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

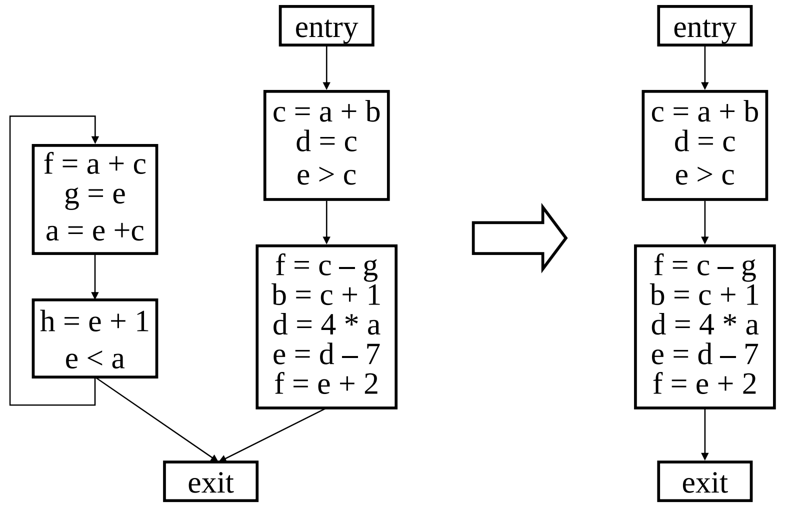
Unreachable code is code that cannot be executed, regardless of the input data

- 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

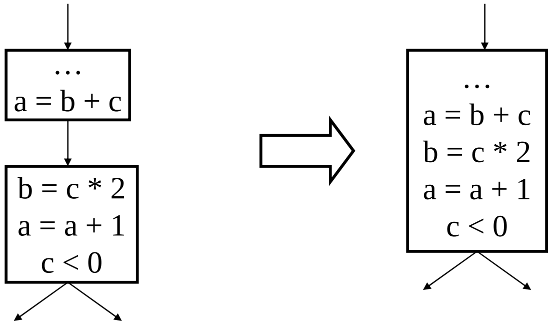
- reduces the code space
- improves instruction-cache utilization
- enables other control-flow transformations

Unreachable-Code Elimination



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



Straightening Example

Straightening in the presence of fall-throughs is tricky...

```
L1:  …
     a = b + c
     goto L2

L6:  …
     goto L4

L2:  b = c * 2
     a = a + 1
     if c < 0 goto L3
L5:  …
```

⇒

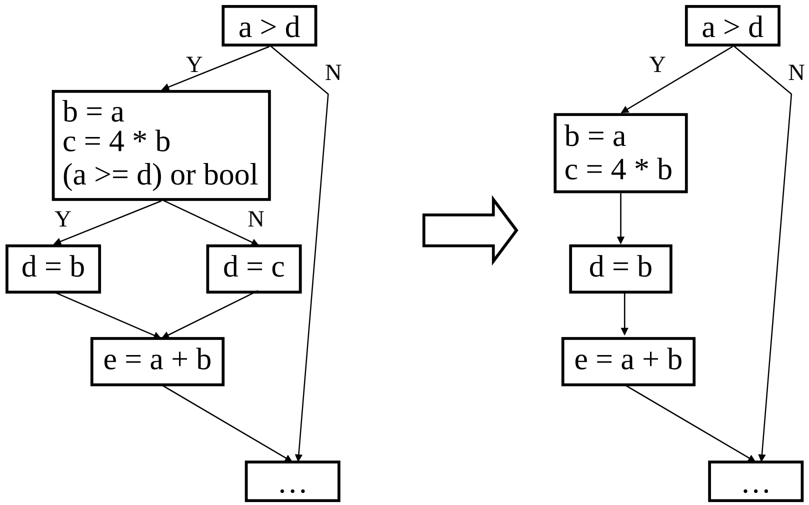
```
L1:  …
     a = b + c
     b = c * 2
     a = a + 1
     if c < 0 goto L3
     goto L5

L6:  …
     goto L4

L5: …
```

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)

If Simplification Example



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 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

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

```
     s = 0
     i = 0
     i = i + 1
     s = s + i
     i = i + 1
     s = s + i
     i = i + 1
     s = s + i
     i = i + 1
     s = s + i
L2: …
```

```
     i = 4
     s = 10
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

```
for (i = 0; i < 100; i++) {
    a[i] = i + 1;
}
```

Loop bounds are known

```
i = 0;
while (i < 100) {
    a[i] = i + 1;
    i++;
}
```

```
i = 0;
do {
    a[i] = i + 1;
    i++;
} while (i < 100)
```

Loop Inversion Example 2

```
for (i = k; i < n; i++) {
    a[i] = i + 1;
}
```

⇒

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

Loop bounds
are unknown

Unswitching

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

```
for (i = 1; i < 100; i++) {
    if (k == 2)
        a[i] = a[i] + 1;
    else
        a[i] = a[i] – 1;
}
```

⇒

```
if (k == 2) {
    for (i = 1; i < 100; i++)
        a[i] = a[i] + 1;
} else {
    for (i = 1; i < 100; i++)
        a[i] = a[i] – 1;
}
```

Unswitching Example

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

⇒

```
if (k == 2) {
    for (i = 1; i < 100; i++) {
        if (a[i] > 0)
            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: if  a >= 0 goto L2
     …
L2: …
```

⇒

```
     if a == 0 goto L2
     …
L1: if  a >= 0 goto L2
     …
L2: …
```

```
     goto L1
L1: …
```

⇒

```
L1: …
```

```
     if a == 0 goto L1
     goto L2
L1: …
```

⇒

```
     if a != 0 goto L2
L1: …
```

Eliminating Useless Control-Flow

<u>The Problem:</u>

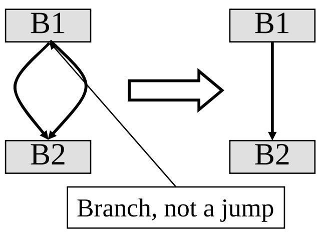
- After optimization, the CFG might contain empty blocks
- "Empty" blocks still end with either a branch or jump
- Produces jump to jump, which wastes time and space

<u>The Algorithm:</u> (Clean)

- Use four distinct transformations
- Apply them in a carefully selected order
- Iterate until done

Eliminating Useless Control-Flow

<u>Transformation 1</u>



Eliminating redundant branches

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?

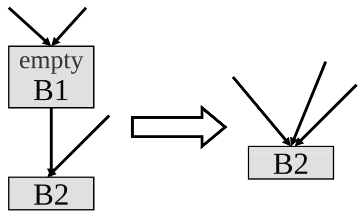
- By rewriting other branches

How do we recognize it?

- Check each branch

Eliminating Useless Control-Flow

<u>Transformation 2</u>



Eliminating empty blocks

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

How does this happen?

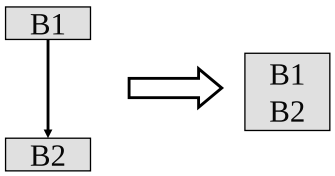
- By eliminating operations in B1

How do we recognize it?

- Test for empty block

Eliminating Useless Control-Flow

Transformation 3



Eliminating non-empty blocks

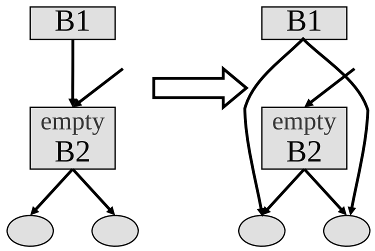
Coalescing blocks
- Neither block must be empty
- B1 ends with a jump to B2
- B2 has one predecessor
- Combine the two blocks
- Eliminates a jump

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

How do we recognize it?
- Check target of jump

Eliminating Useless Control-Flow

Transformation 4



Hoisting branches
from empty blocks

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.

```
      …
     r1 = r2 + r3
     r4 = r3 shl 2
     r2 = r2 + 1
     r2 = r4 – r2
     goto L1
      …
     r5 = r4 – 6
     r4 = r3 shl 2
     r2 = r2 + 1
     r2 = r4 – r2
L1:   …
```

⇒

```
      …
     r1 = r2 + r3
     goto L2
      …
     r5 = r4 – 6
L2:  r4 = r3 shl 2
     r2 = r2 + 1
     r2 = r4 – r2
L1:   …
```

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

```
    if a > b goto L1
    max = b
    goto L2
L1: max = a
L2: …
```

```
t1 = a > b
max = b
max = (t1) a
```

Conditional Moves Help Loop Unrolling

```
for (i = 1; i <= n; i++) {
   x = a[i];
   if (x>0) u = z * x;
   else u = b[i];
   s = s + u;
}
```

```
for (i = 1; i <= n; i++) {
   x = a[i];
   w = z * x;  u = b[i];
   u = (x>0) w;
   s = s + u;
}
```

- 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

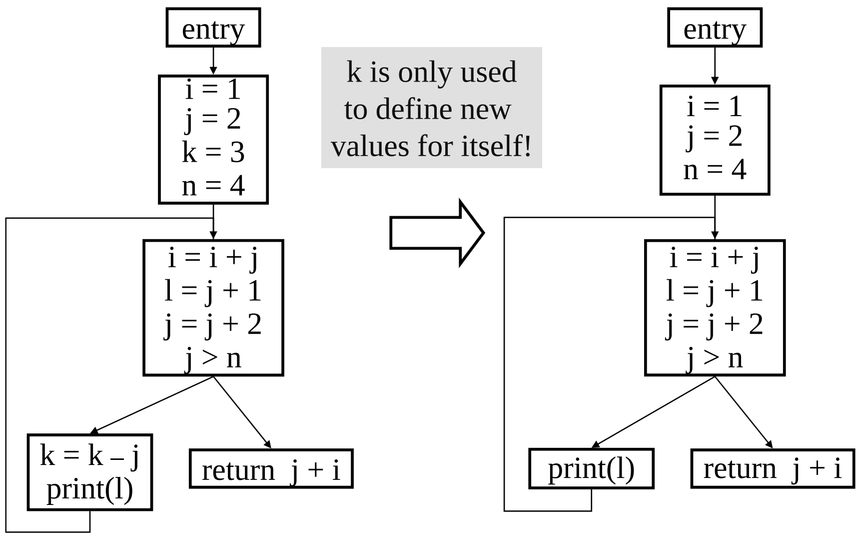
Dead-Code Elimination

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



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 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

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!

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:

  move r1⇒r2, move r2⇒r1 → move r1⇒r2

  – Works if move r2⇒r1 is not the target of a jump
- Another example:

  addiu r1, i ⇒ r1 addiu r1, j ⇒ r1 → addiu r1, i+j ⇒ r1

Peephole Optimization Examples

store r1 ⇒ r0, 8 load r0, 8 ⇒ r2	store r1 ⇒ r0, 8 move r1 ⇒ r2
addiu r1, 0 ⇒ r2 mult r3, r2 ⇒ r2	mult r3, r1 ⇒ r2
jumpl L1 L1: jumpl L2	jumpl L2 L1: jumpl L2

Peephole Optimization (Cont.)

- Many (but not all) of the basic block (i.e. local) optimizations can be cast as peephole optimizations
  – Example: add r1, 0 ⇒ r2 → move r1 ⇒ r2

  – Example: move r ⇒ r →

  – These two together eliminate add r, 0 ⇒ r
- Just like most compiler optimizations, peephole optimizations need to be applied repeatedly to achieve maximum effect

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
```