Register Assignment and Code Generation

Code Generation

Assume that we have a flow graph. We want to generate machine instructions. We have to

- Remove the ϕ functions.
- Choose registers for the intermediate results.
- Select machine instructions.

Register Allocation

The final aim is to generate efficient machine code.

Processors have local variables, called registers which can be read and written very efficiently. (Typically one clock cycle.)

Modern memory is much slower than modern processors. (333 DDR, vs. 1.8 Ghz.) There is roughly a factor of 6 between them.

In addition, storing a variable in memory requires address calculation, which adds to the processing time.

Register Allocation (2)

```
move D0, D1
```

is quicker than:

```
move $4(A7), $8(A7)
```

(Assuming that `A7` is the stack register.)

Register Allocation (3)

Unfortunately, processors do not have many registers.

I think that the problem is not that there is no space for registers on the chip.

The problem is that registers spoil the opcode space.

(But this was in 1992, maybe times have changed.)

Register Allocation for Tree-Like Expressions

For simple, tree-like expressions, register allocation is easy:

How many registers are required to evaluate: $a + (b \times (c + d))$?

Tree-Like Expressions

We give a simple, recursive algorithm for determining how many registers are needed for evaluation a tree-like expression.

- For a constant c or variable v, $\#c = \#v = 1$. (Assuming that there is no address calculation involved. Otherwise, one has to include the address calculation in the tree.)
- If both t_1, t_2 are variables or constants, and there exists a simple machine instruction for computing $f(t_1, t_2)$, then $\#f(t_1, t_2) = 1$.
- If t_1 is a variable or constant, and there exists a simple machine instruction for computing $x = f(t_1, x)$, then $\#f(t_1, t_2) = \#t_1$.
- If t_2 is a variable or constant, and there exists a simple machine instruction for computing $x = f(x, t_2)$, then $\#f(t_1, t_2) = \#t_2$.

Tree-Like Expressions (2)

- If both of t_1, t_2 are not variable or constant, and $\#t_1 \neq \#t_2$, then $\#f(t_1, t_2) = \max(\#t_1, \#t_2)$.
- If both $\#t_1 = \#t_2$ are not variable or constant, then $\#f(t_1, t_2) = 1 + \#t_1$.

(The last two steps can be extended to more arguments in the case where f is associative.)

Tree-Like Expressions (3)

In case that $\#t_1 < \#t_2$, the following code can be generated:

- Compute t_2 in R_i;
- Compute t_1 in R_j;
- $R_j = f(R_j, R_i)$ or $R_i = f(R_j, R_i)$;

Non-Tree-Like Expressions

Unfortunately, reality is non-tree like. (This is what redundancy analysis is based on.)

The algorithm on the previous slides cannot be used in practice.

But there is an important message that should be remembered: Complicated expressions (of which one expects that they will need many registers) should be evaluated first.

Choosing the Order of Evaluation

Remember the slides on optimization:

The normalization algorithm uses a set of rewrite rules $\mathcal{R}$ for storing the normalizations of expressions. After analysis, it generates a minmal sequence of assignments that compute the output of the block.

This minimal sequence of assignments should be generated in such an order, that the most complicated expressions are calculated first.

Register Assignment

Definition Assume that $\mathcal{G}$ is a flow graph. The conflict graph (V, E) of $\mathcal{G}$ is defined as

- V the set of variables that occur in $\mathcal{G}$.
- $E = \{(v_1, v_2) \mid v_1 \text{ and } v_2 \text{ are in conflict with each other .}\}$

Let $R = \{R_0, \ldots, R_7\}$ (or something similar.)

A register assignment is a function $\phi : V \to R$, s.t.
$\forall e_1, e_2, \quad (e_1, e_2) \in E \Rightarrow \phi(e_1) \neq \phi(e_2)$.

Question: Find ϕ with smallest range.

Problem is NP-complete. (Because it is equivalent to graph colouring.)

Spilling

If no ϕ can be found, then there are not enough registers. (Or the heuristic failed.)

1. Assume that $R = \{R_0, \ldots, R_7\}$. Try to find a register assignment $\phi : V \to R$.
2. If no ϕ can be found, then select one $v \in V$. (Typically with the biggest amount of conflicts. One could also try to estimate how often the variable is used, etc.) Store this variable in memory. Compute the new graph. Go to the first step.

Moving local variables to memory is called spilling.

Code Generation

We assume that we have somehow managed to assign registers to the intermediate values.

Some machine instructions are quite complicated, and it would be nice if the compiler could use these instructions.

Code Generation (2)

Translation 1:

```
move $4(A6), (A5) ++
```

Translation 2:

```
move A6, A4
add 4, A4
move (A4), R0
move R0, (A5)
add 1, A5
```

It is not hard to guess which translation will be more efficient.

Code Generation (3)

As usual, a lot is written, but it is hard to find something concrete in the literature. I imagine that it works as follows:

Determine which registers are used only once. Put some mark on these registers.

(In the previous example, `A4` would be marked.)

This marking is essential, because marked registers can be skipped in the translation. If for example on the previous slide, the value `A6+4` in `A4` is reused later, then the first translation is impossible.

The best approach is to have the SIMP-function and the $\mathcal{R}$-datastructure generate the markings together with the assignments.

Instruction Selection

Instructions can be represented by tree rewrite rules:

- `move R0, R1` $R_1 \rightarrow R_0,$
- `move 4(A5), R1` $R_1 \rightarrow M(+(4, A_5)).$
- `add 4(A5), R1` $R_1 \rightarrow +(M(+(4, A_5)), R_1)$
- `move R0, 4(A5)` $W(R_0, +(4, A_5))$

Instruction Selection

Suppose one has the sequence:
$A_4^* = A_6^* + 4;$
$R_0^* = M(A_4^*);$
$W(R_0^*, A_5);$
$A_5 = A_5 + 1;$

Find a maximal sequence of assignments that assigns marked variables.

We now have a tree that assigns a value to a non-marked variable. (That is a variable that we want to remember the result of.)

Let R be the set of marked registers. These are the registers that we are allowed to store intermediate results in. For each node n, we will compute a set $C(n)$, as explained on the next slide.

Instruction Selection

For every node, (except those with label W), the set $C(n)$ is a set of pairs (R_i, x), where R_i is a register and x is a natural number. The meaning of $(R_i, x) \in C(n)$ is: Let t be the subtree on n. It is possible to have a sequence of instructions that puts t in register R_i and which has computational cost x.

Leaves (containing a register R) are initialized with $C(n) = \{(R, 0\}$. (This means that we get the premisses for free.)

Inner nodes are initialized with $\emptyset$.

After that, tree matching is used to compute other $C(n)$.