Control-Flow and Low-Level Optimizations

Outline

- Unreachable-Code Elimination
- Straightening
- If and Loop Simplifications
- Loop Inversion and Unswitching
- Branch Optimizations
- Tail Merging (Cross Jumping)
- Conditional Moves
- Dead-Code Elimination
- Branch Prediction
- Peephole Optimization
- Machine Idioms & Instruction Combining

Unreachable-Code Elimination

Unreachable code is code that cannot be executed, regardless of the input data
- code that is never executable for any input data
- code that has become unreachable due to a previous compiler transformation

Unreachable code elimination removes this code
- reduces the code space
- improves instruction-cache utilization
- enables other control-flow transformations
Straightening

Straightening is applicable to pairs of basic blocks such that the first has no successors other than the second and the second has no predecessors other than the first.

If Simplifications

If simplifications apply to conditional constructs one or both of whose branches are empty:

- if either the then or the else part of an if-construct is empty, the corresponding branch can be eliminated
- one branch of an if with a constant-valued condition can also be eliminated
- we can also simplify if s whose condition, C, occurs in the scope of a condition that implies C (and none of the condition’s operands has changed value)
Loop Simplifications

- A loop whose body is empty can be eliminated if the iteration-control code has no side-effects (Side-effects might be simple enough that they can be replaced with non-looping code at compile time).

- If the number of iterations is small enough, loops can be unrolled into branchless code and the loop body can be executed at compile time.

Loop Simplification Example

- Initial code:
  ```
  s = 0
  i = 0
  L1: if i >= 4 goto L2
      i = i + 1
      s = s + i
      goto L1
  L2: …
  ```

- Optimized code:
  ```
  s = 0
  i = 0
  L1: if i >= 4 goto L2
      i = i + 1
      s = s + i
  L2: …
  ```

Loop Inversion

Loop inversion transforms a `while` loop into a `repeat` loop (i.e., moves the loop-closing test from before the loop to after it).

- Has the advantage that only one branch instruction needs to be executed to close the loop.
- Requires that we determine that the loop is entered at least once!

Loop Inversion Example 1

- Loop bounds are known:
  ```
  for (i = 0; i < 100; i++) {
      a[i] = i + 1;
  }
  ```

- Optimized code:
  ```
  i = 0;
  while (i < 100) {
      a[i] = i + 1;
      i++;
  }
  ```
Loop Inversion Example 2

for (i = k; i < n; i++) {
    a[i] = i + 1;
}
Loop bounds are unknown

Unswitching

Unswitching is a control-flow transformation that moves loop-invariant conditional branches out of loops

if (k >= n) goto L
i = k;
do {
    a[i] = i + 1;
i++;
} while (i < n)
L:

Unswitching Example

for (i = 1; i < 100; i++) {
    if (k == 2) {
        for (i = 1; i < 100; i++) {
            if (a[i] > 0)
                a[i] = a[i] + 1;
        }
        else {
            a[i] = a[i] - 1;
        }
    } else {
        i = 100;
    }
}

Branch Optimizations

Branches to branches are remarkably common!
- An unconditional branch to an unconditional branch can be replaced by a branch to the latter’s target
- A conditional branch to an unconditional branch can be replaced by the corresponding conditional branch to the latter branch’s target
- An unconditional branch to a conditional branch can be replaced by a copy of the conditional branch
- A conditional branch to a conditional branch can be replaced by a conditional branch with the former’s test and the latter’s target as long as the latter condition is true whenever the former one is
Branch Optimization Examples

if a == 0 goto L1
L1: ...
L2: ...

goto L1
L1: ...

if a == 0 goto L1
if a != 0 goto L2
L1: ...
L2: ...

Eliminating Useless Control-Flow

The Problem:
– After optimization, the CFG might contain empty blocks
– “Empty” blocks still end with either a branch or jump
– Produces jump to jump, which wastes time and space

The Algorithm: (Clean)
– Use four distinct transformations
– Apply them in a carefully selected order
– Iterate until done

Eliminating Useless Control-Flow

Transformation 1

Both sides of branch target B2
– Neither block must be empty
– Replace it with a jump to B1
– Simple rewrite of the last operation in B1

How does this happen?
– By rewriting other branches

Eliminating redundant branches

How do we recognize it?
– Check each branch

Transformation 2

Merging an empty block
– Empty B1 ends with a jump
– Coalesce B1 and B2
– Move B1’s incoming edges
– Eliminates extraneous jump
– Faster, smaller code

Eliminating empty blocks

How does this happen?
– By eliminating operations in B1

How do we recognize it?
– Test for empty block
Eliminating Useless Control-Flow

**Transformation 3**

- Coalescing blocks
  - Neither block must be empty
  - B1 ends with a jump to B2
  - B2 has one predecessor
  - Combine the two blocks
  - Eliminates a jump

- How does this happen?
  - By simplifying edges out of B1

- How do we recognize it?
  - Check target of jump

- Eliminating non-empty blocks

**Transformation 4**

- Jump to a branch
  - B1 ends with a jump, B2 is empty
  - Eliminates pointless jump
  - Copy branch into end of B1
  - Might make B2 unreachable

- How does this happen?
  - By eliminating operations in B1

- How do we recognize it?
  - Jump to empty block

Eliminating Useless Control-Flow

**Putting the transformations together**

- Process the blocks in postorder
  - Clean up Bi’s successors before Bi
  - Simplifies implementation and understanding

- At each node, apply transformations in a fixed order
  - Eliminate redundant branch
  - Eliminate empty block
  - Merge block with successors
  - Hoist branch from empty successor

- May need to iterate
  - Postorder ⇒ unprocessed successors along back edges
  - Can bound iterations, but deriving a tight bound is hard
  - Must recompute postorder between iterations

Tail Merging (Cross Jumping)

Tail merging applies to basic blocks whose last few instructions are identical and that continue to the same location.

It replaces the matching instructions of one block with a branch to the corresponding point in the other.
Conditional Moves

Conditional moves are instructions that copy a source to a target if and only if a specified condition is satisfied:
- available in several modern architectures (SPARC-V9, PentiumPro)
- are used to replace simple branching code sequences with non-branching code

\[
\begin{align*}
\text{if } a > b \ & \text{ goto } L1 \\
\ & \text{ max } = b \\
\ & \text{ goto } L2 \\
L1: \ & \text{ max } = a \\
L2: \ & \ldots
\end{align*}
\]

Conditional Moves Help Loop Unrolling

\[
\begin{align*}
\text{for } (i = 1; i <= n; i++) \{ \\
\ & x = a[i]; \\
\ & \text{if } (x > 0) \ u = z * x; \\
\ & \text{else } u = b[i]; \\
\ & s = s + u;
\}
\end{align*}
\]

- By using conditional move instructions, we can unroll loops containing internal control-flow and end up with “straight-line” code
  - helps because instruction scheduling is then more effective
  - works if the two instruction blocks of the if are small in size

Dead-Code Elimination

A variable is dead if it is not used on any path from the location in the code where it is defined to the exit point of the routine.
An instruction is dead if it computes values that are not used on any executable path leading from the instruction.

- Many compiler optimizations create dead code as part of the division of labor principle: *keep each optimization phase as simple as possible (to make it easy to implement and maintain) and leave it to other passes to clean up the mess...*
- Detecting dead code local to a procedure is simple
- Interprocedural analysis is required to detect dead variables with wider visibility

Dead-Code Elimination Example

\[
\begin{align*}
\text{entry} & \quad k \text{ is only used to define new values for itself!} \\
 i = 1 & \quad \text{print}(l) \\
 j = 2 & \quad \text{return } j + i \\
 k = 3 & \\
 n = 4 &
\end{align*}
\]

\[
\begin{align*}
\text{entry} & \\
 i = 1 & \quad \text{print}(l) \\
 j = 2 & \quad \text{return } j + i \\
 n = 4 &
\end{align*}
\]
Branch Prediction

Branch prediction refers to predicting whether a conditional branch transfers flow of control or not.
Modern machines rely on branch prediction to make the right guess on which instructions to fetch after a branch.
Static prediction: the compiler predicts which way the branch is likely to go and places its prediction in the branch instruction itself.
Dynamic prediction: the hardware remembers for each recently executed branch, which way it went the previous time and predicts that it will go the same way.

Static vs. Dynamic Branch Prediction

Perfect static production results in a dynamic misprediction rate of about 9% for C and about 6% for Fortran programs.
Profile-based prediction approaches the accuracy of perfect static prediction.
Heuristic-based static prediction results in a dynamic misprediction rate of about 20% (for C).
Hardware-based prediction typically results in a misprediction rate of about 11% (for C).
Relying on heuristics that mispredict 20% of branches is better than no prediction, but does not suffice in practice!

Static Branch Prediction

A simple rule used by many machines:
Backward branches are assumed to be taken, forward branches are assumed to be not-taken.
• When generating code for machines following this prediction rule, a compiler can order the basic blocks in such a way that the predicted-taken branches go towards lower addresses.
• Several empirically validated heuristics help the compiler predict the direction of a branch.

Peephole Optimization

Peephole optimization is an effective post-pass technique for improving assembly code.
Basic Idea:
– Discover local improvements by looking at a window of the code (a peephole).
  Peephole: a short sequence of (usually contiguous) instructions.
  • slide the peephole over the code, and examine the contents.
– The optimizer replaces the sequence with another equivalent one (but faster).
Peephole Optimization (Cont.)

Write peephole optimizations as rewrite rules

\[ i_1, \ldots, i_n \rightarrow j_1, \ldots, j_m \]

where the RHS is the improved version of the LHS

- Example:
  \( \text{move } r_1 \Rightarrow r_2, \text{move } r_2 \Rightarrow r_1 \rightarrow \text{move } r_1 \Rightarrow r_2 \)
  - Works if \( \text{move } r_2 \Rightarrow r_1 \) is not the target of a jump
- Another example:
  \( \text{addiu } r_1, i \Rightarrow r_1 \text{ addiu } r_1, j \Rightarrow r_1 \rightarrow \text{addiu } r_1, i+j \Rightarrow r_1 \)

Peephole Optimization (Cont.)

- Many (but not all) of the basic block (i.e. local) optimizations can be cast as peephole optimizations
  - Example: \( \text{add } r_1, 0 \Rightarrow r_2 \rightarrow \text{move } r_1 \Rightarrow r_2 \)
  - Example: \( \text{move } r \Rightarrow r \rightarrow \)
  - These two together eliminate \( \text{add } r, 0 \Rightarrow r \)
- Just like most compiler optimizations, peephole optimizations need to be applied repeatedly to achieve maximum effect

Peephole Optimization Examples

\[
\begin{align*}
\text{store } r_1 & \Rightarrow r_0, 8 \\
\text{load } r_0, 8 & \Rightarrow r_2 \\
\text{move } r_1 & \Rightarrow r_2 \\
\text{addiu } r_1, 0 & \Rightarrow r_2 \\
\text{mult } r_3, r_2 & \Rightarrow r_2 \\
\text{jumpl } L1 & \Rightarrow \text{jumpl } L2 \\
L1: \text{jumpl } L2 & \Rightarrow L1: \text{jumpl } L2
\end{align*}
\]

Machine Idioms & Instruction Combining

Machine idioms are (sequences of) instructions for a particular architecture that provide a more efficient way of performing a computation than one might use if compiling for a more generic architecture.

Pattern matching is used to recognize opportunities where

- Individual instructions can be substituted by faster and more specialized instructions that achieve the same purpose
- Groups of instructions can be combined into a shorter or faster sequence
Examples of Instruction Combining

If high-order 20 bits of const are all 0

- `sethi %hi(const) ⇒ r1`
- `or r1, %lo(const) ⇒ r1`
- `⇒ add r0, const ⇒ r1`

- `mult r1, 5 ⇒ r2`
- `⇒ shl r1, 2 ⇒ r2`
- `add r1, r2 ⇒ r2`

- `sub r1, r2 ⇒ r3`
- `....`
- `subcc r1, r2 ⇒ r0`
- `bg L1`
- `⇒ subcc r1, r2 ⇒ r3`
- `....`
- `bg L1`