

Code Generation for Data Processing

Lecture 8: Instruction Selection

Alexis Engelke

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

Winter 2025/26

Code Generation – Overview

- ▶ Instruction Selection
 - ▶ Map IR to assembly
 - ▶ Keep code shape and storage; change operations
- ▶ Instruction Scheduling
 - ▶ Optimize order to hide latencies and reduce register pressure
 - ▶ Keep operations unmodified
- ▶ 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 = add %8, %9
i8 %11 = trunc %10
i64 %12 = const 24
i64 %13 = add %7, %12
store %11, %13
```

```
i64 %10 = ADD %8, %9
STRB %10, [%7+24]
```

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 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., no program with better “performance” exists
 - ▶ Performance \approx instructions associated with specific costs
- ▶ Problem: optimal code generation is **undecidable**
- ▶ Alternative: optimal *tiling* of IR with machine code instructions
 - ▶ IR as dataflow graph, instr. tiles to optimally cover graph
 - ▶ \mathcal{NP} -complete⁴¹
 - ▶ Additional complication: many different ways to express same computation

⁴¹DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54. .

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

<code>%5 = add %1, 12345</code>	→	<code>%5a = movz 12345</code> <code>%5 = add %1, %5a</code>
<code>%6 = and %2, 7</code>	→	<code>%6 = and %2, 7</code>
<code>%7 = shl %5, %6</code>	→	<code>%7a = lsl %5, %6</code> <code>%7b = cmp %6, 64</code> <code>%7 = csel %7a, xzr, %7b, lo</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 sequences⁴²
- ▶ Originally: physical window over assembly code
 - ▶ Replace with more efficient instructions having same effects
 - ▶ Possibly with allocated registers
- ▶ Extension: do expansion before register allocation⁴³
 - ▶ Expand IR into Register Transfer Lists (RTL) with temporary registers
 - ▶ While *combining*, ensure that each RTL can be implemented as single instr.

⁴²WM McKeeman. "Peephole optimization". In: *CACM* 8.7 (1965), pp. 443–444. .

⁴³JW Davidson and CW Fraser. "Code selection through object code optimization". In: *TOPLAS* 6.4 (1984), pp. 505–526. .

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

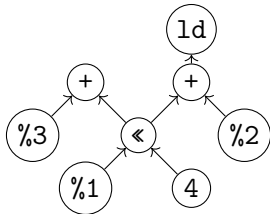
- ▶ Idea: represent program as data flow graph
- ▶ Tree: expression, comb. of single-use SSA instructions *(local ISel)*
- ▶ DAG: data flow in basic block, e.g. SSA block *(local ISel)*
- ▶ Graph: data flow of entire function, e.g. SSA function *(global ISel)*
- ▶ ISA “defines” *pattern set* of trees/DAGs/graphs for instrs.
- ▶ Cover data flow tree/DAG/graph with least-cost combination of patterns
 - ▶ Patterns in data flow graph may overlap
 - ▶ For non-global ISel: values used outside of block must be generated

Tree Covering: Converting SSA into Trees

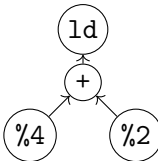
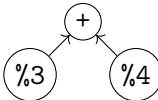
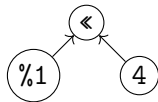
► SSA form:

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

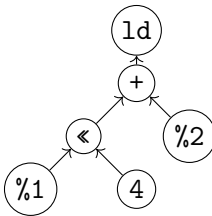
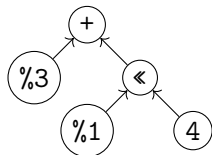
- ▶ Data flow graph:



- ▶ Method 1:
Edge Splitting



- ▶ Method 2:
Node Duplication



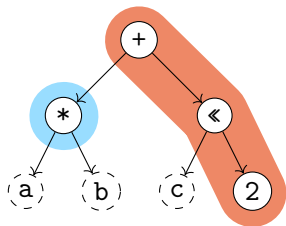
Tree Covering: Patterns

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

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:

- ▶ $+$: P_1 – cost 1 – covered nodes: 1
- ▶ $+$: P_2 – cost 2 – covered nodes: 3
- ▶ $+$: 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? Yes!⁴⁴
 - ▶ 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

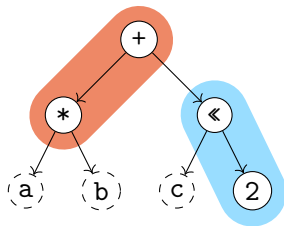
⁴⁴RS Glanville and SL Graham. "A new method for compiler code generation". In: *POPL*. 1978, pp. 231–254. 

Tree Covering: Dynamic Programming⁴⁵

- ▶ Step 1: compute cost matrix, bottom-up for all nodes
 - ▶ Matrix: tree node \times register bank
(different patterns might yield the same result in different register banks)
 - ▶ 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, follow rules in matrix
- ▶ Time linear w.r.t. tree size

⁴⁵AV Aho, M Ganapathi, and SWK Tjiang. "Code generation using tree matching and dynamic programming". In: *TOPLAS* 11.4 (1989), pp. 491–516. .

Tree Covering: Dynamic Programming – Example

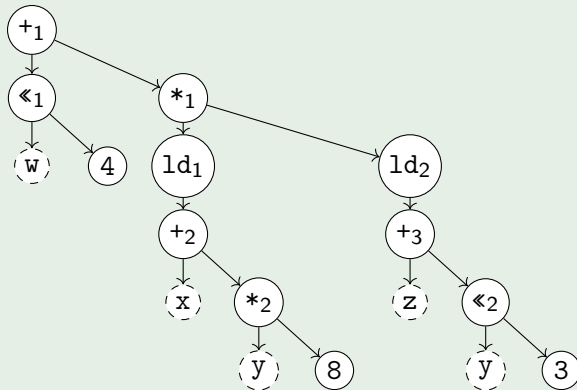


Node: +
Pattern: $P_9: GP \rightarrow +(* (GP, GP), GP)$
Pat. Cost: 3
Cost Sum: 4

		Node	+	*	«	2
GP	Cost		4	3	1	1
	Pattern		P_9	P_8	P_0	P_{10}

Tree Covering: Dynamic Programming – Example

Find a tree covering using the dynamic programming algorithm.



Tree Covering: Dynamic Programming – Off-line Analysis

- ▶ Cost analysis can actually be *precomputed*⁴⁶
- ▶ Idea: annotate each node with a state based on child states
- ▶ Lookup node label from precomputed table (one per register bank)
- ▶ Significantly improves compilation time
- ▶ But: Tables can be large, need to cover all possible (sub-)trees
- ▶ Variation: dynamically compute and cache state tables⁴⁷

⁴⁶A Balachandran, DM Dhamdhere, and S Biswas. "Efficient retargetable code generation using bottom-up tree pattern matching". In: *Computer Languages* 15.3 (1990), pp. 127–140.

⁴⁷MA Ertl, K Casey, and D Gregg. "Fast and flexible instruction selection with on-demand tree-parsing automata". In: *PLDI* 41.6 (2006), pp. 52–60.

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 \Rightarrow 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 ☹
- ▶ Least-cost covering is \mathcal{NP} -complete⁴⁸

⁴⁸DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: *CGO*. 2008, pp. 45–54. .

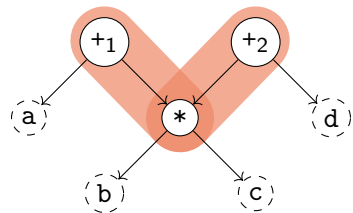
DAG Covering: Adapting Dynamic Programming I⁴⁹

- ▶ 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/regbank combinations: avoid redundant emit

+ Linear time

– Generally not optimal, only for specific grammars

DAG Covering: Adapting Dynamic Programming I – Example



Total cost: 6

madd %1, %b, %c, %a

madd %2, %b, %c, %d

Optimal cost: 5 \rightsquigarrow non-optimal result

		Node	+ ₂	+ ₁	*
GP	Cost		3	3	3
	Pattern		P_9	P_9	P_8

DAG Covering: Adapting Dynamic Programming II⁵⁰

- ▶ 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

⁵⁰DR Koes and SC Goldstein. “Near-optimal instruction selection on DAGs”. In: *CGO*. 2008, pp. 45–54. .

DAG Covering: ILP⁵¹

- ▶ 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

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

Minimize cost for all matched patterns s.t. every root has a match and every input of a match has a match.

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

⁵¹DR Koes and SC Goldstein. "Near-optimal instruction selection on DAGs". In: CGO. 2008, pp. 45–54. .

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.
- ▶ 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 \rightsquigarrow much slower code!

Instruction Selection in Practice

- ▶ Most compilers use some form of greedy tree/DAG pattern matching
- ▶ Later stages use peephole optimizations
 - ▶ Basically also tree/DAG matching on machine operations
- ▶ Distinction between tree/DAG/graph matching somewhat artificial⁵²

Problem in practice: implementing the huge amount of required patterns

- ▶ LLVM X86 back-end has 73k lines C++ for lowering + auto-generated patterns
- ▶ Needs lots of handling for corner cases, e.g. immediates
- ▶ Coming up with the patterns is often non-trivial

⁵²My personal opinion.

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 MIR Example

```
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 MIR Example

Analyze the Machine IR of the following code⁵⁵. (Also consult the reference⁵⁶.)

- ▶ What is the difference between physical and virtual registers?
- ▶ What do killed and implicit-def mean?
- ▶ How do branches and calls differ from LLVM-IR?

```
// clang --target=aarch64 -c -mllvm -stop-after=finalize-isel -O1 -o -  
// Also try -O0, -O2, -g, and --target=x86_64.  
[[clang::noinline]] int foo(int n) {  
    int r = 1;  
    while (n) { r *= n << n; n--; }  
    return r;  
}  
int bar(int n) { return foo(n) + n; }
```

⁵⁵<https://godbolt.org/z/zj9WM84Wh>

⁵⁶<https://llvm.org/docs/MIRLangRef.html>

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

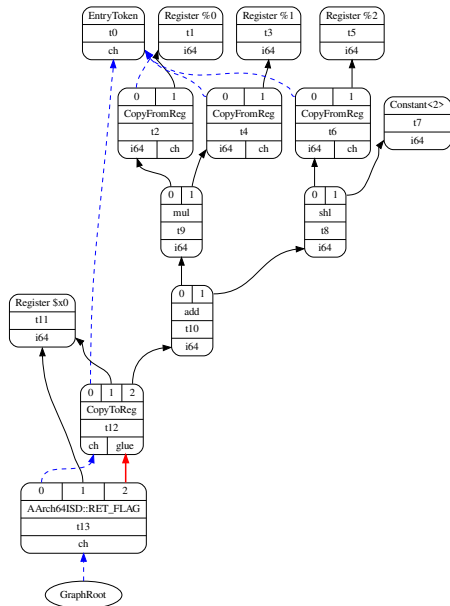
- ▶ Conv. to generic-MIR then legalize to MIR
- ▶ Reuses SD patterns
- ▶ Faster than SelDAG
- ▶ Few architectures
- ▶ Handles many cases, SelDAG-fallback
- ▶ Default AArch64 -O0

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



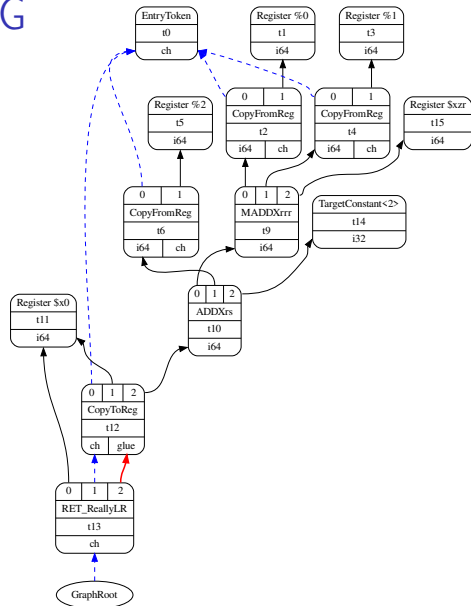
isel input for fn:

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



scheduler input for fn:

LLVM GlobalSel

- ▶ DAGs do have limitations w.r.t. instruction selection
- ▶ Graph loses original scheduling information (why problematic?)
- ▶ Building a separate IR is expensive

- ▶ Core idea of GlobalSel: do legalization and ISel in MachineIR
- ▶ MIR is target-specific \rightsquigarrow introduce generic operations
- ▶ Permit matching/combining instructions from different blocks

LLVM GlobalSel: Passes

- ▶ IRTranslator: translate LLVM-IR into gMIR
 - ▶ One-to-one mapping, except call and switch
- ▶ PreLegalizerCombiner
- ▶ Localizer: try to improve placement of constant materialization
- ▶ Legalizer: replace illegal instructions with legal gMIR instrs
- ▶ PostLegalizerCombiner
- ▶ RegBankSelect: assign register bank (reduce copies between banks)
- ▶ InstructionSelect

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/Graph Covering: support multi-res instrs., but \mathcal{NP} -complete
- ▶ Practical problems are very different from theory

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?
- ▶ What are advantages/disadvantages of SelDAG and GlobalISel?