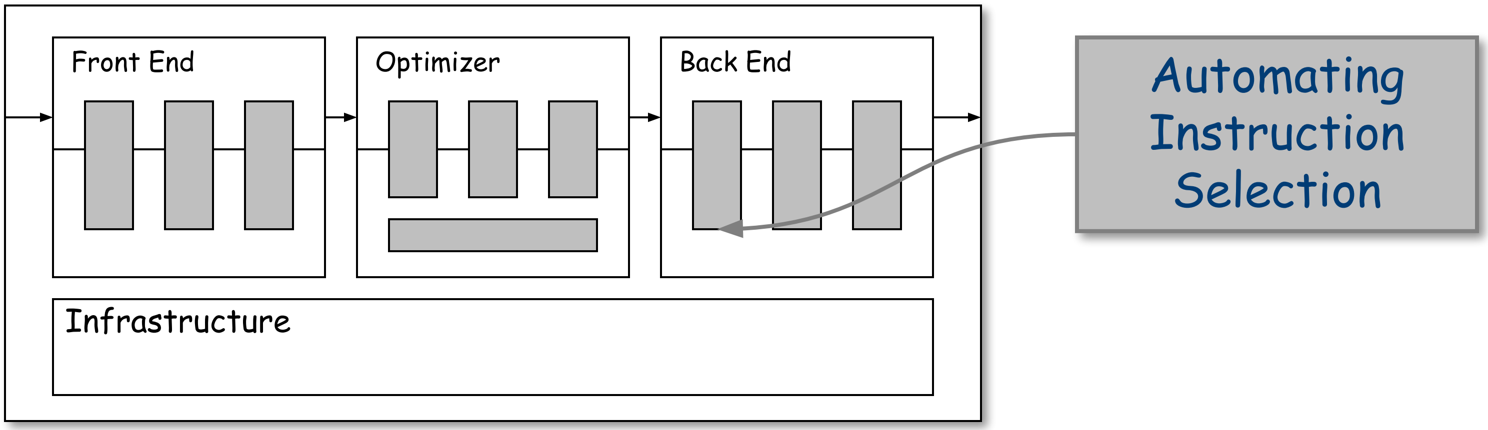
CENG513 Compiler Design and Construction
Instruction Selection

Note by Işıl ÖZ:
Our slides are adapted from Cooper and Torczon's slides that are prepared for COMP 412 at Rice.

The Problem

Writing a compiler is a lot of work

- Would like to reuse components whenever possible
- Would like to automate construction of components



- Front end construction is largely automated
- Middle is largely hand crafted
- (Parts of) back end can be automated

Definitions

Instruction selection

- Mapping <u>*IR*</u> into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

Instruction scheduling

- Reordering operations to hide latencies
- Assumes a fixed program *(set of operations)*
- Changes demand for registers

Register allocation

- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing
- Concerns about placement of data & memory operations

ILOC - Instruction Set Review

Linear assembly code for a simple abstract RISC machine

Typical ILOC instructions (EaC Appendix A)

load	r_1	$\Rightarrow r_2$	$r_2 = \text{Mem}[\ r_1\ ]$
loadI	c_1	$\Rightarrow r_1$	$r_1 = c_1$
loadAI	$r_1,\ c_1$	$\Rightarrow r_2$	$r_2 = \text{Mem}[\ r_1 + c_1\ ]$
loadAO	$r_1,\ r_2$	$\Rightarrow r_3$	$r_3 = \text{Mem}[\ r_1 + r_2\ ]$
store	r_1	$\Rightarrow r_2$	$\text{Mem}[\ r_2\ ] = r_1$
storeAI	r_1	$\Rightarrow r_2,\ c_1$	$\text{Mem}[\ r_2 + c_1\ ] = r_1$
storeAO	r_1	$\Rightarrow r_2,\ r_3$	$\text{Mem}[\ r_2 + r_3\ ] = r_1$
i2i	r_1	$\Rightarrow r_2$	$r_2 = r_1$
add	$r_1,\ r_2$	$\Rightarrow r_3$	$r_3 = r_1 + r_2$
addI	$r_1,\ c_1$	$\Rightarrow r_2$	$r_2 = r_1 + c_1$
	Similar for arithmetic, logical, and shifts		
jump		r_1	$\text{PC} = r_1$
jumpI		l_1	$\text{PC} = l_1$
cbr	r_1	$\Rightarrow l_1,\ l_2$	$\text{PC} = r_1\ ?\ l_1 : l_2$

The Problem

Modern computers (still) have many ways to do anything

Consider register-to-register copy in ILOC

- Obvious operation is `i2i` $r_i \Rightarrow r_j$
- Many others exist

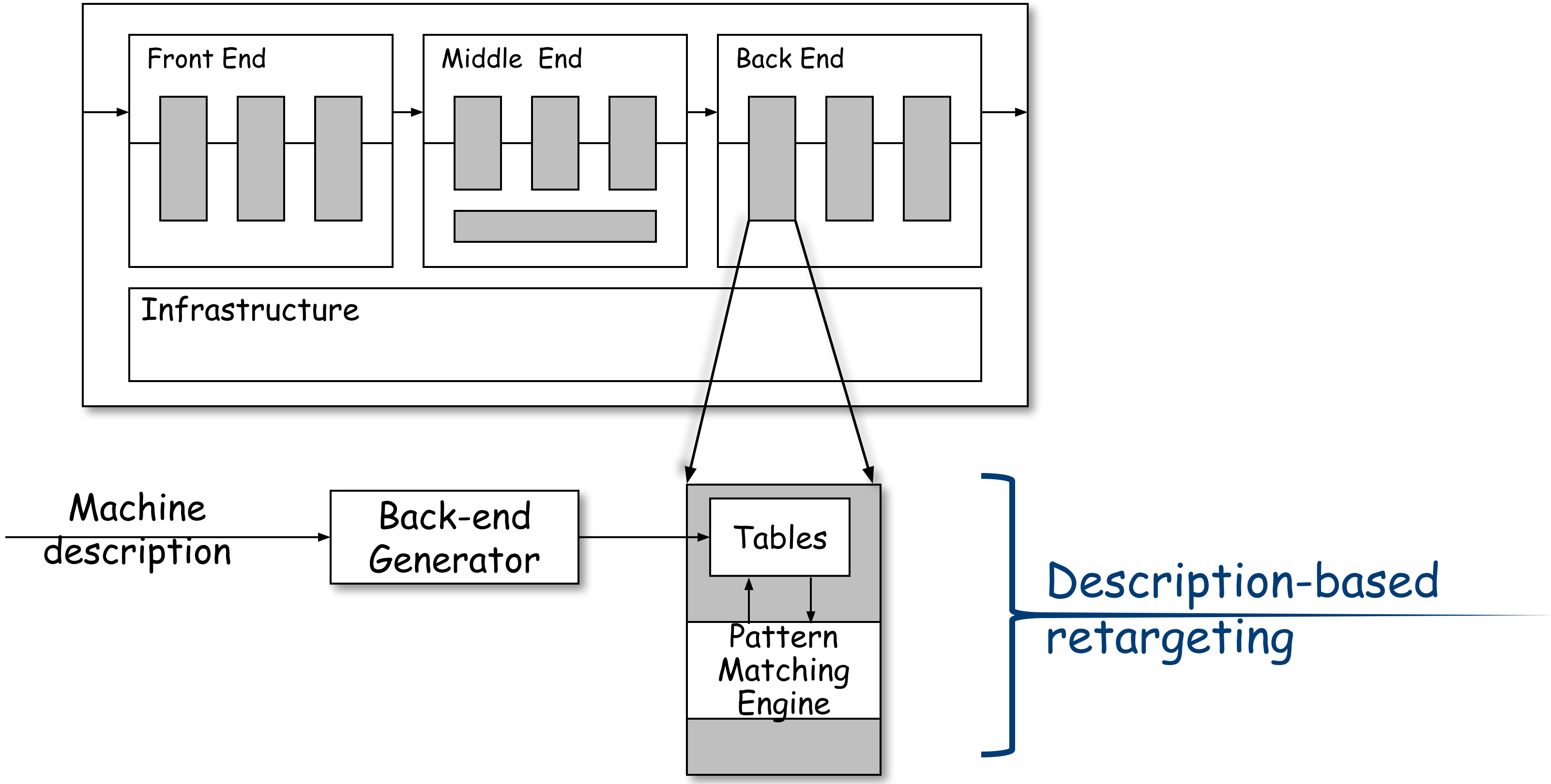
addI $r_i,0 \Rightarrow r_j$	subI $r_i,0 \Rightarrow r_j$	lshiftI $r_i,0 \Rightarrow r_j$
multI $r_i,1 \Rightarrow r_j$	divI $r_i,1 \Rightarrow r_j$	rshiftI $r_i,0 \Rightarrow r_j$
orI $r_i,0 \Rightarrow r_j$	xorI $r_i,0 \Rightarrow r_j$	*... and others ...*

- Human would ignore all of these
- Algorithm must look at all of them & find low-cost encoding
  - Take context into account (*busy functional unit?*)

And ILOC is an overly-simplified case

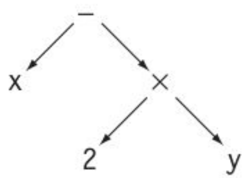
The Goal

Want to automate generation of instruction selectors



mapping from the IR to the target ISA

Simple Tree-Walk for Expressions



Abstract Syntax Tree for
x - 2 × y

```
loadI   @x         ⇒ r1
loadAO  rarp,r1    ⇒ r2
loadI   2          ⇒ r3
loadI   @y         ⇒ r4
loadAO  rarp, r4   ⇒ r5
mult    r3, r5     ⇒ r6
sub     r2, r6     ⇒ r7
```

Tree-Walk Code Generator

```
expr(node) {
  int result, t1, t2;
  switch(type(node)) {
    case ×, ÷, +, -:
     t1 ← expr(LeftChild(node));
     t2 ← expr(RightChild(node));
     result ← NextRegister( );
     emit(op(node), t1, t2, result);
     break;

    case IDENT:
     t1 ← base(node);
     t2 ← offset(node);
     result ← NextRegister( );
     emit(loadAO, t1, t2, result);
     break;

    case NUM:
     result ← NextRegister( );
     emit(loadI, val(node), none,
          result);
     break;
  }
  return result;
}
```

Simple Tree-Walk Routine for Variables and Numbers

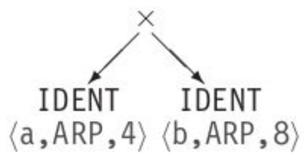
```
case IDENT:
  t1 ← base(node);
  t2 ← offset(node);
  result ← NextRegister();
  emit (loadA0, t1, t2, result);
  break;
```

```
case NUM:
  result ← NextRegister();
  emit (loadI, val(node),
        none, result);
  break;
```

emit (Op, src1, src2, dest)

Naive Selection - Tree Walk

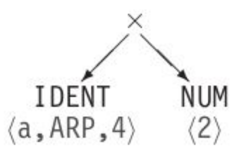
a × b



loadI 4 $\Rightarrow r_5$
loadAO $r_{arp}, r_5 \Rightarrow r_6$
loadI 8 $\Rightarrow r_7$
loadAO $r_{arp}, r_7 \Rightarrow r_8$
mult $r_6, r_8 \Rightarrow r_9$

loadAI $r_{arp}, 4 \Rightarrow r_5$
loadAI $r_{arp}, 8 \Rightarrow r_6$
mult $r_5, r_6 \Rightarrow r_7$

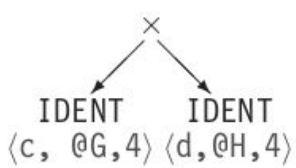
a × 2



loadI 4 $\Rightarrow r_5$
loadAO $r_{arp}, r_5 \Rightarrow r_6$
loadI 2 $\Rightarrow r_7$
mult $r_6, r_7 \Rightarrow r_8$

loadAI $r_{arp}, 4 \Rightarrow r_5$
multI $r_5, 2 \Rightarrow r_6$

c × d



loadI @G $\Rightarrow r_5$
loadI 4 $\Rightarrow r_6$
loadAO $r_5, r_6 \Rightarrow r_7$
loadI @H $\Rightarrow r_8$
loadI 4 $\Rightarrow r_9$
loadAO $r_8, r_9 \Rightarrow r_{10}$
mult $r_7, r_{10} \Rightarrow r_{11}$

loadI 4 $\Rightarrow r_5$
loadAI r_5, @G $\Rightarrow r_6$
loadAI r_5, @H $\Rightarrow r_7$
mult $r_6, r_7 \Rightarrow r_8$

Pattern Matching

Need pattern matching techniques to transform IR sequences to assembly sequences

- Must produce good code (*some metric for good*)
- Must run quickly

When the code generator considers multiple possible matches for a given sub-tree, it needs a way to choose among them

If the compiler writer can associate a cost with each pattern, then the matching scheme can select patterns in a way that minimizes the costs

Need to describe the target machine's ISA in a formal notation

- Tree pattern matching
- Peephole optimization

How do we perform this kind of matching ?

Tree-oriented IR suggests pattern matching on trees

- Process takes tree-patterns as input, matcher as output
- Each pattern maps to a target-machine instruction sequence
- Use dynamic programming or bottom-up rewrite systems

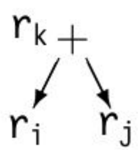
Linear IR suggests using some sort of string matching

- Process takes strings as input, matcher as output
- Each string maps to a target-machine instruction sequence
- Use text matching (Aho-Corasick) or peephole matching

In practice, both work well; matchers are quite different

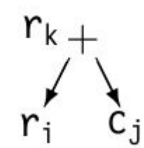
Tree Pattern Matching - Low-Level AST

Both the IR form of the program and the target machine's instruction set must be expressed as trees



$+(r_i, r_j)$

$\texttt{add}\ r_i, r_j \Rightarrow r_k$



$+(r_i, c_j)$

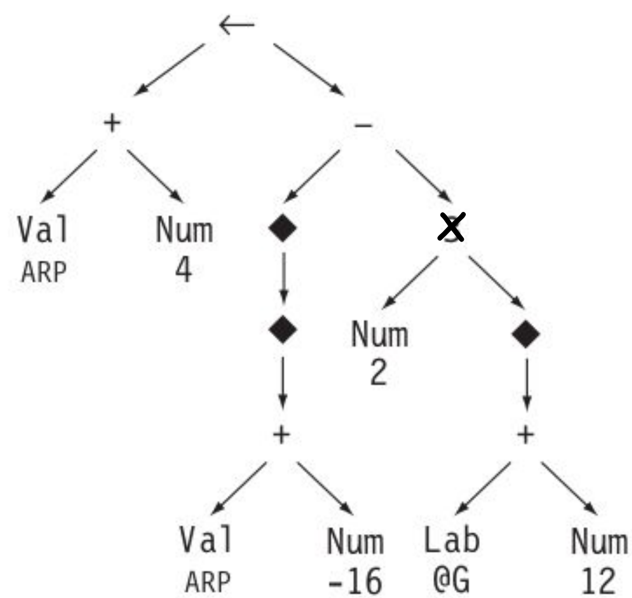
$\texttt{addI}\ r_i, c_j \Rightarrow r_k$

IR is in low level AST form exposing storage type of operands

Tile AST with operation trees

Recursively tile tree and bottom-up select the cheapest tiling

Low-Level AST for $w \leftarrow x - 2 \times y$



$\leftarrow(+(\mathrm{Val}_1,\mathrm{Num}_1),\ -(\blacklozenge(\blacklozenge(+(\mathrm{Val}_2,\mathrm{Num}_2))),\ \times(\mathrm{Num}_3,\ \blacklozenge(+(\mathrm{Lab}_1,\mathrm{Num}_4)))))$

Tree Pattern Matching - Rewrite Rules

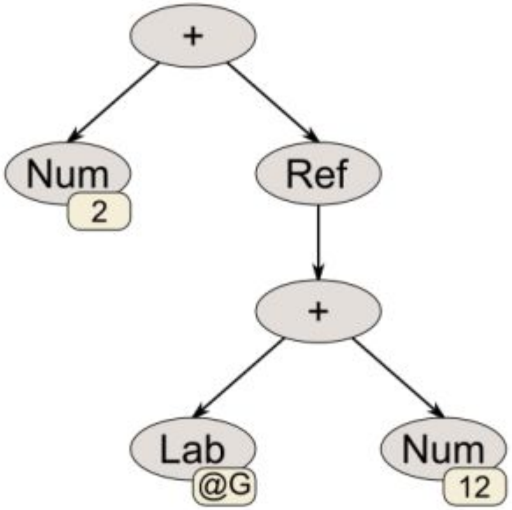
Operations are connected to AST subtrees by a set of ambiguous rewrite rules

Rules have costs - ambiguity allows cost based choice

Subset of rules

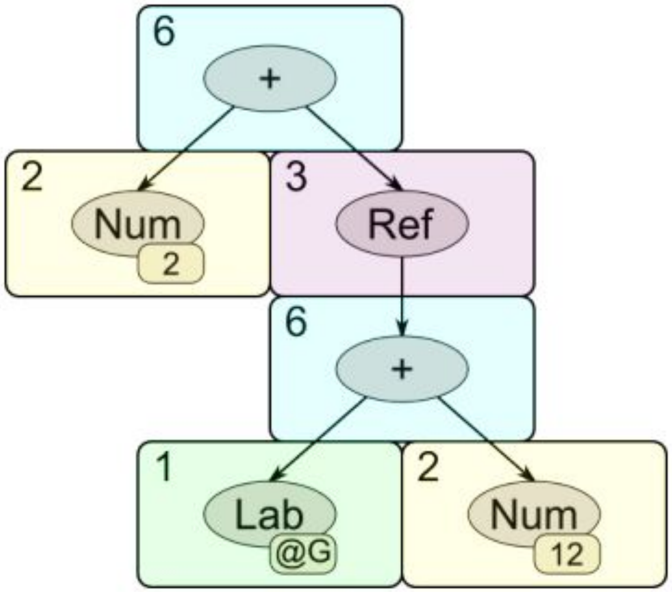
Id	Production	Code Template	
1:	$Reg \rightarrow Lab$	`loadI`	$lbl \Rightarrow r_{new}$
2:	$Reg \rightarrow Num$	`loadI`	$n_1 \Rightarrow r_{new}$
3:	$Reg \rightarrow Ref(Reg)$	`load`	$r_1 \Rightarrow r_{new}$
4:	$Reg \rightarrow Ref(+(Reg_1, Reg_2))$	`loadAO`	$r_1, r_2 \Rightarrow r_{new}$
5:	$Reg \rightarrow Ref(+(Reg, Num))$	`loadAI`	$r_1, n_1 \Rightarrow r_{new}$
6:	$Reg \rightarrow +(Reg_1, Reg_2))$	`add`	$r_1, r_2 \Rightarrow r_{new}$
7:	$Reg \rightarrow +(Reg, Num))$	`addI`	$r_1, n_1 \Rightarrow r_{new}$
8:	$Reg \rightarrow +(Num, Reg))$	`addI`	$r_1, n_1 \Rightarrow r_{new}$

Tree Pattern Matching - Tiling



Begin tiling AST bottom up

Tree Pattern Matching - Tiling

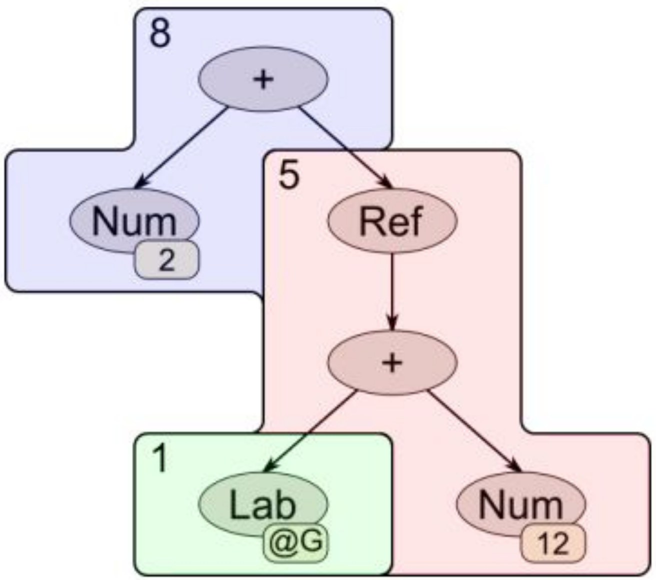


Code produced

loadI	@G	$\Rightarrow r_1$
loadI	12	$\Rightarrow r_2$
add	r_1, r_2	$\Rightarrow r_3$
load	r_3	$\Rightarrow r_4$
loadI	2	$\Rightarrow r_5$
add	r_4, r_5	$\Rightarrow r_6$

Bad tiling: productions used	
1: $Reg \rightarrow Lab$	loadI $lbl \Rightarrow r_{new}$
2: $Reg \rightarrow Num$	loadI $n_1 \Rightarrow r_{new}$
3: $Reg \rightarrow Ref(Reg)$	load $r_1 \Rightarrow r_{new}$
6: $Reg \rightarrow +(Reg_1, Reg_2))$	add $r_1, r_2 \Rightarrow r_{new}$

Tree Pattern Matching - Tiling



- Many different sequences available
- Selecting lowest cost bottom-up gives

Code produced

`loadI`	$@G$	$\Rightarrow r_1$
`loadAI`	$r_1, 12$	$\Rightarrow r_2$
`addI`	$r_2, 2$	$\Rightarrow r_3$

Good tiling: productions used

1: $Reg \rightarrow Lab$	`loadI` $lbl \Rightarrow r_{new}$
5: $Reg \rightarrow Ref(+(Reg, Num))$	`loadAI` $r_1, n_1 \Rightarrow r_{new}$
8: $Reg \rightarrow +(Num, Reg))$	`addI` $r_1, n_1 \Rightarrow r_{new}$

Tree Pattern Matching - Cost-Based Selection

- If, at each match, the code generator retains the lowest-cost matches, it will produce a locally optimal tiling
- This bottom-up accumulation of costs implements a dynamic-programming solution to find the minimal-cost tiling

- The cost function depends, inherently, on the target processor; it cannot be derived automatically from the grammar
- It must encode properties of the target machine and reflect the interactions that occur between operations in an assembly program—particularly the flow of values from one operation to another

- Examples assume all operations are equal cost, certain ops may be more expensive - divs

Peephole Matching

Basic idea

- Compiler can discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a "peephole"-sliding window- over code
  - Search for improvement
- Classic example was store followed by load

Original code	Improved code
storeAI $r_1 \Rightarrow r_0,8$	storeAI $r_1 \Rightarrow r_0,8$
loadAI $r_0,8 \Rightarrow r_{15}$	i2i $r_1 \Rightarrow r_{15}$

Peephole Matching

Basic idea

- Compiler can discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a "peephole"-sliding window- over code
  - Search for improvement
- Simple algebraic identities

Original code	Improved code
addI $r_2,0 \Rightarrow r_7$ mult $r_4,r_7 \Rightarrow r_{10}$	mult $r_4,r_2 \Rightarrow r_{10}$
multI $r_5,2 \Rightarrow r_7$	add $r_2,r_2 \Rightarrow r_7$

Peephole Matching

Basic idea

- Compiler can discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a "peephole"-sliding window- over code
  - Search for improvement
- Jump to a jump

Original code	Improved code
jumpI $\rightarrow L_{10}$ L_{10}: jumpI $\rightarrow L_{11}$	L_{10}: jumpI $\rightarrow L_{11}$

Must be within the window

Peephole Matching

Implementing it

- Early systems used limited set of hand-coded patterns
- Window size ensured quick processing

Modern peephole instruction selectors

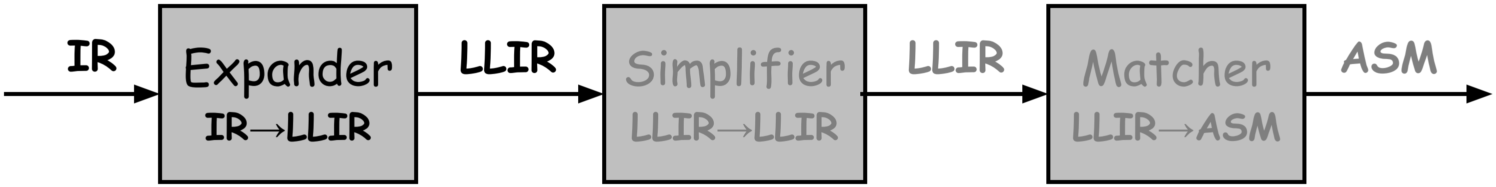
- Break problem into three tasks

- Apply symbolic interpretation & simplification systematically

Peephole Matching

Expander

- Turns IR code into a low-level IR (LLIR) such as RTL
- Operation-by-operation, template-driven rewriting
- LLIR form includes all direct effects of the operations
- Significant, albeit constant, expansion of size



Peephole Matching

Simplifier

- Looks at LLIR through <u>window</u> and rewrites it
- Uses forward substitution, algebraic simplification, local constant propagation, and dead-effect elimination
- Performs local optimization within window

- This is the heart of the peephole system
  - Benefit of peephole optimization shows up in this step

Peephole Matching

Matcher

- Compares simplified LLIR against a library of patterns
- Picks low-cost pattern that captures effects
- Must preserve LLIR effects, may add new ones
- Generates the assembly code output

Example

w = x - 2 * y *becomes*

Original IR Code

OP	Arg_1	Arg_2	Result
mult	2	y	t_1
sub	x	t_1	w

Symbolic names for memory-bound variables

Example

w = x - 2 * y *becomes*

Original IR Code

OP	Arg_1	Arg_2	Result
mult	2	y	t_1
sub	x	t_1	w

Symbolic names for memory-bound variables

Expand

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow MEM(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow MEM(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$MEM(r_{20}) \leftarrow r_{18}$

Example

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \mathrm{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \mathrm{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\mathrm{MEM}(r_{20}) \leftarrow r_{18}$

Simplify

LLIR Code

$r_{13} \leftarrow \mathrm{MEM}(r_0 + @y)$
$r_{14} \leftarrow 2 \times r_{13}$
$r_{17} \leftarrow \mathrm{MEM}(r_0 + @x)$
$r_{18} \leftarrow r_{17} - r_{14}$
$\mathrm{MEM}(r_0 + @w) \leftarrow r_{18}$

Example

LLIR Code		ILOC Code		
$r_{13} \leftarrow \text{MEM}(r_0 + @y)$	Match	loadAI	$r_0, @y$	$\Rightarrow r_{13}$
$r_{14} \leftarrow 2 \times r_{13}$		multI	$2 \times r_{13}$	$\Rightarrow r_{14}$
$r_{17} \leftarrow \text{MEM}(r_0 + @x)$		loadAI	$r_0, @x$	$\Rightarrow r_{17}$
$r_{18} \leftarrow r_{17} - r_{14}$		sub	$r_{17} - r_{14}$	$\Rightarrow r_{18}$
$\text{MEM}(r_0 + @w) \leftarrow r_{18}$		storeAI	r_{18}	$\Rightarrow r_0, @w$

- Introduced all memory operations & temporary names
- Turned out pretty good code

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$

$r_{11} \leftarrow @y$

$r_{12} \leftarrow r_0 + r_{11}$

$r_{13} \leftarrow \text{MEM}(r_{12})$

$r_{14} \leftarrow r_{10} \times r_{13}$

$r_{15} \leftarrow @x$

$r_{16} \leftarrow r_0 + r_{15}$

$r_{17} \leftarrow \text{MEM}(r_{16})$

$r_{18} \leftarrow r_{17} - r_{14}$

$r_{19} \leftarrow @w$

$r_{20} \leftarrow r_0 + r_{19}$

$\text{MEM}(r_{20}) \leftarrow r_{18}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \mathrm{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \mathrm{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\mathrm{MEM}(r_{20}) \leftarrow r_{18}$

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$

$\Rightarrow$

$r_{10} \leftarrow 2$
$r_{12} \leftarrow r_0 + @y$
$r_{13} \leftarrow \text{MEM}(r_{12})$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{10} \leftarrow 2$
$r_{12} \leftarrow r_0 + @y$
$r_{13} \leftarrow \text{MEM}(r_{12})$

$\Rightarrow$

$r_{10} \leftarrow 2$
$r_{13} \leftarrow \text{MEM}(r_0 + @y)$
$r_{14} \leftarrow r_{10} \times r_{13}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{10} \leftarrow 2$
$r_{13} \leftarrow \text{MEM}(r_0 + @y)$
$r_{14} \leftarrow r_{10} \times r_{13}$

$\Rightarrow$

$r_{13} \leftarrow \text{MEM}(r_0 + @y)$
$r_{14} \leftarrow 2 \times r_{13}$
$r_{15} \leftarrow @x$

Folding 2 into computation of r_{14} made the 1st op *dead*.

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{13} \leftarrow \text{MEM}(r_0 + @y)$
$r_{14} \leftarrow 2 \times r_{13}$
$r_{15} \leftarrow @x$

$\Rightarrow$

$r_{14} \leftarrow 2 \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$

Simplifier emits ops that are *live* when they roll out of the window.

Steps of the Simplifier *(3-operation window)*

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$$\boxed{\begin{array}{l} r_{14} \leftarrow 2 \times r_{13} \\ r_{15} \leftarrow @x \\ r_{16} \leftarrow r_0 + r_{15} \end{array}} \Rightarrow \boxed{\begin{array}{l} r_{14} \leftarrow 2 \times r_{13} \\ r_{16} \leftarrow r_0 + @x \\ r_{17} \leftarrow \text{MEM}(r_{16}) \end{array}}$$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{14} \leftarrow 2 \times r_{13}$
$r_{16} \leftarrow r_0 + @x$
$r_{17} \leftarrow \text{MEM}(r_{16})$

$\Rightarrow$

$r_{14} \leftarrow 2 \times r_{13}$
$r_{17} \leftarrow \text{MEM}(r_0 + @x)$
$r_{18} \leftarrow r_{17} - r_{14}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{14} \leftarrow 2 \times r_{13}$
$r_{17} \leftarrow \text{MEM}(r_0+@x)$
$r_{18} \leftarrow r_{17} - r_{14}$

$\Rightarrow$

$r_{17} \leftarrow \text{MEM}(r_0+@x)$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{17} \leftarrow \text{MEM}(r_0+@x)$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$

$\Rightarrow$

$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$

$\Rightarrow$

$r_{18} \leftarrow r_{17} - r_{14}$
$r_{20} \leftarrow r_0 + @w$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

Steps of the Simplifier (*3-operation window*)

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{18} \leftarrow r_{17} - r_{14}$
$r_{20} \leftarrow r_0 + @w$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

$r_{18} \leftarrow r_{17} - r_{14}$
$\text{MEM}(r_0 + @w) \leftarrow r_{18}$

Example

LLIR Code

$r_{10} \leftarrow 2$
$r_{11} \leftarrow @y$
$r_{12} \leftarrow r_0 + r_{11}$
$r_{13} \leftarrow \text{MEM}(r_{12})$
$r_{14} \leftarrow r_{10} \times r_{13}$
$r_{15} \leftarrow @x$
$r_{16} \leftarrow r_0 + r_{15}$
$r_{17} \leftarrow \text{MEM}(r_{16})$
$r_{18} \leftarrow r_{17} - r_{14}$
$r_{19} \leftarrow @w$
$r_{20} \leftarrow r_0 + r_{19}$
$\text{MEM}(r_{20}) \leftarrow r_{18}$

Simplify

LLIR Code

$r_{13} \leftarrow \text{MEM}(r_0 + @y)$
$r_{14} \leftarrow 2 \times r_{13}$
$r_{17} \leftarrow \text{MEM}(r_0 + @x)$
$r_{18} \leftarrow r_{17} - r_{14}$
$\text{MEM}(r_0 + @w) \leftarrow r_{18}$

References

Chapter sections from the book:

- 11.1-11.4, Appendix A

Selected videos from compiler course from California State University:

- https://www.youtube.com/watch?v=jKN_kjtb128&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=25