Code Generation

– Wilhelm/Maurer: Compiler Design, Chapter 12 –
Mooly Sagiv
Tel Aviv University
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

19. Dezember 2007
“Standard” Structure

source(text) →
lexical analysis(7) →
tokenized-program →
syntax analysis(8) →
syntax-tree →
semantic-analysis(9) →
decorated syntax-tree →
optimizations(10) →
intermediate rep. →
code-generation(11, 12) →
machine-program

finite automata

pushdown automata

attribute grammar evaluators

abstract interpretation + transformations

tree automata + dynamic programming + …
Code Generation

Real machines (instead of abstract machines):
- Register machines,
- Limited resources (registers, memory),
- Fixed word size,
- Memory hierarchy,
- Intraprocessor parallelism.
Architectural Classes: CISC vs. RISC

**CISC** IBM 360, PDP11, VAX series, INTEL 80x86, Pentium, Motorola 680x0
- A large number of addressing modes
- Computations on stores
- Few registers
- Different instruction lengths
- Different execution times for instructions
- Microprogrammed instruction sets

**RISC** Alpha, MIPS, PowerPC, SPARC
- One instruction per cycle (with pipeline for load/stores)
- Load/Store architecture – Computations in registers (only)
- Many registers
- Few addressing modes
- Uniform lengths
- Hard-coded instruction sets
- Intra-processor parallelism: Pipeline, multiple units, Very Long Instruction Words (VLIW), Superscalarity, Speculation
Phases in code generation

**Code Selection:** selecting semantically equivalent sequences of machine instructions for programs,

**Register Allocation:** exploiting the registers for storing values of variables and temporaries,

**Code Scheduling:** reordering instruction sequences to exploit intraprocessor parallelism.

Optimal register allocation and instruction scheduling NP-hard.
Phase Ordering Problem

Partly contradictory optimization goals:

Register allocation: minimize number of registers used $\Rightarrow$ reuse registers,

Code Scheduling: exploit parallelism $\Rightarrow$ keep computations independent, no shared registers

Issues:
- Software Complexity
- Result Quality
- Order in Serialization
Challenges in real machines: CISC vs. RISC

**CISC** IBM 360, PDP11, VAX series, INTEL 80x86, Motorola 680x0
- A large number of addressing modes
- Computations on stores
- Few registers
- Different instruction lengths
- Different execution times for instructions
- Microprogrammed instruction sets

**RISC** Alpha, MIPS, PowerPC, SPARC
- One instruction per cycle (with pipeline for load/stores)
- Load/Store architecture – Computations in registers (only)
- Many registers
- Few addressing modes
- Uniform lengths
- Hard-coded instruction sets
- Intra-processor parallelism: Pipeline, multiple units, Very Long Instruction Words (VLIW), Superscalarity, Speculation
Example: \( x = y + z \)

**CISC/Vax**  \texttt{addl3 4(fp), 6(fp), 8(fp)}

**RISC**

- load \( r_1, 4(fp) \)
- load \( r_2, 6(fp) \)
- add \( r_1, r_2, r_3 \)
- store \( r_3, 8(fp) \)
The VLIW Architecture

- Several functional units,
- One instruction stream,
- Jump priority rule,
- FUs connected to register banks,
- Enough parallelism available?
# Instruction Pipeline

Several instructions in different states of execution Potential structure:

1. instruction fetch and decode,
2. operand fetch,
3. instruction execution,
4. write back of the result into target register.

<table>
<thead>
<tr>
<th>Pipeline stage</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B_1$</td>
<td>$B_2$</td>
<td>$B_3$</td>
<td>$B_4$</td>
<td>$B_1$</td>
<td>$B_2$</td>
<td>$B_3$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Pipeline hazards

- Cache hazards: Instruction or operand not in cache,
- Data hazards: Needed operand not available,
- Structural hazards: Resource conflicts,
- Control hazards: (Conditional) jumps.
Program Representations

- Abstract syntax tree: algebraic transformations, code generation for expression trees,
- Control Flow Graph: Program analysis (intraproc.)
- Call Graph: Program analysis (interproc.)
- Static Single Assignment: optimization, code generation
- Program Dependence Graph: instruction scheduling, parallelization
- Register Interference graph: register allocation
Code Generation: Integrated Methods

- Integration of register allocation with instruction selection,
- Simple machine model:
  - \( r \) general purpose register \( R_0, \ldots, R_{r-1} \),
  - Two address instructions
    \[
    \begin{align*}
    R_i & := M[V] & \text{Load} \\
    M[V] & := R_i & \text{Store} \\
    R_i & := R_i \text{ op } M[V] & \text{Compute} \\
    R_i & := R_i \text{ op } R_j
    \end{align*}
    \]
- Two phases:
  1. Computing register requirements,
  2. Generating code, allocating registers and temporaries.
Example Tree

Source \( r := (a + b) - (c - (d + e)) \)

Tree

```
:=
  r
  -
    -
      +
        a
        b
      -
        c
        +
          d
          e
```
## Generated Code

2 Registers $R_0$ and $R_1$

Two possible code sequences:

<table>
<thead>
<tr>
<th>$R_0$</th>
<th>$M[a]$</th>
<th>$R_0$</th>
<th>$M[c]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_0$</td>
<td>$R_0 + M[b]$</td>
<td>$R_0$</td>
<td>$R_0 - R_1$</td>
</tr>
<tr>
<td>$R_1$</td>
<td>$M[d]$</td>
<td>$R_1$</td>
<td>$R_1 + M[e]$</td>
</tr>
<tr>
<td>$R_1$</td>
<td>$R_1 + M[e]$</td>
<td>$R_0$</td>
<td>$R_0 - R_1$</td>
</tr>
<tr>
<td>$M[t_1]$</td>
<td>$R_1$</td>
<td>$R_1$</td>
<td>$M[a]$</td>
</tr>
<tr>
<td>$R_1$</td>
<td>$M[c]$</td>
<td>$R_1$</td>
<td>$R_1 + M[b]$</td>
</tr>
<tr>
<td>$R_1$</td>
<td>$R_1 - M[t_1]$</td>
<td>$R_1$</td>
<td>$R_1 - R_0$</td>
</tr>
<tr>
<td>$R_0$</td>
<td>$R_0 - R_1$</td>
<td>$M[f]$</td>
<td>$R_1$</td>
</tr>
<tr>
<td>$M[f]$</td>
<td>$R_0$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

stores result for $c - (d + 2)$ evaluates $c - (d + 2)$ first
in a temporary (needs 2 registers)
no register available saves one instruction
The Algorithm

Principle: Given tree $t$ for expression $e_1 \ op \ e_2$
$t_1$ needs $r_1$ registers, $t_2$ needs $r_2$ registers,

$r \geq r_1 > r_2$: After evaluation of $t_1$:
   $r_1 - 1$ registers freed, one holds the result,
   $t_2$ gets enough registers to evaluate, hence
   $t$ can be evaluated in $r_1$ registers,

$r_1 = r_2$: $t$ needs $r_1 + 1$ registers to evaluate,

$r_1 > r$ or $r_2 > r$: spill to temporary required.
Labeling Phase

- Labels each node with its register needs,
- Bottom-up pass,
- Left leaves labeled with '1' have to be loaded into register,
- Right leaves labeled with '0' are used as operands,
- Inner nodes:
  \[
  \text{regneed}(\text{op}(t_1, t_2)) = \begin{cases} 
  \max(r_1, r_2), & \text{if } r_1 \neq r_2 \\
  r_1 + 1, & \text{if } r_1 = r_2 
  \end{cases}
  \]
  where \( r_1 = \text{regneed}(t_1) \), \( r_2 = \text{regneed}(t_2) \)
Example

\[
\begin{array}{c}
\text{:=} \\
\text{−} 2 \\
\text{+} 1 \\
\text{−} 2 \\
\text{+} 1 \\
a \quad b \\
1 \quad 0 \\
1 \\
d \quad e \\
1 \quad 0
\end{array}
\]
Generation Phase

Principle:

- Generates instruction \textbf{Op} for operator \textit{op} in \textit{op}(t_1, t_2) after generating code for \textit{t}_1 and \textit{t}_2.
- Order of \textit{t}_1 and \textit{t}_2 depends on their register needs,
- The generated \textbf{Op}–instruction finds value of \textit{t}_1 in register,
- \textit{RSTACK} – available registers, initially all registers,
  
  \textbf{Before processing \textit{t}}: top(\textit{RSTACK}) is determined as result register for \textit{t},
  
  \textbf{After processing \textit{t}}: all registers available, but top(\textit{RSTACK}) is result register for \textit{t}.
- \textit{TSTACK} – available temporaries.
Algorithm Gen_Opt_Code

Algorithm

\[
\begin{array}{l}
\text{var } RSTACK: \text{ stack of register;} \\
\text{var } TSTACK: \text{ stack of address;} \\
\text{proc Gen\_Code}(t: \text{ tree}); \\
\text{var } R: \text{ register, } T: \text{ address;} \\
\text{case } t \text{ of} \\
(\text{leaf } a, 1): \quad (\text{*left leaf*}) \\
\quad \text{emit}(\text{top}(RSTACK) := a); \\
(\text{op}((t_1, r_1), (\text{leaf } a, 0))): \quad (\text{*right leaf*}) \\
\quad \text{Gen\_Code}(t_1); \\
\quad \text{emit}(\text{top}(RSTACK) := \text{top}(RSTACK) \text{ Op } a); \\
\end{array}
\]

<table>
<thead>
<tr>
<th>$RSTACK$-Contents</th>
<th>result register</th>
</tr>
</thead>
<tbody>
<tr>
<td>((R', R'', \ldots))</td>
<td>(\text{result in } R')</td>
</tr>
</tbody>
</table>
\[ \text{op}((t_1, r_1), (t_2, r_2)) : \]

\begin{cases} 
    r_1 < \min(r_2, r): \\
    \text{begin} \\
    \text{exchange}(\text{RSTACK}); \\
    \text{Gen\_Code}(t_2); \\
    R := \text{pop}(\text{RSTACK}); \\
    \text{Gen\_Code}(t_1); \\
    \text{emit}(\text{top}(\text{RSTACK}) := \text{top}(\text{RSTACK}) \text{ Op } R); \\
    \text{push}(\text{RSTACK}, R); \\
    \text{exchange}(\text{RSTACK}); \\
    \text{end} ; \\
\end{cases}

(R', R'', \ldots)
\[ r_1 \geq r_2 \land r_2 < r: \]
\[
\text{begin}
\quad \text{Gen\_Code}(t_1);
\quad R := \text{pop}(RSTACK);
\quad \text{Gen\_Code}(t_2);
\quad \text{emit}(R := R \text{ Op top}(RSTACK));
\quad \text{push}(RSTACK, R);
\text{end} ;
\]
\[
(R', R'', \ldots)
\]
result in \( R' \)
\[
(R'', \ldots)
\]
result in \( R'' \)
\[
(R', R'', \ldots)
\]
\( r_1 \geq r \land r_2 \geq r \):

\[
\begin{align*}
\text{begin} \\
\quad \text{Gen\_Code}(t_2); \\
\quad T := \text{pop}(TSTACK); \\
\quad \text{emit}(M[T] := \text{top}(RSTACK)); \\
\quad \text{Gen\_Code}(t_1); \\
\quad \text{emit}(\text{top}(RSTACK) := \text{top}(RSTACK) \text{ Op } M[T]); \\
\quad \text{push}(TSTACK, T); \\
\text{end}; \\
\text{endcases} \\
\text{endcase} \\
\text{endproc}
\end{align*}
\]
Register Allocation and Instruction Selection by Dynamic Programming

- More complex architecture,
  - \( r \) general purpose registers \( R_0, \ldots, R_{r-1} \),
    \[
    R_i \quad := \quad e \quad \text{Compute}
    \]
  - Instruction formats: \( R_i \quad := \quad M[V] \quad \text{Load} \)
    \[
    M[V] \quad := \quad R_i \quad \text{Store}
    \]
  - \( e \) term with registers and memory cells,
    costs associated with each instruction.

- Goal: Generate cheapest instruction sequence using no more than \( r \) registers.
- Assume contiguous computation of subtrees \( \implies \) only one register required to hold the result
- Use instruction selection techniques with tree parsing, compute cheapest derivation.
Canonical recursive solution

- Assume $e$ of instruction $R_i := e$ matches tree $t$
- some subtrees of $t$ corresp. to memory operands of $e$ – computed into memory, no registers occupied after that
- let $e$ have $k$ register operands: compute corresponding subtrees $t_1, t_2, \ldots, t_k$ into these registers
- assume order $i_1, i_2, \ldots, i_k$ on computation and $j$ available registers
- $t_{i_1}$ has $j$ registers available, $t_{i_2}$ has $j - 1$, \ldots, $t_k$ has $j - k$ available
- if this fits for all subtrees $(j - k - \text{regnee}(t_{i_k}) \geq 0)$, add the minimal costs for computing all subtrees in this way to the costs of $e$ to yield the minimal costs for this combination
- if not enough registers are available, compute enough subtrees into memory, sum up costs like above
Canonical recursive solution (cont’d)

Doing it for all potential combinations recomputes the costs for subtrees $\Rightarrow$ exponential complexity
Dynamic Programming

- Convert top-down algorithm into bottom-up algorithm tabulating partial solutions
- Associate cost vector $C[0..r]$ with each node $n$, $C[0]$ cheapest costs for computing $t/n$ into a temporary, $C[i]$ cheapest costs computing $t/n$ into a register using $i$ registers.
- Compute cost vector at $n$ minimizing over all “legal” combinations of
  - one applicable instruction,
  - the cost vectors of the nodes “under” non-terminal nodes in the applied rule.
- What is a legal combination for $C[j], j > 0$?
  A combination of generated code for subtrees needing $\leq j$ registers.
- Extract cheapest instruction sequence in a second pass.
Global Register Allocation

So far, register allocation for assignments. Now, register allocation across whole procedures/programs.

Tasks of the Register Allocator:

1. determine candidates, i.e., variables and intermediate results, called Symbolic Registers, to keep in real registers, and determine their “life spans”.
2. assign symbolic registers without “collisions” to real registers using some optimality criterion,
3. modify the code to implement the decisions.

Constraint for assignment:

- Two symbolic registers collide if their contents are “live” at the same time,
- Colliding symbolic registers cannot be allocated to the same real register.
Definitions

- A **definition** of a symbolic register is the computation of an intermediate result or the modification of a variable,
- A **use** of a symbolic register is a reading access to the corresponding variable or a use of the intermediate value,
  Note: uses of symbolic registers in an individual computation step, e.g. execution of an instruction or of an assignment **precede** definitions of symbolic registers.
- A **definition–path** of $s$ to program point $p$ is a path from the entry point of the program to $p$ containing a definition of $s$,
- A **use–path** from $p$ is a definition-free path starting at $p$ containing a use of $s$,
- Symbolic register $s$ is **live** at program point $p$ if exists a definition–path to $p$ and a use–path from $p$,
The **life span** of \( s \) is the set of all program points, on which \( s \) is live.
Value of a live symbolic register may still be used.

Two life spans of symbolic registers **collide** if one of the registers is set in the life span of the other.

A life span for variable \( X \)
Computation of life ranges

Needs du (definition-use) chains.
A du (definition-use) chain connects a definition of a variable to all
the associated uses, i.e., uses that a value set at the definition may
flow to.
Two du chains are use-connected iff they share a use.
One could say, shared uses were vel-defined\(^1\).
A life range of a variable is the connected component of all
use-connected du chains of that variable.

\(^1\)Thanks to Raimund Seidel
Register Interference Graph

- nodes – life spans,
- edge between colliding life spans.

Allows to view the register-allocation problem as a graph coloring problem.

- $k$ physical registers available,
- Solve $k$–coloring problem,
- NP–complete for $k > 2$,
- Use heuristics.
Build constructs the register interference graph $G$,
Reduce initializes an empty stack;
repeatedly removes locally colorable nodes and pushes them onto the stack.
Continue at Assign Colours, if arrived at the empty graph: $G$ is $k$-colorable
Continue at Spill if locally uncolorable nodes remain in the graph.
Algorithm cont’d

**Assign Colours** pops nodes from the stack, reinserts them into the graph, and assigns a color not assigned to any neighbour.

**Spill** uses heuristics to select one node (variable) to spill to memory, inserts a **load** before each use of the variable and a **store** after each definition.
Then continues with **Build**.

The classical method by Chaitin uses $\text{degree}(n) < k$ as local-colorability criterion.
It means, $n$ and its neighbours can be colored with different colors.
Properties

- **Assign Colours** pops nodes off the stack in reverse order as Reduce pushed them onto the stack.
- The $degree(n) < k$ criterium holding, when $n$ was pushed, guarantees colorability.
- **Termination:**
  - Reduce repeatedly removes nodes from the finite set of nodes; each cycle through Spill reduces the graph by 1 node.
Heuristics for Node Removal

1. degree of the node: high degree causes many deletions of edges,
2. costs of spilling.
Example

Input-program
x := 1
y := 2
w := x + y
u := y + 2
z := x * y
x := u + z
print x,z,u
Example

Input-program    Symbolic Reg. Assign.
\[ x := 1 \]    \[ s_1 := 1 \]
\[ y := 2 \]    \[ s_2 := 2 \]
\[ w := x + y \]    \[ s_3 := s_1 + s_2 \]
\[ u := y + 2 \]    \[ s_4 := s_2 + 2 \]
\[ z := x \times y \]    \[ s_5 := s_1 \times s_2 \]
\[ x := u + z \]    \[ s_6 := s_4 + s_5 \]
\[ \text{print } x, z, u \]    \[ \text{print } s_6, s_5, s_4 \]
Example

Input-program  Symbolic Reg. Assign.
x := 1          s1 := 1
y := 2          s2 := 2
w := x + y      s3 := s1 + s2
u := y + 2      s4 := s2 + 2
z := x * y      s5 := s1 * s2
x := u + z      s6 := s4 + s5
print x,z,u     print s6,s5,s4

Register interference graph

s3  s2  s6

s1  s4  s5
Example

Input-program  Symbolic Reg. Assign.
\(x := 1\)  \(s_1 := 1\)
\(y := 2\)  \(s_2 := 2\)
\(w := x + y\)  \(s_3 := s_1 + s_2\)
\(u := y + 2\)  \(s_4 := s_2 + 2\)
\(z := x \times y\)  \(s_5 := s_1 \times s_2\)
\(x := u + z\)  \(s_6 := s_4 + s_5\)
\(\text{print } x, z, u\)  \(\text{print } s_6, s_5, s_4\)

Register interference graph
Example

<table>
<thead>
<tr>
<th>Input-program</th>
<th>Symbolic Reg. Assign.</th>
<th>After Register Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1</td>
<td>s1 := 1</td>
<td>r1 := 1</td>
</tr>
<tr>
<td>y := 2</td>
<td>s2 := 2</td>
<td>r2 := 2</td>
</tr>
<tr>
<td>w := x + y</td>
<td>s3 := s1 + s2</td>
<td>r3 := r1 + r2</td>
</tr>
<tr>
<td>u := y + 2</td>
<td>s4 := s2 + 2</td>
<td>r3 := r2 + 2</td>
</tr>
<tr>
<td>z := x * y</td>
<td>s5 := s1 * s2</td>
<td>r1 := r1 + r2</td>
</tr>
<tr>
<td>x := u + z</td>
<td>s6 := s4 + s5</td>
<td>r2 := r3 + r1</td>
</tr>
<tr>
<td>print x, z, u</td>
<td>print s6, s5, s4</td>
<td>print r2, r1, r3</td>
</tr>
</tbody>
</table>

Register interference graph
Problems

Architectural irregularities:

- not every physical register can be allocated to every symbolic register,
- some symbolic registers need combinations of physical registers, e.g. pairs of aligned registers.

Dedication: Some registers are dedicated for special purposes, e.g. transfer of arguments.
Extensions

Remember: An edge in the interference graph means: the connected objects cannot be allocated to the same physical register.
Assume, that physical register \( r \) cannot be allocated to symbolic register \( s \).
Solution: Add nodes for physical registers to the interference graph; connect \( r \) with \( s \).
Disadvantage: Graph now describes program-specific constraints (\( s_1 \) and \( s_2 \) live at the same time) and architecture-specific constraints (fixed-point operands should not be allocated to floating-point registers).
Separating Architectural and Program Constraints

**Machine description:**

- **Regs** register names,
- **Conflict** relation on Regs,
  
  \((r_1, r_2) \in \text{Conflict} \iff r_1 \text{ and } r_2 \text{ can not be allocated simultaneously.} \)

  Example: registers and register pairs containing them.

- **Class** Subsets of registers
  - required as operands of instructions, or
  - dedicated for special purposes of the run-time system

**Constraints on allocation** (connection between symb. and phys. registers)

- Association of register classes with symbolic registers
- Conjunction of constraints \(\implies\) intersection of register classes is new register class.
Generalized Interference Graph

extended by assoc. register classes to symbolic registers.
Assignment for $S \subseteq SymbRegs$ is $A : S \mapsto Regs$ such that $A(s) \in \text{class}(s)$ for all $s \in S$.
New local colorability criterion:
$s \in S \subseteq SymbRegs$ is locally colorable iff
for all assignments $A$ of the neighbours of $s$
there exists a register $r \in \text{class}(s)$
that does not conflict with the assignment on any neighbour.
Coloring the Generalized Interference Graph

Register classes with conflicts and generalized interference graph.

s1 and s2 are locally colorable, s3 is not. Old local-colorability criterion is satisfied, \(\text{degree} = 2\) for all three symb. registers.
Efficient Approximative Test for Local Colorability

Let $A$, $B$ be two register classes.

$$\maxConflict_A(B) = \max_{a \in A} \left| \{ b \in B | (a, b) \in \text{Conflict} \} \right|$$

maximal nume of registers in $B$, that a single register in $A$ can conflict with.

**Approximative colorability test** for $s$ with $\text{class}(s) = B$:

$$\sum_{(s, s') \in E, \text{class}(s') = A} \maxConflict_A(B) < |B|$$

Precompute $\maxConflict_A(B)$ for all $A$ and $B$, depends only on the architecture!
Example

<table>
<thead>
<tr>
<th>C \ D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

Tabulating $\maxConflicts_A(B)$

<table>
<thead>
<tr>
<th>$C \setminus D$</th>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$B$</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>