High-Level Compiler Structure

- **Lexer**
  - Input: string of chars
  - Output: string of tokens

- **Parser**
  - Input: string of tokens
  - Output: parse tree

- **Semantic Analyzer**
  - Input: parse tree
  - Output: Intermediate Representation

- **Optimizer**
  - Input: Intermediate Representation
  - Output: Optimized IR

- **Code Generator**
  - Input: Optimized IR
  - Output: Object code

- **Linker**
  - Input: Object code
  - Output: Executable

Optimizing Compilers

- **Analyzes & Transforms**
- **Guided by Cost Models**
- **Executes**

Theory

Practice
Lexing and Parsing

Lexer          Forms tokens from character stream

Parser        Syntax-directed translation to parse tree

Production    Rule

E → E + T     E.ptr := mknode('+', E.ptr, T.ptr)
E → E - T     E.ptr := mknode('-', E.ptr, T.ptr)
E → T         E.ptr := T.ptr
T → id        T.ptr := mkleaf(id, id.entry)
T → num       T.ptr := mkleaf(num, num.val)

LR Parsing

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Stream</th>
<th>Stack</th>
<th>Rule Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E + T</td>
<td>1 7-x+5-x</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>E → E + T</td>
<td>2 -x+5-x</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>E → T</td>
<td>3 x+5-x</td>
<td>E</td>
<td>3</td>
</tr>
<tr>
<td>T → id</td>
<td>4 +5-x</td>
<td>E-x</td>
<td>4</td>
</tr>
<tr>
<td>T → num</td>
<td>5 5-x</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>-x</td>
<td>E+5</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>-x</td>
<td>E+T</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>-x</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>x</td>
<td>E-x</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>E-T</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Why is Optimization needed?

\[
X := Y + 3 \\
\text{temp1 := inttoreal(3)} \\
\text{load(temp2, Y)} \\
\text{temp3 := temp2 + temp1} \\
\text{store(X, temp3)}
\]

\[
Z := X + 5 \\
\text{temp4 := inttoreal(5)} \\
\text{load(temp5, X)} \\
\text{temp6 := temp5 + temp4} \\
\text{store(Z, temp6)}
\]

X already in temp3, so needn’t reload!

Not because of dumb code!

Different operations (loads, addressing) exposed at different levels

Why Optimization not solved

Machines evolve

New Features
Ex. EPIC, Multithreading, ...

New ways of using machines
Ex. Grids, Peer-to-peer,...

Changing Costs
Ex. L1 cache misses less costly

Languages evolve

New
Ex. Java

Higher level, special purpose
Ex. Matlab

Applications evolve

Ex. adaptive, web-based,...
Why Optimization not solved

Finding "optimal" undecidable

Most interesting properties of languages are
Undecidable

Ex. Determining whether variable is used
assigned a value

Ex. Determining if program terminates

Ex. Determining if 2 programs have same
Input/Output behavior

Halting Problem for T.M.

No general algorithm
All optimization techniques are approximate
Good news: better optimization always possible

When is Optimization needed?

High-level

Close to source

Loop structure evident, e.g., CFG
Ex. Loop Fusion, Distribution

Medium-level

Language- and machine independent
Registers, temporaries exposed
Ex. Constant propagation, value numbering

Low-level

Close to machine

May be actual machine instructions
Ex. Register allocation, Scheduling

Link-time

Library routines available
Ex. Interprocedural Register allocation

Run-time

Runtime constants available
Ex. Selection of code variants using run-time info
What about optimization order?

<table>
<thead>
<tr>
<th>Theory</th>
<th>Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformations interact</td>
<td>Pick &quot;right&quot; order &amp; hope for the best</td>
</tr>
<tr>
<td></td>
<td>Architectural complexity of machines</td>
</tr>
<tr>
<td>Take interactions into account</td>
<td></td>
</tr>
</tbody>
</table>

Theory: Transformations Interact

Constant Propagation enables Dead Code Elimination

Ex. \( A = 4 \)

If \( A < 17 \) then ...

Can eliminate test!
Theory: Transformations Interact

Loop Distribution

Do i = 1, 100
a(i) = b(i) + c(i)
d(i) = e(i) * 2
Enddo

Loop Fusion

Do i = 1, 100
a(i) = b(i) + c(i)
Enddo

d(i) = e(i) * 2
Enddo

Instruction scheduling interferes with
Register allocation

Ex.  A =
     = A
B =
     = B

Instruction level parallelism
Reuse of registers

Integration: [Norris, Pollock], [Bradlee, Eggers, Henry]
Practice: Transformation Order
Pick order $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k$

Ex.
Enabling:
Constant Propagation
Dead Code Elimination

Theory: Transformation Order
Partial Order: [Whitfield, Soffa]
Constant Propagation
Dead Code Elimination
Code Motion
Loop
Loop Fusion
Interchange

General case: Guidance needed!
**Guidance in Practice**

Separate architectural features

optimized separately

using independent cost functions

**Ex.**

- Parallelism
- Locality
  - Locality
  - Parallelism

SUIF [Lam et al] [McKinley et al]

---

**Guidance in Practice**

Single compiler for multiple architectures

needs architecture-based guidance

**Ex.**

- Coarse-grain Parallelism:
  - Use Loop Fusion

- Vector processor:
  - Use Loop Distribution
Analysis = Basis of Optimization

To transform program, must determine

Legality

*Is X already in register temp3?*
If not, then cannot remove load of X, or
could get wrong value

Profitability for Guidance

Some optimizations always profitable
Others may depend on program or machine characteristics

Analysis of program can help with both

Data Flow Analysis

Collects (small) facts about program
at compile-time

Ex. \( X := \ldots \)
\[
\begin{align*}
\text{No assignment to } X \\
P: \ldots := X
\end{align*}
\]
Initial assignment of \( X \) is the one one that reaches \( P \)

At runtime, may or may not hold

Ex. \( X := \ldots \)
\[
\begin{align*}
X := \ldots \\
P: \ldots := X
\end{align*}
\]
Kinds of Data Flow Information

What assignments to a variable reach a point P?

*Can help determine if variable is a constant*

What variables are modified or used as a result of calling a function or procedure?

*Otherwise must assume all may be modified!*

Is an occurrence of a variable the last one in the procedure (no longer live)?

*If so, can reuse register!*

Compiler Challenges

Interaction with other tools, runtime

Higher level, special purpose languages

Optimization tradeoffs, interactions

New platforms

New applications of techniques