Final Review

Today
- Overview of what we have learned so far
- data-flow analysis review
- pointer and alias analysis
- interprocedural analysis
- register allocation
- loop skewing

Studying
- make sure to review terminology
  - (i.e. what does flow-sensitive mean?)

Big Picture: Traditional View of Compilers

Compiling down
- Translate high-level language to machine code

High-level programming languages
- Increase programmer productivity
- Improve program maintenance
- Improve portability

Low-level architectural details
- Instruction set
- Addressing modes
- Pipelines
- Registers, cache, and the rest of the memory hierarchy
- Instruction-level parallelism

Structure of a Typical Compiler

Analysis
- character stream
  - lexical analysis
  - tokens
  - syntactic analysis
  - AST
  - semantic analysis
  - annotated AST

Synthesis
- IR code generation
  - IR
  - optimization
  - IR
  - code generation
  - target language

Topics

1. Introduction
- Scanning and parsing

III. Low-Level Optimizations
- Register allocation
  - difference between Briggs and Chaitin
  - heuristics to determine spilling will be provided
- Instruction scheduling
  - list scheduling

III. Data-Flow and Control-Flow Analysis
- Dataflow analysis (review the projects)
  - Theoretic framework built on lattices
  - Control flow analysis: control-flow graphs, dominators, dominance frontiers, irreducibility
  - How do you create a control-flow graph and then perform a specified data-flow analysis?
  - Program optimizations (also example data-flow analyses)
  - dead-code elimination, constant propagation, CSE, loop-invariant code motion, copy propagation, induction variable elimination, strength reduction
III. Static Single Assignment
- SSA Form: types of data dependences, how to translate to minimal SSA
- global value numbering

IV. Advanced Analysis and Optimization
- Aliases
  - how do data-flow analysis algorithms use aliasing information?
  - how do we characterize alias analysis algorithms?
- Interprocedural Analysis
  - how do different levels of context information affect analysis results?
- Register allocation
  - difference between Briggs and Chairin
  - heuristics to determine spilling will be provided
- Profile-guided and dynamic optimizations
  - what types of profiling information can we collect and how is it useful?
  - when are dynamic optimizations profitable?

V. OOP and GC
- compiling for inheritance, polymorphism, and dynamic binding
- how does mark and sweep work?

VI. Parallelism and Locality
- Dependence analysis
  - what are dependence vectors and how are they calculated?
- Loop transformations
  - unimodular transformation framework
    - how is this used to show transformation legality?
  - Kelly and Pugh transformation framework
    - what transformations can be expressed in K&P and not in unimodular?
    - How is transformation legality shown?
- Tiling and Unroll and Jam
  - when is tiling legal?
  - how is tiling expressed in K&P framework?

Data-flow Equations for Reaching Definitions

Symmetry between reaching definitions and liveness
- Swap in[] and out[] and swap the flow direction

<table>
<thead>
<tr>
<th>Reaching Definitions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>in[n] = \bigcup_{p \in pred[n]} out[s]</td>
<td>out[n] = \bigcup_{s \in succ[n]} in[s]</td>
</tr>
</tbody>
</table>

Entry

Def of x = x

Is x def’d along this path?

Use of x = x

Entry

main() {
  int *a, *b, c, d;
  a = &c;
  b = &d;
  foo(&a, &a);
  foo(&b, &a);
}

void foo(int** x, int **y) {
  *x = *y;
  **x = 3;
}
**Garbage Collection Example**

```java
class MultipleParams {
    public static void main(String[] a) {
        System.out.println(new Foo().testing()); // call 1, call 2, call 3
    }
}

class Foo {
    Foo m1;
    public int testing() {
        Foo x; Foo y;
        x = new Foo(); m1 = new Foo(); // call 4, call 5, call 6
        y = new Foo(); y.m1 = x; // call 7
        return thisfoobar(x, y); // call 8
    }
}
```

**Example**

```java
Sample code
do i = 1, 6
do j = 1, 5
A(2i, j) = A(i, j-1)
enddo
```

**Dependence**
- \(2i+1, j = 0, j_i = j_2 - 1\), solution: YES

**Distance/Direction Vector**
- \((i_1, j_1) + (d_i, d_j) = (i_2, j_2)\), \(d_j = 1, d_i = ?, d = (<,1)\)

**Dependence Relation**
- \(\{i, j\} \rightarrow \{2i, j + 1\} | 1 < i < 3 \land 1 < j < 4\)

**Tiling**

**Sample code**
do i = 1, 6
do j = 1, 5
A(2i, j) = A(i, j-1)
enddo

**Dependence Relation**
- \(\{i, j\} \rightarrow \{2i, j + 1\} | 1 < i < 3 \land 1 < j < 4\)

**Tiling Both Loops with tile size 4**
- \(\{i, j\} \rightarrow \{(j-1)/4, (i-1)/4, i, j\}\)

**Loop Transformations**

**Original code**
do i = 1, 6
do j = 1, 7
A(i, j) = A(i-1, j-1) + 1
class Example {
    Sample code
do i = 1, 6
do j = 1, 5
    A(2i, j) = A(i, j-1)
    enddo
    enddo
    Dependence
    \(2i+1, j = 0, j_i = j_2 - 1\), solution: YES
    Distance/Direction Vector
    \((i_1, j_1) + (d_i, d_j) = (i_2, j_2)\), \(d_j = 1, d_i = ?, d = (<,1)\)
    Dependence Relation
    \(\{i, j\} \rightarrow \{2i, j + 1\} | 1 < i < 3 \land 1 < j < 4\)
    Which loop can we parallelize?
### Loop Skewing

**Original code**

```plaintext
do  i = 1, 6
do  j = 1, 5
   A(i, j) = A(i-1, j+1)+1
  enddo
enddo
```

**Distance vector:** (1, -1)

Can we permute the original loop?

**Skewing:**

\[
\begin{bmatrix}
1 & 0 \\
1 & 1
\end{bmatrix}
\begin{bmatrix}
i \\
j
\end{bmatrix} =
\begin{bmatrix}
i \\
i+j
\end{bmatrix}
\]

---

### Transforming the Dependences to Check Legality

**Original code**

```plaintext
do  i = 1, 6
  do  j = 1, 5
    A(i, j) = A(i-1, j+1)+1
  enddo
enddo
```

**Dependence vector:**

\[
\begin{bmatrix}
1 & 0 \\
1 & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
-1
\end{bmatrix} =
\begin{bmatrix}
1 \\
0
\end{bmatrix}
\]