Code Generation

Mooly Sagiv
Tel Aviv University
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

July 9, 1997

“Standard” Structure

source(text)

lexical analysis(7) — finite automata

tokenized-program

syntax analysis(8) — pushdown automata

syntax-tree

semantic-analysis(9) — attribute grammar evaluators

decorated syntax-tree

optimizations(10) — abstract interpretation + transformations

intermediate rep.

code-generation(11, 12) — tree automata + dynamic programming + $\cdots$

machine-program

Challenges in real machines: Memory Hierarchy

1. Registers (Special and general purpose)
2. Cache (multi level)
3. Memory (segments, stack, heap)
4. $\cdots$

Code Generation

Real machines instead of abstract machines:

- Register machines,
- Limited resources (registers, memory),
- Fixed word size,
- Storage hierarchy,
- Intraprocessor parallelism.

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,

instruction scheduling: reordering instruction sequences to exploit intraprocessor parallelism.

Challenges in real machines: CISC vs. RISC

Complex Instruction Set Computer 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

Reduced Instruction Set Computer Alpha, MIPS, PowerPC, SPARC

- One instruction per cycle (with pipe for load/stores)
- 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)

Example: $x = y + z$

CISC/Vax

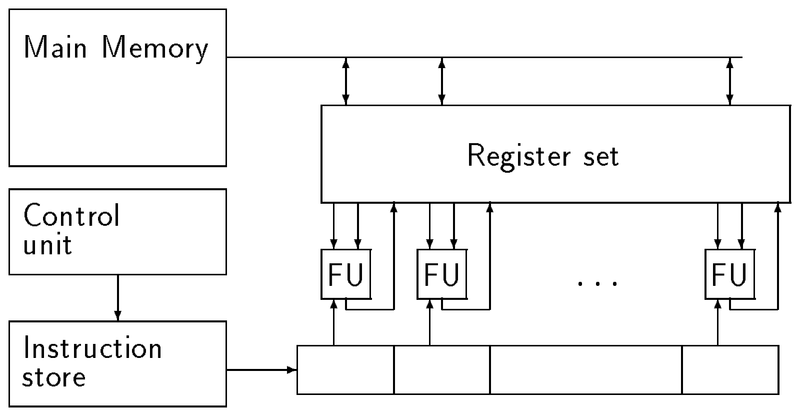
addl3 $4(fp), 6(fp), 8(fp)$

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.

		cycle						
		1	2	3	4	5	6	7
Pipe-	1	B_1	B_2	B_3	B_4			
line-	2		B_1	B_2	B_3	B_4		
stage	3			B_1	B_2	B_3	B_4	
	4				B_1	B_2	B_3	B_4

Problems:

- Pipeline hazards: Needed operand not available,
- Resource collision: e.g., bus collision,
- Hot spots,
- (Conditional) jumps.

Program Representations

- Abstract syntax tree
- Control Flow Graph
- Call Graph
- Static Single Assignment
- Program Dependence Graph
- Register Interference graph

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

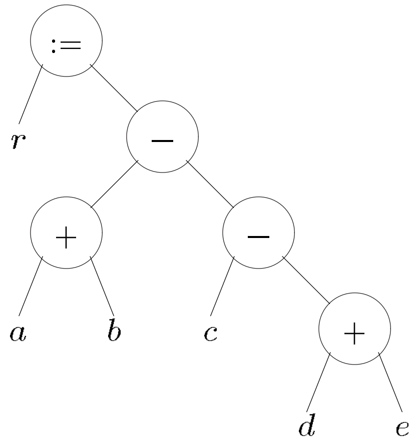
R_i	$:=$	$M[V]$	Load
$M[V]$	$:=$	R_i	Store
R_i	$:=$	$R_i\ op\ M[V]$	Compute
R_i	$:=$	$R_i\ op\ R_j$	

- Two phases:
  1. Computing register requirements,
  2. Generating code, allocating registers and temporaries.

Example Tree

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

Tree



Generated Code

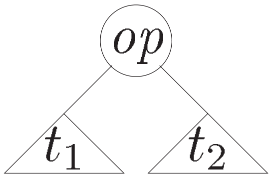
2 Registers R_0 and R_1
Two possible code sequences:

$\begin{array}{lcl} R_0 & := & M[a] \\ R_0 & := & R_0 + M[b] \\ R_1 & := & M[d] \\ R_1 & := & R_1 + M[e] \\ M[t_1] & := & R_1 \\ R_1 & := & M[c] \\ R_1 & := & R_1 - M[t_1] \\ R_0 & := & R_0 - R_1 \\ M[f] & := & R_0 \end{array}$	$\begin{array}{lcl} R_0 & := & M[c] \\ R_1 & := & M[d] \\ R_1 & := & R_1 + M[e] \\ R_0 & := & R_0 - R_1 \\ R_1 & := & M[a] \\ R_1 & := & R_1 + M[b] \\ R_1 & := & R_1 - R_0 \\ M[f] & := & R_1 \end{array}$

- Left code
  - stores result for $c - (d + 2)$ in a temporary,
  - no register available,
- Right code
  - evaluates $c - (d + 2)$ first (needs 2 registers),
  - 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,
- Let $r \geq r_1 > r_2$:
  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,
- Let $r_1 = r_2$:
  t needs $r_1 + 1$ registers to evaluate,
- Let one of t_1 or t_2 need more than r registers,
  spill to temporary required.

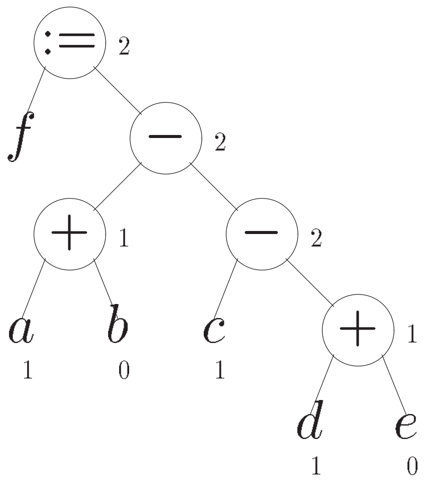
Labeling Phase

- 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:

  $$regneed(op(t_1, t_2)) = \begin{cases} max(r_1, r_2), & \text{if } r_1 \neq r_2 \\ r_1 + 1, & \text{if } r_1 = r_2 \end{cases}$$

  where $r_1 = regneed(t_1)$, $r_2 = regneed(t_2)$

Example



Generation Phase

Principle:

- Generates instruction **Op** for operator op in $op(t_1, t_2)$ after generating code for t_1 and t_2.
- Order of t_1 and t_2 depends on their register needs,
- Upon execution of the generated **Op**–instruction: value of t_1 in register,
- $RSTACK$ – available registers, init. all registers,
  Before processing t: $top(RSTACK)$ is result register for t,
  After processing t: all registers available,
  $top(RSTACK)$ is result register for t.
- $TSTACK$ – available temporaries.

Algorithm Gen_Opt_Code

Algorithm	$RSTACK$-Contents resp. result register
var $RSTACK$: **stack of register**;	
var $TSTACK$: **stack of address**;	
proc $Gen\_Code\,(t:$ **tree**);	$(R', R'', \ldots)$
var R: **register**, T: **address**;	
case t **of**	
$(leaf\ a, 1)$: (∗left leaf∗)	
$emit(top(RSTACK)\ := a)$;	result in R'
$op((t_1, r_1), (leaf\ a, 0))$: (∗right leaf∗)	
$Gen\_Code(t_1)$;	
$emit(top(RSTACK)\ :=\ top(RSTACK)$ **Op** $a)$;	result in R'

$op((t_1, r_1), (t_2, r_2))$:	$(R', R'', \ldots)$
cases	
$r_1 < \ min(r_2, r)$:	$(R', R'', \ldots)$
begin	
exchange$(RSTACK)$;	$(R'', R', \ldots)$
$Gen\_Code(t_2)$;	result in R''
$R :=$ **pop**$(RSTACK)$;	$(R', \ldots)$
$Gen\_Code(t_1)$;	result in R'
$emit(top(RSTACK) \ := \ top(RSTACK)$ **Op** $R)$;	result in R'
push$(RSTACK, R)$;	$(R'', R', \ldots)$
exchange$(RSTACK)$;	$(R', R'', \ldots)$
end ;	

$r_1 \geq r_2 \wedge r_2 < r$:	$(R', R'', \ldots)$
begin	
$Gen\_Code(t_1)$;	result in R'
$R :=$ **pop**$(RSTACK)$;	$(R'', \ldots)$
$Gen\_Code(t_2)$;	result in R''
$emit(R\ :=\ R$ **Op** $top(RSTACK))$;	result in R'
push$(RSTACK, R)$;	$(R', R'', \ldots)$
end ;	

$r_1 \geq r \wedge r_2 \geq r$:	$(R', R'', \ldots)$
begin	
$Gen\_Code(t_2)$;	result in R'
$T :=$ **pop**$(TSTACK)$;	
$emit(M[T] := top(RSTACK))$;	result in $M[T]$
$Gen\_Code(t_1)$;	result in R'
$emit(top(RSTACK) := top(RSTACK)$ **Op** $M[T])$;	result in R'
push$(TSTACK, T)$;	
end ;	
endcases	
endcase	
endproc	

Register Allocation and Instruction Selection by Dynamic Programming

- More complex architecture,
  - r general purpose registers $R_0, \ldots, R_{r-1}$,
  - Instruction formats:

R_i	$:=$	e	Compute
R_i	$:=$	$M[V]$	Load
$M[V]$	$:=$	R_i	Store

    e term with registers and memory cells,
    costs associated with each instruction.

- Goal: Generate cheapest instruction sequence using no more than r registers.

- Use instruction selection techniques with tree parsing,
  compute cheapest derivation.

Dynamic Programming

- Canonical recursive solution, exponential complexity,
- Compute bottom up tabulating partial solutions.
- Associate cost vector $K[0..r]$ with each node n,
  K[0] cheapest costs for computing t/n into a temporary,
  K[i] cheapest costs computing t/n into a register using i registers.
- Compute cost vector at node 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 $C[j], j > 0$?
  Any combination of generated code for subtrees not needing more than j registers.
- Extract cheapest instruction sequence in a second pass.

Global Register Allocation

- Register allocation across whole procedures/programs,
- **Symbolic Registers** for intermediate results and modified variables,
- Two symbolic registers **collide** if their contents are "live" at the same time,
- Colliding symbolic registers cannot be allocated to the same real register,
- Goal: allocate the (unbounded number of) symbolic registers to the fixed number of real registers without collision.

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 an access to the corresponding variable or a use of the intermediate value,

  In the following, "paths" are paths in the control flow graph.
- A **definition–path** of r to program point p is a path from the starting point to p with a definition of r,
- A **use–path** is a definition free path containing a use of r,
- Symbolic register r is **live** at program point p if exists a definition–path to p and a use–path from p,
- The **life span** of r is the set of all program points, on which r 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.

- The **register collision graph**:

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

- Allows viewing register allocation as graph colouring.

Graph Colouring

- k real registers available,
- Solve k–colouring problem,
- NP–complete for $k > 2$,
- Use heuristics.

Algorithm:
Case:
G contains a node n with degree $< k$.
n and its neighbours can be coloured with different colours.
Remove n from G, decreasing the size of G.
G is k-colourable if we arrive at the empty graph.

Case:
G not empty and exists no node with degree $< k$,
use heuristics to select one node to remove next.
Removal corresponds to spilling,

Program has to be modified, inserting spills at definitions and loads at uses.

Possible heuristics:

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

Example

Input-program

$$
\begin{aligned}
&x := 1;\\
&y := 2;\\
&z := x + y;\\
&x := x + z;
\end{aligned}
$$

Register collision graph

Instruction Scheduling

- Reorders instruction stream as generated by instruction selection,
- Goal: Exploitation of intraprocessor parallelism,
  - Filling very long instruction words (VLIWs), or
  - Avoiding resource collisions and pipeline hazards.
- Must be semantics preserving,
- Basis: Program dependences.

Program Dependences

- Relation between definitions and uses of resources (program variables, memory cells or blocks, symbolic or real registers),

- Here mainly machine resources, i.e. registers, memory cells, status words.

Definitions :

- modifications of register contents by loads or operations, pre-, postincrement/decrement,
- setting carry, overflow, condition bits in status words,
- storing values in memory cells,
- modifying registers as side effects of e.g. pop, push.

Uses :

- Using register contents in operations and for addressing,

- Storing register contents,
- Loading contents of memory cells,
- Testing the program status word.

Types of Dependences

A sequence of definitions ($X :=$) and uses ($:= X$) of X:

$$
\begin{array}{lll}
a: & X & := \\
b: & X & := \\
c: & & := X \\
d: & X & :=
\end{array}
$$

Output dependences: Dependence between definitions, e.g., b on a,

True dependence: Uses on definitions, e.g., c on b,

Antidependence: Definitions on uses, e.g., d on c.

Objects of dependences: Machine resources.