### 3 Main Tasks in Code Generation

- **Instruction Selection**
  - Choose set of instructions equivalent to IR code
  - Minimize (locally) execution time, # used registers, code size
  - Example: `INC RM #4(fp)` vs. `LOAD #4(fp), R1
ADD R1, #1, R1
STORE R1, #4(fp)`

- **Instruction Scheduling**
  - Reorder instructions to better utilize processor architecture
  - Minimize temporary space (#registers, #stack locations) used, execution time, or energy consumption

- **Register Allocation**
  - Keep frequently used values in registers (limited resource)
  - Some registers are reserved, e.g., sp, fp, pc, sr, retval...
  - Minimize #loads and #stores (which are expensive instructions!)

### Machine model (here: a simple register machine)

- **Register set**
  - E.g., 32 general-purpose registers R0, R1, R2...
  - Some of them reserved (sp, fp, pc, sr, retval, par1, par2...)

- **Instruction set** with different addressing modes
  - Cost (usually, time / latency) depends on the operation and the addressing mode

### Some Code Generation Algorithms

- **Macro-expansion of IR operations (quadruples)**
- "Simple code generation algorithm" (ALSU2e Section 8.6)
- Trade-off:
  - Registers vs. memory locations for temporaries
  - Sequencing
  - Code generation for expression trees
    - Labeling algorithm [Ershov 1958] [Sethi, Ullman 1970]
    - (see later)
  - Code generation using pattern matching
    - For trees: Aho, Johnson 1976 (dynamic programming),
      Graham/Glanville 1978 (LR parsing),
      Fraser/Hanson/Proebsting 1992 (IBURG tool), ...
    - For DAGs: [Ertl 1999], [K., Bednarski 2006] (DP, ILP)

### Towards code generation by pattern matching

- Example: Data flow graph (expression tree) for `i = c + 4`
- In LCC-IR (DAGs of quadruples) [Fraser/Hanson’95]
  - i, c: local variables

- In quadruple form:
  - (Convention: last letter of opcode gives result type: i=int, c=char, p-pointer)
  - For `i = c + 4`
    - `(ADDRLP, i, 0, 11)` // `i <- i + 4`
    - `(ADDRLP, c, 0, 12)` // `c <- c + 4`
    - `(INDIRC, 12, 0, 13)` // `i <- i + 4`
    - `(CVCI, 12, 0, 14)` // convert char to int
    - `(CNSTI, 4, 0, 15)` // create int const
    - `(ADDI, 14, 15, 16)` // `i <- i + 4`
    - `(ASGIN, 16, 11)` // `i <- i + 4`

### Macro expansion of quadruples

- Each quadruple is translated to a sequence of one or several target instructions that performs the same operation.

  - **very simple**
  - **bad code quality**
    - Cannot utilize powerful instructions/addressing modes that do the job of several quadruples in one step
    - Poor use of registers

- Simple code generation algorithm, see TDDD44/TDDD16 ([ALSU2e] 8.6)
Recall: Macro Expansion

- For the example tree:
  - $s1, s2, s3...$: “symbolic” registers (allocated but not assigned yet)
  - Target processor has delayed load (1 delay slot)

```c
{ int i; char c; i+=e+4; }
```

Using tree pattern matching...

- Utilizing the available addressing modes of the target processor, 3 instructions and only 2 registers are sufficient to cover the entire tree:

```c
{ int i; char c; i+=e+4; }
```

Tree patterns vs. Complex patterns

- Complex patterns
  - Forest patterns (several pattern roots)
  - DAG patterns (common subexpressions in pattern)

```c
       *       *
       |       |
       |       |
Tree pattern (Multiply-add)  Forest pattern (SIMD instruction)  DAG pattern (Memory incr)
```

No match of `mad` (pending use of `MUL`)

Code generation by pattern matching

- Powerful target instructions/addressing modes may cover the effect of several quadruples in one step.

```
For each instruction and addressing mode,
define a pattern that describes its behavior in terms of quadruples resp. data-flow graph nodes and edges
(usually limited to tree fragment shapes: tree pattern).
```

- A pattern matches at a node $v$ if pattern nodes, pattern operators and pattern edges coincide with a tree fragment rooted at $v$

- Each instruction (tree pattern) is associated with a cost, e.g. its time behavior or space requirements

- Optimization problem: Cover the entire data flow graph (expression tree) with matching tree patterns such that each node is covered exactly once, and the accumulated cost of all covering patterns is minimal.

Tree grammar (Machine grammar)

The target processor is described by a tree grammar $G = (N, T, s, P)$

```
Nonterminals $N = \{ \text{stmt, reg, con, addr, mem, ...} \}$ (start symbol is stmt)
```

Production rules $P$

<table>
<thead>
<tr>
<th>Production</th>
<th>Target instruction for pattern</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg $\rightarrow$ reg (reg)</td>
<td>$\text{add } k, k, k$</td>
<td>1</td>
</tr>
<tr>
<td>smt $\rightarrow$ ASGN (reg, reg)</td>
<td>$\text{store } k, k$</td>
<td>0</td>
</tr>
<tr>
<td>reg $\rightarrow$ ADDRLP (reg, reg)</td>
<td>$\text{add } f, k, k$</td>
<td>1</td>
</tr>
<tr>
<td>reg $\rightarrow$ addr</td>
<td>$\text{add } k, k$</td>
<td>1</td>
</tr>
<tr>
<td>reg $\rightarrow$ INDIRC (addr)</td>
<td>$\text{load } k, k$, nop</td>
<td>2</td>
</tr>
<tr>
<td>reg $\rightarrow$ ADDRP (addr)</td>
<td>$\text{add } k, k$</td>
<td>1</td>
</tr>
<tr>
<td>reg $\rightarrow$ CONTI</td>
<td>$\text{add } k, k$</td>
<td>1</td>
</tr>
</tbody>
</table>

Tree Grammar / Machine Grammar

Formally, we specify for each target machine a tree grammar $G = (N, T, s, P)$

```
$N$ = set of nonterminals
  representing value / storage classes: addr, reg, const, mem, ...
$T$ = set of terminals
  “leaf” LIR operators: CNST, ADDRLP, ...
$P$ = list of production rules
```

For each production rule $lhs \rightarrow rhs; i, c$

- $i$: production rule for each possible target instr. + addr-mode
- $c$: cost of this production (not including subtree costs)

$s$: start symbol
  nonterminal representing a statement
Derivation of the expression tree

Here: Top-down derivation

Derivation using a LR parser (bottom-up derivation)

Some methods for tree pattern matching

- Use a LR-parser for matching [Graham, Glanville 1978]
  - compact specification of the target machine using a context-free grammar ("machine grammar")
  - quick matching
  - not total-cost aware (greedy local choices at reduce decisions → suboptimal)
- Combine tree pattern matching with dynamic programming for total cost minimization [Aho, Ganapathi, Tjiang '89] [Fraser, Hanson, Proebsting '92]
  - A LR parser is stronger than what is really necessary for matching tree patterns in a tree.
  - Right machine model is a tree automaton = a finite automaton operating on input trees rather than flat strings [Ferdinand, Seidl, Wilhelm '92]
  - By Integer Linear Programming [Wilson et al. '94] [K., Bednarski '06]

Example: IBURG

Phase 1: bottom-up labeller

- annotations each node v of the input tree with
  - the set of tree patterns that match v and their accumulated costs;
  - if multiple rules apply, pick one with locally minimum cost for each lhs nonterminal;
- Apply chain rules nonterm1 → nonterm2 as far as possible.

Tree Pattern Matching by Dynamic Programming (1)

- Derivation is not unique
  - find a least-cost derivation of the LIR tree
- cost of a derivation = sum over costs of productions applied
- A greedy approach is not sufficient
  - initially cheap derivations may later turn out to be expensive
  - naive: backtracking (= enumerate all possible coverings)
  - fast: dynamic programming [Aho/Johnson '76]
- bottom-up rewrite machine (BURM), for code generator generators:
  - TWIG [Aho/Ganapathi/Tjiang '89],
  - BURG [Fraser/Henry/Proebsting '92],
  - IBURG [Fraser/Hanson/Proebsting '92] [Fraser/Hanson '95]
Example: IBURG

Phase 2: Top-down reducer
- root of the labeled tree must correspond to start symbol (stmt)
- choose best production for root node (accumulated costs),
  apply the corresponding productions,
- and do this recursively for each nonterminal in the rhs term

```
Example: IBURG
```

Found least-cost derivation:
Example: IBURG

Phase 3: Emitter
- in reverse order of the derivation found in phase 2:
  - emit the assembler code for each production applied
  - execute additional compiler code associated with these rules
    - e.g. register allocation.

Example: IBURG

Emitter result:

Example: IBURG

Given: a tree grammar describing the target processor
1. parse the tree grammar
2. generate:
   - bottom-up labeller,
   - top-down reducer,
   - emitter automaton
→ retargetable code generation!

Complexity of Tree Pattern Matching
- NP-complete if associativity / commutativity included
- Naive: time $O(\# \text{tree patterns} \times \text{size of input tree})$
- Preprocessing initial tree patterns
  - may require exponential space / time
  - but then tree pattern matching in time $O(\text{size of input tree})$
- Theory of (non)deterministic tree automata
  - [Ferdinand/Seidl/Wilhelm'92]

Instruction selection for DAGs

Computing a minimal cost covering (with tree patterns) for DAGs?
- NP-complete [Proebsting'98]
  - For common subexpressions, only one of possibly several possible coverings can be realized.
- Dynamic programming algorithm for trees OK as heuristic for regular processor architectures
- The algorithm for trees may create optimal results for DAGs for special tree grammars (usually for regular register sets).
  - This can be tested a priori! [Ertl POPL'99]

Complex Patterns (1)
- Several roots possible
- Common subexpressions possible:
  - SIMD instructions
  - DIVU instruction on Motorola 68K (simultaneous div + mod)
  - Read/Modify/Write instructions on IA32
  - Autocrement / autodecrement memory access instructions
- Min-cost covering of a DAG with complex patterns?
  - Can be formulated as PBQP instance [Scholz,Eckstein '03] (partitioned boolean quadratic programming)
  - Or as ILP (integer linear programming) instance
- Caution: Risk of creating artificial dependence cycles!
Complex Patterns (2)

**Caution:** Risk of creating artificial dependence cycles!

**Example** [Ebner 2009]:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Resulting Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>p := r+4;</td>
<td>ADD</td>
<td>p' = p + 4</td>
</tr>
<tr>
<td>q := p+4;</td>
<td>ADD</td>
<td>q' = q + 4</td>
</tr>
<tr>
<td>r := q+4;</td>
<td>ADD</td>
<td>r' = r + 4</td>
</tr>
</tbody>
</table>

Use postdecr. store instruction:

- Cycle between resulting instructions → No longer schedulable
- Solution [Ebner 2009]: Add constraints to guarantee schedulability (some topological order exists)

Interferences with instruction scheduling and register allocation

- The cost attribute of a production is only a rough estimate
  - E.g., best-case latency or occupation time
- The actual impact on execution time is only known for a given scheduling situation:
  - currently free functional units
  - other instructions that may be executed simultaneously
  - latency constraints due to previously scheduled instructions
  - Integration with instruction scheduling would be great!!
- Mutations with different unit usage may be considered:
  - $a = 2^b \text{ equivalent to } a = b \ll 1 \text{ and } a = b + b \text{ (integer)}$
- Different instruction selections may result in different register need.

Retargetable Compilers

- Variant 1: Use a Code Generator Generator
- Variant 2: Parameterizable Code Generator

Excerpt from an OLIVE tree grammar

```plaintext
%term AND  // declare terminal AND
%declare-char > reg;  // declare nonterminal reg, whose
                       // action function returns a string
reg: AND ( reg, reg )  // rule for a bitwise AND instruction
  { $cost[0] = 1 + $cost[2] + $cost[3];  // cost = 1 plus cost of subtrees
    char "v1", "v2", "v3";  // local variables in action function
    vr1 = $action[2];      // get virtual register name for argument 1
    vr2 = $action[3];      // get virtual register name for argument 2
    vr3 = NewVirtualName();  // get virtual register name for destination
    printf(" AND %s, %s, %s", vr1, vr2, vr3);  // emit assembler instruction
    return strdup(vr3);  // pass a copy of destination name upwards in tree
  }
```

Some Literature on Instruction Selection