Objectives

Brief Overview of Optimization

1. *Classifying optimizations*: 3 axes for classifying optimizations
2. *Criteria for choosing optimizations*: Safety, profitability, opportunity

Control Flow Analysis

1. *Control Flow Graphs*
2. *Dominators*: Dominance; the dominance graph
3. *Defining Loops in Flow Graphs*: Natural loops, Intervals.
4. *Reducibility*: Reducible vs. irreducible flow graphs; T1/T2 transformations; eliminating irreducibility.

A brief catalog of optimizations

activation record merging
branch folding
branch straightening
carry optimization
code hoisting
common subexpression elim.
constant folding
constant propagation
copy propagation
dead (unreachable) code elim.
dead (unused) code elim.
dead space reclamation
detection of parallelism
expression simplification
heap-to-stack promotion
instruction scheduling
live range shrinking

loop distribution
loop fusion
(loop) induction variable elim.
loop invariant code motion
loop interchange
(loop) linear func. test replacement
(loop) software pipelining
loop splitting
loop test elision
loop tiling
loop unrolling
loop unswitching
(loop) unroll and jam
. . .

machine idiom recognition
operator strength reduc’n
partial redundancy elim.
peephole optimization
procedure inlining
procedure integration
procedure specialization
software prefetching
special-case code gen.
register allocation
register assignment
register promotion
reassociation
scalar expansion
scalar replacement
stack height reduction
value numbering (local)
value numbering (global)

Axes of Classification 1 – Machine Dependence

Machine independent transformations

- applicable across broad range of machines
- remove redundant computations
- decrease (*overhead/real work*)
- reduce running time or space

Machine dependent transformations

- apitalize on specific machine properties
- improve mapping from *il* onto machine
- use “exotic” instructions

The distinction is not always clear cut:

consider replacing `multiply` with `shifts` and `adds`

Machine independent ⇒ deliberately ignore hardware parameters
Machine dependent ⇒ explicitly consider hardware parameters

Axes of Classification 2 – Scope

Local

- confined to straight line code
- simplest to analyze, prove strongest theorems
- code quality suffers at block boundaries

Regional

- multiple, related basic blocks
- use results from one block to improve its neighbors
- common choices:
  - single loop or loop nest
  - extended basic block
  - dominator region

Global (*intra-procedural*)

- consider the whole procedure
- classical data-flow analysis, dependence analysis
- make best choices for entire procedure

Interprocedural (*whole program*)

- analyze whole programs
- less information is discernible
- move knowledge or code across calls

Axes of Classification 3 - Effect

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

Three Considerations

In discussing any optimization, look for three issues: safety, profitability, opportunity

Safety — *Does it change the results of the computation?*

- dataflow analysis
- alias/dependence analysis
- special-case analysis

Profitability — *Is it expected to speed up execution?*

- always profitable
- seat of the pants rules

Opportunity — *Can we locate application sites efficiently?*

- find all sites
- updates and ordering

Flow Graphs

A fundamental representation for global optimizations.

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 $n \in N$.

Properties:

- An *entry node* in a flow graph has no predecessors.
  An *exit node* in a flow graph has no successors.
- There is exactly one entry node, s.
  We can modify a general DAG to ensure this. *How?*
- In a control flow graph, any node unreachable from s can be safely deleted.
- Control flow graphs are usually *sparse*. That is, $\mid A \mid = O(\mid N \mid)$. In fact, if only binary branching is allowed $\mid A \mid \leq 2 \mid N \mid$.

Control Flow Graph: CFG

Definitions

Basic Block $\equiv$ a sequence of statements (or instructions) $S_1 \ldots S_n$ such that execution control must reach S_1 before S_2, and, if S_1 is executed, then $S_2 \ldots S_n$ are all executed in that order (unless one of the statements causes the program to halt)

Leader $\equiv$ the first statement of a basic block

Maximal Basic Block $\equiv$ a *maximal-length* basic block

CFG $\equiv$ a directed graph (usually for a single procedure) in which:

- Each node is a single basic block
- There is an edge $b_1 \rightarrow b_2$ if block b_2 *may* be executed after block b_1 in *some* execution

NOTE: A CFG is a conservative approximation of the control flow! Why?

Homework: Read Section 9.4 of *Aho, Sethi & Ullman*: algorithm to partition a procedure into basic blocks.

Dominance in Flow Graphs

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

Definitions

d **dominates** n (write "$d\ dom\ n$") *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 "$d\ idom\ n$") if d is the last dominator on any path from initial node to n, $d \neq n$

DOM(x) denotes the set of dominators of x.

Properties

Lemma 1: DOM(s) = { s }.

Lemma 2: $s\ dom\ d, \ \forall$ nodes d in G.

Lemma 3: The dominance relation on nodes in a flow graph is a *partial ordering*:

Reflexive : $n\ dom\ n$ is true $\forall\ n$.

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

Transitive : $d_1\ dom\ d_2 \ \wedge\ d_2\ dom\ d_3 \implies d_1\ dom\ d3$

The Dominator Graph

Lemma 4: The dominators of a node form a chain.

Proof : Suppose x and y dominate node z. Then there is a path from s to z where both x and y appear only once (if not "cut and paste" the path). Assume that x appears first. Then if x does not dominate y there is a path from s to y that does not include x contradicting the assumption that x dominates z.

Lemma 5 : Every node except s has a unique immediate dominator.

Proof : The dominators form a chain. The last node in the chain is the immediate dominator, and the last node always exists and is unique.

Lemma 6 : Create the **dominator graph** for G:

- Same nodes in G.
- Edge $n_1 \rightarrow n_2$ *iff* $n_1\ idom\ n_2$.

The dominator graph is a ____________?

Dominator Construction

Algorithm DOM : Finding Dominators in A Flow Graph

Input : A flow graph $G = (N, A.s)$.

Output : The sets DOM(x) for each $x \in N$.

Algorithm:

```
DOM(s) := { s }

forall n ∈ N − {s} do
   DOM(n) := N
od

while changes to any DOM(n) occur do
   forall n in N − {s} do
      DOM(n) := {n} ⋃ ⋂_{p→n} DOM(p)
   od
od
```

Identifying Program "Loops" in Control Flow Graphs

Properties of a Good Definition

Q. What properties of "loops" do we want to extract from the CFG?

- Represent cycles in flow graph, with precise entries/exits
- . Distinguish nested loops and nesting structure
- . Extract loop bounds, loop stride, iterations: Symbolic analysis

Q. What kinds of loops do we need to allow?

- . Loops intuitively defined by programmers (including all high-level language loops)
- . Support all kinds of loops: structured and unstructured

Identifying Program "Loops" in Control Flow Graphs

The right definition of "loop" is not obvious. Obviously bad definitions:

Cycle: Not necessarily properly nested or disjoint

SCC: Too coarse; no nesting information

Easy *Harder* *Difficult*

Natural Loops

Def. Back Edge: An edge $n \to d$ where $d\ dom\ n$

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

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

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

Intervals

Idea

Partition flow graph into disjoint subgraphs where each subgraph has a single entry (header).

Intervals are closely connected to concept of reducibility (see later slides).

Definition

The interval with node h as header, denoted $I(h)$, is the subset of nodes of G obtained as follows:

$I(h) := \{h\}$

while $\exists$ node m such that $m \notin I(h)$ and $m \neq s$ and
all arcs entering m leave nodes in $I(h)$
do
$I(h) := I(h) + m$
od

Lemma 7. $I(h)$ is unique: does not depend on order of node insertion.

Proof: See *Hecht*

Properties of Intervals

Lemma 8. The subgraph generated by $I(h)$ is itself a flow graph.

Lemma 9.

(a) Every arc entering a node of the interval $I(h)$ from the outside enters the header h.

(b) h dominates every node in $I(h)$

(c) every cycle in I(h) includes h

Proof.

(a) Consider a node $m \in h$ that is also an entry node of $I(h)$. Then m could not have been added to $I(h)$.

(b) Consider a node $m \in I(h)$ not dominated by h. Then m could not have been added to $I(h)$.

(c) Suppose there is a cycle in $I(h)$ that does not include h. Then no node in the cycle could have been added to $I(h)$, because before any such node could be added the preceeding node in the cycle would have to be added.

Comparing Loop Definitions

Natural loop: Defined using dominators

+ Intuitive, and similar to SCC.
+ Single entry point: “loop header”.
+ Identifies nested loops (if different headers)

- Nested loops are not disjoint.
- Some nodes are not part of any natural loop.
- Does not include some cycles in “irreducible” flow graphs.

Intervals: Defined in terms of reachability in flow graph

+ Single entry point: “loop header”.
+ Identifies nested loops
+ Nested loops are disjoint.
+ In reducible graphs, all nodes are part of some interval.

- Not an intuitive definition.
- Does not include some cycles in “irreducible” flow graphs.

Reducible and Irreducible Flow Graphs

Def. Reducible flow graph: *the two easier cases*

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 $n \to d$ where $d\ dom\ n$

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

$\Rightarrow$ All “cycles” are natural loops

Otherwise graph is called irreducible. *the difficult case*

Reducibility by Intervals

Definition: If G is a flow graph, then the derived flow graph of G, $I(G)$, is:

(a) The nodes of $I(G)$ are the intervals of G

(b) The initial node of $I(G)$ is $I(s)$

(c) There is an arc from node $I(h)$ to $I(k)$ in $I(G)$ if there is any arc from a node in $I(h)$ to node k in G.

Definition: The sequence $G = G_0, G_1, ..., G_k$ is called the derived sequence for G *iff* $G_{i+1} = I(Gi)$ for $0 \leq i < k, G_{k-1} \neq Gk, I(Gk) = Gk$. G_k is called the limit flow graph of G.

Definition: A flow graph is reducible *iff* its limit flow graph is a single node with no arc. Otherwise it is called irreducible.

The T1 and T2 Transformations

T1 : Reduce a self-loop $x \rightarrow x$ to a single node

T2 : If $x \rightarrow y$, and there is no other predecessor of y, then reduce x and y to a single node.

Example 1:

The T1 and T2 Transformations

Example 2: An Irreducible graph

Important: If G is reducible, successive applications of T1 and T2 produce the trivial graph.
$\Rightarrow$ Reducibility by T1 and T2 is equivalent to reducibility by intervals.

Properties of Irreducible Graphs

The (*) subgraph:

The lines represent edge-disjoint paths.
Nodes s and a may be the same node.

Lemma. The absence of the (*) subgraph in a flow graph is preserved by T1 and T2.

Proof.

- If there is no path $x \rightsquigarrow y$, then T1 and T2 cannot create such a path.
- If two paths are not edge-disjoint, then T1 and T2 will not make them so.

Removing Irreducibility by Node Splitting

If a node has $n > 1$ predecessors and $m > 1$ successors, split the node into n copies:

Claim: T2 is always applicable to a graph after a node is split.

$\Rightarrow$ Any graph can be reduced to the trivial graph by applying T1, T2, and splitting.

Challenge: Finding a “minimal” splitting of a graph is not easy. Typically involves an NP-complete problem.