**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 than 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.
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:

```
   A
  /|
 / | 
C  D
 / | 
F  1  E
```

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

```
[C,F]  7  [A,D]  1  2
  1   E
```

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

- A-D-C-F \equiv F-C-D-A
- A-D-F-C \equiv C-F-D-A
- D-A-C-F \equiv F-C-A-D
- D-A-F-C \equiv 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:

```
   D-A-C-F  3  E
```

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
  | 6500
  | 6500
  C
  | 500
  | 900
  | 4000
  | 2500
  D
  | 1000
  E
  | 500
  | 2500
  | 4000
  F
  | 900
  | 500
  G
  H
  | 1000
  | 1000
  I
```
Initially, each block is in its own chain.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>7000</td>
<td>Form B-C</td>
</tr>
<tr>
<td>6500</td>
<td>Form B-C-D</td>
</tr>
<tr>
<td>6500</td>
<td>Form H-B-C-D</td>
</tr>
<tr>
<td>4000</td>
<td>Form H-B-C-D-F</td>
</tr>
<tr>
<td>4000</td>
<td>H is already placed</td>
</tr>
<tr>
<td>2500</td>
<td>E can't be placed after D, leave it alone</td>
</tr>
<tr>
<td>2500</td>
<td>H is already placed</td>
</tr>
<tr>
<td>1000</td>
<td>A can't be placed before B, leave it alone</td>
</tr>
<tr>
<td>900</td>
<td>I can't be placed after B, leave it alone</td>
</tr>
<tr>
<td>500</td>
<td>G can't be placed after C, leave it alone</td>
</tr>
<tr>
<td>500</td>
<td>Form G-I</td>
</tr>
</tbody>
</table>

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→E and C→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 we expect to make better use of page frames and I-cache space, filling them with mostly active basic blocks.