Low-level optimization

T-106.5450 Advanced Course on Compilers  
Spring 2015 (III-V): Lecture 6

Vesa Hirvisalo  
ESG/CSE/Aalto

2015-02-25
Today

- Introduction to code generation
  - finding the best translation
- Instruction selection
  - matching the machine instructions
  - optimization means the best match
- Code-generator compilers
  - also known as code-generator generators
  - LR-based, TWIG-like, and BURS-based as examples
- Register allocation
  - overview of the problem
  - interference graph coloring as a solution
- Instruction scheduling
  - only an overview
  - we will get back to this (is essential for parallelism!)
Introduction to code generation
Code generation

The two basic problems are:

- finding the (semantically) correct translations
- selecting the best one

We will mostly consider latter, i.e., *low-level optimizations*.

But also the former, finding correct translations, is difficult.

For both, we are facing severe problems

- machine languages are large
- they are complex
- they are ambiguous
- parallelism is a real challenge
  - in our introduction, we will omit this
Low-level optimizations

Selecting the best (optimal) translation is NP-complete for typical target machines. Even selecting a good translation is difficult:

- huge search space
- complex cost functions
  - especially memory
  - but we will omit such complications in our simplified introduction

The *classical solution* is to split code generation into phases:

- register allocation (data)
- instruction selection (code)
- execution order (connecting the code and data)
Ambiguity

Considered as a formal (context-free) language, typical IL grammar of machine language is *massively* ambiguous.

- This means that there exists a huge number of (parse) trees for a IL program.

Ambiguity leads to conflicts that must be resolved:

- **dynamic resolution**
  - during compilation
  - can be slow
  - soft-coded generators
  - typically a program in IL is covered by ML patterns in IL

- **static resolution**
  - resolved when the compiler is built
  - hard-coded generators
  - often based on LR-parsing a program in IL
  - parser state space is often huge
Instruction selection
Instruction selection

Instruction selection translates from IL to machine language.

- instruction selection methods are translation methods
- typically much more advanced (and complex) than the *syntax-directed translation* method for front-ends

Code generation can be viewed as a tree rewriting process

- an instruction is described by a tree pattern
- patterns may have nonterminal symbols
- nonterminals stand for an intermediate results
- description for storage location

Actually, a true IL representation is typically a SSA-wired graph consisting of basic block DAGs

- we simplify (once more) to introduce the basics
Tree coverage

We cover the source (language) tree

- using the target language patterns (matching)
- each match corresponds to a cover node

The *cover tree* is the translation result.

A cover tree fulfills the following conditions:

- all cover nodes have pattern that matches the original intermediate representation.
- a cover node has as many sons as the pattern of the associated rule contains nonterminals.
- the i-th nonterminal of the rule of a node n is equal to the result nonterminal of the rule of the i-th son of n.
Example (1/3)

Let machine instructions and their patterns in IL be:

- **ADD ri, rj, rk**  BINOP(+, TEMP(rj), TEMP(rk))
- **MUL ri, rj, rk**  BINOP(*, TEMP(rj), TEMP(rk))
- **SUB ri, rj, rk**  BINOP(-, TEMP(rj), TEMP(rk))
- **DIV ri, rj, rk**  BINOP(/, TEMP(rj), TEMP(rk))
- **ADDI ri, rj, c**  BINOP(+, TEMP(rj), CONST(c))
- **ADDI ri, r0, c**  CONST(c)
- **SUBI ri, rj, c**  BINOP(-, TEMP(rj), CONST(c))
- **LOAD ri, rj[c]**  MEM(BINOP(+, TEMP(rj), CONST(c)))
- **LOAD ri, r0[c]**  MEM(CONST(c))
- **LOAD ri, rj[0]**  MEM(TEMP(rj))
- **STORE rj[c], ri**  MOVE(MEM(BINOP(+, TEMP(rj), CONST(c))), TEMP(ri))
- **STORE r0[c], ri**  MOVE(MEM(CONST(c)), TEMP(ri))
- **STORE rj[0], ri**  MOVE(MEM(TEMP(rj)), TEMP(ri))
- **MOVEM rj, ri**  MOVE(MEM(TEMP(rj)), MEM(TEMP(ri)))

Note: Register r0 is always zero. In the above, we omitted listing the symmetries (e.g., switching places of TEMPs, MEMs, and CONSTs).
Example (2/3)

Consider the following IL expression

\[
\text{MOVE} \left( \text{MEM}(\text{MEM}(\text{CONST}(y)), \text{MEM}(\text{BINOP}(+, \text{TEMP}(FP), \text{CONST}(x)))) \right)
\]

to be covered with

![Diagram of IL expression]

\[
\text{MOVE} \quad \text{MEM} \quad \text{MEM} \quad + \quad \text{FP} \quad \text{CONST} \; x
\]

\[
\text{MEM} \quad \text{MEM} \quad \text{CONST} \; y
\]
Example (3/3)

The cover forms a tree (in the machine language):

```
MOVEM M[ri], M[rj]
LOAD r1, r0[y]
ADDI rj, fp, x
```

Note that register allocation and execution order are left free.

- but we have limited the choices by them
- doing instruction selection first has consequences

Note also, we did not describe how we selected the tree coverage.
Greedy selection

A simple approach is to take “maximal munch” at each time.

- go top-down until the whole tree is covered
- at each IL node the largest fitting pattern is selected
- continue from the uncovered border

For the root node of the example, we can select

\[
\text{MOVE}(	ext{MEM}(	ext{TEMP}(r_j), \text{TEMP}(r_i))) \quad \text{or} \quad \text{MOVE}(	ext{MEM}(	ext{TEMP}(r_j), \text{MEM}(	ext{TEMP}(r_i))))
\]

The latter is bigger, thus it is selected. For the left branch:

\[
\text{MEM}(	ext{CONST}(y))
\]

and for the right:

\[
\text{BINOP}(+, \text{TEMP}(FP), \text{CONST}(x))
\]
Code-generator compilers
Automated Tools

The evolution of programming languages and computer architectures has created the need for automating the code-synthesis phase in compilers.

Several techniques have been proposed.

- some of these techniques have become feasible
- they are gaining acceptance in practice

However compared to front ends

- LL, LR, etc. parser really do parse the input
- automated code generators usually do partial generation
  - they help to programmer with trivial code
  - difficult situations are usually manually coded
  - this is laborious as instruction sets are large
Challenges

Front ends are easy to write. What is the problem with the back ends?

- it is enough for a front end translation to be correct
- a back end translation must be "optimal"

The success of a code generator compiler is measured by the following key criteria:

- minimization of machine-dependent modules
- reasonable speed and compactness of target code generated by the code generator
- reasonable speed and compactness of the code generator
- reasonable speed of the code generator compiler
Using context-free grammars

A code generator can be construed as a syntax-directed translator

- source representation
  - a linearized prefix form of the intermediate representation
  - for simplicity, we assume trees in our introduction
- parsed by a LR parser built for context-free grammar
  - the grammar describes the target machine

Representing machines as LR grammars

- every instruction variant of the target machine is described by a grammar rule
- uses a prefix form of an intermediate representation
- nonterminals for the results of instructions
- nonterminals for the addressing modes
Code generation with parsers

A parser attempts to find a parse tree

- for the entire source tree
  - in the intermediate language
  - the tree is linearized without loss of information
- along parsing generates code for it
- the parse tree is the translation result
  - in the machine language

Since there are (typically) several ways to parse

- there are several machine code sequences that may correspond to it
- the ambiguities are usually resolved by some heuristics and simplifications
Example (1/2): Prefix form of trees

Figure: An example input, nodes numbered in their linearization order
Example (2/2): Prefix form of patterns

An example of a code generator rule is

\[
\text{reg.1} ::= + \text{deref} + \text{const.1}\ \text{reg.2} \\
\text{reg.1, const.1 (reg.2)}
\]

The rule matches an addition instruction

\[
+ \ [\text{reg}] \\
\downarrow \\
\text{deref} \quad \text{reg} \\
\downarrow \\
+ \\
\downarrow \\
\text{const} \quad \text{reg}
\]

**Figure:** The pattern for the addition instruction \text{add reg, const(reg)}
Code generation with an LR-parser

This can be viewed as a form of tree rewriting

- A parse corresponds to a covering of the tree with instruction templates
- the instruction templates are also trees
- however, we use linearized forms of both

We can use parsing because of the linearization

- but this means that the input is seen only once, LR:
  - L=”from left” consumption of input
  - R=”rightmost derivation” of a parser tree

As any parser generating output, the parser has actions.

- actions are done at reductions
- code is emitted during the actions (reductions)
Problems with parser-based generation

With LR-parsing there are several practical difficulties

▶ a grammar describing a target machine can be quite large
▶ the resulting automaton can be of infeasible size
▶ an LR parser is left-operand biased

Another problem in purely syntactic approach (context-free grammar) is the difficulty of specifying target-machine architectural constraints

▶ register restrictions on addressing-modes
▶ results in multiple locations
▶ condition codes
Generator based on tree rewriting

Code generation can be viewed as a global tree rewriting process.

- an instruction is described by a tree pattern
- patterns may have nonterminal symbols
- nonterminals stand for an intermediate results

Instead of simple greedy approach, global optimization can be done by using global cost computations.

- a cover tree is called a minimal cover of an intermediate representation if no cheaper cover exists
  - the cost of a cover tree is the sum of the costs of each cover node
  - the cost of a node is the cost of the rule associated with it

The problem: how to find the minimal cost cover?
TWIG-like generators

The original Twig (by Aho & all.) is a code generator compiler that uses a tree rewriting scheme. Since, other similar generators have been developed.

Twig uses tree rewrite rules of the form
replacement <- template { cost } = { action }
where

- \textit{replacement} is a single node,
- \textit{template} is a tree,
- \textit{cost} is a code fragment that computes the cost of the template, and
- \textit{action} is a code fragment.
TWIG pattern matching and cost computation

The tree-matching algorithm is a generalization of Aho and Corasick’s linear-time keyword matching algorithm as suggested by Hoffman and O’Donnell.

- intermediate representation as a string
- patterns as strings
- several matches at the same time
- linear time

Twig uses dynamic programming to find the minimal cover. The dynamic programming algorithm is a simplification of Aho and Johnson’s optimal code-generation algorithm.

- instead of a cost vector twig uses a single scalar cost

You can find descriptions of these in the literature (omitted from the slides of the course).
Register allocation
Introduction to register allocation

Register allocation can be divided into sub-problems.

- register recognition: what values need to be stored (pseudos)
- register allocations: what are the values stored
- register assignment: where they are stored

Register allocation is about finding storage for computation using static methods.

- keeping data close to the functional units
- caches do this dynamically
  - scratchpad store chunks of data (compared to registers)
  - we will omit scratchpad allocation by compilers

Instructions using registers are cheaper than memory-based instructions (time-wise, energy-wise, etc.).
Fixed register allocations

Registers have different usages. Typical is to use some for special purposes and allocate the others freely.

Typical special uses.

- stack pointer
- frame pointer
- heap pointer
- indexing

Register allocators target the “freely allocatable” register

- however, there can be (and often there are!) restrictions on their usage
- such restrictions complicate the allocation
Registers and data flow

Register allocation is typically based on data flow constraints and heuristic cost analysis.

The allocation is usually divided into:

- local register allocation (basic blocks)
- global register allocation
  - Once again: optimizing for loops is the key

Registers allocated for values (not for variables):

- one variable can have several values
- variables can share values
- this is closely connected to liveness and SSA

**Note:** Typically we use liveness data (or next uses, or du chains), but backward chains can also be needed for the heuristics.
Interference graphs

Register allocation can be done by comparing liveness regions of values. This approach is closely connected to SSA.

We build an interference graph between values:

- the nodes are the values
- there is an edge between two nodes, if one is alive while the other is defined
- values cannot be put into the same registers, if there is an edge between their nodes

The allocation can be computed by coloring the graph.

Note: Graph coloring is a common NP-complete graph problem. Heuristic approaches for compilers include priority-based (Chow-style) and bottom-up (Chaitin-style) methods.
The next example uses a heuristic. Consider the program:

S1: \[ v3 <- v1 / v2 \]
S2: \[ v5 <- a[v3] \]
S3: \[ a[v7] <- v2 \]
S4: \[ v4 <- v3 / v2 \]
S5: \[ v6 <- v5 - v4 \]

Liveness

before S1 \[ v1 \ v2 \ v7 \]
between S1/S2 \[ v2 \ v3 \ v7 \]
between S2/S3 \[ v2 \ v3 \ v5 \ v7 \]
between S3/S4 \[ v2 \ v3 \ v5 \]
between S4/S5 \[ v4 \ v5 \]
after S5 \[ v6 \]
Thus, we get the graph:

![Graph Diagram]

Example (2/4)
Example (3/4)

We use a stack to do the coloring. We stack nodes that have less than four neighbors or make a spill if there is not other choice.

Assume three registers. We can stack, e.g.:

\[ v_6 \ v_1 \ v_4 \]

All the rest have more than two neighbors. Let us select \( v_5 \) to be spilled. Thus, we get:

\[ v_6 \ v_1 \ v_4 \ v_5 \text{ (potential spill)} \ v_2 \ v_3 \ v_7 \]

Make an assignment:

\[ v_7 \text{ (R0)} \ v_3 \text{ (R1)} \ v_2 \text{ (R2)} \]

Neighbors of \( v_5 \) occupy all the registers (colors), thus it becomes a true spill.
Based on the coloring, we end up with the allocation:

<table>
<thead>
<tr>
<th>node</th>
<th>edge-to</th>
<th>selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>v7</td>
<td>-</td>
<td>R0</td>
</tr>
<tr>
<td>v3</td>
<td>R0</td>
<td>R1</td>
</tr>
<tr>
<td>v2</td>
<td>R0,R1</td>
<td>R2</td>
</tr>
<tr>
<td>v5</td>
<td>R0,R1,R2</td>
<td>mem</td>
</tr>
<tr>
<td>v4</td>
<td>-</td>
<td>R0</td>
</tr>
<tr>
<td>v1</td>
<td>R0,R2</td>
<td>R1</td>
</tr>
<tr>
<td>v6</td>
<td>-</td>
<td>R0</td>
</tr>
</tbody>
</table>
Instruction scheduling
Introduction to instruction scheduling

Order of the execution must be decided

▶ this is often done in several phases (and using several levels of abstraction)
▶ in the following, we use our simple example machine

For a basic blocks, its DAG gives the all the correct execution orders. Thus any traversal following the restrictions is good.

▶ different orders can yield different performance
▶ selecting the best one is called instruction scheduling

Instruction scheduling is in theory NP-complete (and difficult in practice), so heuristics must be used.

▶ the issue is even more complex for parallel architectures
▶ scheduling constraints (etc.) must be added to IL
Consider the following basic block:

\[
\begin{align*}
i &\leftarrow i / j \\
m &\leftarrow i \\
k &\leftarrow i / j \\
i &\leftarrow a[i] \\
n &\leftarrow i - k \\
s &\leftarrow m / j \\
m &\leftarrow i - s \\
a[t] &\leftarrow m
\end{align*}
\]

We apply the local value numbering algorithm to the code.
Figure: The value DAG for the basic block
Example (3/4)

The final execution order is usually decided after doing the instruction selection (e.g., assume the DAG to be represented in our IL and covered by our machine language templates).

Topological ordering of the computational nodes gives the possible execution orders.

Figure: Execution orders for the DAG (using value node numbers). The figure is to be read from left to right and machine instructions are represented by the value numbers of the root nodes of they cover.
We get the following code, if we select the top-most execution order and assume two registers available (assuming $R_1$ to be holding $i$ and $R_2$ to be holding $j$ at the beginning of the basic block).

```assembly
div R1, R1, R2 ; (3) note the order of the operands!
div R2, R1, R2 ; (4)
load R1, R2[a] ; (6)
sub R1, R1, R2 ; (7)
load R2, R0[t] ; remember semantics of R0!
store R2[a], R1 ; (9)
```

Note that the choice of the order affected our register allocation. The decision of our register allocator is to keep variable $t$ in memory (i.e., loading it into a register just prior the use).