Lecture 1

Advanced Compilers Course Introduction

I. Why Study Compilers?
II. Mathematical Abstractions: with Examples
III. Course Syllabus

Chapters 1.1-1.5, 8.4, 8.5, 9.1

Why Study Compilers?

Impact!

Techniques in compilers help all programmers
Why Study Compilers?

Bridge the semantic gap between programmers and machines

Concepts in programming languages
- High-level programming languages
- Domain-specific languages
- Natural language

Concepts in computer architecture
- RISC vs CISC
- Locality: Caches, memory hierarchy
- Parallelism:
  - Instruction-level parallelism
  - Multi-processors

Programmers

Programming Language

Compilers

Machine

Programming Tools
- Security audits
- Binary translations

Compiler Study Trains Good Developers

- Reasoning about programs makes better programmers
- Tool building: there are programmers and there are tool builders ...
- Excellent software engineering case study: Compilers are hard to build
  - Input: all programs
  - Objectives:
  - Methodology for solving complex real-life problems
    - Build upon mathematical / programming abstractions
Compilers: Where theory meets practice

- Desired solutions are often NP-complete / undecidable
- Key to success: Formulate the right abstraction / approximation
  - Can’t be solved by just pure hacking
    - Theory aids generality and correctness
  - Can’t be solved by just theory
    - Experimentation validates & provides feedback to problem formulation
- Tradeoffs: Generality, power, simplicity, and efficiency

Why Study Compilers?

Impact!
Techniques in compilers help all programmers

Better Programmer
Reasoning about programs
Mathematical Abstractions
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

Topics in this Course

1. Concepts in programming languages
   - High-level programming languages
   - Domain-specific languages
   - Natural language

2. Concepts in computer architecture
   - RISC vs CISC
   - Locality: Caches, memory hierarchy
   - Parallelism:
     - Instruction-level parallelism
     - Multi-processors

3. Programming Tools
   - Security audits
   - Binary translations

4. Humans
The Rest of this Lecture

• Goal
  – Explain why I chose the topics
  – Emphasize the abstraction methodology
• For each topic:
  – Motivate importance
  – Show an example to illustrate the complexity
  – What’s the abstraction?
  – Impact

1. Optimizing Compilers for High-Level Programming Languages

• 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**

- \( i := n - 2 \)
- \( t13 = j + 1 \)
- \( S5: \text{if } i<n \text{ goto } s1 \)
- \( j := 0 \)
- \( t14 = 4 \times t13 \)
- \( t15 = 6A \)
- \( s4: \text{if } j>i \text{ goto } s2 \)
- \( t16 = t15 + t14 \)
- \( t17 = 4 \times t16 ; A[j+1] \)
- \( t18 = 4 \times j \)
- \( t19 = 6A \)
- \( t2 = 6A \)
- \( t20 = t19 + t18 ; 4A[j] \)
- \( t21 = j + 1 \)
- \( t22 = 4 \times t21 \)
- \( t23 = 6A \)
- \( t3 = t2 + t1 \)
- \( t24 = t23 + t22 \)
- \( t4 = t3 ; A[j] \)
- \( t5 = j + 1 \)
- \( t6 = 4 \times t5 \)
- \( t7 = 6A \)
- \( j := n \)
- \( \text{store value of } t17 \text{ into memory at address in } t20 \)
- \( t8 = t7 + t6 \)
- \( t9 = t8 ; A[j+1] \)
- \( t10 = 4 \times j \)
- \( t11 = 6A \)
- \( \text{goto } s4 \)
- \( t12 = t11 + t10 \)
- \( s2: i = i - 1 \)
- \( \text{goto } s5 \)
- \( s3: j = j + 1 \)
- \( \text{goto } s4 \)

(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)

---

**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!

- \( i := n - 2 \)
- \( t27 = 4 \times i \)
- \( t28 = 6A \)
- \( t29 = t27 + t28 \)
- \( t30 = t28 + 4 \)
- \( S5: \text{if } t29 < t28 \text{ goto } s1 \)
- \( t25 = t28 \)
- \( t26 = t30 \)
- \( s4: \text{if } t25 > t29 \text{ goto } s2 \)
- \( t4 = t25 ; A[j+1] \)
- \( t9 = t26 ; A[j] \)
- \( \text{if } t4 = t9 \text{ goto } s3 \)
- \( \text{temp} = t25 ; \text{temp} = A[j] \)
- \( t17 = t26 ; A[j+1] \)
- \( \text{*t25 = t17} ; A[j+1] = \text{temp} \)
- \( \text{*t26 = temp} ; A[j] = \text{temp} \)
- \( s3: t25 = t25 + 4 \)
- \( t26 = t26 + 4 \)
- \( \text{goto } s4 \)
- \( s2: t29 = t29 - 4 \)
- \( \text{goto } s5 \)
- \( s1: \)
Redundancy Elimination: Data flow

- Impact: Programmers write in high-level programming languages without worrying about efficiency

2. High-Performance Computing (Machine Learning)

1 ExaFLOPS = 10^{18} FLOPS
Nvidia Volta GV100 GPU

218 transistors
815 mm²
80 Stream Multiprocessors (SM)
1455 Mhz

In Each SM

64 FP32 cores
64 int cores
32 FP64 cores
8 Tensor cores

Tensor Cores
D = A x B + C; A, B, C, D are 4x4 matrices
4 x 4 x 4 matrix processing array
1024 floating point ops / clock

FP32: 15 TFLOPS
FP64: 7.5 TFLOPS
Tensor: 120 TFLOPS
Parallelism and Locality

- Can programmers focus on high-level programming & get performance?
- **Example:** matrix multiply: core kernel in neural networks

```c
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
            m3(i, j) += m1(i, k) * m2(k, j);
        }
    }
}
```

- Lots of parallelism in the program: \( N^2 \)
- Poor sequential / parallel performance without locality optimization

---

Optimizing for Single Core: Permuting Loops

```c
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
            m3(i, j) += m1(i, k) * m2(k, j);
        }
    }
}
```

Permute loop to make data access contiguous for vectorization:

```c
for (k = 0; k < N; k++) {
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            m3(i, j) += m1(i, k) * m2(k, j);
        }
    }
}
```
**Tiling: to Increase Reuse**

for \( k = 0; k < N; k++ \) {
    for \( i = 0; i < N; i++ \) {
        for \( j = 0; j < N; j++ \) {
            \( m3(i, j) += m1(i, k) \ast m2(k, j); \)
        }
    }
}

Assume cache size \( < N^2 \)

Tile the outermost loop

for \( k1 = 0; k1 < N; k1 += B \) {
    for \( i = 0; i < N; i++ \) {
        for \( k2 = k1; k2 < k1 + B; k2++ \) {
            for \( j = 0; j < N; j++ \) {
                \( m3(i, j) += m1(i, k2) \ast m2(k2, j); \)
            }
        }
    }
}

Assume \( N \) is divisible by \( B \)

---

**Experiment**

- Square float32 matrix of various sizes
- Initialized with random \((0, 1)\) normal
- Average of 10 iterations
- Intel i7-4770HQ CPU @ 2.20GHz (Haswell), no turbo
  - Number of cores: 4
  - Number of threads: 8
  - SSE4.2 and AVX2: 256 bit SIMD instructions
- 32k L1 cache, 256k L2, 6M L3, 132M L4 cache (LLC, GPU shared)
- Compiled with g++ 7.2.1 20170915, as provided in Fedora 27
- Common options: --std=c++14 -Wall -g
- (The production version of clang does not support loop optimizations)
Sequential Performance

Parallel scaling (matrix size 1500)
Parallelism and Locality Optimization

- Impact:
  Programmers write in high-level programming languages without knowledge of how to use parallelism in hardware

3. Security of Web Applications

Hacker → Browser → Web App → Database

Evil Input
Confidential information leak
**SQL Injection Errors**

- Hacker
- Browser
- Web App
- Database

**Give me Bob's credit card #**
**Delete all records**

---

**Happy-go-lucky SQL Query**

User supplies: *name, password*

```
"SELECT UserID, Creditcard FROM CCRec WHERE Name = "
  name " AND PW = "
password "
```
Fun with SQL

“—”: “the rest are comments” in Oracle SQL

SELECT UserID, CreditCard FROM CCRec
WHERE:
  Name = bob AND PW = foo
  Name = bob— AND PW = x
  Name = bob or 1=1— AND PW = x
  Name = bob; DROP CCRec— AND PW = x

Dynamic vs. Static Pattern

Dynamically:
  o = req.getParameter ( );
  stmt.executeQuery (o);

Statically:
  p_1 = req.getParameter ( );
  stmt.executeQuery (p_2);

p_1 and p_2 point to same object?
  Pointer alias analysis
In Practice

ParameterParser.java:586
String session.ParameterParser.getRawParameter(String name)

```java
public String getRawParameter(String name)
    throws ParameterNotFoundException {
    String[] values = request.getParameterValues(name);
    if (values == null) {
        throw new ParameterNotFoundException(name + " not found");
    } else if (values[0].length() == 0) {
        throw new ParameterNotFoundException(name + " was empty");
    }
    return (values[0]);
}
```

ParameterParser.java:570
String session.ParameterParser.getRawParameter(String name, String def)

```java
public String getRawParameter(String name, String def) {
    try {
        return getRawParameter(name);
    } catch (Exception e) {
        return def;
    }
}
```

In Practice (II)

ChallengeScreen.java:194
Element lessons.ChallengeScreen.doStage2(WebSession s)

```java
String user = s.getParser().getRawParameter( USER, "");
StringBuffer tmp = new StringBuffer();
tmp.append("SELECT cc_type, cc_number from user_data WHERE userid = ");
tmp.append(user);
tmp.append(" ");
query = tmp.toString();
Vector v = new Vector();
try {
    ResultSet results = statement3.executeQuery(query);
    ...
```
Why is Pointer Alias Analysis Hard?

• Unbounded number of dynamically allocated objects

• An indirect write via an unknown pointer can write to all possible locations of the same type.

• Must analyze across procedures

• Must keep track of the calling contexts (exponential)

Vulnerabilities Found in 9 Programs

<table>
<thead>
<tr>
<th></th>
<th>SQL injection</th>
<th>HTTP splitting</th>
<th>Cross-site scripting</th>
<th>Path traversal</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Parameter</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>Cookie</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Non-Web</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9</strong></td>
<td><strong>11</strong></td>
<td><strong>5</strong></td>
<td><strong>5</strong></td>
<td><strong>30</strong></td>
</tr>
</tbody>
</table>
**Automatic Analysis Generation**

Programmer: Security analysis in 10 lines

Compiler Writer: Flow-insensitive Context-sensitive Ptr analysis in 10 lines

1000s of lines 1 year tuning

- Datalog: Logic database programming language
- Binary Decision Diagrams (BDD): Exponential state operations
- PQL: Domain specific language

- PQL
- Datalog
- Binary Decision Diagrams (BDD)

**Security Audits**

- Programs
  - Static statements
  - Dynamic execution
  - Generated code

- Abstraction
  - PQL Logic database (Datalog) Binary decision diagrams (BDDs)
  - Solutions

**Impact:**

- Programmers can ensure code is secure without tedious and error-prone manual inspection
- Illustrates use of multiple levels of abstraction
4. Programming in Natural Language

• Today’s software
  – All possible combinations are hardcoded
  – Users choose from a menu of choices
  – Limited choices to keep the interface manageable

• Can consumers code in the highest programming language?
  – Natural language!

• What kind of programs?
  – Not C, Java, Python
  – Many useful APIs: virtual assistants
  – Our target: connect virtual assistant primitives

Asthma Patient

- *Asthma Patient*

- **people**
  - "If Bob’s peak flow-meter drops below 180L/min notify me"

- **location**
  - "Let my Dad know if I am at the hospital"

- **environment**
  - "when the ragweed pollen count is high and Bob is running, warn him"

- **devices**
  - "log where I am when I use my inhaler"
Natural Language Programming

“When I use my inhaler, get my GPS location, if it is not home, write it to logfile in Box.”

- Event-driven program
- Multiple function calls
- Parameter passing
- Filters on values

Almond: 1st Programmable Virtual Assistant

“When I use my inhaler, get my GPS location, if it is not home, write it to logfile in Box.”

monitor @Inhaler-use(),
=> @GPS(), location <> "home"
=> @Box-write(file="logfile", data=location)

Giovanni, Rameihi, Xu, Fischer, Lam, WWW 2017
Thingpedia: Encyclopedia of Things

- **Interoperability**
  - API signatures + corresponding NL
- **Open repository**
  - Available to all assistants

> 60 devices / 200 functions

<table>
<thead>
<tr>
<th>Natural Language</th>
<th>API Signatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHEN @Stanford tweets</td>
<td>Monitor (@home_timeline(), ...) author==&quot;Stanford&quot;)</td>
</tr>
<tr>
<td>GET tweets matching “#Cardinal”</td>
<td>search(...), contains (hashtag, ...)</td>
</tr>
<tr>
<td>DO tweet “Stanford won!”</td>
<td>post (status)</td>
</tr>
</tbody>
</table>

Real Natural Language Input

When I tweet, share the text on LinkedIn

Share my tweets on my LinkedIn
Whenever I tweet, post the same message on LinkedIn
Post all my tweets on LinkedIn
LUInet Results

- Dataset (60+ devices, 200+ functions)
  - Synthetic: 515K programs, 2.9M sentences
  - Paraphrased: 175K programs, 400K sentences
- Model:
  - Seq2seq bi-LSTM with attention, pointer network
- Accuracy: 89%
- Future work: real user input

Natural Language -> ThingTalk

- Impact:
  Consumers can automate personal and professional tasks without relying on professional programmers
Mathematic Abstraction: Where theory meets practice

- Compiler optimizations
- Parallelism and locality optimizations
- Pointer alias analysis
- Natural language programming
- Garbage collection

Tentative Course Schedule

<table>
<thead>
<tr>
<th></th>
<th>Course introduction</th>
<th>Data-flow analysis: introduction</th>
<th>Data-flow analysis: theoretic foundation</th>
<th>Optimization: constant propagation</th>
<th>Optimization: redundancy elimination</th>
<th>Register allocation</th>
<th>Non-numerical code scheduling</th>
<th>Software pipelining</th>
<th>Parallelization</th>
<th>Loop transformations</th>
<th>Pipelined Parallelism</th>
<th>Algorithm</th>
<th>BDDs in pointer analysis</th>
<th>Introduction</th>
<th>Neural network application</th>
<th>Algorithms</th>
</tr>
</thead>
</table>
Homework

- Due Wednesday
- Read Chapter 9.1 for introduction of the optimizations