Introduction to Control-flow Analysis

CIS410/510 Program Analysis and Transformation

Why do we need analysis?

Analysis (and transformation) are useful everywhere

- Analysis and transformations can be performed at run time and link time, not just at “compile time”
- Translation can be used to improve security
- Analysis can be used in software engineering
- Program understanding
- Reverse engineering
- Increased interaction between hardware and compilers can improve performance

Bottom line

- Analysis and transformation play essential roles in computer systems
- Computation important $\Rightarrow$ understanding computation important
Consider optimization

An *optimization* is a transformation that is expected to improve the program in some way; often consists of analysis and transformation, e.g., decreasing the running time or decreasing memory requirements.

- Machine-independent optimizations
  - Eliminate redundant computation
  - Move computation to less frequently executed place
  - Specialize some general purpose code
  - Remove useless code

Consider optimizations (cont.)

- Machine-dependent optimizations
  - Replace a costly operation with a cheaper one
  - Replace a sequence of operations with a cheaper one
  - Hide latency
  - Improve locality
  - Reduce power consumption

- Enabling transformations
  - Expose opportunities for other optimizations
  - Help structure optimizations
Examples

- **Arithmetic simplification**
  - **Constant folding**
    - e.g., \( x = \frac{8}{2}; \rightarrow x = 4; \)
  - **Strength reduction**
    - e.g., \( x = y \times 4; \rightarrow x = y << 2; \)
  - **Constant propagation**
    - e.g., \( x = 3; \rightarrow x = 3; \)
    - \( y = 4 + x; \rightarrow y = 4 + 3; \rightarrow y = 7; \)
  - **Copy propagation**
    - e.g., \( x = z; \rightarrow x = z; \)
    - \( y = 4 + x; \rightarrow y = 4 + z; \)

---

Example optimizations (cont.)

- **Common subexpression elimination (CSE)**
  - e.g., \( x = a + b; \rightarrow t = a + b; \)
  - \( y = a + b; \rightarrow x = t; \)
  - \( y = t; \)
- **Dead (unused) assignment elimination**
  - e.g., \( x = 3; \)
    - \( \ldots x \text{ not used...} \)
    - \( x = 4; \)
- **Dead (unreachable) code elimination**
  - e.g., \( \text{if (0) \{ } \)
    - \( \text{printf(“Output...”);} \)
    - \( \text{\}} \)
Example optimizations (cont.)

- Loop-invariant code motion
  e.g.,
  ```c
  for(i=0; i<10; ++i){
      x = 2;
      ...
  }
  ```
  →
  ```c
  x = 2;
  for(i=0; i<10; ++i){
      ...
  }
  ```

- Loop unrolling
  e.g.,
  ```c
  for(i=0; i<10; ++i){
      a[i] = a[i] + 1;
  }
  ```
  →
  ```c
  for(i=0; i<10; i+=2){
      a[i] = a[i] + 1;
      a[i+1] = a[i+1] + 1;
  }
  ```

Is an optimization legal or worthwhile?

- Safety: does it preserve behavior?
- Profitability: does it actually improve the code?
- Opportunity: is it widely applicable?
- Cost (compilation): can it be practically performed?
- Cost (complexity): can it be practically implemented?
Scope of analysis/transformations

- **Peephole**
  - Consider a small window of instructions
  - Usually machine specific
  - Know nothing about context

- **Local**
  - Consider blocks of straight-line code (no control flow)
  - Simple to analyze
  - Know nothing about context

- **Global (intraprocedural)**
  - Consider entire procedures
  - Must consider control flow (branches, loops, merging of control flow)
  - Use dataflow analysis
  - Make certain assumptions in method calls

- **Whole program (interprocedural)**
  - Consider multiple procedures
  - Analysis even more complex (calls, returns)
  - Hard with separate (file) compilation

Limits of compiler optimization

- **Fully optimizing compiler (FOC)**
  - FOC(P) = P_{opt}
  - P_{opt} is the smallest program with the same I/O behavior as P

- **Observe**
  - If program Q produces no output and never halts, then FOC(Q) = L: goto L (or something equivalent)
  - So -- > we solved the halting problem?!'
    - Halting problem: Given a description of an arbitrary computer program, decide whether the program finishes running or continues to run forever. (Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist).

- **Moral**
  - Cannot build FOC
  - But, can always build a better optimizing compiler
    - *Full employment theorem* for compiler writers
“Optimizations” don’t always help

- Common subexpression elimination
  \[ x = a + b; \]
  \[ y = a + b; \]
  \[ t = a + b; \]
  \[ x = t; \]
  \[ y = t; \]
  2 additions,
  4 reads, 2 writes

  1 addition,
  4 reads, 3 writes

Phase ordering problem

- In what order should optimizations be performed?
- Simple dependencies
  - One optimization creates opportunity for another (e.g., copy propagation and dead code elimination)
- Cyclic dependencies
  - E.g., constant folding and constant propagation
- Adverse interactions
  - E.g., common subexpression elimination and register allocation
  - E.g., register allocation and instruction scheduling
Engineering issues

- Building compilers (and source analysis and transformation tools in general) requires balancing of multiple goals
  - Benefit for *typical* programs (or you could target an important domain)
  - Complexity of implementation
  - Compilation speed

- Overall goal
  - Identify a small set of general (or DSL) analyses and transformations
  - Typically never finished (just one more...)

Beyond optimization

- Security and correctness
  - Can we check whether variables are initialized correctly?
  - Can we check whether pointers and addresses are valid?
  - Can we detect when untrusted code accesses a sensitive part of the system?
  - Can we detect whether locks are used properly in concurrent codes?
  - Can we use compilers to certify that code is “correct”?
  - Can we use compilers to obfuscate code?
  - ...
Data-flow vs control-flow analysis

- **Data-flow**
  - Flow of data values from defs to uses
    - \( x = y + 4; \) // \( \text{defs} = \{x\}, \text{uses} = \{y\} \)

- **Control-flow analysis**
  - Sequencing of operations
    - e.g., evaluation of then-code and else-code depends on if-test

Control-flow analysis

- **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

- **To perform transformations safely, control flow information**
  - Must be computed statically (at compile time)
  - Must characterize all dynamic (run-time) executions

- **Control-flow analysis**
  - Building basic blocks
  - Building control-flow graphs
  - Loops
Representing control flow

- High-level representation
  - Control flow is implicit in an AST

- Low-level representation
  - Use a control-flow graph (CFG)
    - Nodes represent statements (low-level linear IR)
    - Edges represent explicit flow of control

- Other possibilities
  - Control dependencies in program dependence graph (PDG) [Ferrante87]

What is control-flow analysis?

Control-flow analysis discovers the flow of control within a procedure (e.g., by building a CFG, identifying loops)

Example:

```plaintext
1   a = 0;
2   b = a * b;
3 L1: c = b / d;
4   if (c < x) goto L2;
5   e = b / c;
6   f = e + 1;
7 L2: g = f;
8   h = t - g;
9   if (e > 0) goto L3;
10  goto L1;
11 L3: return;
```

CFG models all program executions!
Basic blocks

- Definition
  - A basic block is a sequence of straight-line code that can be entered only at the beginning and exited only at the end

```
g = f;
h = t – g;
e > 0?
  No
  Yes
```

Building basic blocks

- Identify leaders
  - The first instruction in a procedure, or
  - The target of any branch, or
  - An instruction immediately following a branch (implicit target)
- Include all subsequent instructions until the next leader
Algorithm for building basic blocks

**Input**: List of n instructions (instr[i] = i\textsuperscript{th} instruction)

**Output**: Set of leaders & list of basic blocks
(block[x] is block with leader x)

leaders = \{1\} \quad \text{\textit{// First instruction is a leader}}

\textbf{for} i = 1 \textbf{to} n \quad \text{\textit{// Find all leaders}}

\textbf{if} instr[i] is a branch
leaders = leaders \cup \text{set of potential targets of instr[i]}

\textbf{foreach} x \in leaders
block[x] = \{ x \}
i = x + 1 \quad \text{\textit{// Fill out x’s basic block}}

\textbf{while} i \leq n \textbf{ and } i \notin leaders
block[x] = block[x] \cup \{ i \}
i = i + 1

Building basic blocks example (low-level IR)

**Example:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a = 0;</td>
</tr>
<tr>
<td>2</td>
<td>b = a * b;</td>
</tr>
<tr>
<td>3</td>
<td>L1: c = b / d;</td>
</tr>
<tr>
<td>4</td>
<td>\textbf{if} (c &lt; x) \textbf{goto} L2;</td>
</tr>
<tr>
<td>5</td>
<td>e = b / c;</td>
</tr>
<tr>
<td>6</td>
<td>f = e + 1;</td>
</tr>
<tr>
<td>7</td>
<td>L2: g = f;</td>
</tr>
<tr>
<td>8</td>
<td>h = t - g;</td>
</tr>
<tr>
<td>9</td>
<td>\textbf{if} (e &gt; 0) \textbf{goto} L3;</td>
</tr>
<tr>
<td>10</td>
<td>\textbf{goto} L1;</td>
</tr>
<tr>
<td>11</td>
<td>L3: \textbf{return};</td>
</tr>
</tbody>
</table>

**Leaders?**
- \{1,3,5,7,19,11\}

**Blocks?**
- \{1,2\}
- \{3,4\}
- \{5,6\}
- \{7,8,9\}
- \{10\}
- \{11\}
Extended basic blocks

- Extended basic blocks
  - A maximal sequence of instructions that has no merge points in it (except perhaps in the leader)
  - Single entry, multiple exits

- How are these useful?
  - Increases the scope of any local analysis that "flows forwards" (e.g., copy propagation, register renaming, instruction scheduling)

- Reverse extended basic blocks
  - Useful for "backward flow" problems

Building a CFG from basic blocks

- Construction
  - Each CFG node represents a basic block
  - There is an edge from node $i$ to $j$ if
    - The last statement of block $i$ branches to the first statement of $j$, or
    - Block $i$ does not end with an unconditional branch and is immediately followed in program order by block $j$ (fall through)

**Input:** A list of $m$ basic blocks (block)
**Output:** A CFG where each node is a basic block

```plaintext
for i = 1 to m
    x = last instruction of block[i]
    if instr x is a branch
        for each target (to block j) of instr x
            create an edge from block i to block j
    if instr x is not an unconditional branch
        create an edge from block i to block i+1
```

L1:
  4 goto L1;

1 2

3 5
Details

- Multiple edges between two nodes
  
  ... 

  If (a < b) goto L2

  L2: ...

  - Combine these edges into one edge

- Unreachable code

  ... 

  goto L1

  L0: a = 10

  L1: ...

  - Perform DFS from entry node
  - Mark each basic block as it is visited
  - Unmarked blocks are unreachable

Recall: Depth-first and Breadth-first search

Depth-first search

Breadth-first search
CFG for high-level IR

- \( \text{CFG}(S) = \text{flow graph of high-level statement } S \)
- \( \text{CFG}(S) \) is single-entry, single-exit graph:
  - One entry node (basic block)
  - One exit node (basic block)

Recursively define \( \text{CFG}(S) \)
CFG for block statement

- CFG \((S_1; S_2; \ldots; S_N)\) =

  - CFG\((S_1)\)
  - CFG\((S_2)\)
  - \ldots
  - CFG\((S_N)\)

CFG for if-then-else statement

- CFG\(\text{if}(E) S_1 \text{else} S_2\)

  - if\((E)\)
    - T \rightarrow CFG\((S_1)\)
    - F \rightarrow CFG\((S_2)\)
  - Empty basic block
CFG for if-then statement

- CFG ( \( \text{if (E) S1} \) )

CFG for while statement

- CFG ( \( \text{while (E) S} \) )
Recursive CFG construction

- Nested statements: recursively construct CFG while traversing IR nodes

Example

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

Recursive CFG construction

- Nested statements: recursively construct CFG while traversing IR nodes

Example

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

High-level vs low-level CFG
Loop concepts

- **Loop:** Strongly connected component of CFG
- **Loop entry edge:** Source not in loop & target in loop
- **Loop exit edge:** Source in loop & target not in loop
- **Loop header node:** Target of loop entry edge
- **Natural loop:** Loop with only a single loop header
- **Back edge:** Target is loop header & source is in loop
- **Loop tail node:** Source of back edge
- **Loop preheader node:** Single node that’s source of the loop entry edge
- **Nested loop:** Loop whose header is inside another loop
- **Reducible flow graph:** CFG whose loops are all natural loops
Picturing loop terminology

Not all loops have preheaders
- Sometimes it’s useful to create them

Without preheader node
- There can be multiple entry edges

With single preheader node
- There is only one entry edge

Useful when moving code outside the loop
- Don’t have to replicate code for multiple entry edges

The value of preheader nodes
Identifying loops

- **Why?**
  - Most execution time spent in loops so optimizing loops will often give the most benefit

- **Many approaches**
  - Interval analysis
    - Exploit the natural hierarchical structure of programs
    - Decompose the program into nested regions called intervals
  - Structural analysis: a generalization of interval analysis
  - Identify *dominators* to discover loops

Dominator terminology

- **D dominators**
  - d dom i if all paths from entry to node i include d

- **Strict dominators**
  - d sdom i if d dom i and d ≠ I

- **Immediate dominators**
  - a idom b if a sdom b and there does not exist a node c

- **Post dominators**
  - p pdom i if every possible path from i to exit includes p
    (p dom i in the flow graph whose arcs are reversed and entry and exit are interchanged)
Why Go To All This Trouble?

- **Modern languages provide structured control flow**
  - Shouldn’t the compiler remember this information rather than throw it away and then re-compute it?

- **Answers?**
  - We may want to work on the binary code in which case such information is unavailable
  - Most modern languages still provide a `goto` statement
  - Languages typically provide multiple types of loops. This analysis lets us treat them all uniformly
  - We may want a compiler with multiple front ends for multiple languages; rather than translate each language to a CFG, translate each language to a canonical LIR, and translate that representation once to a CFG

Using CFGs

- Next: use CFG to statically extract information about the program
  - Reason at compile time about the runtime values of variables and expressions in *all* program executions

- Extracted information example
  - Live variables

- **Idea**
  - Define *program points* in the CFG
  - Reason statically about how the information flows between these program points
Program points

- **Two program points** for each instruction:
  - There is a program point *before* each instruction
  - There is a program point *after* each instruction
    (think of them as the instruction’s imaginary friends)

- **In a basic block**:
  - Program point after an instruction = program point before the successor instruction

Program points example

- Multiple successor blocks means that the point at the end of the block has *multiple successor program points*
- Depending on the execution, *control flows from a program point to its successors*
- Allow multiple predecessors
- How does information propagate between program points?
Flow of extracted information

- **Question 1:** How does information flow between the program points before and after an instruction?
- **Question 2:** How does information flow between successor and predecessor basic blocks?
- ... in other words:
  - Q1: what is the effect of instructions?
  - Q2: what is the effect of control flow?

Using CFGs

- **To extract information:** reason about how it propagates between program points

- Rest of this lecture: how to use CFGs to compute information at each program point for:
  - *Live variable analysis*, which computes which variables are live at each program point
  - Very similar -- *Copy propagation analysis*, which computes the variable copies available at each program point.
Live variable analysis

- Computes live variables at each program point
  - Live variables: those variables holding values that may be used later (in some execution of the program)
  - Used, for example, in register allocation
- For an instruction $I$, consider:
  - $\text{in}[I] =$ live variables at program point before $I$
  - $\text{out}[I] =$ live variables at program point after $I$
- For a basic block $B$, consider:
  - $\text{in}[B] =$ live variables at beginning of $B$
  - $\text{out}[B] =$ live variables at end of $B$

- If $I =$ first instruction in $B$, then $\text{in}[B] = \text{in}[I]$
- If $I' =$ last instruction in $B$, then $\text{out}[B] = \text{out}[I']$

How to compute liveness?

- Answer Q1: for each instruction $I$, what is the relation between $\text{in}[I]$ and $\text{out}[I]$?
- Answer Q2: for each basic block $B$ with successor blocks $B_1, ..., B_n$, what is the relation between $\text{out}[B]$ and $\text{in}[B_1], ..., \text{in}[B_n]$?
Part 1: Analyze instructions

- Question: what is the relation between sets of live variable before and after an instruction?

- Examples:

  - conclude
    - in[1] = {y,z}
    - x = y + z;
    - out[1] = {z}

  - assume
    - in[1] = {y,z,t}
    - x = y + z;
    - out[1] = {x,t}

  - in[1] = {x,t}

- Yes! Knowing the live variables after I, we can compute variables before I.

- Mathematically:

  \[ \text{in}[I] = (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] \]

  where

  - \text{def}[I] = \text{variables defined (written) by instruction I}
  - \text{use}[I] = \text{variables used (read) by instruction I}
Computing use/def

- Compute use[I] and def[I] for each instruction I
  - If I is \( x = y \) OP \( z \): \( \text{use}[I] = \{y, z\} \) \( \text{def}[I] = \{x\} \)
  - If I is \( x = \text{OP} \) \( y \): \( \text{use}[I] = \{y\} \) \( \text{def}[I] = \{x\} \)
  - If I is \( x = y \): \( \text{use}[I] = \{y\} \) \( \text{def}[I] = \{x\} \)
  - If I is \( x = \text{addr} \) \( y \): \( \text{use}[I] = \{\} \) \( \text{def}[I] = \{x\} \)
  - If I is \( \text{if}(x) \): \( \text{use}[I] = \{x\} \) \( \text{def}[I] = \{\} \)
  - If I is \( \text{return} \) \( x \): \( \text{use}[I] = \{x\} \) \( \text{def}[I] = \{\} \)
  - If I is \( x = f(y_{1}, \ldots, y_{n}) \): \( \text{use}[I] = \{y_{1}, \ldots, y_{n}\} \) \( \text{def}[I] = \{x\} \)
  - Etc...

Example

- Example: block B with three instructions I1, I2, and I3
  
  \[
  \begin{align*}
  \text{Live}_1 &= \text{in}[B] = \text{in}[I1] \\
  \text{Live}_2 &= \text{out}[I1] = \text{in}[I2] \\
  \text{Live}_3 &= \text{out}[I2] = \text{in}[I3] \\
  \text{Live}_4 &= \text{out}[I3] = \text{out}[B]
  \end{align*}
  \]

- Relation between Live sets:
  
  \[
  \begin{align*}
  \text{Live}_1 &= (\text{Live}_2 \setminus \{x\}) \cup \{y\} \\
  \text{Live}_2 &= (\text{Live}_3 \setminus \{y\}) \cup \{z\} \\
  \text{Live}_3 &= (\text{Live}_4 \setminus \{\}) \cup \{d\}
  \end{align*}
  \]
Backward flow

- Relation
  \[ \text{in}[l] = (\text{out}[l] - \text{def}[l]) \cup \text{use}[l] \]

- The information flows backward!

- Instructions: can compute \text{in}[l] if we know \text{out}[l]

- Basic blocks: information about live variables flows from \text{out}[B] to \text{in}[B]

---

Part 2: Analyze control flow

- Question: for each basic block \( B \) with successor blocks \( B_1, \ldots, B_n \), what is the relation between \text{out}[B] and \text{in}[B_1], \ldots, \text{in}[B_n]?

- Examples:

- What is the general rule?
Analyze control flow (cont.)

- Rule: A variable is live at end of block B if it is live at the beginning of one (or more) successor blocks
- Characterizes all possible program executions
- Mathematically

\[
\text{out}[B] = \bigcup_{B' \in \text{succ}(B)} \text{in}[B']
\]

- Again, information flows backward: from successors B’ of B to basic block B

Constraint system

- Put the parts together: start with CFG and derive a system of constraints between live variable sets:

\[
\begin{align*}
\text{in}[I] &= (\text{out}[I] - \text{def}[I]) \cup \text{use}[I] & \text{For each instruction I} \\
\text{out}[B] &= \bigcup_{B' \in \text{succ}(B)} \text{in}[B'] & \text{For each basic block B}
\end{align*}
\]

- Solve constraints:
  - Start with empty sets of live variables (conservative)
  - Iteratively apply constraints
  - Stop when we reach a fixed point
    - Definition: a value \( v \) is a fixed point of a function \( f \) if and only if \( f(v) = v \)
    (More on fixed point iterations later)
Constraint solving algorithm

\[\text{for all instructions } i \text{ do } \text{in}[i] = \text{out}[i] = \emptyset;\]
\[\text{repeat}\]
\[\text{select an instruction } i \text{ (or a basic block } B) \text{ such that}\]
\[\text{in}[i] \neq (\text{out}[i] - \text{def}[i]) \cup \text{use}[i]\]
or (respectively)
\[\text{out}[B] \neq \bigcup_{B' \in \text{succ}(B)} \text{in}[B']\]

and update \(\text{in}[i]\) (or \(\text{out}[B]\)) accordingly
\[\text{until no such change is possible}\]

Example

\begin{align*}
\text{def} = \emptyset, \text{ use} = \{c\} & \quad \rightarrow \quad \text{if (c)} \\
\text{def} = \{x\}, \text{ use} = \{y\} & \quad \rightarrow \quad x = y + 1 \\
\text{def} = \{y\}, \text{ use} = \{z\} & \quad \rightarrow \quad y = 2 \times z \quad \text{if (d)} \\
\text{def} = \emptyset, \text{ use} = \{d\} & \quad \rightarrow \quad x = y + z \\
\text{def} = \{x\}, \text{ use} = \{y,z\} & \quad \rightarrow \quad x = y + z \\
\text{def} = \{z\}, \text{ use} = \emptyset & \quad \rightarrow \quad z = 1 \\
\text{def} = \{z\}, \text{ use} = \{x\} & \quad \rightarrow \quad z = x
\end{align*}
def = {}, use = {c}

def = {x}, use = {y}

def = {y}, use = {z}

def = {}, use = {d}

def = {x}, use = {y,z}

def = {z}, use = {}

def = {z}, use = {x}

---

strategy: pick program points in postorder