Code Generation for Data Processing
Lecture 6: Instruction Selection

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2022/23
Code Generation – Overview

- Instruction Selection
- Map IR to assembly
- Keep code shape and storage; change operations

- Instruction Scheduling
- Optimize order to hide latencies
- Keep operations, may increase demand for registers

- Register Allocation
- Map virtual to architectural registers and stack
- Adds operations (spilling), changes storage
Code Generation – Overview

- Instruction Selection
  - Map IR to assembly
  - Keep code shape and storage; change operations
Code Generation – Overview

- Instruction Selection
  - Map IR to assembly
  - Keep code shape and storage; change operations

- Instruction Scheduling
  - Optimize order to hide latencies
  - Keep operations, may increases demand for registers
Code Generation – Overview

- Instruction Selection
  - Map IR to assembly
  - Keep code shape and storage; change operations

- Instruction Scheduling
  - Optimize order to hide latencies
  - Keep operations, may increases demand for registers

- Register Allocation
  - Map virtual to architectural registers and stack
  - Adds operations (spilling), changes storage
Instruction Selection (ISel) – Overview

- Find machine instructions to implement abstract IR
- Typically separated from scheduling and register allocation

- Input: IR code with abstract instructions
- Output: lower-level IR code with target machine instructions

\[
i64 \ %10 = \text{add} \ %8, \ %9 \\
i8 \ %11 = \text{trunc} \ %10 \\
i64 \ %12 = \text{const} \ 24 \\
i64 \ %13 = \text{add} \ %7, \ %12 \\
\text{store} \ %11, \ %13
\]

\[
i64 \ %10 = \text{ADD} \ %8, \ %9 \\
\text{STRB} \ %10, \ [%7+24]
\]
Target offers multiple ways to implement operations:
- `imul x, 2`, `add x, x`, `shl x, 1`, `lea x, [x+x]`

Target operations have more complex semantics:
- E.g., combine truncation and offset computation into store
- Can have multiple outputs, e.g. value+flags, quotient+remainder

Target has multiple register sets, e.g. GP and FP/SIMD:
- Important to consider even before register allocation

Target requires specific instruction sequences:
- E.g., for macro fusion
- Often represented as pseudo-instructions until assembly writing
ISel – Typical Constraints

▸ Target offers multiple ways to implement operations
  ▸ `imul x, 2, add x, x, shl x, 1, lea x, [x+x]`
ISel – Typical Constraints

- Target offers multiple ways to implement operations
  - `imul x, 2, add x, x, shl x, 1, lea x, [x+x]`
- Target operations have more complex semantics
  - E.g., combine truncation and offset computation into store
  - Can have multiple outputs, e.g. value+flags, quotient+remainder
ISel – Typical Constraints

- Target offers multiple ways to implement operations
  - `imul x, 2, add x, x, shl x, 1, lea x, [x+x]`
- Target operations have more complex semantics
  - E.g., combine truncation and offset computation into store
  - Can have multiple outputs, e.g. value+flags, quotient+remainder
- Target has multiple register sets, e.g. GP and FP/SIMD
  - Important to consider even before register allocation
Target offers multiple ways to implement operations

- `imul x, 2, add x, x, shl x, 1, lea x, [x+x]`

Target operations have more complex semantics

- E.g., combine truncation and offset computation into store
- Can have multiple outputs, e.g. value+flags, quotient+remainder

Target has multiple register sets, e.g. GP and FP/SIMD

- Important to consider even before register allocation

Target requires specific instruction sequences

- E.g., for macro fusion
- Often represented as pseudo-instructions until assembly writing
Optimal ISel

- Find *most performant* instruction sequence with same semantics (?)
  - i.e., there no program with better “performance” exists
  - Performance = instructions associated with specific costs

---

Optimal ISel

- Find *most performant* instruction sequence with same semantics (?)
  - I.e., there no program with better “performance” exists
  - Performance = instructions associated with specific costs

- Problem: optimal code generation is **undecidable**

---

Optimal ISel

- Find *most performant* instruction sequence with same semantics (?)
  - i.e., there no program with better “performance” exists
  - Performance = instructions associated with specific costs

- Problem: optimal code generation is **undecidable**

- Alternative: optimal *tiling* of IR with machine code instrs
  - IR as dataflow graph, instr. tiles to optimally cover graph
  - $\mathcal{NP}$-complete\(^{20}\)

---

Avoiding ISel Altogether

- Use an interpreter
  - Fast "compilation time", easy to implement
- Slow execution time
  - Best if code is executed once
Avoiding ISel Altogether

Use an interpreter

+ Fast “compilation time”, easy to implement
- Slow execution time

▶ Best if code is executed once
Macro Expansion

- Expand each IR operation with corresponding machine instrs

\[
\begin{align*}
%5 &= \text{add} \ %1, \ 12345 & \rightarrow & \%5a &= \text{movz} \ 12345 \\
%6 &= \text{and} \ %2, \ 7 & \rightarrow & \%6 &= \text{and} \ %2, \ 7 \\
%7 &= \text{shl} \ %5, \ %6 & \rightarrow & \%7a &= \text{lsl} \ %5, \ %6 \\
\end{align*}
\]
Macro Expansion

- Oldest approach, historically also does register allocation
  - Also possible by walking AST
Macro Expansion

- Oldest approach, historically also does register allocation
  - Also possible by walking AST

+ Very fast, linear time, simple to implement, easy to port
- Inefficient and large output code
Macro Expansion

- Oldest approach, historically also does register allocation
  - Also possible by walking AST

  + Very fast, linear time, simple to implement, easy to port
  - Inefficient and large output code

- Used by, e.g., LLVM FastISel, Go, GCC
Peephole Optimization

- Plain macro expansion leads to suboptimal results
- Idea: replace inefficient instruction sequences\textsuperscript{21}

- Originally: physical window over assembly code
  - Replace with more efficient instructions having same effects
  - Possibly with allocated registers

- Extension: do expansion before register allocation\textsuperscript{22}
  - Expand IR into Register Transfer Lists (RTL) with temporary registers
  - While \textit{combining}, ensure that each RTL can be implemented as single instr.

\textsuperscript{21} WM McKeeman. “Peephole optimization”. In: \textit{CACM} 8.7 (1965), pp. 443–444.

Peephole Optimization

- Originally covered only adjacent instructions
- Can also use logical window of data dependencies
  - Problem: instructions with multiple uses
  - Needs more sophisticated matching schemes for data deps.
    ⇒ Tree-pattern matching
Peephole Optimization

- Originally covered only adjacent instructions
- Can also use logical window of data dependencies
  - Problem: instructions with multiple uses
  - Needs more sophisticated matching schemes for data deps.
    ⇒ Tree-pattern matching

+ Fast, also allows for target-specific sequences
- Pattern set grows large, limited potential
Peephole Optimization

- Originally covered only adjacent instructions
- Can also use logical window of data dependencies
  - Problem: instructions with multiple uses
  - Needs more sophisticated matching schemes for data deps.
    ⇒ Tree-pattern matching

+ Fast, also allows for target-specific sequences
- Pattern set grows large, limited potential

- Widely used today at different points during compilation
ISel as Graph Covering – High-level Intuition

- Idea: represent program as data flow graph
ISel as Graph Covering – High-level Intuition

- Idea: represent program as data flow graph

- Tree: expression, comb. of single-use SSA instructions

- DAG: data flow in basic block, e.g. SSA block

- Graph: data flow of entire function, e.g. SSA function
ISel as Graph Covering – High-level Intuition

- Idea: represent program as data flow graph
  - Tree: expression, comb. of single-use SSA instructions
  - DAG: data flow in basic block, e.g. SSA block
  - Graph: data flow of entire function, e.g. SSA function

- ISA “defines” pattern set of trees/DAGs/graphs for instrs.
- Cover data flow tree/DAG/graph with least-cost combination of patterns
  - Patterns in data flow graph may overlap
Tree Covering: Converting SSA into Trees

SSA form:
%4 = shl %1, 4
%5 = add %2, %4
%6 = add %3, %4
%7 = load %5
live-out: %6, %7
Tree Covering: Converting SSA into Trees

- **SSA form:**
  - %4 = shl %1, 4
  - %5 = add %2, %4
  - %6 = add %3, %4
  - %7 = load %5
  - live-out: %6, %7

- **Data flow graph:**

$$
\text{ld} \quad + \quad + \quad - \quad + \quad 4
\quad %3 \quad %3 \quad %2 \quad %1 \quad %1
$$
Tree Covering: Converting SSA into Trees

- **SSA form:**
  - %4 = shl %1, 4
  - %5 = add %2, %4
  - %6 = add %3, %4
  - %7 = load %5
  - live-out: %6, %7

- **Data flow graph:**

- **Method 1:**
  - Edge Splitting

- **Method 2:**
  - Node Duplication
Tree Covering: Converting SSA into Trees

- SSA form:
  
  \[
  \begin{align*}
  \%4 &= \text{shl} \ \%1, 4 \\
  \%5 &= \text{add} \ \%2, \ %4 \\
  \%6 &= \text{add} \ \%3, \ %4 \\
  \%7 &= \text{load} \ \%5 \\
  \text{live-out:} &\ %6, \ %7
  \end{align*}
  \]

- Data flow graph:

- Method 1: Edge Splitting

- Method 2: Node Duplication
## Tree Covering: Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Cost</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 ) ( GP_{R1} \rightarrow \ll (GP_{R2}, K_1) )</td>
<td>1</td>
<td>lsl ( R_1, R_2, #K_1 )</td>
</tr>
<tr>
<td>( P_1 ) ( GP_{R1} \rightarrow + (GP_{R2}, GP_{R3}) )</td>
<td>1</td>
<td>add ( R_1, R_2, R_3 )</td>
</tr>
<tr>
<td>( P_2 ) ( GP_{R1} \rightarrow + (GP_{R2}, \ll (GP_{R3}, K_1)) )</td>
<td>2</td>
<td>add ( R_1, R_2, R_3, ) lsl #K_1</td>
</tr>
<tr>
<td>( P_3 ) ( GP_{R1} \rightarrow + (\ll (GP_{R2}, K_1), GP_{R2}) )</td>
<td>2</td>
<td>add ( R_1, R_3, R_2, ) lsl #K_1</td>
</tr>
<tr>
<td>( P_4 ) ( GP_{R1} \rightarrow ld (GP_{R2}) )</td>
<td>2</td>
<td>ldr ( R_1, [R_2] )</td>
</tr>
<tr>
<td>( P_5 ) ( GP_{R1} \rightarrow ld (+ (GP_{R2}, GP_{R3})) )</td>
<td>2</td>
<td>ldr ( R_1, [R_2, R_3] )</td>
</tr>
<tr>
<td>( P_6 ) ( GP_{R1} \rightarrow ld (+ (GP_{R2}, \ll (GP_{R3}, K_1))) )</td>
<td>3</td>
<td>ldr ( R_1, [R_2, R_3, ) lsl #K_1]</td>
</tr>
<tr>
<td>( P_7 ) ( GP_{R1} \rightarrow ld (+ (\ll (GP_{R2}, K_1), GP_{R3}) )</td>
<td>3</td>
<td>ldr ( R_1, [R_3, R_2, ) lsl #K_1]</td>
</tr>
<tr>
<td>( P_8 ) ( GP_{R1} \rightarrow * (GP_{R2}, GP_{R3}) )</td>
<td>3</td>
<td>madd ( R_1, R_2, R_3, ) xzr</td>
</tr>
<tr>
<td>( P_9 ) ( GP_{R1} \rightarrow + (* (GP_{R2}, GP_{R3}), GP_{R4}) )</td>
<td>3</td>
<td>madd ( R_1, R_2, R_3, R_4 )</td>
</tr>
<tr>
<td>( P_{10} ) ( GP_{R1} \rightarrow K_1 )</td>
<td>1</td>
<td>mov ( R_1, K_1 )</td>
</tr>
</tbody>
</table>

...
Tree Covering: Greedy/Maximal Munch

- Top-down always take largest pattern
- Repeat for sub-trees, until everything is covered

+ Easy to implement, fast
Tree Covering: Greedy/Maximal Munch

- Top-down always take largest pattern
- Repeat for sub-trees, until everything is covered

+ Easy to implement, fast
- Result might be non-optimum
Tree Covering: Greedy/Maximal Munch – Example

Matching Patterns:

```
+  
  *  
  /  
  a  (b)  c  2
```

Total cost: 5
Tree Covering: Greedy/Maximal Munch – Example

Matching Patterns:
▶ +: $P_1$ – cost 1 – covered nodes: 1
Tree Covering: Greedy/Maximal Munch – Example

Matching Patterns:
- +: $P_1$ – cost 1 – covered nodes: 1
- +: $P_2$ – cost 2 – covered nodes: 3
Tree Covering: Greedy/Maximal Munch – Example

Matching Patterns:

- +: $P_1$ – cost 1 – covered nodes: 1
- +: $P_2$ – cost 2 – covered nodes: 3
- +: $P_9$ – cost 3 – covered nodes: 2

Total cost: 5

```
madd %1, %a, %b, xzr
add %2, %1, %c, lsl #2
```
Tree Covering: Greedy/Maximal Munch – Example

Matching Patterns:

- +: $P_1$ – cost 1 – covered nodes: 1
- +: $P_2$ – cost 2 – covered nodes: 3 – best
- +: $P_9$ – cost 3 – covered nodes: 2
Tree Covering: Greedy/Maximal Munch – Example

Matching Patterns:

- ➤ +: $P_1$ – cost 1 – covered nodes: 1
- ➤ +: $P_2$ – cost 2 – covered nodes: 3 – best
- ➤ +: $P_9$ – cost 3 – covered nodes: 2
- ➤ *: $P_8$ – cost 3 – covered nodes: 1
Tree Covering: Greedy/Maximal Munch – Example

Matching Patterns:

- +: $P_1$ – cost 1 – covered nodes: 1
- +: $P_2$ – cost 2 – covered nodes: 3 – best
- +: $P_9$ – cost 3 – covered nodes: 2
- *: $P_8$ – cost 3 – covered nodes: 1 – best

Total cost: 5
Tree Covering: Greedy/Maximal Munch – Example

Matching Patterns:

- +: $P_1$ – cost 1 – covered nodes: 1
- +: $P_2$ – cost 2 – covered nodes: 3 – best
- +: $P_9$ – cost 3 – covered nodes: 2
- *: $P_8$ – cost 3 – covered nodes: 1 – best

Total cost: 5
Tree Covering: Greedy/Maximal Munch – Example

Matching Patterns:
- ➤ +: \( P_1 \) – cost 1 – covered nodes: 1
- ➤ +: \( P_2 \) – cost 2 – covered nodes: 3 – best
- ➤ +: \( P_9 \) – cost 3 – covered nodes: 2
- ➤ *: \( P_8 \) – cost 3 – covered nodes: 1 – best

Total cost: 5

```
madd %1, %a, %b, xzr
add %2, %1, %c, lsl #2
```
Tree Covering: with LR-Parsing?

- Can we use (LR-)parsing for instruction selection?

Tree Covering: with LR-Parsing

- Can we use (LR-)parsing for instruction selection? Yes!\(^{23}\)
  - Pattern set = grammar; IR (in prefix notation) = input

Advantages

Tree Covering: with LR-Parsing

Can we use (LR-)parsing for instruction selection? Yes!\textsuperscript{23}

- Pattern set = grammar; IR (in prefix notation) = input

### Advantages
- Possible in linear time
- Can be formally verified
- Implementation can be generated automatically

### Disadvantages
- Constraints must map to non-terminals
- Constant ranges, reg types, ...
- CISC: handle all operand combinations
- Large grammar (impractical)
- Refactoring into non-terminals
- Ambiguity hard to handle optimally

Can we use (LR-)parsing for instruction selection? Yes!\textsuperscript{23}

- Pattern set = grammar; IR (in prefix notation) = input

**Advantages**

- Possible in linear time
- Can be formally verified
- Implementation can be generated automatically

**Disadvantages**

- Constraints must map to non-terminals
  - Constant ranges, reg types, ...
- CISC: handle all operand combinations
  - Large grammar (impractical)
  - Refactoring into non-terminals
- Ambiguity hard to handle optimally

Tree Covering: Dynamic Programming

- Step 1: compute cost matrix, bottom-up for all nodes
  - Matrix: tree node × non-terminal
    (different patterns might yield different non-terminals)
  - Cost is sum of pattern and sum of children costs
  - Always store cheapest rule and cost

- Step 2: walk tree top-down using rules in matrix
  - Start with goal non-terminal, follow rules in matrix

- Time linear w.r.t. tree size

---

### Tree Covering: Dynamic Programming – Example

#### Diagram:

```
+  
|   
*--|--
  b | c
```

#### Table:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>∞</td>
</tr>
<tr>
<td><code>*</code></td>
<td>∞</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>∞</td>
</tr>
<tr>
<td><code>2</code></td>
<td>∞</td>
</tr>
</tbody>
</table>

#### Node:

- 2

#### Pattern:

- `+`  
- `*`  
- `<`  
- `2`  

#### Cost Sum:

<table>
<thead>
<tr>
<th>GP</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td><code>∞</code></td>
</tr>
<tr>
<td><code>*</code></td>
<td><code>∞</code></td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td><code>∞</code></td>
</tr>
<tr>
<td><code>2</code></td>
<td><code>∞</code></td>
</tr>
</tbody>
</table>

---

185
Tree Covering: Dynamic Programming – Example

Node: 2
Pattern: \( P_{10}: \text{GP} \rightarrow K_1 \)
Pat. Cost: 1
Cost Sum: 1

<table>
<thead>
<tr>
<th>Node</th>
<th>+</th>
<th>*</th>
<th>( \prec )</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td>( P_{10} )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tree Covering: Dynamic Programming – Example

Node:  «
Pattern: 
Pat. Cost: 
Cost Sum: 

<table>
<thead>
<tr>
<th>Node</th>
<th>+</th>
<th>*</th>
<th>«</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td></td>
<td></td>
<td></td>
<td>$P_{10}$</td>
</tr>
</tbody>
</table>
Tree Covering: Dynamic Programming – Example

Node: «
Pattern: $P_?: GP \rightarrow «(GP, GP)$
Pat. Cost: 1
Cost Sum: 2

<table>
<thead>
<tr>
<th>Node</th>
<th>+</th>
<th>*</th>
<th>«</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_?$</td>
<td>$P_{10}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tree Covering: Dynamic Programming – Example

Node: «
Pattern: \( P_1: GP \rightarrow «(GP, K_1) \)
Pat. Cost: 1
Cost Sum: 2

<table>
<thead>
<tr>
<th>Node</th>
<th>+</th>
<th>*</th>
<th>«</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td>( P_1 )</td>
<td>( P_{10} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tree Covering: Dynamic Programming – Example

Node: *
Pattern:
Pat. Cost:
Cost Sum:

<table>
<thead>
<tr>
<th>Node</th>
<th>+</th>
<th>*</th>
<th>«</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP Cost</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_1$</td>
<td>$P_{10}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tree Covering: Dynamic Programming – Example

Node: *  
Pattern: $P_8: GP \rightarrow \ast (GP, GP)$  
Pat. Cost: 3  
Cost Sum: 3

<table>
<thead>
<tr>
<th>Node</th>
<th>+</th>
<th>*</th>
<th>«</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>∞</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_8$</td>
<td>$P_1$</td>
<td>$P_{10}$</td>
<td></td>
</tr>
</tbody>
</table>
Tree Covering: Dynamic Programming – Example

Node: +
Pattern:
Pat. Cost:
Cost Sum:

<table>
<thead>
<tr>
<th>Node</th>
<th>+</th>
<th>*</th>
<th>«</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP Cost</td>
<td>∞</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_8$</td>
<td>$P_1$</td>
<td>$P_{10}$</td>
<td></td>
</tr>
</tbody>
</table>
Tree Covering: Dynamic Programming – Example

Node: +
Pattern: $P_1: GP \rightarrow + (GP, GP)$
Pat. Cost: 1
Cost Sum: 5

<table>
<thead>
<tr>
<th>Node</th>
<th>+</th>
<th>*</th>
<th>«</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_1$</td>
<td>$P_8$</td>
<td>$P_1$</td>
<td>$P_{10}$</td>
</tr>
</tbody>
</table>
Tree Covering: Dynamic Programming – Example

Node: +
Pattern: $P_2: GP \rightarrow + (GP, \langle GP, K_1 \rangle)$
Pat. Cost: 2
Cost Sum: 5

<table>
<thead>
<tr>
<th>Node</th>
<th>+</th>
<th>*</th>
<th>«</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_1$</td>
<td>$P_8$</td>
<td>$P_1$</td>
<td>$P_{10}$</td>
</tr>
</tbody>
</table>
Tree Covering: Dynamic Programming – Example

Node: +
Pattern: $P_9: GP \rightarrow +(*(GP, GP), GP)$
Pat. Cost: 3
Cost Sum: 4

<table>
<thead>
<tr>
<th>Node</th>
<th>+</th>
<th>*</th>
<th>«</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_9$</td>
<td>$P_8$</td>
<td>$P_1$</td>
<td>$P_{10}$</td>
</tr>
</tbody>
</table>
Cost analysis can actually be *precomputed*.

Idea: annotate each node with a state based on child states.

Lookup node label from precomputed table (one per non-terminal).

Significantly improves compilation time.

But: Tables can be large, need to cover all possible (sub-)trees.

Variation: dynamically compute and cache state tables.

---


Tree Covering

- Efficient: linear time to find local optimum
- Better code than pure macro expansion
- Applicable to many ISAs

- Common sub-expressions cannot be represented
  - Need either edge split (prevents using complex instructions)
  - or node duplication (redundant computation ⇒ inefficient code)
- Cannot make use of multi-output instructions (e.g., divmod)
Tree Covering

+ Efficient: linear time to find local optimum
+ Better code than pure macro expansion
+ Applicable to many ISAs
Tree Covering

+ Efficient: linear time to find local optimum
+ Better code than pure macro expansion
+ Applicable to many ISAs

− Common sub-expressions cannot be represented
  ▶ Need either edge split (prevents using complex instructions) or node duplication (redundant computation ⇒ inefficient code)
− Cannot make use of multi-output instructions (e.g., divmod)
DAG Covering

- Idea: lift restriction of trees, operate on data flow DAG
  - Reminder: an SSA basic block already forms a DAG

- Trivial approach: split into trees

---

DAG Covering

- Idea: lift restriction of trees, operate on data flow DAG
  - Reminder: an SSA basic block already forms a DAG

- Trivial approach: split into trees 😞

- Least-cost covering is \( \mathcal{NP} \)-complete\(^{27} \)

Step 1: compute cost matrix, bottom-up for all nodes
  - As before; make sure to visit each node once
Step 2: iterate over DAG top-down
  - Respect that multiple roots exist: start from all roots
  - Mark visited node/non-terminal combinations: avoid redundant emit

DAG Covering: Adapting Dynamic Programming

- Step 1: compute cost matrix, bottom-up for all nodes
  - As before; make sure to visit each node once
- Step 2: iterate over DAG top-down
  - Respect that multiple roots exist: start from all roots
  - Mark visited node/non-terminal combinations: avoid redundant emit

+ Linear time
- Generally not optimal, only for specific grammars

---

DAG Covering: Adapting Dynamic Programming I – Example

Node: *
Pattern:
Pat. Cost:
Cost Sum:

<table>
<thead>
<tr>
<th>GP</th>
<th>Cost Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2</td>
<td>∞</td>
</tr>
<tr>
<td>+1</td>
<td>∞</td>
</tr>
<tr>
<td>*</td>
<td>∞</td>
</tr>
</tbody>
</table>

Node | +2 | +1 | *  
---|----|----|---
+1 |    |    |   
+2 |    |    |   
*  |    |    |   

Diagram:

```
+1
  /   
 b   *   d
  /   
 a   c
```

Node: *
Pattern:
Pat. Cost:
Cost Sum:
DAG Covering: Adapting Dynamic Programming I – Example

Node: *
Pattern: \( P_8: GP \rightarrow * (GP, GP) \)
Pat. Cost: 3
Cost Sum: 3

<table>
<thead>
<tr>
<th>Node</th>
<th>( +_2 )</th>
<th>( +_1 )</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>3</td>
</tr>
<tr>
<td>Pattern</td>
<td>( P_8 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DAG Covering: Adapting Dynamic Programming I – Example

Node: +₁
Pattern: 
Pat. Cost: 
Cost Sum: 

<table>
<thead>
<tr>
<th>Node</th>
<th>+₂</th>
<th>+₁</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>∞</td>
<td>∞</td>
<td>3</td>
</tr>
<tr>
<td>Pattern</td>
<td>P₈</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Diagram of a DAG with nodes a, b, c, d, labeled +₁ and +₂, and a pattern node labeled *.]
DAG Covering: Adapting Dynamic Programming I – Example

Node: $+_1$

Pattern: $P_1: GP \rightarrow + (GP, GP)$

Pat. Cost: 1

Cost Sum: 4

<table>
<thead>
<tr>
<th>Node</th>
<th>$+_2$</th>
<th>$+_1$</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>$\infty$</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_1$</td>
<td>$P_8$</td>
<td></td>
</tr>
</tbody>
</table>
DAG Covering: Adapting Dynamic Programming I – Example

Node: \(+_1\)
Pattern: \(P_9: GP \rightarrow +(\ast(GP, GP), GP)\)
Pat. Cost: 3
Cost Sum: 3

<table>
<thead>
<tr>
<th></th>
<th>(+_2)</th>
<th>(+_1)</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP Cost</td>
<td>(\infty)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Pattern</td>
<td>(P_9)</td>
<td>(P_8)</td>
<td></td>
</tr>
</tbody>
</table>
Node: $+_2$
Pattern:
Pat. Cost:
Cost Sum:

<table>
<thead>
<tr>
<th>GP</th>
<th>Pattern</th>
<th>$+_2$</th>
<th>$+_1$</th>
<th>$*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>Cost</td>
<td>$\infty$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_9$</td>
<td>$P_8$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Node: $+_2$

Pattern: $P_1: GP \rightarrow +(GP, GP)$

Pat. Cost: 1

Cost Sum: 4

<table>
<thead>
<tr>
<th>Node</th>
<th>$+_2$</th>
<th>$+_1$</th>
<th>$\ast$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_1$</td>
<td>$P_9$</td>
<td>$P_8$</td>
</tr>
</tbody>
</table>
DAG Covering: Adapting Dynamic Programming I – Example

Node: \( +_2 \)
Pattern: \( P_9: GP \rightarrow +(*(GP, GP), GP) \)
Pat. Cost: 3
Cost Sum: 3

<table>
<thead>
<tr>
<th></th>
<th>( +_2 )</th>
<th>( +_1 )</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>Cost</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Pattern</td>
<td>( P_9 )</td>
<td>( P_9 )</td>
<td>( P_8 )</td>
</tr>
</tbody>
</table>
DAG Covering: Adapting Dynamic Programming I – Example

Total cost: 6

\[
\text{madd } \%1, \%b, \%c, \%a \\
\text{madd } \%2, \%b, \%c, \%d
\]

<table>
<thead>
<tr>
<th>Node</th>
<th>+2</th>
<th>+1</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP Cost</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Pattern</td>
<td>$P_9$</td>
<td>$P_9$</td>
<td>$P_8$</td>
</tr>
</tbody>
</table>
DAG Covering: Adapting Dynamic Programming II\textsuperscript{29}

- Step 1: compute cost matrix, bottom-up (as before)
- Step 2: iterate over DAG top-down (as before)
- Step 3: identify overlaps and check whether split is beneficial
  - Mark nodes which should not be duplicated as \textit{fixed}
- Step 4: as step 1, but skip patterns that \textit{include} fixed nodes
- Step 5: as step 2

\textsuperscript{29} DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: CGO. 2008, pp. 45–54.
DAG Covering: Adapting Dynamic Programming II

- Step 1: compute cost matrix, bottom-up (as before)
- Step 2: iterate over DAG top-down (as before)
- Step 3: identify overlaps and check whether split is beneficial
  - Mark nodes which should not be duplicated as fixed
- Step 4: as step 1, but skip patterns that include fixed nodes
- Step 5: as step 2

+ Probably fast? “Near-optimal”?
− Generally not optimal, superlinear time

DAG Covering: ILP\textsuperscript{30}

- Idea: model ISel as integer linear programming (ILP) problem
- $P$ is set of patterns with cost and edges, $V$ are DAG nodes
- Variables: $M_{p,v}$ is 1 iff a pattern $p$ is rooted at $v$

\[
\begin{align*}
\text{minimize} & \quad \sum_{p,v} p\.cost \cdot M_{p,v} \\
\text{subject to} & \quad \forall r \in \text{roots}. \sum_{p} M_{p,r} \geq 1 \\
& \quad \forall p, v, e \in p\.edges(v). M_{p,v} - \sum_{p'} M_{p',e} \leq 0 \\
& \quad M_{p,v} \in \{0, 1\}
\end{align*}
\]

DAG Covering: ILP\textsuperscript{30}

- Idea: model ISel as integer linear programming (ILP) problem
- $P$ is set of patterns with cost and edges, $V$ are DAG nodes
- Variables: $M_{p,v}$ is 1 iff a pattern $p$ is rooted at $v$

\[
\begin{align*}
\text{minimize} & \quad \sum_{p,v} p.\text{cost} \cdot M_{p,v} \\
\text{subject to} & \quad \forall r \in \text{roots}. \quad \sum_p M_{p,r} \geq 1 \\
& \quad \forall p, v, e \in p.\text{edges}(v). \quad M_{p,v} - \sum_{p'} M_{p',e} \leq 0 \\
& \quad M_{p,v} \in \{0, 1\}
\end{align*}
\]

+ Optimal result
- Practicability beyond small programs questionable (at best)

DAG Covering: Greedy/Maximal Munch

- Top-down, start at roots, always take largest pattern
- Repeat for remaining roots until whole graph is covered

Easy to implement, reasonably fast

Result often non-optimal

Used by: LLVM SelectionDAG
DAG Covering: Greedy/Maximal Munch

- Top-down, start at roots, always take largest pattern
- Repeat for remaining roots until whole graph is covered

+ Easy to implement, reasonably fast
- Result often non-optimal

Used by: LLVM SelectionDAG
DAG Covering: Greedy/Maximal Munch

- Top-down, start at roots, always take largest pattern
- Repeat for remaining roots until whole graph is covered

+ Easy to implement, reasonably fast
- Result often non-optimal

Used by: LLVM SelectionDAG
Graph Covering

- Idea: lift limitation of DAGs, cover entire function graphs
- Better handling of predication and VLIW bundling
  - E.g., hoisting instructions from a conditional block
- Allows to handle instructions that expand to multiple blocks
  - switch, select, etc.
Graph Covering

- Idea: lift limitation of DAGs, cover entire function graphs
- Better handling of predication and VLIW bundling
  - E.g., hoisting instructions from a conditional block
- Allows to handle instructions that expand to multiple blocks
  - `switch`, `select`, etc.

- May need new IR to model control flow in addition to data flow
Graph Covering

- Idea: lift limitation of DAGs, cover entire function graphs
- Better handling of predication and VLIW bundling
  - E.g., hoisting instructions from a conditional block
- Allows to handle instructions that expand to multiple blocks
  - switch, select, etc.

- May need new IR to model control flow in addition to data flow
- In practice: only used by adapting methods showed for DAGs
- Used by: Java HotSpot Server, LLVM GlobalISel (all tree-covering)
Flawed Assumptions

Cost model is fundamentally flawed
⇒ "Optimal" ISel doesn't really mean anything

Out-of-order execution: costs are not linear
Instructions executed in parallel, might execute for free
Possible contention of functional units

Register allocator will modify instructions
"Bad" instructions boundaries increase register requirements
More stack spilling
⇝ much slower code!
Flawed Assumptions

- Cost model is fundamentally flawed
- “Optimal” ISel doesn’t really mean anything
Flawed Assumptions

- Cost model is fundamentally flawed

  ⇒ “Optimal” ISel doesn’t really mean anything

- Out-of-order execution: costs are not linear
  - Instructions executed in parallel, might execute for free
  - Possible contention of functional units

- Register allocator will modify instructions

- “Bad” instructions boundaries increase register requirements
  - More stack spilling ⇒ much slower code!
LLVM Back-end: Overview

- LLVM-IR → Machine IR: instruction selection + scheduling
- MIR is SSA-representation of target instructions
- Selectors: SelectionDAG, FastISel, GlobalISel
- Also selects register bank (GP/FP/...) – required for instruction
- Annotates registers: calling convention, encoding restrictions, etc.
- MIR: minor (peephole) optimizations
- MIR: register allocation
- MIR: prolog/epilog insertion (stack frame, callee-saved regs, etc.)
- MIR → MC: translation to machine code
LLVM Back-end: Overview

- LLVM-IR → Machine IR: instruction selection + scheduling
  - MIR is SSA-representation of target instructions
  - Selectors: SelectionDAG, FastISel, GlobalISel
  - Also selects register bank (GP/FP/...) – required for instruction
  - Annotates registers: calling convention, encoding restrictions, etc.
LLVM Back-end: Overview

- LLVM-IR $\rightarrow$ Machine IR: instruction selection + scheduling
  - MIR is SSA-representation of target instructions
  - Selectors: SelectionDAG, FastISel, GlobalISel
  - Also selects register bank (GP/FP/…) – required for instruction
  - Annotates registers: calling convention, encoding restrictions, etc.
- MIR: minor (peephole) optimizations
- MIR: register allocation
- MIR: prolog/epilog insertion (stack frame, callee-saved regs, etc.)
LLVM Back-end: Overview

- LLVM-IR → Machine IR: instruction selection + scheduling
  - MIR is SSA-representation of target instructions
  - Selectors: SelectionDAG, FastISel, GlobalISel
  - Also selects register bank (GP/FP/...) – required for instruction
  - Annotates registers: calling convention, encoding restrictions, etc.

- MIR: minor (peephole) optimizations
- MIR: register allocation
- MIR: prolog/epilog insertion (stack frame, callee-saved regs, etc.)
- MIR → MC: translation to machine code
define i64 @fn(i64 %a, i64 %b, i64 %c) {
    %shl = shl i64 %c, 2
    %mul = mul i64 %a, %b
    %add = add i64 %mul, %shl
    ret i64 %add
}

# YAML with name, registers, frame info
body: |
    bb.0 (%ir-block.0):
        liveins: $x0, $x1, $x2
        %2:gpr64 = COPY $x2
        %1:gpr64 = COPY $x1
        %0:gpr64 = COPY $x0
        %3:gpr64 = MADDXrrr %0, %1, $xzr
        %4:gpr64 = ADDXrs killed %3, %2, 2
        $x0 = COPY %4
        RET_ReallyLR implicit $x0

llc -march=aarch64 -stop-after=finalize-isel
LLVM: Instruction Selectors

- FastISel
  - Uses macro expansion
  - Low compile-time
  - Code quality poor
  - Only common cases
  - Otherwise: fallback to SelectionDAG
  - Default for -O0

- SelectionDAG
  - Converts each block into separate DAGs
  - Greedy tree matching
  - Slow, but good code
  - Handles all cases
  - No cross-block opt. (done in DAG building)
  - Default

- GlobalISel
  - Converts to generic-MIR then legalize to MIR
  - Reuses SD patterns
  - Faster than SelDAG
  - Few architectures
  - Handles many cases, SelDAG-fallback
LLVM: Instruction Selectors

FastISel

- Uses macro expansion
- Low compile-time
- Code quality poor

- Only common cases
- Otherwise: fallback to SelectionDAG

- Default for -00
PLLVM: Instruction Selectors

**FastISel**
- Uses macro expansion
- Low compile-time
- Code quality poor
- Only common cases
- Otherwise: fallback to SelectionDAG
- Default for `-00`

**SelectionDAG**
- Converts each block into separate DAGs
- Greedy tree matching
- Slow, but good code
- Handles all cases
- No cross-block opt.
  (done in DAG building)
- Default
<table>
<thead>
<tr>
<th>FastISel</th>
<th>SelectionDAG</th>
<th>GlobalISel</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Uses macro expansion</td>
<td>- Converts each block into separate DAGs</td>
<td>- Conv. to generic-MIR then legalize to MIR</td>
</tr>
<tr>
<td>- Low compile-time</td>
<td>- Greedy tree matching</td>
<td>- Reuses SD patterns</td>
</tr>
<tr>
<td>- Code quality poor</td>
<td>- Slow, but good code</td>
<td>- Faster than SelDAG</td>
</tr>
<tr>
<td>- Only common cases</td>
<td>- Handles all cases</td>
<td>- Few architectures</td>
</tr>
<tr>
<td>- Otherwise: fallback to SelectionDAG</td>
<td>- No cross-block opt. (done in DAG building)</td>
<td>- Handles many cases, SelDAG-fallback</td>
</tr>
<tr>
<td>- Default for -00</td>
<td>- Default</td>
<td>-</td>
</tr>
</tbody>
</table>

**FastISel**
- Uses macro expansion
- Low compile-time
- Code quality poor
- Only common cases
- Otherwise: fallback to SelectionDAG
- Default for -00

**SelectionDAG**
- Converts each block into separate DAGs
- Greedy tree matching
- Slow, but good code
- Handles all cases
- No cross-block opt. (done in DAG building)
- Default

**GlobalISel**
- Conv. to generic-MIR then legalize to MIR
- Reuses SD patterns
- Faster than SelDAG
- Few architectures
- Handles many cases, SelDAG-fallback
LLVM SelectionDAG: IR to ISelDAG

- Construct DAG for basic block
- EntryToken as ordering chain

isel input for fn:
EntryToken
t0
c
Register %0
t1
i64
ch
Register %1
t3
i64
c
Register %2
t5
i64
Constant<2>
t7
i64
mul
t9
i64
shl
t8
i64
add
t10
i64
CopyToReg
t12
c
Register $x0
t11
i64
CopyFromReg
t2
i64
ch
CopyFromReg
t4
i64
ch
CopyFromReg
t6
i64
ch
AArch64ISD::RET_FLAG
t13
ch
GraphRoot
LLVM SelectionDAG: IR to ISelDAG

- Construct DAG for basic block
  - EntryToken as ordering chain
- Legalize data types
  - Integers: promote or expand into multiple
  - Vectors: widen or split (or scalarize)
LLVM SelectionDAG: IR to ISelDAG

- Construct DAG for basic block
  - EntryToken as ordering chain
- Legalize data types
  - Integers: promote or expand into multiple
  - Vectors: widen or split (or scalarize)
- Legalize operations
  - E.g., conditional move, etc.
LLVM SelectionDAG: IR to ISelDAG

- Construct DAG for basic block
  - EntryToken as ordering chain
- Legalize data types
  - Integers: promote or expand into multiple
  - Vectors: widen or split (or scalarize)
- Legalize operations
  - E.g., conditional move, etc.
- Optimize DAG, e.g. some pattern matching, removing unneeded sign/zero extensions

```
llc -march=aarch64 -view-isel-dags
```

Note: needs LLVM debug build
LLVM SelectionDAG: ISelDAG to DAG

- Mainly pattern matching
- Simple patterns specified in TableGen
  - Matching/selection compiled into bytecode
    - SelectionDAGISel::SelectCodeCommon()
- Complex selections done in C++
- Scheduling: linearization of graph

`llc -march=aarch64 -view-sched-dags`
Note: needs LLVM debug build
Instruction Selection – Summary

- Instruction Selection: transform generic into arch-specific instructions
- Often focus on optimizing tiling costs
- Target instructions often more complex, e.g., multi-result
- Macro Expansion: simple, fast, but inefficient code
- Peephole optimization on sequences/trees to optimize
- Tree Covering: allows for better tiling of instructions
- DAG Covering: support for multi-res instrs., but \( \text{NP} \)-complete
- Graph Covering: mightiest, but also most complex, rarely used
Instruction Selection – Questions

- What is the (nowadays typical) input and output IR for ISel?
- Why is good instruction selection important for performance?
- Why is peephole optimization beneficial for nearly all ISel approaches?
- How can peephole opt. be done more effectively than on neighboring instrs.?
- What are options to transform an SSA-IR into data flow trees?
- Why is a greedy strategy not optimal for tree pattern matching?
- When is DAG covering beneficial over tree covering?
- Which ISel strategies does LLVM implement? Why?