Improving the Speed of Instruction Selection

As we have presented it, instruction selection looks rather slow—for each node in the IR tree, we must match productions, compare costs, and select least-cost productions.

Since compilers routinely generate program with tens or hundreds of thousands of instructions, doing a lot of computation to select one instruction (even if it's the *best* instruction) could be too slow.

Fortunately, this need not be the case.

Instruction selection using BURS can be made *very* fast.

Adding States to BURG

We can *precompute* a set of *states* that represent possible labelings on IR tree nodes. A table of node names and subtree states then is used to select a node's state. Thus labeling becomes nothing more than repeated table lookup.

For example, we might create a state s0 that corresponds to the labeling {Reg:R1:0, Adr:R2:0}.

A state selection function, *label*, defines label(r) = s0. That is, whenever r is matched as a leaf, it is to be labeled with s0.

If a node is an operator, label uses the name of the operator and the labeling

assigned to its children to choose the operator's label. For example,
   label(+,s0,s1)=s2
says that a + with children labeled as s0 and s1 is to be labeled as s2.

In theory, that's all there is to building a fast instruction selector.

We generate possible labelings, encode them as states, and table all combinations of labelings.

But,

how do we know the set of possible labelings is even finite?

In fact, it isn't!

Normalizing Costs

It is possible to generate states that are identical except for their costs.

For example, we might have
 s1 = {Reg:R1:0, Adr:R2:0},
 s2 = {Reg:R1:1, Adr:R2:1},
 s3 = {Reg:R1:2, Adr:R2:2}, etc.

Here an important insight is needed—the *absolute* costs included in states aren't really essential. Rather *relative* costs are what is important. In s1, s2, and s3, Reg and Adr have the same cost. Hence the same decision in choosing between Reg and Adr will be made in all three states.

We can limit the number of states needed by *normalizing* costs within states so that the lowest cost choice is always 0, and other costs are differences (deltas) from the lowest cost choice.

This observation keeps costs bounded within states (except for pathologic cases).

Using additional techniques to further reduce the number of states needed, and the time needed to generate them, fast and compact BURS instruction selectors are achievable. See

“Simple and Efficient BURS Table Generation,” T. Proebsting, 1992 PLDI Conference.

Example

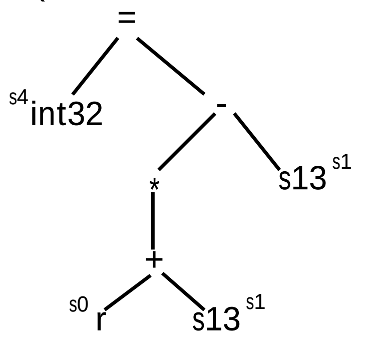
State	Meaning
s0	{Reg:R1:0, Adr:R2:0}
s1	{Imm:R4:0, Reg:R5:1}
s2	{adr:R3:0}
s3	{Reg:R9:0}
s4	{UInt:R0:0}
s5	{Reg:R8:0}
s6	{Void:R10:0}
s7	{Reg:R7:0}

Node	Left Child	Right Child	Result
r			s0
s13			s1
int32			s4
+	s0	s1	s2
*	s2		s3
-	s3	s1	s5
-	s1	s3	s7
=	s4	s5	s6

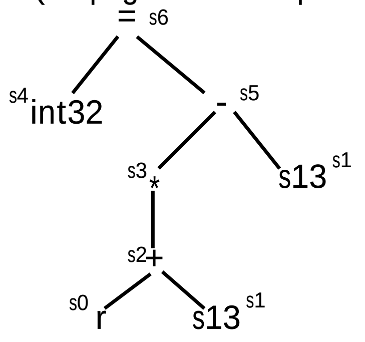
We start by looking up the state assigned to each leaf. We then work upward, choosing the state of a parent based on the parent’s kind and the states assigned to the children. These are all table lookups, and hence very fast.

At the root, we select the nonterminal and production based on the state assigned to the root (any entry with 0 cost). Knowing the production used at the root tells us the nonterminal used at each child. Each state has only one entry per nonterminal, so knowing a node’s state and the nonterminal used to generate it immediately tells us the production used. Hence identifying the production used for each node is again very fast.

Step 1 (Label leaves with states):

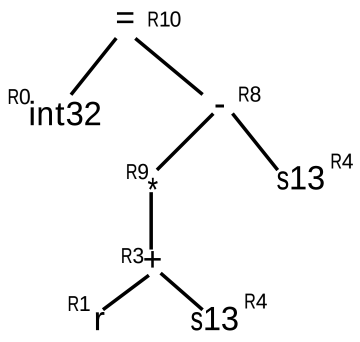


Step 2 (Propagate states upward):



Step 3 (Choose production used at root): R10.

Step 4 (Propagate productions used downward to children):



Code Generation for x86 Machines

The x86 presents several special difficulties when generating code.

- There are only 8 architecturally visible registers, and only 6 of these are allocatable. Deciding what values to keep in registers, and for how long, is a difficult, but crucial, decision.
- Operands may be addressed directly from memory in some instructions. Such instructions avoid using a register, but are longer and add to I-cache pressure.

In “Optimal Spilling for CISC Machines with Few Registers,” Appel and George address both of these difficulties.

They use Integer Programming techniques to directly and optimally solve the crucial problem of deciding which live ranges are to be register-resident at each program point. Stores and loads are automatically added to split long live ranges.

Then a variant of Chaitin-style register allocation is used to assign registers to live ranges chosen to be register-resident.

The presentation of this paper, at the 2001 PLDI Conference, is at

`www.cs.wisc.edu/~fischer/cs701/cisc.spilling.pdf`