Compiler Optimisation

1 – Introductory Lecture

Hugh Leather
IF 1.18a
hleather@inf.ed.ac.uk

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

2018
Textbooks

- **Engineering a Compiler** “EaC” by K. D. Cooper and L. Torczon. Published by Morgan Kaufmann 2003

- **Optimizing Compilers for Modern Architectures: A Dependence-based Approach** “CMA” by R. Allen and K. Kennedy. Published Morgan Kaufmann 2001

- **Advanced Compiler Design and Implementation** by Steven S. Muchnick, published by Morgan Kaufmann. (extra reading - not required)

- Plus research papers in last part of course

*Note: Slides do not replace books. Provide motivation, concepts and examples not details.*
How to get the most out of the course

- Read ahead including exam questions and use lectures to ask questions
- L1 is a recap and sets the stage. Check you are comfortable
- Take notes
- Do the course work and write well. Straightforward - schedule smartly
- Exam results tend to be highly bi-modal
- If you are struggling, ask earlier rather than later
- If you don’t understand - it’s probably my fault - so ask!
Course structure

- L1 Introduction and Recap
- L2 Course Work - again updated from last year
- 4-5 lectures on classical optimisation
  (Based on \textit{EaC})
- 5-6 lectures on high level/parallel
  (Based on \textit{CMA} + papers)
- 4-5 lectures on adaptive compilation
  (Based on papers)
- Additional lectures on course work/ revision/ external talks/ research directions
Compilers review
What is a compiler?

- Translates a program from source language to target language
- Often target is assembly
- If target is a source language then “source-to-source” compiler
Compilers review

What is a compiler?

- Translates a program from source language to target language
- Often target is assembly
- If target is a source language then “source-to-source” compiler
- Compare this to an interpreter
Compilers review
Optimisation

- Just translating not enough - must optimise!
- Not just performance - also *code size, power, energy*
- Generally *undecidable, often NP-complete*
- Gap between potential performance and actual widening
- Many architectural issues to think about
  - Exploiting parallelism: instruction, thread, multi-core, accelerators
  - Effective management of memory hierarchy registers, L1, L2, L3, Mem, Disk

Small architectural changes have big impact - hard to reason about

Program optimised for CPU with Random cache replacement. What do you change for new machine with LRU?
Compilers review
Typical compiler structure

- Front end takes string of characters into abstract syntax tree
- Optimiser does machine independent optimisations
- Back end does machine dependent optimisation and code generation
Compilers review

Typical compiler structure

- Work broken into small passes or phases
- Different IRs used - choice affects later analysis/optimisation
Compilers review
Front end

Front end stages

Lexical Analysis - Scanner
Finds and verifies basic syntactic items - lexemes, tokens using finite state automata

Syntax Analysis - Parser
Checks tokens follow a grammar based on a context free grammar and builds an Abstract Syntax Tree (AST)

Semantic Analysis - Parser
Checks all names are consistently used. Various type checking schemes employed. Attribute grammar to Milner type inference. Builds a symbol table
Compilers review
Lexical analysis

- Find keywords, identifiers, constants, etc. - these are tokens
- A set of rules are expressed as regular expressions (RE)
- Scanner automatically generated from rules
- Transform RE $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Scanner table

Example scanner rules

\[
\ell \rightarrow (\text{'a'}|\text{'b'}|\ldots|\text{'z'}|\text{'A'}|\text{'B'}|\ldots|\text{'Z'})
\]
\[
digit \rightarrow (\text{'0'}|\text{'1'}|\ldots|\text{'9'})
\]
\[
integer \rightarrow digit \ digit^* \\
real \rightarrow digit \ digit^* \ . \ digit \ digit^*
\]
\[
exp \rightarrow digit \ digit^* \ . \ digit \ digit^* \ (\text{'e'} \mid \text{'E'}) \ digit \ digit^*
\]

\(^{1}\)Except in practically every real compiler, where all of this is hand coded
How are the following classified?

0, 01, 2.6, 2., 2.6E2, and 2E20
Compilers review
Lexical analysis

- Each token has at least:
  - Type (Keyword, LBracket, RBracket, Number, Identifier, String, etc.)
  - Text value (and number value etc.)
  - Source file, line number, position

- White space and comments are typically stripped out
- Error tokens may be returned
Compilers review
Syntactic analysis

- REs not powerful enough
  (matched parentheses, operator precedence, etc)
- Syntax parser described by context free grammar (often BNF)
- Care must be taken to avoid ambiguity
  Generators (YACC, BISON, ANTLR) will complain

Example grammar

\[
expr \rightarrow term \ op \ expr \mid term \\
\]
\[
term \rightarrow number \mid id \\
\]
\[
op \rightarrow \ast \mid + \mid - \\
\]
Parse \( x - 2 \ast y \)
Compilers review

Syntactic analysis

Parse tree for $x - 2 \times y$

Notice this is parsed as $x - (2 \times y)$
What about $x \times 2 - y$?
Compilers review

Syntactic analysis

- Parse trees have irrelevant intermediate nodes
- Removing them gives AST

Simplified parse tree for $x - 2 \ast y$
Compilers review
Syntactic analysis

- Arbitrary CFGs can be expensive to parse
- Simple dynamic programming $T(n) = O(n^3)$
- Restricted classes of CFG with more efficient parsers

<table>
<thead>
<tr>
<th>CFG classes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR(1)</td>
<td>Left to right scan, Rightmost derivation with 1 symbol lookahead</td>
</tr>
<tr>
<td>LL(1)</td>
<td>Left to right scan, Leftmost derivation with 1 symbol lookahead; cannot handle left-recursive grammars</td>
</tr>
<tr>
<td>Others$^a$</td>
<td>LR(k), LL(k), SLR(k), LALR(k), LR(k), IELR(k), GLR(k), LL(*), etc</td>
</tr>
</tbody>
</table>

$^a$Some represent the same languages
Compilers review
Semantic analysis

- Syntactic analysis produces **abstract** syntax tree
  Program may still be invalid
- Semantic analysis checks correct meaning and decorates AST
- Symbol tables record what names refer to at different scopes
- Semantic actions embedded in grammar allow arbitrary code during parsing
- Attribute grammars propagate information around AST
Symbol tables provide two operations
  \text{lookup(name)} \text{ retrieve record associated with name}
  \text{insert(name, record)} \text{ associate record with name}

Stack of symbol tables manages lexical scopes

Lookup searches stack recursively for name

Scope example

(0) char* n = "N";
(0) char* fmt = "\%d";
(0) void foo() {
(1)   int n = 10;
(2)   for( int i = 0; i < n; ++i ) {
(3)     printf(fmt, n);
(2)   }
(0) }

Semantic actions allow arbitrary code to be executed during parsing. Action executed only on successful parse of rule or Action provides conditional check to help parser choose between rules. Side effects can cause trouble with backtracking.

**Semantic actions**

```
\textit{decl} \rightarrow \textit{var id = expr} \quad \{\text{symtab.insert(id.name)}\}

\textit{expr} \rightarrow \textit{number | id} \quad \{\text{assert(symtab.exists(id.name))}\}
```
Compilers review
Semantic analysis - attribute grammars

- Attribute grammar is a CFG with:
  - Attributes associated with each symbol
  - Semantic rules per production to move attributes
- Attributes can be inherited or synthesised
- Semantic rules can access global data structures, such as a symbol table

### Attribute grammar example - types

```
expr → term op expr  \quad \text{expr.type} = F_{op}(\text{term.type}, \text{expr.type})
term → num | id \quad \text{term.type} = \text{num.type} | \text{id.type}
\quad \text{op} → \ast | + | − \quad F_{op} = F_{\ast} | F_{+} | F_{−}
```
Compilers review
Semantic analysis - attribute grammars

Attribute grammar example - $x - 2 \times y$  
x:int, y:real, int < real

Type matrices can encode errors

Example

<table>
<thead>
<tr>
<th>F</th>
<th>int</th>
<th>real</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>real</td>
<td>double</td>
</tr>
<tr>
<td>real</td>
<td>real</td>
<td>real</td>
<td>⊥</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>⊥</td>
<td>real</td>
</tr>
</tbody>
</table>
Translate AST in to assembler - walk through the tree and emit code based on node type

ILOC instruction set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadl 2 → r2</td>
<td>Load constant 2 into r2</td>
</tr>
<tr>
<td>loadl @x → r1</td>
<td>Load value x into r1</td>
</tr>
<tr>
<td>loadA0 r0, r1 → r1</td>
<td>Mem[r0 + r1] → r1</td>
</tr>
<tr>
<td>add r2, r3 → r1</td>
<td>Add integers r1 = r2 + r3</td>
</tr>
</tbody>
</table>

Assume activation record pointer in r0
Typical top down generator - left to right - for simple expressions

Assume activation record pointer in register $r_0$

function gen(node) : Register
Typical top down generator - left to right - for simple expressions

Assume activation record pointer in register $r_0$

```cpp
function gen( node ) : Register
    case num
        r = nextreg()
        emit(loadI value( node ) → r)
    return r
```

Typical top down generator - left to right - for simple expressions

Assume activation record pointer in register $r_0$

```plaintext
function gen( node ) : Register
  case num
    r = nextreg()
    emit(loadI value( node ) → r)
    return r
  case id
    r = nextreg()
    emit(loadI offset( node ) → r)
    emit(loadA $r_0$, r → r)
    return r
```
Typical top down generator - left to right - for simple expressions

Assume activation record pointer in register $r_0$

```
function gen( node ) : Register
  case num
    r = nextreg()
    emit(loadI value( node ) → r)
    return r
  case id
    r = nextreg()
    emit( loadI offset( node ) → r)
    emit( loadA $r_0$, r → r)
    return r
  case binop( left, +, right )
    r_L = gen( left ); r_R = gen( right )
    emit( add $r_L$, $r_R$ → $r_R$ )
    return $r_R$
```
Generate code for $x - 2 \times y$

- **id**: $x$
- **number**: 2
- **binexpr**: $\times$
- **binexpr**: $-$
- **id**: $y$
Generate code for $x - 2 \times y$

- $\text{loadl } @x \rightarrow r_1$
- $\text{loadA0 } r_0, r_1 \rightarrow r_1$
Generate code for $x - 2 \times y$

- **loadl @x → r_1**
- **loadA0 r_0, r_1 → r_1**
- **loadl 2 → r_2**
Generate code for $x - 2 \times y$

- $\text{load@x} \rightarrow r_1$
- $\text{loadA0 } r_0, r_1 \rightarrow r_1$
- $\text{loadl } 2 \rightarrow r_2$
- $\text{load@y} \rightarrow r_3$
- $\text{loadA0 } r_0, r_3 \rightarrow r_3$
Generate code for $x - 2 \times y$

- loadl @x $\rightarrow$ $r_1$
- loadl 2 $\rightarrow$ $r_2$
- loadl @y $\rightarrow$ $r_3$
- mult $r_2$, $r_3$ $\rightarrow$ $r_3$

Diagram:

```

Generate code for x - 2 * y

id
  | binexpr -
  |       |
  |       |
  |       |
  | id  |
x
  |

binexpr
  | *
  |     |
  |     |
  |     |
  | id |
  |    |
y
  |

number
  | 2
  |
```

3 registers used
Generate code for $x - 2 \times y$

- loadl @x $\rightarrow r_1$
- loadA0 $r_0, r_1 \rightarrow r_1$
- loadl 2 $\rightarrow r_2$
- loadl @y $\rightarrow r_3$
- loadA0 $r_0, r_3 \rightarrow r_3$
- mult $r_2, r_3 \rightarrow r_3$
- sub $r_1, r_3 \rightarrow r_3$
Compilers review
Basic Code Generation

Generate code for $x - 2 * y$

id $x$ -> $r_1$
loadA0 $r_0$, $r_1$ -> $r_1$
loadl 2 -> $r_2$
loadl @y -> $r_3$
loadA0 $r_0$, $r_3$ -> $r_3$
mult $r_2$, $r_3$ -> $r_3$
sub $r_1$, $r_3$ -> $r_3$

3 registers used
Compilers review
Optimisation

- Reducing number of registers used *usually* good
- Current traversal order left to right
  \( (r_L = \text{gen(left)}; r_R = \text{gen(right)}) \)
- Instead traverse child needing most registers first
- `nextreg()` must know which regs unused

Most registers first traversal order

loadl @y → r₁
loadA0 r₀, r₁ → r₁
loadl 2 → r₂
mult r₂, r₁ → r₁
loadl @x → r₂
loadA0 r₀, r₂ → r₂
sub r₂, r₁ → r₂

2 registers used
Expression, \( x - 2 \times y \) will have context

Subtrees of expression already evaluated?

**Common subexpression elimination**

\[
\begin{align*}
a &= 2 \times y \times z \\
b &= x - 2 \times y
\end{align*}
\]

\[
\begin{align*}
t &= 2 \times y \\
a &= t \times z \\
b &= x - t
\end{align*}
\]
In first part of course
- Assume uni-processor with instruction level parallelism, registers and memory
- Generated assembler should not perform any redundant computation
- Should utilise all available functional units and minimise impact of latency
- Register access is fast compared to memory but limited in number. Use wisely
- Two flavours considered superscalar out-of-order vs VLIW: Dynamic vs static scheduling

Later consider multi-core architecture
Compilation as translation and optimisation
Compiler structure
Phase order lexical, syntactic, semantic analysis
Naive code generation and optimisation
Next lecture course work
Then scalar optimisation - middle end
The biggest revolution in the technological landscape for fifty years

Now accepting applications!

Find out more and apply at: pervasiveparallelism.inf.ed.ac.uk

• 4-year programme: MSc by Research + PhD

• Research-focused: Work on your thesis topic from the start

• Collaboration between:
  ▶ University of Edinburgh’s School of Informatics
    ★ Ranked top in the UK by 2014 REF
  ▶ Edinburgh Parallel Computing Centre
    ★ UK’s largest supercomputing centre

• Research topics in software, hardware, theory and application of:
  ▶ Parallelism
  ▶ Concurrency
  ▶ Distribution

• Full funding available

• Industrial engagement programme includes internships at leading companies

The biggest revolution in the technological landscape for fifty years

Now accepting applications!
Find out more and apply at: pervasiveparallelism.inf.ed.ac.uk