Optimizations

- Code transformations to improve program
  - Mainly: improve execution time
  - Also: reduce program size
- Can be done at high level or low level
  - E.g. constant folding
- Optimizations must be safe
  - Execution of transformed code must yield same results as the original code for all possible executions

Optimization Safety

- Safety of code transformations usually requires certain information which may not explicit in the code
- Example: dead code elimination
  1. \( x = y + 1; \)
  2. \( y = 2 * z; \)
  3. \( x = y + z; \)
  4. \( z = 1; \)
  5. \( z = x; \)
- What statements are dead and can be removed?

Dead Code Example

- Add control flow to example:
  \[
  x = y + 1; \\
  y = 2 * z; \\
  \text{if} \; (d) \; x = y + z; \\
  z = 1; \\
  z = x;
  \]
- Is ‘\( x = y + 1 \)’ dead code? Is ‘\( z = 1 \)’ dead code?
**Dead Code Example**

- Add more control flow:
  ```
  while (c) {
    x = y + 1;
    y = 2 * z;
    if (d) x = y+z;
    z = 1;
  }
  z = x;
  ```
  Is 'x = y+1' dead code? Is 'z = 1' dead code?

---

**Low-level Code**

- Much harder to eliminate dead code in low-level code:
  ```
  label L1
  fjump c L2
  x = y + 1;
  y = 2 * z;
  fjump d L3
  x = y+z;
  label L3
  z = 1;
  jump L1
  label L2
  Z = x;
  ```

---

**Optimizations and Control Flow**

- Application of optimizations requires information
  - Dead code elimination: need to know if variables are dead when assigned values

- Required information:
  - Not explicit in the program
  - Must compute it statically (at compile-time)
  - Must characterize all dynamic (run-time) executions

- Control flow makes it hard to extract information
  - Branches and loops in the program
  - Different executions = different branches taken, different number of loop iterations executed

---

**Control Flow Graphs**

- **Control Flow Graph (CFG)** = graph representation of computation and control flow in the program
  - framework to statically analyze program control-flow

- Nodes are **basic blocks** = sequences of consecutive non-branching statements

- Edges represent possible flow of control from the end of one block to the beginning of the other
  - There may be multiple incoming/outgoing edges for each block
**CFG Example**

**Program**

\[
\begin{align*}
x &= z - 2; \\
y &= 2 \times z; \\
\text{if (c) } &\{ \\
&\quad x = x + 1; \\
&\quad y = y + 1; \\
\} \\
\text{else } &\{ \\
&\quad x = x - 1; \\
&\quad y = y - 1; \\
\} \\
z &= x + y;
\end{align*}
\]

**Control Flow Graph**

\[
B_1 \quad B_2 \quad B_3 \quad B_4
\]

\[
\begin{align*}
T &\quad F \\
x &= x + 1; \\
y &= y + 1; \\
x &= x - 1; \\
y &= y - 1; \\
z &= x + y;
\end{align*}
\]

**Basic Blocks**

- **Basic block** = sequence of consecutive statements such that:
  - Control enters only at beginning of sequence
  - Control leaves only at end of sequence

\[
\begin{align*}
a &= a + 1; \\
b &= c \times b; \\
d &= c \times b;
\end{align*}
\]

- No branching in or out in the middle of basic blocks

**Computation and Control Flow**

- **Basic Blocks**:
  - **Nodes in the graph** = computation in the program
  - **Edges in the graph** = control flow in the program

**Multiple Program Executions**

- **CFG models all program executions**
- **Possible execution** = path in the graph
- **Multiple paths = multiple possible program executions**

**Execution 1**

- CFG models all program executions
- Possible execution = path in the graph
- Execution 1:
  - C is true
  - Program executes basic blocks $B_1$, $B_2$, $B_3$, $B_4$

**Execution 2**

- CFG models all program executions
- Possible execution = path in the graph
- Execution 2:
  - C is false
  - Program executes basic blocks $B_1$, $B_2$, $B_4$
Edges Going Out

- Multiple outgoing edges
- Basic block executed next may be one of the successor basic blocks
- Each outgoing edge = outgoing flow of control in some execution of the program

Basic Block

outgoing edges

Edges Coming In

- Multiple incoming edges
- Control may come from any of the successor basic blocks
- Each incoming edge = incoming flow of control in some execution of the program

incoming edges

Basic Block

Building the CFG

- Currently the compiler represents the program using either High IR or low IR
- Can construct CFG for either of the two intermediate representations

  - Build CFG for High IR
    - Construct CFG for each High IR node
  - Build CFG for Low IR
    - Analyze jump and label statements

CFG for High-level IR

- CFG(S) = flow graph of high level statement S
- CFG (S) is single-entry, single-exit graph:
  - one entry node (basic block)
  - one exit node (basic block)

  CFG(S) =

  Entry

  Exit

  Recursively define CFG(S)

CFG for Block Statement

- CFG( S1; S2; …; SN ) =

  CFG(S1)

  CFG(S2)

  …

  CFG(SN)

CFG for If-then-else Statement

- CFG ( if (E) S1 else S2 )

  if (E)

  T

  F

  CFG(S1)

  CFG(S2)

Empty basic block
Recursive CFG Construction

- Nested statements: recursively construct CFG while traversing IR nodes
- Example:

```
while (c) {
    x = y + 1;
    y = 2 * z;
    if (d) x = y + z;
    z = 1;
}
```

Recursive CFG Construction

- Nested statements: recursively construct CFG while traversing IR nodes
- Example:

```
while (c) {
    x = y + 1;
    y = 2 * z;
    if (d) x = y + z;
    z = 1;
}
```

CFG for If-then Statement

- CFG( if (E) S )

```
if (E) -> T -> CFG(S) -> F
```

CFG for While Statement

- CFG for: while (e) S

```
if (e) -> T -> CFG(S) -> F
```

Recursive CFG Construction

- Nested statements: recursively construct CFG while traversing IR nodes
- Example:

```
while (c) {
    x = y + 1;
    y = 2 * z;
    if (d) x = y + z;
    z = 1;
}
```

Recursive CFG Construction

- Nested statements: recursively construct CFG while traversing IR nodes
- Example:

```
while (c) {
    x = y + 1;
    y = 2 * z;
    if (d) x = y + z;
    z = 1;
}
```
Recursive CFG Construction

- Simple algorithm to build CFG
- Generated CFG
  - Each basic block has a single statement
  - There are empty basic blocks
- Small basic blocks = inefficient
  - Small blocks = many nodes in CFG
  - Compiler uses CFG to perform optimization
  - Many nodes in CFG = compiler optimizations will be time- and space-consuming

Efficient CFG Construction

- Basic blocks in CFG:
  - As few as possible
  - As large as possible
- There should be no pair of basic blocks (B1,B2) such that:
  - B2 is a successor of B1
  - B1 has one outgoing edge
  - B2 has one incoming edge
- There should be no empty basic blocks

Example

- Efficient CFG:

    while (c) {
        x = y + 1;
        y = 2 * z;
        if (d) x = y + z;
        z = 1;
    }
    z = x;

    if (c)
    x = y+1
    y = 2*z
    if (d) x = y+z
    z = 1
    z = x

CFG for Low-level IR

- Basic block start:
  - At label instructions
  - After jump instructions
- Basic blocks end:
  - At jump instructions
  - Before label instructions

    label L1
    jump c L2
    x = y + 1;
    y = 2 * z;
    jump d L3
    x = y+z;
    label L3
    z = 1;
    jump L1
    label L2
    z = x;

CFG for Low-level IR

- Conditional jump:
  2 successors
- Unconditional jump:
  1 successor
CFG for Low-level IR

- If (c)
  - x = y + 1
  - y = 2 * y
- If (d)
  - x = y + 2
  - z = 1
  - z = x

- Jump c L2
- Label L1
- x = y + 1;
- y = 2 * z;
- Jump d L3
- Label L3
- z = 1;
- Jump L1
- Label L2
- z = x;