations (1 of 3)

(1) Unreachable code elimination

- Code after an unconditional jump and with no branches to it
- Code in a branch never taken
(Often eliminated during global constant propagation)

(2) Flow-of-control optimizations

- *If simplification*: constant conditions, nested equivalent conditions
- *Straightening*: merge basic blocks that are always consecutive
- *Branch folding*:
 - unconditional jump to unconditional jump
 - conditional jump to unconditional jump
 - unconditional jump to conditional jump

Local and Peephole Optimizations (2 of 3)

(3) Algebraic simplifications

- exploit algebraic identities, commutativity, ...
- $x + 0$, $y * 1$, $18 * z * 14$

(4) Redundant instruction elimination

- Redundant loads and stores:
LD a $\rightarrow$ R0
ST R0 $\rightarrow$ a
 - Usually caused by compiler-generated code
- Conditional branch always taken
 - Usually caused by global constant propagation

Local and Peephole Optimizations (3 of 3)

(5) Reduction in strength

- Replace x^2 by $x * x$
- Replace $2^n * x$ by $x \ll n$ if integer x
- Replace $x/4$ by $x * 0.25$ if real division

(6) Machine idioms and Instruction Combining

(Or could be done during Instruction Selection, e.g., with Burg)

- Multiply-Add instruction: $r3 \leftarrow r3 + r1 * r2$
- Auto-increment or auto-decrement addressing modes
- Conditional move instructions
- Predicated instructions
- See Section 18.1.1 in Muchnick's book for some strange idioms

Flow Graphs

A fundamental representation for global optimizations.

Definitions

Flow Graph: A triple $G=(N,A,s)$, where (N,A) is a (finite) directed graph, $s \in N$ is a designated “initial” node, and there is a path from node s to every node $n \in N$.

Entry node: A node with no predecessors.

Exit node: A node with no successors.

Properties

- There is a unique entry node, which must be s (*Reachability assumption*)
 - Assumption is *safe*: can delete unreachable code
 - Assumption may be *conservative*: some branches never taken.
 - Control Flow Graphs are usually *sparse*. That is, $|A| = O(|N|)$. In fact, if only binary branching is allowed $|A| \leq 2|N|$.
-

Control Flow Graphs

Definitions

Review slides on Control Flow Graphs in the IR lecture

CFG Construction :

Read Section 9.4 of Aho, Sethi & Ullman for algorithm to partition a procedure into basic blocks. *This is required material.*

Dominance in Flow Graphs

Let d, d_1, d_2, d_3, n be nodes in G .

Definitions

d **dominates** n (write “ $d \text{ dom } n$ ”) iff every path in G from s to n contains d .

d **properly dominates** n if d dominates n and $d \neq n$.

d **is the immediate dominator of** n (write “ $d \text{ idom } n$ ”) if d is the last dominator on any path from initial node to n , $d \neq n$

Properties

● $s \text{ dom } d, \forall \text{ nodes } d \text{ in } G$.

● **Partial Ordering:** The dominance relation of a flow graph G is a *partial ordering*:

Reflexive : $n \text{ dom } n$ is true $\forall n$.

Antisymmetric : If $d \text{ dom } n$, then $n \text{ dom } d$ cannot hold.

Transitive : $d_1 \text{ dom } d_2 \wedge d_2 \text{ dom } d_3 \implies d_1 \text{ dom } d_3$

The Dominator Tree

Why it is a Tree

The dominators of a node form a chain:

- If $d_1 \text{ dom } n$ and $d_2 \text{ dom } n$ and $d_1 \neq d_2$, then:
it must hold that $d_1 \text{ dom } d_2$ or $d_2 \text{ dom } d_1$.

$\implies$ Every node $n \neq s$ has a unique immediate dominator.

Definition: Dominator Tree

The **Dominator Tree** of a flow graph G is a graph with the same nodes as G , and an edge $n_1 \rightarrow n_2$ iff $n_1 \text{ idom } n_2$.

Loops in Flow Graphs

Why Defining Loops is Challenging

- *Easy case*: Structured nested loops: FOR or WHILE
- *Harder case*: Arbitrary flow and exits in loop body, but unique loop “entry”
- *Hardest case*: No unique loop “entry” (“irreducible loops”)

Defining Loops

Idea: Use dominance to define Natural Loops

Back Edge : An edge $n \rightarrow d$ where $d \text{ dom } n$

Natural Loop : Given a back edge, $n \rightarrow d$, the natural loop corresponding to $n \rightarrow d$ is the set of nodes $\{d + \text{all nodes that can reach } n \text{ without going through } d\}$

Loop Header: A node d that dominates all nodes in the loop

- Header is unique for each natural loop
- $\Rightarrow d$ is the unique entry point into the loop
- Uniqueness is very useful for many optimizations

Why?

Preheader: An optimization convenience

The Idea

If a loop has multiple incoming edges to the header, moving code out of the loop safely is complicated

Preheader gives a safe place to move code before a loop

The Transformation

Introduce a pre-header p for each loop (let loop header be d):

1. Insert node p with one out edge: $p \rightarrow d$
2. All edges that previously entered d should now enter p

Reducible and Irreducible Flow Graphs

Reducible flow graph:

A flow graph G is called reducible iff we can partition the edges into 2 sets:

1. *forward edges*: should form a DAG in which every node is reachable from initial node
2. *other edges must be back edges*: i.e., only those edges $n \rightarrow d$ where $d \text{ dom } n$

Otherwise graph is called irreducible.

Idea: Every “cycle” has at least one back edge

$\Rightarrow$ All “cycles” in a reducible graph are natural loops

Not true in an irreducible graph!