Productions

Rule #	Production	Cost	SPARC Code
R0	UInt → Int32	0	
R1	Reg → r	0	
R2	Adr → r	0	
R3	$\text{Adr} \rightarrow \overset{+}{\text{Reg}\quad\text{Imm}}$	0	
R4	Imm → s13	0	
R5	Reg → s13	1	`mov s13,Reg`

R6	Reg → int32	2	`sethi %hi(int32),%g1`
			`or %g1, %lo(int32),Reg`

R7	$\text{Reg} \rightarrow \overset{-}{\text{Reg}\quad\text{Reg}}$	1	`sub Reg,Reg,Reg`

Rule #	Production	Cost	SPARC Code
R8	$Reg \rightarrow \begin{tikzcd} & - \arrow[dl, no head] \arrow[dr, no head] & \\ Reg & & Imm \end{tikzcd}$	**1**	`sub Reg,Imm,Reg`
R9	$Reg \rightarrow \begin{tikzcd} * \arrow[d, no head] \\ Adr \end{tikzcd}$	**1**	`ld [Adr],Reg`

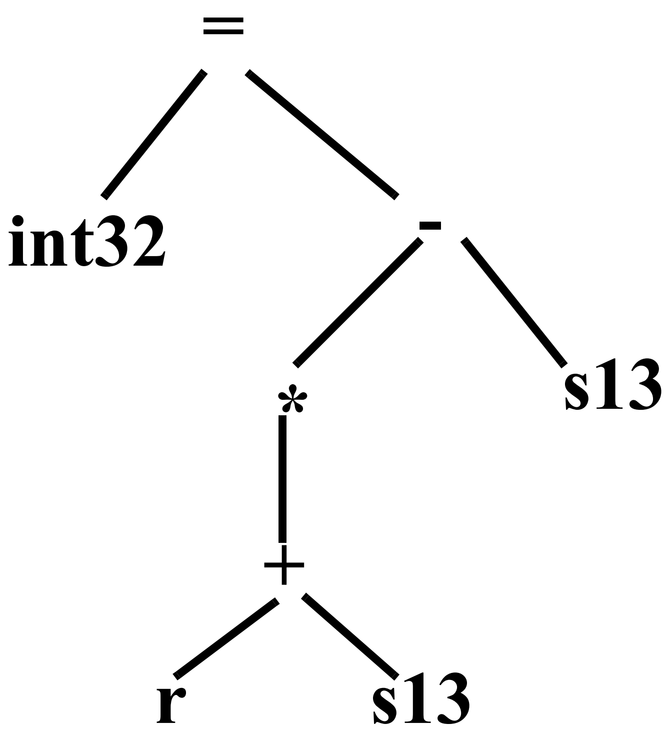
R10	$Void \rightarrow \begin{tikzcd} & = \arrow[dl, no head] \arrow[dr, no head] & \\ UInt & & Reg \end{tikzcd}$	**2**	`sethi`
			`%hi(UInt),%g1`
			`st Reg,`
			`[%g1+%lo(Uint)]`

Example

Let's look at instruction selection for

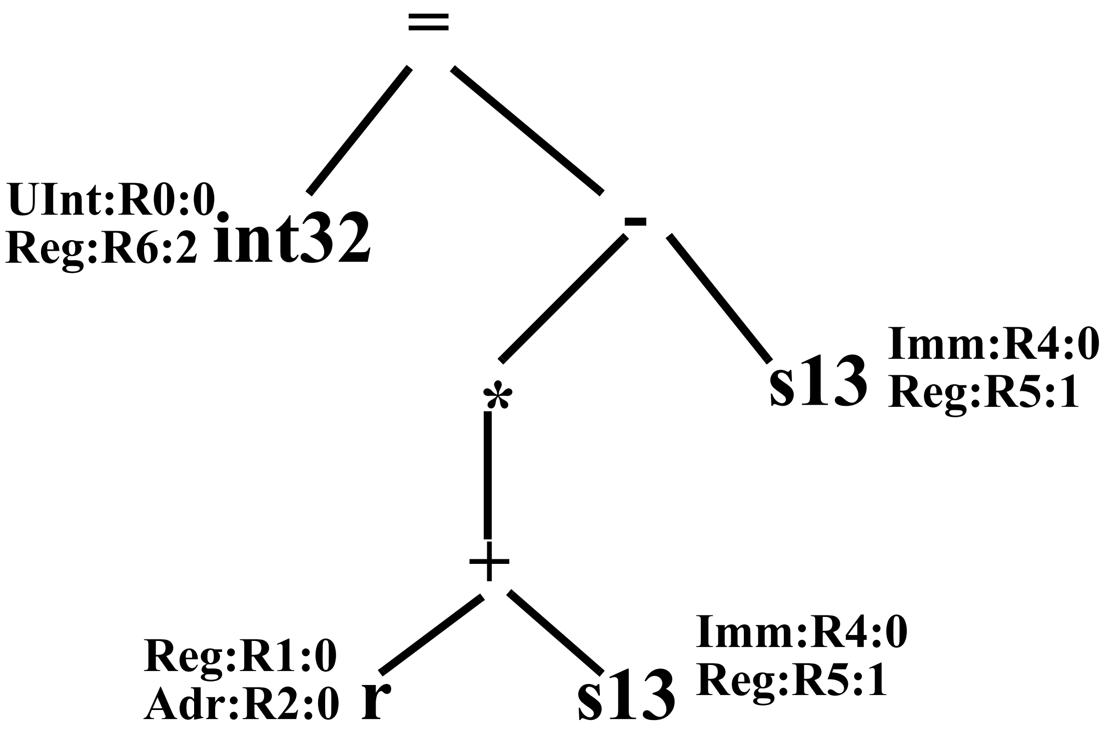
```
a = b - 1;
```

where `a` is a global int, accessed with a 32 bit address and `b` is a local int, accessed as an offset from the frame pointer.



We match tree nodes *bottom-up*. Each node is labeled with the nonterminals it can be reduced to, the production used to produce the nonterminal, and the cost to generate the node (and its children) from the nonterminal.

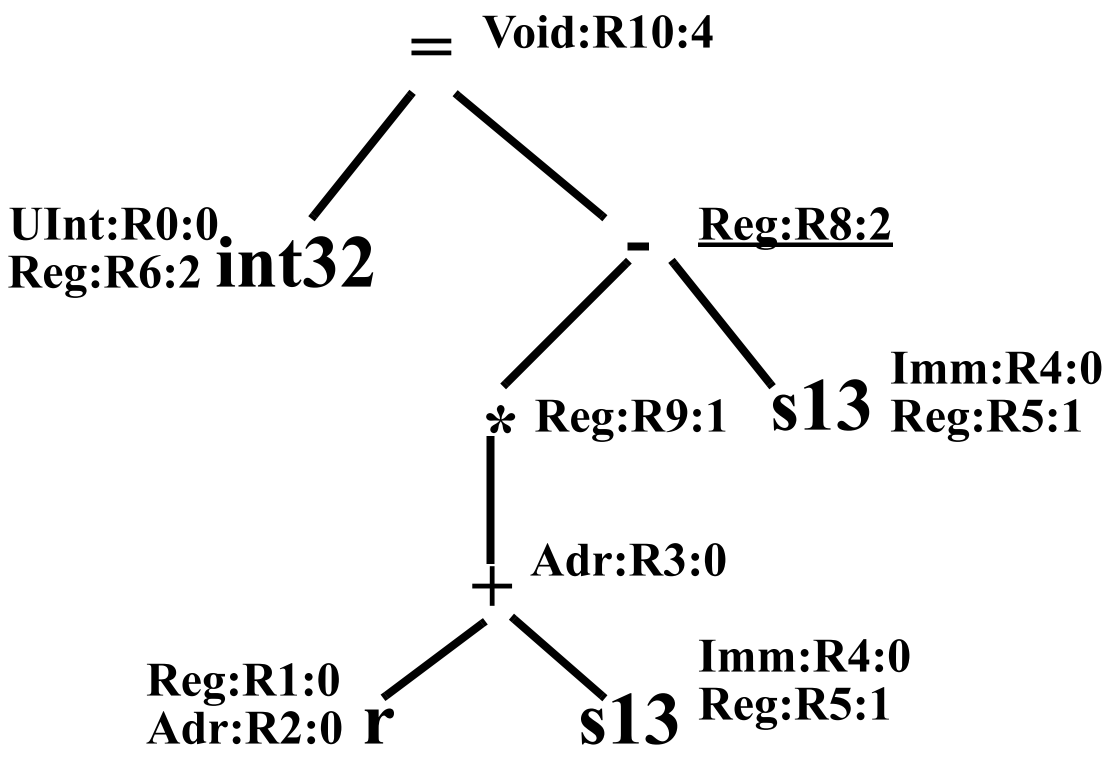
We match leaves first:



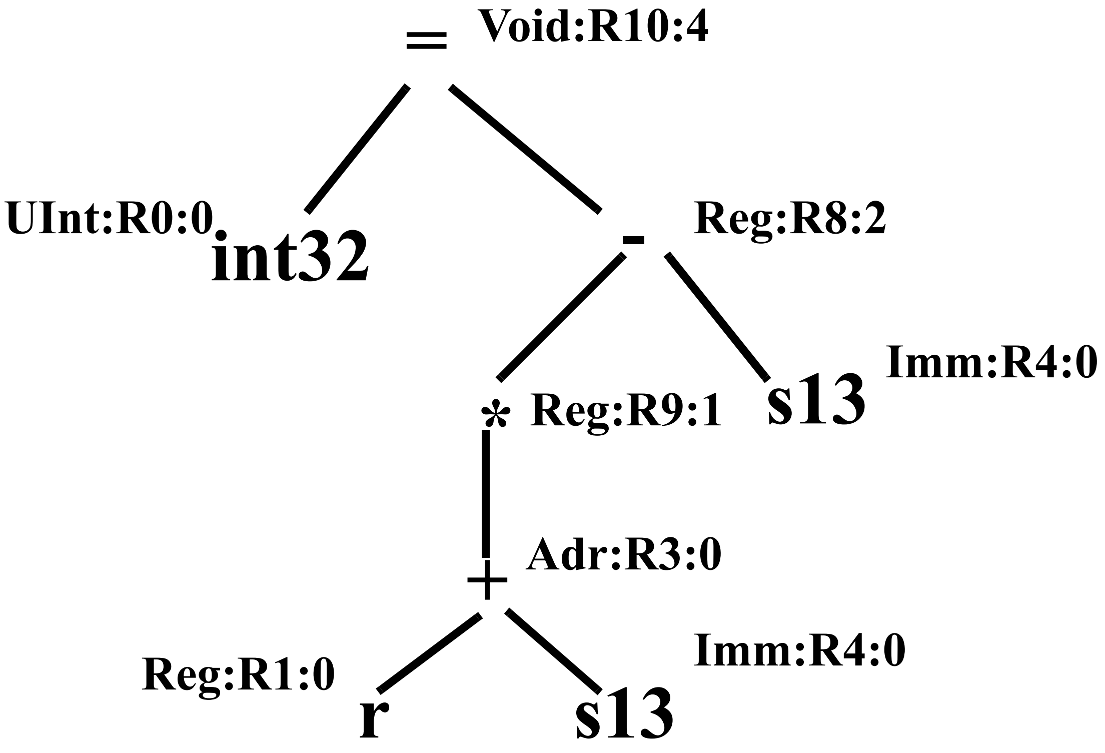
We now work *upward*, considering operators whose children have been labeled. Again, if an operator can be generated by a nonterminal, we mark the operator with the nonterminal, the production used to generate the operator, and the total cost (including the cost to generate all children).

If a nonterminal can generate the operator using more than one production, the *least-cost* derivation is chosen.

When we reach the root, the nonterminal with the lowest overall cost is used to generate the tree.

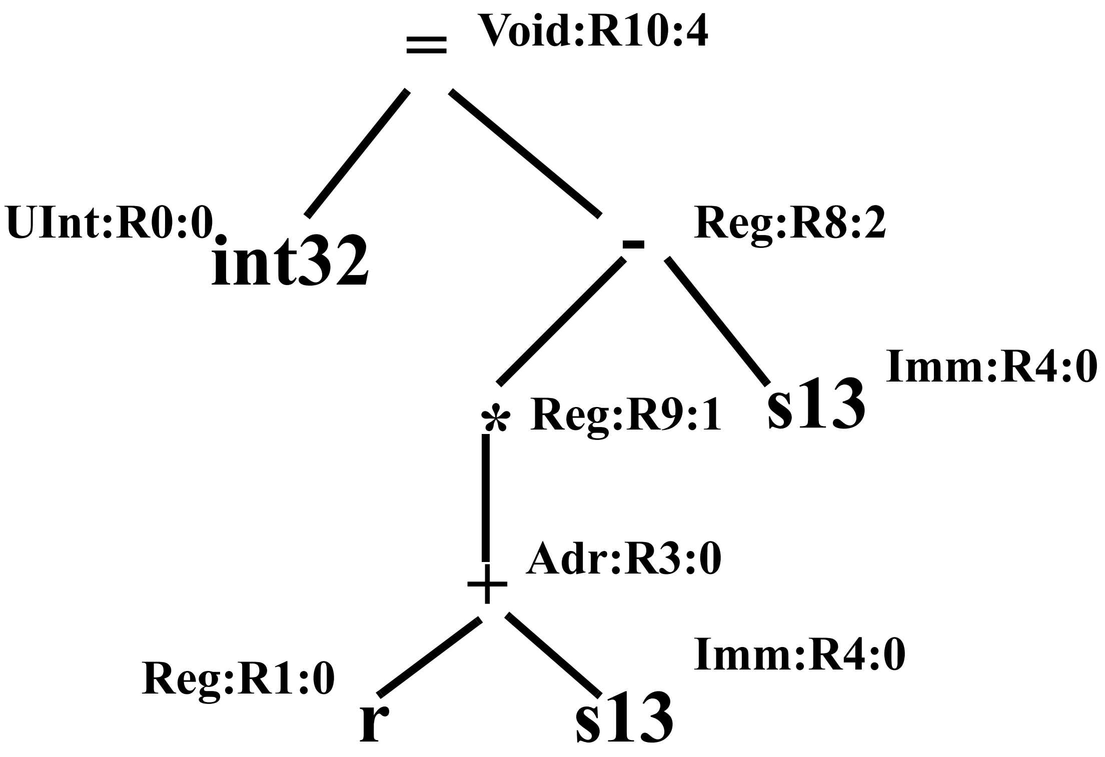

=
Void:R10:4
UInt:R0:0
Reg:R6:2
int32
-
Reg:R8:2
*
Reg:R9:1
s13
Imm:R4:0
Reg:R5:1
+
Adr:R3:0
Reg:R1:0
Adr:R2:0
r
s13
Imm:R4:0
Reg:R5:1

Note that once we know the production used to generate the root of the tree, we know the productions used to generate each subtree too:



We generate code by doing a depth-first traversal, generating code for a production after all the production's children have been processed.

We need to do register allocation too; for our example, a simple on-the-fly generator will suffice.



```
ld    [%fp+b],%l0
sub   %l0,1,%l0
sethi %hi(a),%g1
st    %l0,[%g1+%lo(a)]
```

Had we translated a slightly difference expression,

```
a = b - 1000000;
```

we would *automatically* get a different code sequence (because 1000000 is an int32 rather than an s13):

```
ld    [%fp+b],%l0
sethi %hi(1000000),%g1
or    %g1,%lo(1000000),%l1
sub   %l0,%l1,%l0
sethi %hi(a),%g1
st    %l0,[%g1+%lo(a)]
```

Adding New Rules

Since instruction selectors can be automatically generated, it’s easy to add “extra” rules that handle optimizations or special cases.

For example, we might add the following to handle addition of a left immediate operand or subtraction of 0 from a register:

Rule #	Production	Cost	SPARC Code
R11	$\begin{tikzcd} & & + \arrow[dl, dash] \arrow[dr, dash] & \\ \text{Reg} \rightarrow & \text{Imm} & & \text{Reg} \end{tikzcd}$	1	`add Reg,Imm,Reg`
R12	$\begin{tikzcd} & & - \arrow[dl, dash] \arrow[dr, dash] & \\ \text{Reg} \rightarrow & \text{Reg} & & \text{Zer} \end{tikzcd}$	0	

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

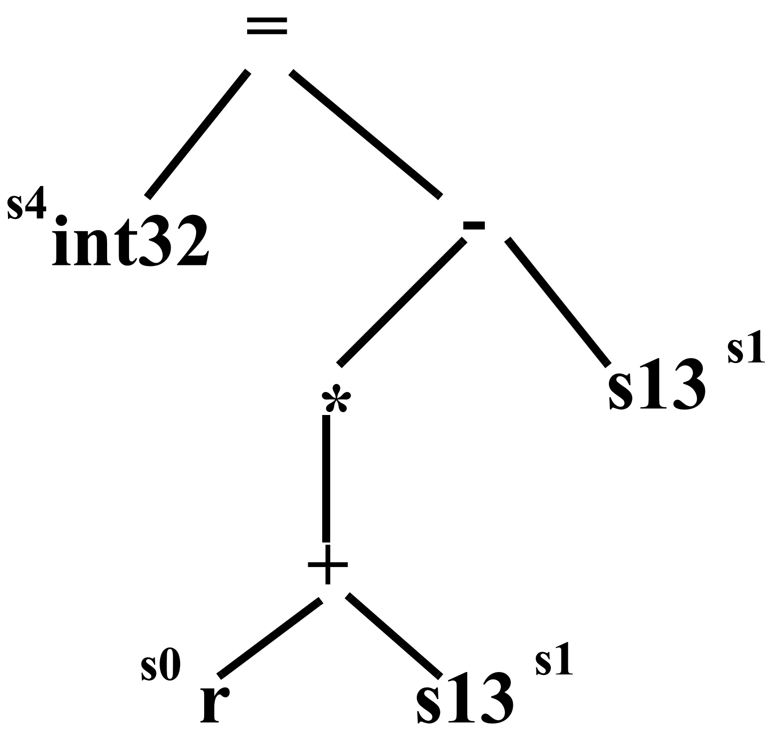
State	Meaning
s0	{Reg:R1:0, Adr:R2:0}
s1	{Imm:R4:0, Reg:R5:1}
s2	{adr:R3:0}
s3	{Reg:R9:0}
s4	{UInt:R0:0}
s5	{Reg:R8:0}
s6	{Void:R10:0}
s7	{Reg:R7:0}

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

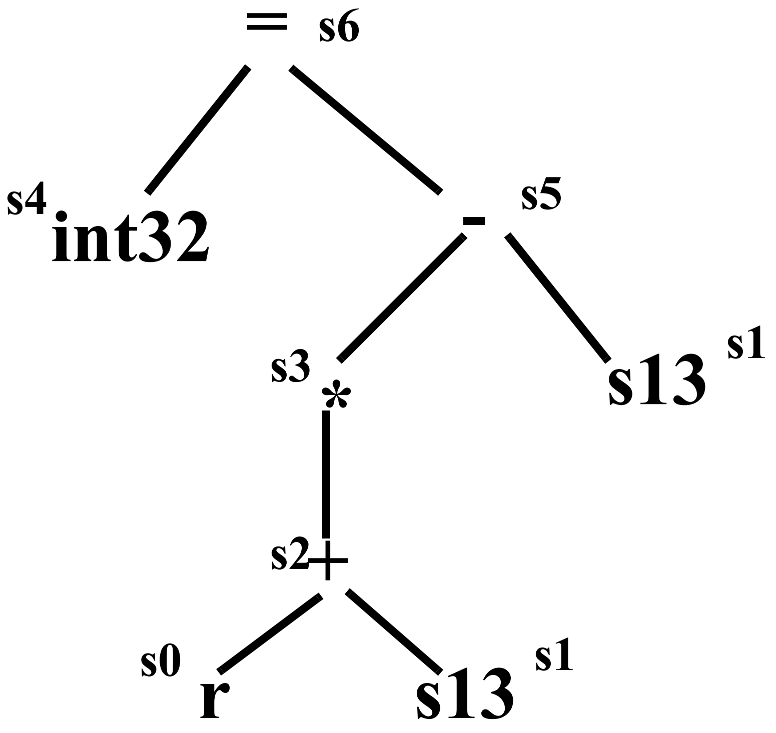
We start by looking up the state assigned to each leaf. We then work upward, choosing the state of a parent based on the parent's kind and the states assigned to the children. These are all table lookups, and hence very fast.

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):

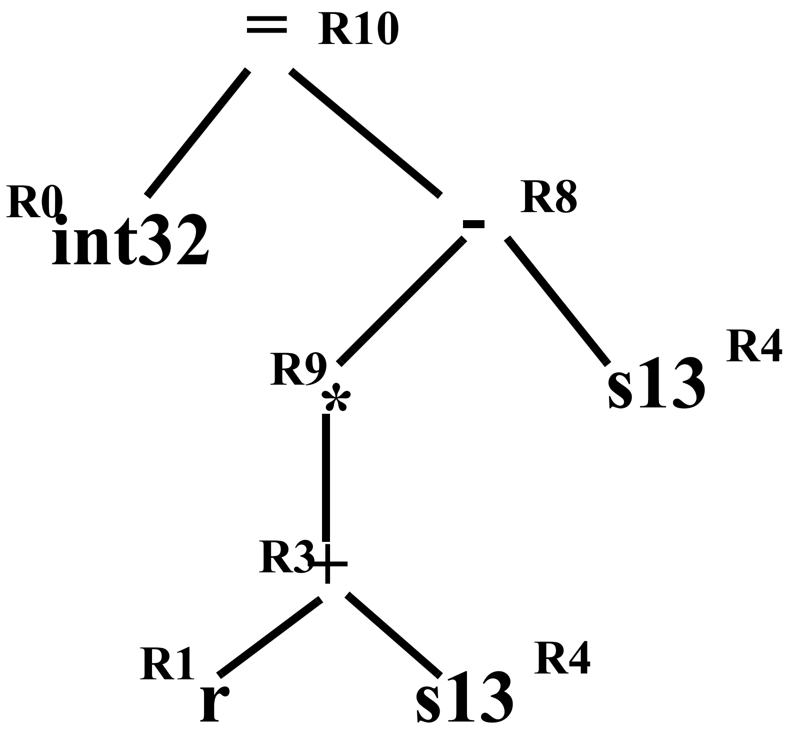


Step 2 (Propagate states upward):



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`

Optimistic Coalescing

Given R allocatable registers, Appel and George guarantee that no more than R live ranges are marked as register resident.

This doesn't always guarantee that an R coloring is possible.

Consider the following program fragment:

```
x=0;
while (...) {
  y = x+1;
  print(x);
  z = y+1;
  print(y);
  x = z+1;
  print(z);
}
```

At any given point in the loop body only 2 variables are live, but 3 registers are needed (`x` interferes with `y`, `y` interferes with `z` and `z` interferes with `x`).

We know that we have enough registers to handle all live ranges marked as register-resident, but we may need to "shuffle" register allocations at certain points.

Thus at one point x might be allocated R1 and at some other point it might be placed in R2. Such shuffling implies register to register copies, so we'd like to minimize their added cost.

Appel and George suggest allowing changes in register assignments between program points. This is done by creating multiple variable names for a live range (`x`$_1$, `x`$_2$, `x`$_3$, ...), one for each program point. Variables are connected by assignments between points. Using coalescing, it is expected that most of the assignments will be optimized away.

Using our earlier example, we have the following code with each variable expanded into 3 segments (one for each assignment). Copies of dead variables are removed to simplify the example:

```
x₃=0;
while (...) {
  x₁ = x₃;
  y₁ = x₁+1;
  print(x₁);
  y₂ = y₁;
  z₂ = y₂+1;
  print(y₂);
  z₃ = z₂;
  x₃ = z₃+1;
  print(z₃);
}
```

Now a 2 coloring is possible:

x_1: R1, y_1: R2

z_2: R1, y_2: R2

z_3: R1, x_3: R2

(and only $x_1 = x_3$ is retained).

Appel and George found that iterated coalescing wasn’t effective (too many copies, most of which are useless).

Instead they recommend *Optimistic Coalescing*. **The idea is to first do Chaitin-style reckless coalescing of all copies, even if colorability is impaired.**

Then we do graph coloring register allocation, using the cost of copies as the “spill cost.” As we select colors, a coalesced node that can’t be colored is simply split back to the original source and target variables. Since we always limit the number of live ranges to the number of colors, we know the live ranges must be colorable (with register to register copies sometimes needed).

Using our earlier example, we initially merge x_1 and x_3, y_1 and y_2, z_2 and z_3. We already know this can't be colored with two registers. All three pairs have the same costs, so we arbitrarily stack x_1-x_3, then y_1-y_2 and finally z_2-z_3.

When we unstack, z_2-z_3 gets R1, and y_1-y_2 gets R2. x_1-x_3 must be split back into x_1 and x_3. x_1 interferes with y_1-y_2 so it gets R1. x_3 interferes with z_2-z_3 so it gets R2, and coloring is done.

x_1: R1, y_1: R2

z_2: R1, y_2: R2

z_3: R1, x_3: R2

Procedure & Code Placement

We have seen many optimizations that aim to reduce the number of instructions executed by a program.

Another important class of optimizations derives from the fact that programs often must be paged in virtual memory and almost always are far bigger then the I-cache.

Hence how procedures and basic blocks are placed in memory is important. Page faults and I-cache misses can be *very* costly.

In “Profile Guided Code Positioning,” Pettis and Hansen explore three kinds of code placement optimizations:

1. Procedure Positioning.

   Try to keep procedures that often call each other close together.

2. Basic Block Positioning.

   Try to place the most frequently executed series of basic blocks “in sequence.”

3. Procedure Splitting.

   Place infrequently executed “fluff” in a different memory area than heavily executed code.

Procedure Placement

Procedures (and classes in Java) are normally separately compiled. They are then placed in memory by a linker or loader in an arbitrary order.

This arbitrary ordering can be problematic:

If A calls B frequently, and A and B happen to be placed far apart in memory, the calls will cross page boundaries and perhaps cause I-cache conflicts (if code in A and B happen to map to common cache locations).

However,
if A and B are placed close together in memory, they may both fit on the same page *and* **fit into the I-cache without conflicts.**

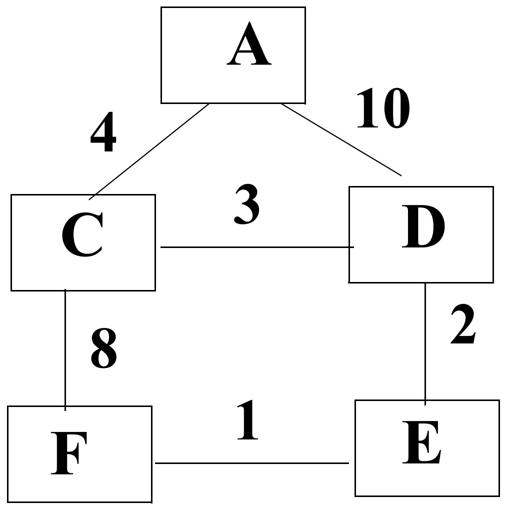
Pettis & Hansen suggest a "closest is best" procedure placement policy.

That is, they recommend that we place procedures that often call each other as close together as possible.

How?

First, we must obtain dynamic call frequencies using a profiling tool like gprof or qpt.

Given call frequencies, we create a call graph, with edges annotated with call frequencies:

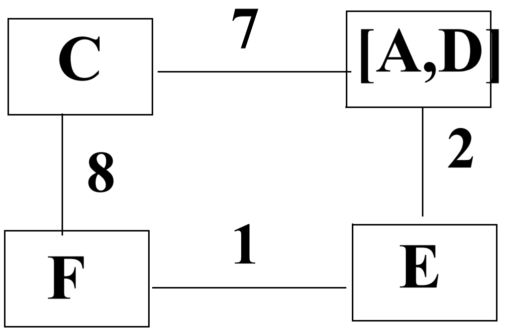


Group Procedures by Call Frequency

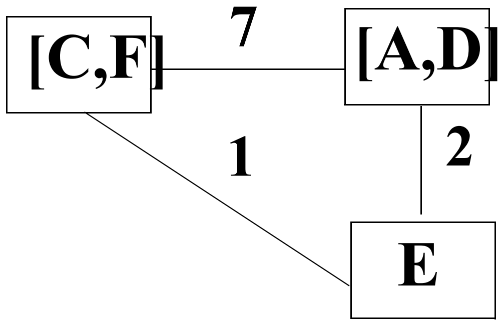
We find the pair of procedures that call each other most often, and group them for contiguous positioning.

The notation [A,D] means A and D will be adjacent (either in order A-D or D-A).

The two procedures chosen are combined in the call graph, which is simplified (much like move-related nodes in an interference graph):



Now C and F are grouped, without their relative order set (as yet):



Next [A,D] and [C,F] are to be joined, but in what exact order?

Four orderings are possible:

A-D-C-F ≡ F-C-D-A

A-D-F-C ≡ C-F-D-A

D-A-C-F ≡ F-C-A-D

D-A-F-C ≡ C-F-A-D

Are these four orderings equivalent?

No—Look at the original call graph. At the boundary between [A,D] and [C,F], which of the following is best:

D-C (3 calls),

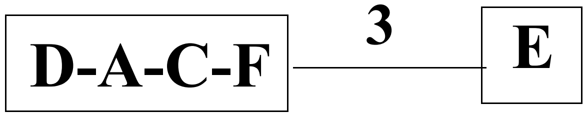
D-F (0 calls)

A-C (4 calls)

A-F (0 calls)

A-C has the highest call frequency, so we choose D-A-C-F.

Finally, we have:



We place E near D (call frequency 2) rather than near F (call frequency 1).

Our final ordering is
E-D-A-C-F.

Basic Block Placement

We often see conditionals of the form

```
if (error-test)
   {Handle error case}
{Rest of Program}
```

Since error tests rarely succeed (we hope!), the error handling code "pollutes" the I-cache.

In general, we'd like to order basic blocks not in their order of appearance in the source program, but rather in order of their execution along frequently executed paths.

Placing frequently executed basic blocks together in memory fills the I-cache nicely, leads to a smaller

working set *and* makes branch prediction easier.

Pettis & Hansen suggest that we profile execution to determine the frequency of inter-block transitions. We then will group blocks together that execute in sequence most often.

At the start, all basic blocks are grouped into singleton chains of one block each.

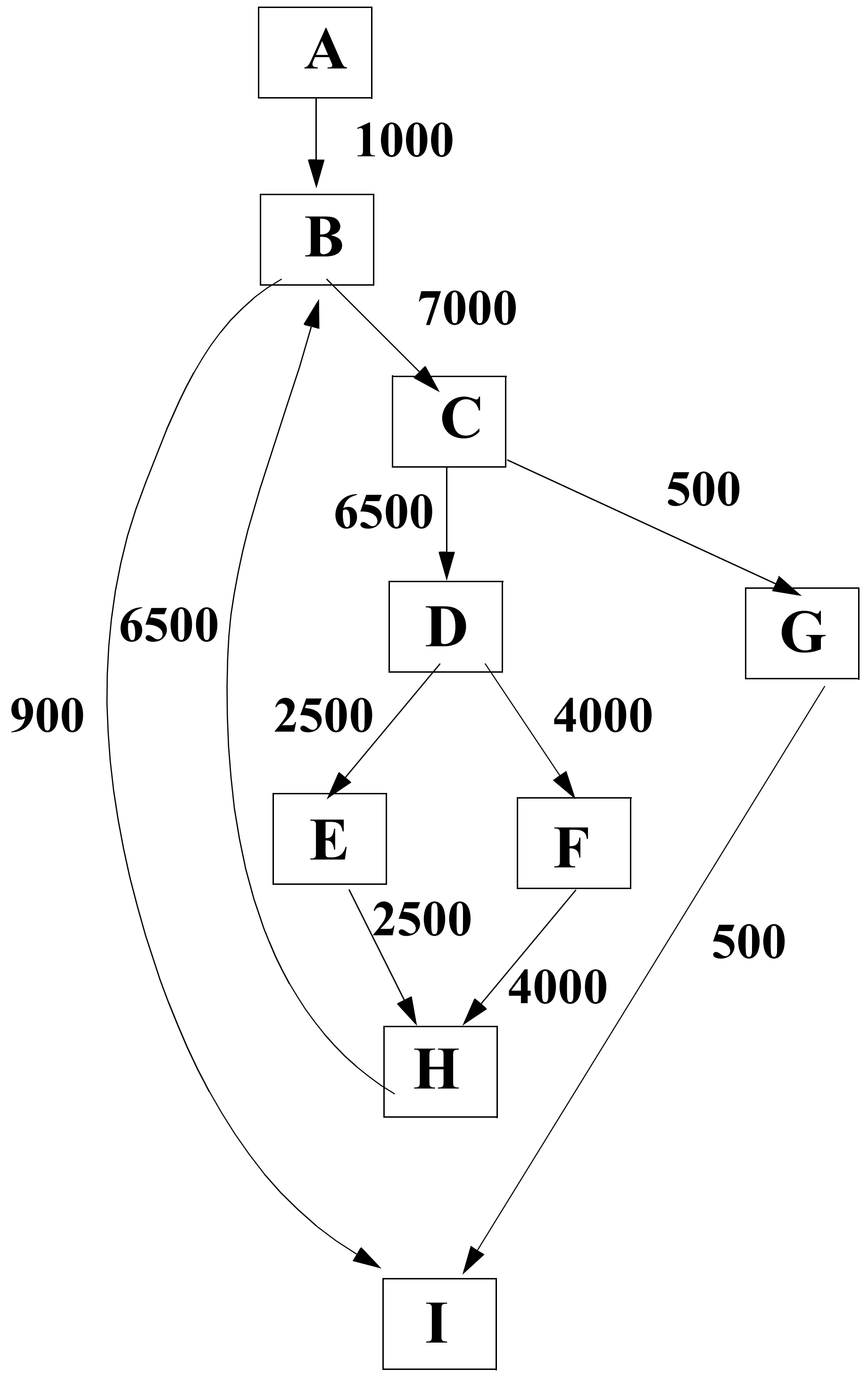
Then, in decreasing order of transition frequency, we visit arcs in the CFG.

If the blocks in the source and target can be linked into a longer chain
then do so, else skip to the next transition.

When we are done, we have linked together blocks in paths in the CFG that are most frequently executed.

Linked basic blocks are allocated together in memory, in the sequence listed in the chain.

Example


A
1000
B
7000
C
500
6500
D
G
6500
900
2500
4000
E
F
2500
500
4000
H
I

Initially, each bock is in its own chain.

Frequency	Action
7000	Form B-C
6500	Form B-C-D
6500	Form H-B-C-D
4000	Form H-B-C-D-F
4000	H is already placed
2500	E can't be placed after D, leave it alone
2500	H is already placed
1000	A can't be placed before B, leave it alone
900	I can't be placed after B, leave it alone
500	G can't be placed after C, leave it alone

Form G-I

We will place in memory the following chains of basic blocks:

H-B-C-D-F, E, A, G-I

On some computers, the direction of a conditional branch predicts whether the branch is expected to be taken or not (e.g., the HP PA-RISC). On such machines, a backwards branch (forming a loop) is assumed taken; a forward branch is assumed not taken.

If the target architecture makes such assumptions regarding conditional branches, we place chains to (where possible) correctly predict the branch outcome.

Thus E and G-I are placed after H-B-C-D-F since D$\rightarrow$E and C$\rightarrow$G normally aren't taken.

On the SPARC (V 9) you can set a bit in each conditional branch indicating expected taken/not taken status.

On many machines internal branch prediction hardware can over-rule poorly made (or absent) static predictions.

Procedure Splitting

When we profile the basic blocks within a procedure, we'll see some that are frequently executed, and others that are executed rarely or never.

If we allocate all the blocks of a procedure contiguously, we'll intermix frequently executed blocks with infrequently executed ones.

An alternative is "fluff removal." We can split a procedure's body into two sets of basic blocks: these executed frequently and those executed infrequently (the dividing line is, of course, somewhat arbitrary).

Now when procedure bodies are placed in memory, frequently executed basic blocks will be placed near each other, and infrequently executed blocks will be placed elsewhere (though infrequently executed blocks are still placed near each other). In this way be expect to make better use of page frames and I-cache space, filling them with mostly active basic blocks.