Lecture 1

Introduction

I. Why Study Compilers?
II. Course Syllabus

Chapters 1.1–1.5, 8.4, 8.5, 9.1
I. Why Study Compilers?
Reasons for Studying Compilers

• **Compilers are important**
  – An essential programming tool
    • Improves software productivity by hiding low-level details
    • Domain-specific languages for encapsulating domain expertise
  – A tool for designing and evaluating computer architectures
    • Inspired RISC, VLIW machines
    • Machines’ performance measured on compiled code
  – Techniques for developing other programming tools
    • Example: error detection tools
  – Program translation can be used to solve other problems
    • Example: Binary translation
      (processor change, adding virtualization support)
  – Implementation of domain-specific languages
    • CAD, database, graphics, networking, bio-computation, IoT, …

• **Compilers have impact: affect all programs**
Compiler Study Trains Good Developers

Excellent software engineering case study

- Optimizing compilers are hard to build
  - Input: all programs
  - Objectives:

- Methodology for solving complex real-life problems
  - Key to success: Formulate the right approximation!
    - Desired solutions are often NP-complete / undecidable
  - Where theory meets practice
    - Can't be solved by just pure hacking
      - theory aids generality and correctness
    - Can't be solved by just theory
      - experimentation validates and provides feedback to problem formulation

- Reasoning about programs, reliability & security makes you a better programmer

There are programmers, and there are tool builders ...
Example

• Tools for web application security vulnerabilities
• PQL: a general language for describing information flow of interest
• Static techniques to locate errors automatically
• Illustrates:
  – Exciting research area!
  – Importance of programming tools
  – Sophistication of static analysis techniques
  – What static analysis looks like
  – Use of domain-specific languages
  – Combination of theory and hacking
Use of Mathematical Abstraction

- Design of mathematical model & algorithm
  - Generality, power, simplicity and efficiency
Course Syllabus

1. Basic compiler optimizations

<table>
<thead>
<tr>
<th>Goal</th>
<th>Eliminates redundancy in high-level language programs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Allocates registers</td>
</tr>
<tr>
<td></td>
<td>Schedules instructions (for instruction-level parallelism)</td>
</tr>
<tr>
<td>Scope</td>
<td>Simple scalar variables, intraprocedural, flow-sensitive</td>
</tr>
<tr>
<td>Theory</td>
<td>Data-flow analysis (graphs &amp; solving fix-point equations)</td>
</tr>
</tbody>
</table>

2. Pointer alias analysis

<table>
<thead>
<tr>
<th>Goal</th>
<th>Used in program understanding, concrete type inference in OO programs (resolve target of method invocation, inline, and optimize)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Pointers, interprocedural, flow-insensitive</td>
</tr>
<tr>
<td>Theory</td>
<td>Relations, Binary decision diagrams (BDD)</td>
</tr>
</tbody>
</table>

3. Parallelization and memory hierarchy optimization

<table>
<thead>
<tr>
<th>Goal</th>
<th>Parallelizes sequential programs (for multiprocessors)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Optimizes for the memory hierarchy</td>
</tr>
<tr>
<td>Scope</td>
<td>Arrays, loops</td>
</tr>
<tr>
<td>Theory</td>
<td>Linear algebra</td>
</tr>
</tbody>
</table>

4. Domain-specific languages: graphs, image processing, internet of things

5. Garbage collection (run-time system)
<table>
<thead>
<tr>
<th></th>
<th>Course introduction</th>
<th>Data-flow analysis: introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Basic compilers</td>
<td>Data-flow analysis: theoretic foundation</td>
</tr>
<tr>
<td>3</td>
<td>Data-flow analysis: theoretic foundation</td>
<td>Optimization: constant propagation</td>
</tr>
<tr>
<td>4</td>
<td>Optimization: constant propagation</td>
<td>(joeq)</td>
</tr>
<tr>
<td>5</td>
<td>(joeq)</td>
<td>Register allocation</td>
</tr>
<tr>
<td>6</td>
<td>Optimization: redundancy elimination</td>
<td>Scheduling: non-numerical code</td>
</tr>
<tr>
<td>7</td>
<td>Scheduling: non-numerical code</td>
<td>Scheduling: software pipelining</td>
</tr>
<tr>
<td>8</td>
<td>Scheduling: software pipelining</td>
<td>Dynamic compilation</td>
</tr>
<tr>
<td>11</td>
<td>Pointer alias analysis</td>
<td>Formulation</td>
</tr>
<tr>
<td>12</td>
<td>BDDs in pointer analysis</td>
<td>BDDs in pointer analysis</td>
</tr>
<tr>
<td>13</td>
<td>Domain specific languages</td>
<td>A graph analysis language</td>
</tr>
<tr>
<td>14</td>
<td>Parallelism/Locality</td>
<td>Introduction</td>
</tr>
<tr>
<td>15</td>
<td>Parallelism/Locality</td>
<td>Affine partitioning</td>
</tr>
<tr>
<td>16</td>
<td>Domain specific languages</td>
<td>An image processing language</td>
</tr>
<tr>
<td>17</td>
<td>A language for Internet of Things</td>
<td>A language for Internet of Things</td>
</tr>
<tr>
<td>18</td>
<td>Garbage Collection</td>
<td>Basic concepts</td>
</tr>
<tr>
<td>19</td>
<td>Garbage Collection</td>
<td>Optimizations</td>
</tr>
</tbody>
</table>
Course Emphasis

- **Methodology:** apply the methodology to other real life problems
  - Problem statement
    - Which problem to solve?
  - Theory and Algorithm
    - Theoretical frameworks
    - Algorithms
  - Experimentation: Hands-on experience

- **Compiler knowledge:**
  - Non-goal: how to build a complete optimizing compiler
  - Important algorithms
  - Exposure to new ideas
  - Background to learn existing techniques
Assignment by next class (no need to hand in)

• Think about how to build a compiler that converts the code on page 11 to page 12
  – (Read Chapter 9.1 for introduction of the optimizations)

• Example:
  Bubblesort program that sorts array A allocated in static storage

```c
for (i = n-2; i >= 0; i--) {
    for (j = 0; j <= i; j++) {
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
    }
}
```
Code Generated by the Front End

i := n-2
S5: if i<0 goto s1
j := 0
s4: if j>i goto s2
t1 = 4*j
t2 = &A
t3 = t2+t1
  ;A[j]
t4 = *t3
  ;A[j]
t5 = j+1
t6 = 4*t5
t7 = &A
t8 = t7+t6
  ;A[j+1]
t9 = *t8
  ;A[j+1]
if t4 <= t9 goto s3
          *t24 = temp  ;A[j+1]=temp
  ;A[j]
t10 = 4*j
t11 = &A
t12 = t11+t10
  ;temp=A[j]
t13 = j+1
t14 = 4*t13
t15 = &A
t16 = t15+t14
t17 = *t16  ;A[j+1]
t18 = 4*j
t19 = &A
  ;&A[j]
t20 = t19+t18
  ;&A[j]
t21 = j+1
t22 = 4*t21
t23 = &A
t24 = t23+t22
                *t20 = t17
  ;A[j]=A[j+1]
S3: j = j+1
S2: i = i-1
S1:
  (t4=*t3 means read memory at address in t3 and write to t4:
  *t20=t17: store value of t17 into memory at address in t20)

s1:
After Optimization

Result of applying:
- global common subexpression
- loop invariant code motion
- induction variable elimination
- dead-code elimination

to all the scalar and temp. variables

These traditional optimizations can make a big difference!

\[
\begin{align*}
i &= n-2 \\
t27 &= 4*i \\
t28 &= &A \\
t29 &= t27+t28 \\
t30 &= t28+4 \\
S5: &\text{if } t29 < t28 \text{ goto } s1 \\
t25 &= t28 \\
t26 &= t30 \\
s4: &\text{if } t25 > t29 \text{ goto } s2 \\
t4 &= *t25 \quad ;A[j] \\
t9 &= *t26 \quad ;A[j+1] \\
&\text{if } t4 <= t9 \text{ goto } s3 \\
temp &= *t25 \quad ;\text{temp}=A[j] \\
t17 &= *t26 \quad ;A[j+1] \\
*\text{t25} &= t17 \quad ;A[j]=A[j+1] \\
*\text{t26} &= \text{temp} \quad ;A[j+1]=\text{temp} \\
s3: &t25 = t25+4 \\
t26 &= t26+4 \\
goto S4 \\
s2: &t29 = t29-4 \\
goto s5 \\
s1: 
\end{align*}
\]