Compiler Construction

Instruction Selection

University of Freiburg

Peter Thiemann, *Matthias Keil*

University of Freiburg

12. Dezember 2016

Naive approach

- Macro-expand each IR node into machine instructions
- Independent expansion requires invariants between nodes (e.g., all intermediate results in registers)

⇒ poor code quality

Instruction Selection (cont'd)

Tree matching approach

- Each instruction is associated with a tree pattern ("tile")
- Covering of the IR tree with tiles yields an instruction sequence

Alternative approach

- Model target machine state as IR is expanded (interpretive code generation)

Registers and Temporaries

Temporaries

- Pseudo registers introduced for intermediate values
- Expected to be mapped to registers
- May be *spilled* to stack frame if not enough registers

Register allocation

- Task: Assign processor registers to temporaries
- Limited number of processor registers
- Register allocator
  - chooses temporaries to spill
  - inserts code to spill/restore as needed
  - generates mapping

Tree Patterns

- Express each machine instruction as a *tree pattern* (a fragment of an IR tree with associated cost)
- A tree pattern may contain zero or more *wildcards*, which match all IR trees
- Instruction selection amounts to tiling the IR tree with the patterns available
- The root of a tile matches either the root of the IR tree or the node in a wildcard of another tile
- Cost of the tiling = sum of cost of all tiles

Optimal vs Optimum Tiling

Optimal Tiling

No two adjacent tiles can be replaced by a larger tile of lower cost.

Optimum Tiling

The total cost of the tiling is minimal among all possible tilings.

- Tiling is optimum ⇒ tiling is optimal

Implementation of Optimal Tiling

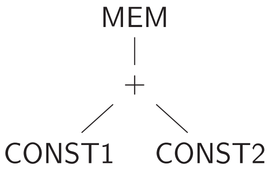
Maximal Munch Algorithm (Top Down)

```
// tiles ordered from largest to smallest cost
List<Pattern> tiles;

Temp munchExpr (Tree.Exp e) {
  foreach (p : tiles)
    if (matches(p, e)) { pattern = p; break; }

  // wildcard(pattern, e) returns the list of
  // subexpressions of e matched to wildcards
  foreach (e_i : wildcard (pattern, e))
    recursively invoke temp_i = munchExpr (e_i)

  emit INS using temp_i as arguments
         putting result into new temp_0

  return temp_0
}
```

Optimum Tiling

Example

University of Freiburg

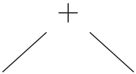
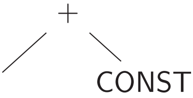
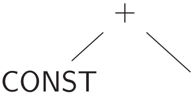


pattern	instr	tile cost	wildcard cost	total cost
CONST	ADDI	1	0	1

Optimum Tiling

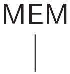
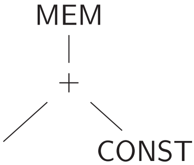
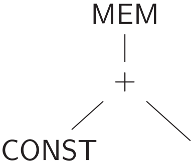
Example (cont'd)

University of Freiburg

pattern	instr	tile cost	wildcard cost	total cost
	ADD	1	1+1	3
	ADDI	1	1	2
	ADDI	1	1	2

Optimum Tiling

Example (cont'd)

pattern	instr	tile cost	wildcard cost	total cost
 MEM	LOAD	1	2	3
 MEM + CONST	LOAD	1	1	2
 MEM + CONST	ADDI	1	1	2

Optimum Tiling

Emitted Code

$$\begin{array}{llll}
\mathrm{ADDI} & r_1 & \leftarrow & r_0 + 1 \\
\mathrm{LOAD} & r_1 & \leftarrow & M[r_1 + 2]
\end{array}$$

Implementation of Optimum Tiling

Dynamic Programming (Bottom Up)

University of Freiburg

```
void matchExpr (Tree.Exp e) {
  for (Tree.Exp kid : e.kids())
    matchExpr (kid);

  cost = INFINITY
  for each pattern P_i
    if (P_i.matches (e)) {
      cost_i = cost(P_i)
             + sum ((wildcard (P_i, e)).mincost)
      if (cost_i < cost) { cost = cost_i; choice = i; }
    }
  e.matched = P_{choice}
  e.mincost = cost
}
```

Implementation of Optimum Tiling

Collecting the Match (Top Down)

```
Temp emission (Tree.Exp e) {
  foreach (e_i : wildcard (e.matched, e)) {
    temp_i = emission (e_i)
  }

  emit INS using temp_i as arguments
         putting result into new temp_0

  return temp_0
}
```

Implementation of Pattern Matching

- Additional side conditions (e.g., size of constants, special constants)
- Matching of patterns can be done with a decision tree that avoids checking the same node twice
- The bottom up matcher can remember partial matches and avoid rechecking the same nodes

⇒ tree automata

Tree Automata

A *bottom-up tree automaton* is $\mathcal{M} = (Q, \Sigma, \delta, F)$ where

- Q is a finite set of states
- Σ a ranked alphabet (the tree constructors)
- $\delta \subseteq \Sigma^{(n)} \times Q^n \times Q$ $(\forall n)$ the transition relation
- $F \subseteq Q$ the set of final states

$\mathcal{M}$ is deterministic if δ is a function.

Define $\Rightarrow$ on $T_{\Sigma+Q}$ (the set of terms/trees where nodes are labels with symbols from Σ or Q) by

$$t[\sigma(q_1, \dots, q_n)] \Rightarrow t[q_0] \qquad \text{if} \qquad \begin{array}{l}(\sigma, q_1, \dots, q_n, q_0) \in \delta \\ \wedge\, \sigma \in \Sigma^{(n)}\end{array}$$

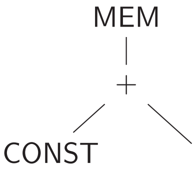
$t \in L(\mathcal{M})$ if $t \Rightarrow^* q$ with $q \in F$

Tree Automata

Example

University of Freiburg

Tree automaton for



- $Q = \{q_t, q_c, q_a, q_m\}$
- $F = \{q_m\}$
- $\delta =$

Σ	q_1	q_2	q_{out}
CONST			q_c
TEMP			q_t
$+$	q_c	q_t	q_a
MEM	q_a		q_m

Optimum Tiling with Tree Automata

- Generate a bu tree automaton for each pattern
- Simulate them in parallel on expression tree
- At each node
  - determine all patterns whose root matches the current node
  - compute their cost and mark the node with the minimum cost pattern
- There are tools to compile a pattern specification to such an automaton ⇒ BURG (Fraser, Hanson, Proebsting)

Tree Grammars

- Tree patterns assume that the result of an IR tree is always used in the same way
- Some architectures habe different kinds of registers that obey different restrictions
- Extension: introduce a different set of patterns for each kind of register
- Example: M680x0 distinguishes data and address registers, only the latter may serve for address calculations and indirect addressing

⇒ *Tree grammar* needed

A *context-free tree grammar* is defined by $\mathcal{G} = (N, \Sigma, P, S)$ where

- N is a finite set of non-terminals
- Σ is a ranked alphabet
- $S \in N$ is the start symbol
- $P \subseteq N \times T_{\Sigma+N}$

Define $\Rightarrow$ on $T_{\Sigma+N}$ by

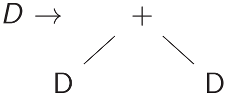
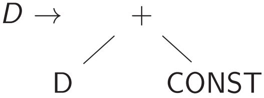
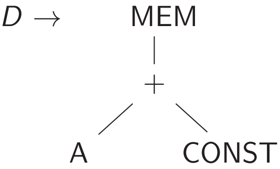
$$t[A] \Rightarrow t[r] \qquad \text{in} \qquad A \to r \in P$$

$t \in L(\mathcal{G})$ if $S \Rightarrow^* t \in T_\Sigma$

Tree Grammars

Example: The Schizo-Jouette Architecture (Excerpt)

University of Freiburg

Instruction	Effect	Pattern
ADD	$d_i \leftarrow d_j + d_k$	
ADDI	$d_i \leftarrow d_j + c$	
MOVEA	$d_i \leftarrow a_j$	$D \rightarrow A$
MOVED	$a_i \leftarrow d_j$	$A \rightarrow D$
LOAD	$d_i \leftarrow M[a_j + c]$	

Efficiency of Tiling

University of Freiburg

- N number of nodes in input tree
- T number of patterns
- K average number of labeled nodes in pattern
- K' maximum number of nodes to check for a match
- T' average number of patterns that match at each node
- **Maximal munch.** Each match consumes K nodes: test for matches at N/K nodes. At each candidate node, choose pattern with $K' + T'$ tests.
  $(K' + T')N/K$ steps on average. Worst case: $K = 1$.
- **Dynamic programming.** Tests every pattern at every node: $(K' + T')N$.

⇒ same linear worst-case complexity. $(K' + T')/K$ is constant, anyway.

CISC vs RISC

Challenges for Instruction Selection and Register Allocation

RISC	CISC
32 registers	few registers (16, 8, 6)
one class of registers	different classes with restricted operations
ALU instructions only between registers	ALU operations with memory operands
three-adress instructions $r_1 \leftarrow r_2 \oplus r_3$	two-address instructions $r_1 \leftarrow r_1 \oplus r_2$
one addressing mode for load/store	several addressing modes
every instruction 32 bits long	different instruction lengths
one result / instruction	instructions w/ side effects

CISC Examples

Pentium / x86 (32-bit)

- six GPR, sp, bp
- multiply / divide only on eax
- generally two-address instructions

MC 680x0 (32-bit)

- 8 data registers, 7 address registers, 2 stack registers
- ALU operations generally on data registers, indirect addressing only through address registers
- generally two-address instructions
- esoteric addressing modes (68030)

Challenges

University of Freiburg

- **[Few Registers]** generate temporaries and rely on register allocation
- **[Restricted Registers]** generate extra moves and hope that register allocation can get rid of them. Example:
    - Multiply on Pentium requires one operand and destination in `eax`
    - Most-significant word of result stored to `edx`

Hence for $t_1 \leftarrow t_2 \cdot t_3$ generate

`mov eax,`t_2	$\mathtt{eax} \leftarrow t_2$
`mul` t_3	$\mathtt{eax} \leftarrow \mathtt{eax} \cdot t_3$; $\mathtt{edx} \leftarrow \textit{garbage}$
`mov` t_1`, eax`	$t_3 \leftarrow \mathtt{eax}$

Challenges II

- **[Two-address instructions]**

  Generate extra move instructions.

  For $t_1 \leftarrow t_2 + t_3$ generate

  $$\begin{array}{lll} \texttt{mov} & t_1, t_2 & t_1 \leftarrow t_2 \\ \texttt{add} & t_1, t_3 & t_1 \leftarrow t_1 + t_3; \end{array}$$

- **[Special addressing modes]**

  Example: memory addressing

  ```
  mov eax,[ebp-8]
  add eax, ecx          add [ebp-8], ecx
  mov [ebp-8], eax
  ```

  Two choices:

  1. Ignore and use separate load and store instructions. Same speed, but an extra register gets trashed.
  2. Avoid register pressure and use addressing mode. More work for the pattern matcher.

Challenges III

- **[Variable-length instructions]**
  No problem for instruction selection or register allocation. Assembler deals with it.
- **[Instructions with side effects]**
  Example: autoincrement after memory fetch (MC 680x0)

  $$r_2 \leftarrow M[r_1]; \qquad r_1 \leftarrow r_1 + 4$$

  Hard to incorporate in tree-pattern based instruction selection.

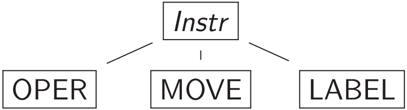
  1. Ignore...
  2. Ad-hoc solution
  3. Different algorithm for instruction selection

Abstract Assembly Language

Output of Instruction Selection

University of Freiburg

Class hierarchy for representing instructions



Each instruction specifies a

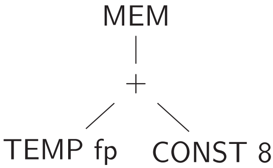
- set of defined temporaries
- set of used temporaries
- set of branch targets

each of which may be empty

Abstract Assembly Language

Creating an Operation

University of Freiburg



```
new OPER ("LOAD 'd0 <- M['s0+8]",
           L (new Temp(), null),// targets: defined
           L (frame.FP, null)); // sources: used
```

- Independent of register allocation and jump labels

Abstract Assembly Language

Important

University of Freiburg

An operation's def and use set must account for *all* defined and used registers.

- Example: the multiplication instruction on Pentium

```
new OPER ("mul 's0",
          L (pentium.EAX, L (pentium.EDX, null)),
          L (argTemp, L (pentium.EAX, null)));
```

- Example: a procedure call trashes many registers (see the calling convention of the architecture)
    - return address
    - return-value register
    - caller-save registers