Instruction Selection: Preliminaries

Comp 412

Chapter 11 in EaC2e
The Problem

Writing a good 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
• Optimizer is largely hand crafted
• (Parts of) the back end can be automated
In discussing the compiler’s back end, we will assume the model shown to the left:

- Instruction selection is largely solved
  - A major problem of the 1970’s & 1980s
- Register allocation & instruction scheduling are more complex
  - Both are NP-complete in any real case
- Operation placement is not yet critical
  - Assume a unified register set
  - Partitioned register sets add another hard problem

Parallelism either comes from the application, or is derived with techniques that are beyond the scope of this course (see COMP 515).
Structure of the Compiler’s Back End

What about the IR?

- Common case today is an ILOC-like IR
  - Branches, memory tags, hierarchy of memory operations, multiple ops/cycle
  - `clang` is ILOC-like but in SSA form, `gcc` is a lower level, linear IR
- Assume that several things are explicit:
  - “Enough” register names
  - Order of operations & flow of values
- All of these techniques also work on a tree-like IR
Definitions

**Instruction selection** (§ 11)
- Maps the IR into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

**Instruction scheduling** (§ 12)
- Reorders the operations to hide latencies
- Assumes a fixed choice of operations
- Changes demand for registers

**Register allocation** (§ 13)
- Decides which values will reside in registers
- Assumes a fixed order of existing operations
- Changes name space; may add false sharing

These three problems are tightly coupled.
The Big Picture

Conventional wisdom says that we lose little by solving these problems independently

Instruction selection

• Use some form of pattern matching
• Assume enough registers or target “important” values

Instruction scheduling

• Within a block, local list scheduling is “close” to optimal
• Across blocks, build framework to apply list scheduling

Register allocation

• Start from virtual registers & map “enough” into $k$
• Global allocation requires global model or paradigm

Local allocation & local scheduling have a similar algorithmic flavor. We don’t have a good paradigm for global scheduling.
The Big Picture

*What are today’s hard issues?*

**Instruction selection**
- Problems are well solved & tools are available
- Impact of choices on energy consumption & functional unit placement

**Instruction scheduling**
- Modulo scheduling loops with control flow
- Schemes for scheduling long latency operations
- Finding enough ILP to keep functional units (& cores) busy

**Register allocation**
- Cost of allocation — a particular problem for JITs
- Better spilling: choice, placement, & granularity
- Efficiency of the allocator

In the original HotSpot Server Compiler (JIT), register allocation took 49% of compile time.
How Hard Are These Problems?

**Instruction selection**
- Can make locally optimal choices quickly, with automated tools
- Global optimality is (undoubtedly) NP-Complete

**Instruction scheduling**
- Single basic block $\Rightarrow$ approximate heuristics work quickly & well
- General problem, with control flow $\Rightarrow$ NP-Complete
- To move beyond a block, find a CFG path to treat as a single block

**Register allocation**
- Single basic block, no spilling, & 1 register size $\Rightarrow$ linear time
- Whole procedure is NP-Complete
- Approximate global allocators are widely used and accepted
Definitions

Instruction selection
• Mapping IR 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 choice of operations
• Changes demand for registers

Register allocation
• Deciding which values will reside in registers
• Assumes a fixed order of existing operations
• Changes name space; may add false sharing
The fundamental problem is that modern computers have many ways to do any single thing

Consider register-to-register copy in ILOC

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

\[
\begin{array}{ccc}
\text{addI} & r_i,0 & \Rightarrow \ r_j \\
\text{subI} & r_i,0 & \Rightarrow \ r_j \\
\text{multI} & r_i,1 & \Rightarrow \ r_j \\
\text{lshiftI} & r_i,0 & \Rightarrow \ r_j \\
\text{orI} & r_i,0 & \Rightarrow \ r_j \\
\text{divI} & r_i,1 & \Rightarrow \ r_j \\
\text{rshiftI} & r_i,0 & \Rightarrow \ r_j \\
\text{xorI} & r_i,0 & \Rightarrow \ r_j \\
\end{array}
\]

\(\text{(any algebraic identity ...)}\)

Some of these are cheap

Some of these are expensive

- Human would ignore most, if not all, of these
- An algorithm will look at all of them & find the low-cost choice
  - Take context into account
    \(\text{(busy functional unit?)}\)

And ILOC is an much simpler than a real ISA
The Goal

We want to automate generation of instruction selectors (as with parsers & scanners)

The machine description should also help with scheduling & allocation

Simplification: we will restrict our discussion to signed integers. Expanding to handle other types is straightforward. It adds more rules to the examples, not more complexity.
The Big Picture

Need pattern matching techniques

- Must produce good code (under some metric for “good”)
- Must run quickly

The simple schemes we saw earlier ran quickly

How good was the code?

<table>
<thead>
<tr>
<th>Tree</th>
<th>AHSĐT Code</th>
<th>Desired Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>loadI 4 ( \Rightarrow r_5 )</td>
<td>loadAI ( r_0,4 \Rightarrow r_5 )</td>
</tr>
<tr>
<td></td>
<td>loadAO ( r_0,r_5 \Rightarrow r_6 )</td>
<td>loadAI ( r_0,8 \Rightarrow r_6 )</td>
</tr>
<tr>
<td></td>
<td>loadI 8 ( \Rightarrow r_7 )</td>
<td>mult ( r_5,r_6 \Rightarrow r_7 )</td>
</tr>
<tr>
<td></td>
<td>loadAO ( r_0,r_7 \Rightarrow r_8 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mult ( r_6,r_8 \Rightarrow r_9 )</td>
<td></td>
</tr>
</tbody>
</table>

Assume ARP is in \( r_0 \)

Notation: Variable named b, stored at offset 8 from the ARP

COMP 412, Fall 2017

Ad Hoc SDT Treewalk on AST
The Big Picture

Need pattern matching techniques
• Must produce good code
• Must run quickly

The simple schemes we saw earlier ran quickly
How good was the code?

This inefficiency is easy to fix.
See digression on page 346 of EaC2e.
The Big Picture

Need pattern matching techniques
• Must produce good code
• Must run quickly

The simple schemes we saw earlier ran quickly
How good was the code?

Tree

<table>
<thead>
<tr>
<th>Tree</th>
<th>AHSĐT Code</th>
<th>Desired Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENT &lt;a,ARP,4&gt;</td>
<td>loadI 4 (\Rightarrow r_5)</td>
<td>loadAI (r_0,4 \Rightarrow r_5)</td>
</tr>
<tr>
<td>NUMBER &lt;2&gt;</td>
<td>loadAO (r_0, r_5 \Rightarrow r_6)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>loadI 2 (\Rightarrow r_7)</td>
<td></td>
</tr>
</tbody>
</table>
The Big Picture

Need pattern matching techniques
• Must produce good code
• Must run quickly

The simple schemes we saw earlier ran quickly
How good was the code?

COMP 412, Fall 2017
The Big Picture

**Need pattern matching techniques**
- Must produce good code
- Must run quickly

The simple schemes we saw earlier ran quickly
How good was the code?

**Pattern matching is not the answer to all problems with code quality.**

**LVN** finds this algebraic identity easily.

Another possibility that might take less time & energy — an algebraic identity
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 sequence of target-machine ops
• 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 sequence of target machine ops
• Use text matching (Aho-Corasick) or peephole matching

In practice, both work well & produce efficient matchers. Either can be implemented as a table-driven or hard-coded matcher.
Equivalence of Trees and Linear Code

Trees and linear code are interchangeable for the kinds of pattern matching done in instruction selection.

To expose low-level detail, we need an AST with a low-level of abstraction.

Quads for \( a \leftarrow b - 2 \times c \)

<table>
<thead>
<tr>
<th>OP</th>
<th>Arg₁</th>
<th>Arg₂</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\times</td>
<td>2</td>
<td>c</td>
<td>( t₁ )</td>
</tr>
<tr>
<td>-</td>
<td>b</td>
<td>( t₁ )</td>
<td>a</td>
</tr>
</tbody>
</table>

NOTE: All interior nodes have values in registers.

Low-level AST for \( a \leftarrow b - 2 \times c \)
Equivalence of Trees and Linear Code

Trees and linear code are interchangeable for the kinds of pattern matching done in instruction selection.

To expose low-level detail, we need an AST with a low-level of abstraction.

VAL in register \( (r_{arp}) \)
NUM: constant
LAB: ASM label

NUM is a constant that fits in an immediate op (e.g., multl).
LAB is a constant that does not, but fits in a loadl.
This kind of mundane detail now matters.

a: at ARP+4
b: at ARP-16 (call-by-reference)
c: at @G+12

Low-level AST for \( w \leftarrow x - 2 \times y \)
Prefix Notation

To describe low-level ASTs, we need a concise notation

\[
+ \quad +
\quad +
\]

\[
\text{VAL} \quad \text{NUM}
\]

\[
\text{REF} \quad \text{Reg}_i
\]

\[
+(\text{VAL,NUM})
\]

\[
+(\text{REF,Reg}_i)
\]

Linear prefix form
To describe low-level ASTs, we need a concise notation

\[
\text{GETS}(+(\text{VAL}_1,\text{NUM}_{1}), -(\text{REF}(\text{REF}(-\text{VAL}_2,\text{NUM}_{2}))), *(\text{NUM}_{3}, \text{REF}(+(\text{LAB}_1,\text{NUM}_{4})))))
\]
Prefix Notation

To describe the machine operations, we need a concise notation

\[ + \]
\[ r_i \quad c_j \]
\[ + (r_i, c_j) \]

\[ + \]
\[ c_j \quad r_i \]
\[ + (c_j, r_i) \]

Pattern for commutative variant of

\[ + (r_i, c_j) \]

With each tree pattern, we associate a code template and a cost

- Template shows how to implement the subtree
- Cost is used to drive process to low-cost code sequences

Prefix notation quickly blurs the distinction between linear and tree IRs
Equivalence of Trees and Linear Code

Of course, we can build low-level trees that model low-level linear code

- Those trees look a lot like dependence graphs built in your schedulers
- The construction algorithm is quite similar

**LLIR Code**

\[
\begin{align*}
  r_{09} & \leftarrow 2 \\
r_{10} & \leftarrow @G \\
r_{11} & \leftarrow 12 \ # @c \\
r_{12} & \leftarrow r_{10} + r_{11} \\
r_{13} & \leftarrow \text{MEM}(r_{12}) \\
r_{14} & \leftarrow r_{10} \times r_{13} \\
r_{15} & \leftarrow 16 \ # @b \\
r_{16} & \leftarrow r_{\text{arp}} - r_{15} \\
r_{17} & \leftarrow \text{MEM}(r_{16}) \\
r_{18} & \leftarrow \text{MEM}(r_{17}) \\
r_{19} & \leftarrow r_{18} - r_{14} \\
r_{20} & \leftarrow 4 \ # @a \\
r_{21} & \leftarrow r_{\text{arp}} + r_{20} \\
\text{MEM}(r_{21}) & \leftarrow r_{19}
\end{align*}
\]