Chapters 4, 5, 6 & 7 in EaC2e
Code Generation

In a modern, multi-IR compiler, code generation happens several times

- Generate an IR straight out of the parser
  - Might be an AST, might be some high-level (abstract) linear form
  - Almost always accompanied by a “symbol table” of some form
- Code passes through one or more “lowering” pass
  - Takes the code and decreases (“lowers”) the level of abstraction
  - Expand complex operations (e.g., call or mvcl), make control-flow explicit

The problems are, essentially, the same
The mechanisms are, likely, quite different
Mechanism versus Content

In the lectures on SDT, we focused on how to generate code.

In the lectures on code shape, we will focus on what code to generate.

Different instruction sequences for the same source code produce significantly different performance:

- When two sequences produce the same result, choose the fastest one.
- The compiler makes fundamental choices that are hard to change.

Tradeoffs in performance often involve tradeoffs in generality:

- Code that handles the general case is robust and, often, slower.
- Code tailored to specific cases is more brittle and, often, faster.
  - Code optimization is, to a large extent, the art of capitalizing on context.
Consider copying characters from `src` to `dst`

```c
char src[256], dst[256];
int len, i;
if (len < 256) {
    i = 0;
    while(i < len) {
        dst[i] = src[i];
        i++;
    }
}
```

```c
char src[256], dst[256];
int64 *p, * q;
int len,i;
if (len < 256 && len % 8 == 0) {
    i = 0;
    p = (int64 *) & src[0];
    q = (int64 *) & dst[0];
    while(i < len) {
        *p++ = *q++;
        i += 8;
    }
}
```

**Generic Version**
len loads & stores

**Aligned Strings of 8x bytes**
(len/8) loads & stores

**Context:**
Declarations ensure that `src` and `dst` force the necessary storage alignment.
Separate declarations mean that `src` and `dst` do not overlap.

Now, imagine the version that uses 8 byte loads as long as it can, then gets that last (<7) bytes with 4, 2, & 1 byte loads.

(as long as the strings do not overlap and start on an aligned boundary)
Code Shape (Chapter 7)

Definition
• All those nebulous properties of the code that affect performance
• Includes code, approach for different constructs, cost, storage requirements & mapping, & choice of operations
• Code shape is the end product of many decisions *(big & small)*

Impact
• Code shape influences algorithm choice & results
• Code shape can encode important facts, or hide them

Rule of thumb: expose as much derived information as possible
• Example: explicit branch targets in ILOC simplify analysis
• Example: hierarchy of memory operations in ILOC *(EaC2e, p 251)*

See Bob Morgan’s book for more ILOC examples [268 in EaC2e Bibliography]
My favorite code shape example

- What if \( x \) is 2 and \( z \) is 3?
- What if \( y+z \) is evaluated earlier?

The “best” shape for \( x+y+z \) depends on contextual knowledge
- There may be several conflicting options

Addition is commutative & associative for integers
Code Shape

Another example — implementing the case statement

• Implement it as cascaded if-then-else statements
  – Cost depends on where your case actually occurs
  – $O(\text{number of cases})$

• Implement it as a binary search
  – Need a dense set of conditions to search
  – Uniform ($\log n$) cost

• Implement it as a jump table
  – Need a compact & computable set of case labels
  – Lookup address in a table & jump to it
  – Uniform ($\text{constant}$) cost

Compiler must choose best implementation strategy
No amount of massaging or transforming in the optimizer will convert one strategy into another

Performance depends on order of cases in the final code ⇒ compiler should reorder based on frequency!
Why worry about code shape? Can’t we just trust the optimizer and the back end?

• Optimizer and back end approximate the answers to many hard problems
• The compiler’s individual passes must run quickly
• It often pays to encode useful information into the IR
  – Shape of an expression or a control structure
  – A value kept in a register rather than in memory
• Deriving such information may be expensive, when possible
• Recording it explicitly in the IR is often easier and cheaper

A good optimizer can tune the engine, but it cannot make a Volkswagen Beetle into a Porsche
Code Shape

The “best” code shape depends on context

• Context in the code being compiled
  – What are the common subexpressions? the known constants?
  – Do case labels form a compact set?
  – What values in registers are live after a call?

• Context in the compiler itself
  – In SDT and optimization, shape the code so that it optimizes well
  – In back end, shape the code so that it runs quickly

And, again, code shape is a content issue not a mechanism issue

Neither case statement nor string copy implementation change between SDT, a tree-walk lowering pass, and a pattern-matching instruction selector

Code shape is one of the places where compilation resembles art.
Consider “a + b + c + d + e + f + g + h”

Left-recursive expression grammar (e.g., LR parser) would produce a left-associative tree

Sequential dependences constrain execution order

Right recursion and right associative trees have a symmetric problem
Consider "a + b + c + d + e + f + g + h"

- No rational parser will produce this tree.
- It results from an explicit reordering pass.
- Best approach might be a post-optimization, pre-scheduling pass.

Sequential dependences are much more relaxed.

Balanced tree

Corresponding code:

\[
\begin{align*}
    t_1 &\leftarrow a + b \\
    t_2 &\leftarrow c + d \\
    t_3 &\leftarrow e + f \\
    t_4 &\leftarrow g + h \\
    t_5 &\leftarrow t_1 + t_2 \\
    t_6 &\leftarrow t_3 + t_4 \\
    t_7 &\leftarrow t_5 + t_6
\end{align*}
\]
### Code Shape & Instruction Scheduling

### Evaluation Order

#### Schedules for Balanced Tree versus Left-Associative Tree

Assumes two functional units

<table>
<thead>
<tr>
<th></th>
<th>Balanced Tree</th>
<th>Left-Associative Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unit 0</td>
<td>Unit 0</td>
</tr>
<tr>
<td>1</td>
<td>( t_1 \leftarrow a + b )</td>
<td>( t_1 \leftarrow a + b )</td>
</tr>
<tr>
<td>2</td>
<td>( t_3 \leftarrow e + f )</td>
<td>( t_2 \leftarrow c + d )</td>
</tr>
<tr>
<td>3</td>
<td>( t_5 \leftarrow t_1 + t_2 )</td>
<td>( t_4 \leftarrow g + h )</td>
</tr>
<tr>
<td>4</td>
<td>( t_7 \leftarrow t_5 + t_6 )</td>
<td>( t_6 \leftarrow t_3 + t_4 )</td>
</tr>
<tr>
<td>5</td>
<td>( t_7 \leftarrow t_5 + t_6 )</td>
<td>( t_5 \leftarrow t_4 + f )</td>
</tr>
<tr>
<td>6</td>
<td>( t_6 \leftarrow t_5 + g )</td>
<td>( t_6 \leftarrow t_5 + g )</td>
</tr>
<tr>
<td>7</td>
<td>( t_7 \leftarrow t_6 + h )</td>
<td>( t_7 \leftarrow t_6 + h )</td>
</tr>
</tbody>
</table>

Multi-unit parallelism is called *instruction-level parallelism* or *ILP*.

Multi for other work
However, deep trees \textit{may} need fewer registers than broad trees.

- Deep trees may lead to less spilling
- Broad trees may lead to more instruction-level parallelism

\textbf{As with most code shape decisions, there is not a simple answer}
Evaluation Order

Order of evaluation for expression trees

• The compiler can use commutativity & associativity to improve code
• This problem is truly hard

Commuting operands at a single operation is much easier

• 1\textsuperscript{st} operand must be preserved while 2\textsuperscript{nd} is evaluated
• Takes an extra register for 2\textsuperscript{nd} operand

⇒ Evaluate the more demanding subtree first

(Ershov in the 1950’s, Sethi in the 1970’s)

This is a local strategy, guided by the actual operations and the actual flow of values (or dependences) in the current tree

Taken to its logical conclusion, this creates Sethi-Ullman scheme for register allocation

See Sethi & Ullman, [311] in EaC2e
See Floyd’s 1961 CACM paper, [150] in EaC2e, also on the website