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 "words"
  syntactic analysis
  AST "sentences"
  semantic analysis
  annotated AST
  interpreter

Synthesis
  IR code generation
    IR
    optimization
    code generation
    target language

Topics

I. The Basics
  - Scanning and parsing
  - Dataflow analysis (review the projects, especially project 2)
    - Theoretic framework built on lattices
    - How might we improve the performance of iterative data-flow analysis by using a worklist-based algorithm?
  - Control flow analysis: control-flow graphs, dominators, dominance frontiers, irreducibility

II. Analyses and Representations
  - SSA Form: types of data dependencies, how to translate to minimal SSA
  - Program optimizations
    - dead-code elimination, constant propagation (simple constants), CSE, loop-invariant code motion, PRE, copy propagation, induction variable elimination, strength reduction, global value numbering
  - 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?
Topics (cont)

III. Low-Level Optimizations
   - Register allocation
     - difference between Briggs and Chaitin
     - heuristics to determine spilling will be provided
   - Instruction scheduling
     - list scheduling
   - Profile-guided and dynamic optimizations
     - what types of profiling information can we collect and how is it useful?
     - when are dynamic optimizations (or run-time reordering transformations) profitable?
   - Predication and speculation
     - what HW features do these techniques attempt to deal with?

Topics (cont)

IV. High-Level Optimizations
   - Dependence analysis
     - is this problem decidable? can we formulate it as decidable?
   - Loop transformations
     - unimodular transformation framework
     - generating a schedule based on schedule constraints
   - Tiling
     - what is it used for?
   - Object-oriented optimizations
     - how does the compiler handle dynamic binding?
Topics (cont)

V. Emerging Topics
   – Garbage collection
     – pros and cons between explicit and implicit garbage collection
     – reference counting, mark and sweep, generational collection, etc.
   – Run-time reordering transformations
     – data reordering improves ?? locality
     – iteration reordering improves ?? locality
   – Security and program checking
     – four different overall strategies

Data-flow Equations for Reaching Definitions

Symmetry between reaching definitions and liveness
   – Swap in[] and out[] and swap the directions of the arcs

<table>
<thead>
<tr>
<th>Reaching Definitions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>(in[n] = \bigcup_{p \in \text{pred}[n]} out[s])</td>
<td>(out[n] = \bigcup_{s \in \text{succ}[n]} in[s])</td>
</tr>
<tr>
<td>(out[n] = \text{gen}[n] \bigcup (in[n] \setminus \text{kill}[n]))</td>
<td>(in[n] = \text{gen}[n] \bigcup (out[n] \setminus \text{kill}[n]))</td>
</tr>
</tbody>
</table>

Def of \(x\)

\(x = \) entry

\(n \) entry

Is \(x\) def’d along this path?

Use of \(x\)

\(n \) entry

Is \(x\) def’d along this path?

\(\text{entry} \)
Reality Check!

Some definitions and uses are ambiguous
- We can’t tell whether or what variable is involved
  \textit{e.g.,} \texttt{*p = x; */ what variable are we assigning?! */}
- Unambiguous assignments are called \textbf{strong updates}
- Ambiguous assignments are called \textbf{weak updates}

Solutions
- Be conservative
  - Sometimes we assume that it could be everything
    \textit{e.g.,} Defining \texttt{*p} (generating reaching definitions)
  - Sometimes we assume that it is nothing
    \textit{e.g.,} Defining \texttt{*p} (killing reaching definitions)
- Try to figure it out: alias/pointer analysis (more later)

Using Alias Information

Example: reaching definitions
- Compute at each point in the program a set of \((s, v)\) pairs, indicating that statement \(s\) may define variable \(v\)

Flow functions
- \(s: \texttt{*p = x;}\)
  \[
  \text{out}_{\text{reach}}[s] = \{(s, z) \mid (p \rightarrow z) \in \text{in}_{\text{may-pt}}[s]\} \cup \\
  \{t \mid (p \rightarrow y) \in \text{in}_{\text{must-pt}}[s]\}
  \]
- \(s: \texttt{x = *p;}\)
  \[
  \text{out}_{\text{reach}}[s] = \{(s, x)\} \cup \{t \mid (p \rightarrow y) \in \text{in}_{\text{must-pt}}[s]\}
  \]
- \ldots
Interprocedural Analysis: Two Types of Information

**Track information that flows into a procedure**
- Sometimes known as **propagation problems**
  e.g., What formals are constant?
  e.g., Which formals are aliased to globals?

**Track information that flows out of a procedure**
- Sometimes known as **side effect problems**
  e.g., Which globals are def’d/used by a procedure?
  e.g., Which locals are def’d/used by a procedure?
  e.g., Which actual parameters are def’d by a procedure?

---

Examples

**Propagation Summaries**
- MAY-ALIAS: The set of formals that may be aliased to globals and each other
- MUST-ALIAS: The set of formals that are definitely aliased to globals and each other
- CONSTANT: The set of formals that must be constant

**Side-effect Summaries**
- MOD: The set of variables possibly modified (defined) by a call to a procedure
- REF: The set of variables possibly read (used) by a call to a procedure
- KILL: The set of variables that are definitely killed by a procedure (e.g., in the liveness sense)
Computing Interprocedural Summaries

Top-down
– Summarize information about the caller (MAY-ALIAS, MUST-ALIAS)
– Use this information inside the procedure body
  int a;
  void foo(int &b, &c){
  ... 
  }
  foo(a,a);

Bottom-up
– Summarize the effects of a call (MOD, REF, KILL)
– Use this information around procedure calls
  x = 7;
  foo(x);
  y = x + 3;

Side-Effect Analysis

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;
}
Register Allocation: Spilling

If we can’t find a k-coloring of the interference graph
– Spill variables (nodes) until the graph is colorable

Choosing variables to spill
– Choose least frequently accessed variables
– Break ties by choosing nodes with the most conflicts in the interference graph
– Yes, these are heuristics!

Example

Sample code
\[
\begin{array}{c}
\text{do } i = 1, 6 \\
\text{do } j = 1, 5 \\
\text{ \hspace{1cm} } A(2i, j) = A(i, j-1) \\
\text{ \hspace{1cm} } \text{enddo} \\
\text{ \hspace{1cm} } \text{enddo}
\end{array}
\]

Dependence
– \(2i - i_2 = 0, j_1 = j_2 - 1\), 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
– \{ \quad [i, j] \mapsto [2i, j + 1] \quad | \quad 1 \leq i \leq 3 \quad \&\& \quad 1 \leq j \leq 4 \quad \}
Loop Transformations

Original code
\[
\begin{align*}
\text{do } & i = 1, 6 \\
\text{do } & j = 1, 5 \\
& A(i,j) = A(i-1,j+1)+1 \\
& \text{enddo} \\
& \text{enddo}
\end{align*}
\]

Distance vector: (1, -1)
Which loop can we parallelize?

Scheduling a SARE

for \(i = 1..n, j = 1..n\), \(A(i,j) = A(i-1,j+1)+1\)

Linear schedule is of the form
\[t(i,j) = a*i + b*j + c\]

Finding constraints on the schedule is similar to checking transformation legality in omega
\[t(i,j) \neq t(i+1,j-1)\]
Loop Skewing

Original code
\[
\begin{align*}
    & \text{do } i = 1, 6 \\
    & \quad \text{do } j = 1, 5 \\
    & \quad \quad A(i,j) = A(i-1,j+1)+1 \\
    & \quad \text{enddo} \\
    & \text{enddo}
\end{align*}
\]

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 and Array Accesses

Original code
\[
\begin{align*}
    & \text{do } i = 1, 6 \\
    & \quad \text{do } j = 1, 5 \\
    & \quad \quad A(i,j) = A(i-1,j+1)+1 \\
    & \quad \text{enddo} \\
    & \text{enddo}
\end{align*}
\]

Dependence vector:
\[
\begin{bmatrix}
    1 & 0 \\
    1 & 1
\end{bmatrix}
\begin{bmatrix}
    1 \\
    1
\end{bmatrix} =
\begin{bmatrix}
    1 \\
    0
\end{bmatrix}
\]

New Array Accesses:
\[
\begin{align*}
    & A\left(\begin{bmatrix}
        1 & 0 \\
        0 & 0
    \end{bmatrix} i + \begin{bmatrix}
        0 & 0 \\
        1 & 1
    \end{bmatrix} j + \begin{bmatrix}
        0 & 0
    \end{bmatrix}\right) = A(i,j) \\
    & A\left(\begin{bmatrix}
        1 & 0 \\
        0 & 0
    \end{bmatrix} i - \begin{bmatrix}
        1 & 0 \\
        1 & 1
    \end{bmatrix} j + \begin{bmatrix}
        0 & 0
    \end{bmatrix}\right) = A(i',j'+i') \\
    & A\left(\begin{bmatrix}
        1 & 0 \\
        0 & 0
    \end{bmatrix} i + \begin{bmatrix}
        -1 & 0 \\
        0 & 1
    \end{bmatrix} j + \begin{bmatrix}
        0 & 1
    \end{bmatrix}\right) = A(i-1,j+1) \\
    & A\left(\begin{bmatrix}
        1 & 0 \\
        0 & 0
    \end{bmatrix} i - \begin{bmatrix}
        1 & 0 \\
        1 & 1
    \end{bmatrix} j + \begin{bmatrix}
        0 & 1
    \end{bmatrix}\right) = A(i-1,j'-i'+1)
\end{align*}
\]
Transforming the Loop Bounds

Original code

\[
\begin{align*}
&\text{do } i = 1,6 \\
&\quad \text{do } j = 1,5 \\
&\quad \quad A(i,j) = A(i-1,j+1) + 1 \\
&\quad \text{enddo} \\
&\text{enddo}
\end{align*}
\]

Bounds:

\[
\begin{bmatrix}
-1 & 0 \\
1 & 0 \\
0 & -1 \\
0 & 1
\end{bmatrix} \leq \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix}
6 & -1 \\
-1 & 5 \\
-6 & -1 \\
-5 & 5
\end{bmatrix}
\]

Transformed code

\[
\begin{align*}
&\text{do } i' = 1,6 \\
&\quad \text{do } j' = 1,5+i',5+i' \\
&\quad \quad A(i',j'-i') = A(i'-1,j'-1'+1) + 1 \\
&\quad \text{enddo} \\
&\text{enddo}
\end{align*}
\]