Computing Dominators and Dominance Frontiers

Preston Briggs
Tim Harvey

Version 0.9
June 26, 1994

Contents

List of Figures

Chapter 1

Introduction

The purpose of this document is to describe and provide the code for an implementation of Lengauer and Tarjan's Algorithm for Finding Dominators in a Flowgraph [6]. We assume some amount of familiarity with the paper, but the purpose of this document is to expose some of the inner-workings of the algorithm which are not readily apparent.

We will also be implementing the algorithm described in Cytron *et al.* for calculating dominance frontiers [4]. This is a natural extension of Lengauer and Tarjan's algorithm which our optimizer passes will need to compute static single assignment (SSA) form.

This document and the resultant code are intended to be part of the shared-code library used in the Massively Scalar Compiler Project.

1.1 Dominators

Let G be a graph made up of vertices (V) connected by edges (E) with a unique start vertex r. A vertex v *dominates* another vertex w if every path from r to w includes v. The *immediate dominator* of some vertex w is the dominating vertex closest to w; *i.e.*, the dominator of w such that all other blocks which dominate w dominate the immediate-dominator as well.

We are interested in finding the immediate dominators (subsequently, simply *dominators*) for all of the nodes in our control-flow graph (CFG).

Dominator information is important because it provides information about what kinds of code motion are safe. We also use them to to compute a secondary form of information, dominance frontiers, which will be useful for performing other forms of data-flow analysis, such as finding SSA form.

To understand the idea of dominators and dominance frontiers, Cytron proposes the following intuitive model for dominators and dominance frontiers: Imagine that the nodes in the CFG are clear marbles and the edges are fiber-optic wires. Imagine further that you have a light source which shines into the start block, subsequently illuminating all of the other blocks in the graph. Now, imagine that the light flowing out of a marble can be blocked, say by putting one's finger over the marble. If we blocked the light flowing out of some node b, all of the nodes which are subsequently darkened are dominated by b. Further, any children of darkened blocks which are still illuminated[1] are in the dominance frontier of b.

1.2 The algorithm

Aho, Sethi, and Ullman present a naive algorithm for finding dominators [1]. They point out that if a block x dominates another block y, then x will also dominate y's predecessors. Thus, to find a particular block's set of dominators, we need only find the intersection of its predecessors' sets of dominators. We can then build the dominator tree from these sets.

[1] To be illuminated, a block must have an incoming fiber-optic wire along which light is still flowing.

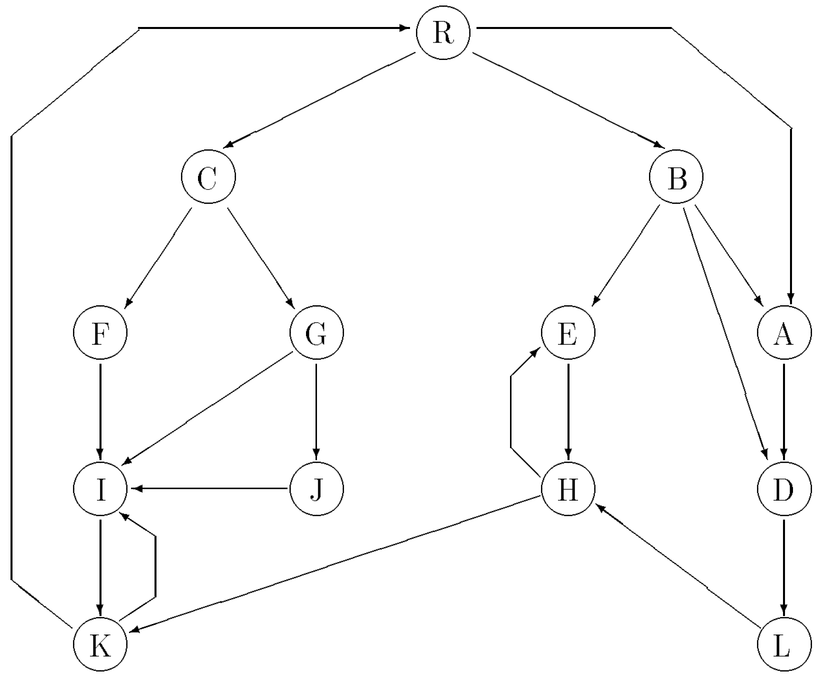


Figure 1.1: Example CFG

This iterative algorithm is both simple and asymptotically slow. Aho *et al.* suggest an implementation using bit vectors to represent the dominator sets. The worst-case complexity for this algorithm is quadratic in the number of nodes in the CFG, although Hecht and Ullman show that careful ordering (*i.e.*, reverse-postorder) of a pass over the graph should cause the algorithm to halt in a small number of iterations [5].

Lengauer and Tarjan developed an algorithm based on a depth-first search of the CFG which runs in $O(N \log N)$, where N is the number the of nodes in the CFG. Actually, Lengauer and Tarjan present *two* variations of their algorithm, which they call *simple* and *sophisticated*, where the sophisticated version runs somewhat more quickly. We implement the simple version here, since our experience has shown that it is quite fast.

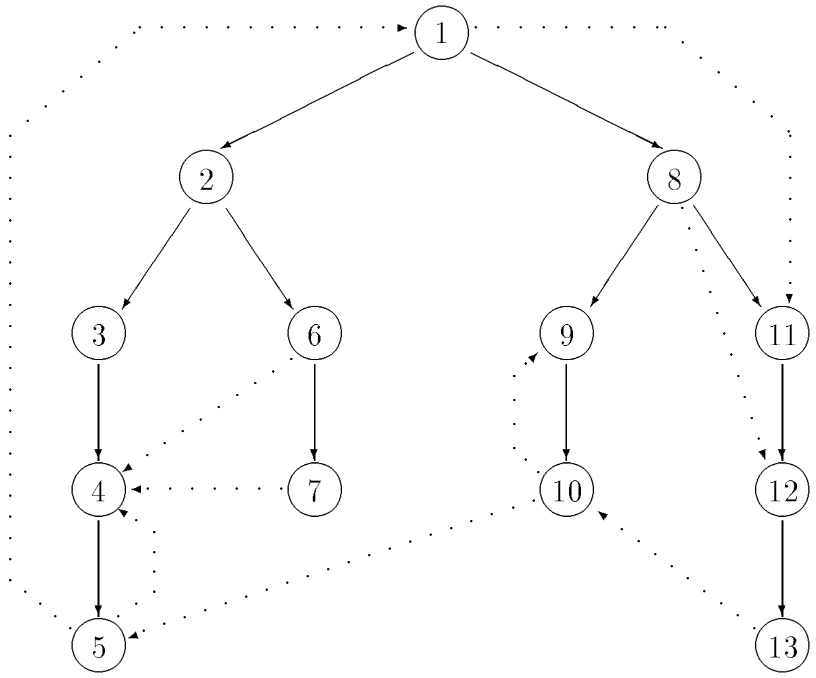
Lengauer and Tarjan's insight into the dominator problem was to show that the depth-first spanning tree (DFS *tree*) is a sufficient representation of the set of dominators. That is, the dominator of a block *must* be a proper ancestor of that block in the DFS tree. This tree will have fewer edges than the original graph, and so will be easier to analyze. But what of the edges in the CFG which are not represented in the DFS tree? Lengauer and Tarjan show how calculating a weak form of dominators, which they call *semidominators*, is sufficient to take into account these edges from the CFG which are not included in the DFS tree. They further show that semidominators completely subsume these extra edges from the CFG. Thus, having calculated semidominators, we can quickly derive the dominators. This leads us to the actual algorithm. There are three steps:

1. Perform a depth-first walk over the CFG, numbering the blocks in depth-first order and building the DFS tree,

2. Build the list of *semidominators*, and

3. Use the semidominators to compute the dominators.

We will use the same CFG as used in the paper to help aid the discussion. The original graph is shown in Figure 1.1.



Node	Parent
1	–
2	1
3	2
4	3
5	4
6	2
7	6
8	1
9	8
10	9
11	8
12	11
13	12

Figure 1.2: DFS tree

1.2.1 The DFS tree

The first step in this algorithm is to perform a depth-first walk over the CFG to build the DFS tree and build a depth-first numbering of the blocks. Each edge taken during the walk (remember that depth-first search only follows an edge if the destination of that edge has not already been reached during the walk) becomes an edge in the DFS tree. We will store the DFS tree in an array called `DFS_tree_parents`. This array will be indexed by a block's depth-first index, and the value at that index will be the block's parent in the DFS tree.

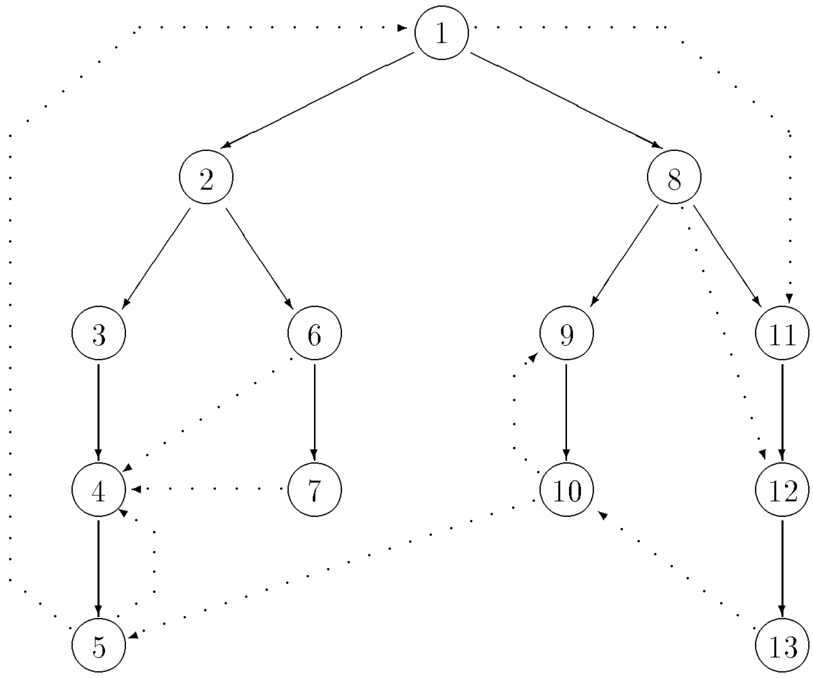
The result of this first step is shown in Figure 1.2. We should note that Lengauer and Tarjan are forced by their implementation to continuously switch between the depth-first indices and the nodes' textual names. Our implementation does not suffer from this handicap; we naturally refer to blocks by number, in this case, the depth-first number. Thus, in Figure 1.2 (and all subsequent uses of this example), we have replaced Lengauer and Tarjan's node names with the actual depth-first indices. The array to the right of the graph is the same as the array which we will build in the first step of our implementation (see Section2.1, starting on page 8). The array is indexed by node number, and the corresponding field shows the DFS tree parent. Note that the solid lines represent edges from the original CFG which are also DFS tree edges, while the dotted lines represent CFG edges which are not DFS tree edges.

1.2.2 Semidominators

The second step in this algorithm is to calculate each block's semidominator. Semidominators can be thought of as an initial guess at a block's dominator. They are easy to compute and, when used in combination with the DFS tree from the first step, can be shown to be a valid intermediate step in calculating each block's dominator.

Lengauer and Tarjan formally show how semidominators can be used to calculate dominators; we will sketch the reasoning they use, since it will be helpful in understanding some of the otherwise arcane pieces of code. To begin, we establish the *paths lemma*, which says:

> In a CFG, if some block, b_1, has a depth-first index which is less than some other block b_2, then any path from b_1 to b_2 must contain a common *ancestor* of both of them in the DFS tree.



Node	Semidominator
1	–
2	1
3	2
4	1
5	1
6	2
7	6
8	1
9	1
10	1
11	1
12	8
13	12

Figure 1.3: DFS tree with semidominators shown

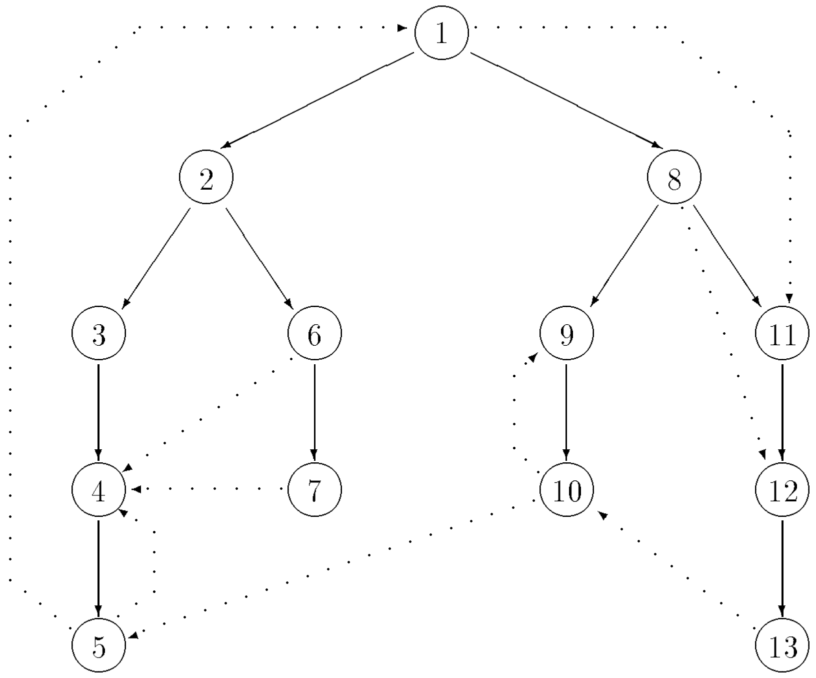
Remember that a block can be thought of as its own ancestor; so, for example, if we were to consider blocks 2 and 3 in Figure 1.2, the common ancestor on the path between these blocks is 2 itself. Likewise, consider blocks 5 and 10. Clearly, we can find a path from 5 to 10 – through blocks 1, 8, and 9. A visual examination of the graph shows that 1 is a common ancestor of 5 and 10. This lemma is essential in any further reasoning about semidominators.

We will develop two definitions for semidominators. The first is a concise definition which is easy to reason about but difficult to implement. The second is a less-compact definition, but one which will lead to a relatively simple implementation. We will give the easily-computable definition at the end of this section, after we use our first definition to show the relationship between semidominators and dominators. This compact definition of a semidominator is as follows:

> With the exception of the root block, which has no dominator, the semidominator of a block b_1 is the smallest-numbered block b_2 such that there is a CFG path from b_2 to b_1 in which all of the intervening blocks have a *larger* depth-first index than b_1.

From this definition, we can look at our example figure and manually find all of the semidominators. For example, block 5 has two incoming edges, so blocks 4 and 10 are candidates. Let us consider the incoming edges to block 4 – we can immediately discount blocks 6 and 7, since they are not smaller than block 4. Block 3 has a lower number than block 4, but it only has one path to block 5. This path has only one intervening block, 4, which is *less than* block 5, so block 3 fails to register as a candidate as well. The same fate awaits block 2 (block 3's predecessor), but we should consider block 1. Here, we can see a path to block 5 which passes through blocks 8, 9, and 10. Since all of these intervening blocks have a depth-first number higher than 5, block 1 is a candidate to be a semidominator of block 5 and, since it has the lowest depth-first index possible, will turn out to be the semidominator of block 5. The full set of semidominators is shown in Figure 1.3. Again, this array is structured the the same way as our implementation will build it – it is indexed by node number, and the corresponding field is the semidominator of that node (for more on this, consult Section 2.2, starting on page 11).

The fact that a block's dominator is defined as being on every path from the root to that block makes it obvious that that dominator must be a proper ancestor of the block in the DFS tree. Lengauer and Tarjan,



Node	Dominator
1	–
2	1
3	2
4	1
5	1
6	2
7	6
8	1
9	1
10	1
11	1
12	1
13	12

Figure 1.4: DFS tree with dominators shown

however, show that a block's semidominator *also* appears as a proper ancestor of the block in the DFS tree. Therefore, the next step is to show how semidominators and dominators are related to each other in the DFS tree.

We know that a block's semidominator and dominator are proper ancestors of that block in the DFS tree. Remember that a block's dominator must have a smaller depth-first index than the block itself. Likewise, the semidominator is not only a proper ancestor of the block on the DFS tree, but it has the smallest depth-first index (*i.e.,* it is as high as possible in the DFS tree) such that all of the blocks along some path in the CFG from the semidominator to the block itself have depth-first indices greater than or equal to the depth-first index of the block. This path in the CFG, then, avoids all proper descendants of the semidominator which are also proper ancestors of the block (in the DFS tree). Thus, the dominator cannot be in the path in the DFS tree from a semidominator to its block; therefore, it must be a (not necessarily proper) ancestor of the semidominator.

These observations are important because they give us an insight into the value of semidominators. We have already asserted that semidominators are easy to compute, and now we have shown that they are related to our ultimate goal, dominators themselves.

But these facts are just the building blocks; they show us that a relationship between semidominators and dominators exists, but they do not define the relationship itself. We do not yet have a strong enough connection between the two that we can calculate dominators. For this, we will need two lemmas about the relationship between semidominators and dominators.

The first lemma assumes that we have some block b and its semidominator $semi(b)$.

> If every block along the path in the DFS tree from b to $semi(b)$ (including b, but not $semi(b)$) has a semidominator with a larger depth-first index than $semi(b)$ itself, $semi(b)$ is b's dominator.

As an example, consider node 7 from Figure 1.3. Here, the path in the DFS tree from node 7 to its semidominator, node 6, has no intervening blocks. Thus, node 7's semidominator is by default the greatest depth-first index along the path, so its dominator is the same as its semidominator, or node 6. This gives us half of the equation which we need to derive dominators from semidominators.

The second lemma once again assumes that we have b and its semidominator $semi(b)$. This time, we find the minimum semidominator of all of the blocks in the path in the DFS tree from $semi(b)$ to b (again, including b and excluding $semi(b)$).

> Having found the minimum semidominator along the path between a block and its semidominator, we can conclude that this minimum semidominator is less than or equal to the semidominator of b, and the dominator of both of these blocks is the same.

Again taking an example from Figure 1.3, consider node 12. In the path from node 12 to node 8 (node 12's semidominator), we find that an intervening node, node 11, has a semidominator which is less than node 12's semidominator. Thus, node 11's dominator will also be node 12's dominator.

We can combine the above two theorems into a single, implementable definition of the relationship between semidominators and dominators. Let us suppose that we once again have a block b and its semidominator, $semi(b)$. Further, we have found the block with the lowest semidominator along the path in the DFS tree, and we shall refer to this minimum semidominator as m and the block itself as b_m. If m is equal to $semi(b)$, then all of the other semidominators are greater than $semi(b)$, so the first theorem applies – that is, the dominator of b is $semi(b)$ itself. Otherwise, we invoke the second theorem, which says that b_m's dominator is also the dominator of b. Figure 1.4 shows the full set of dominators for our example CFG.

Thus, we have shown that dominators can be easily computed from semidominators. But the initial definition of semidominators is not easily computable. By our first formulation, finding a single semidominator would require a walk over the entire CFG, an obviously quadratic proposition, to find the entire set of dominators.

Lengauer and Tarjan redefine the definition of semidominators so that they can be computed recursively. The new definition says that the semidominator of some block b is the minimum element in the union of:

1. the set of (CFG) predecessors of b whose depth-first index is less than b's
2. the set of semidominators taken from the set of blocks which:
   - have depth-first indices greater than b's, and
   - which have a (possibly zero-length) path in the DFS tree to one of b's predecessors.

In practical terms, this second set can be calculated by finding the minimum semidominator along the path in the DFS tree from b's larger-numbered predecessors to the roots of those predecessors' trees in the forest. As we shall see, careful initialization of our data structures will allow us to calculate both sets of candidates in this manner.

The recursion in the second part of the definition allows us to postpone the computation of a block's semidominators until we have accumulated all of the information in its ancestors in the DFS tree. We know that a block's semidominator can only appear above it in the DFS tree, so a reverse pass will not require any backtracking. Thus, the algorithm will require exactly one (backward) pass over the DFS tree.

Chapter 2 will provide the code for the implementation of dominator calculation. First, however, we will provide the high-level specifics of the code.

1.3 Code organization

The code for calculating dominators will be part of the shared library code, so it will reside in a C module. This web will not produce a separate "`.h`" header file, since all of the data structures and entry points are already elaborated in the `shared.h` header file. For an explanation of the data structures which we build, consult the `shared` web [2].

"Dominators.c" 7a ≡

```
#include <Shared.h>

⟨Internal data structures 9c, ... ⟩
⟨Internal global variables 8a, ... ⟩
⟨Internal function prototypes 30, ... ⟩
⟨Internal function bodies 11b, ... ⟩
⟨Externally-visible function bodies 7b, ... ⟩
◇
```

1.4 Entry points

The code in this module will provide two main entry points, one to calculate dominance information, and the other to calculate post-dominance information.

For post-dominance, we assume the existence of a unique ending block in the CFG, much like we assumed the existence of a unique starting block for dominance information. We define the post-dominators of some block b as the set of blocks such that there are no paths from b to the ending block which do not pass through the post-dominators. Like dominators, we will use the term *post-dominator* to indicate a block's *nearest* post-dominator. We haven't mentioned post-dominators in the discussion so far because their computation is carried out using the same algorithm, except that we calculate the information using the depth-first numbering generated by starting at the ending block and working backwards through the graph. Because we build the DFS tree from the bottom up, we often refer to post-dominance as *backwards dominance information*, as opposed to dominance information, which we will refer to as *forward* dominance information.

We call the entry point to calculate forward dominance information `Dominator_CalcDom`. Since we store the CFG in global space, the only parameter this routine needs is an arena it can use to hold dominance information.

The function prototype, and all other function prototypes, are shown in Appendix B.

The body of this routine encapsulates the three steps elaborated in early sections of this introduction for building dominators, as well as the code necessary for building dominance frontiers. Notice that our implementation splits the dominance definition into two parts and moves the calculation of the first part into the code which calculates semidominators. This is an implementation issue which we will discuss in the next chapter.

⟨Externally-visible function bodies 7b⟩ ≡

```
Void Dominator_CalcDom(Arena permanent_memory_bank)
{
    Arena local_memory_bank = Arena_Create();

    ⟨Build the DFS tree and initialize variables 10a⟩
    ⟨Calculate semidominators and implicitly define immediate dominators 14c⟩
    ⟨Explicitly define immediate dominators 18b⟩

    ⟨Rewrite into shared data structures 20b⟩

    ⟨Calculate dominance frontiers 25b⟩

    ⟨Restore original block_extension values 10e⟩

    Arena_Destroy(local_memory_bank);
}
◇
```

Macro defined by scraps 7b, 33a, 34ab, 35a.
Macro referenced in scrap 7a.

The second entry point provided by this module, `Dominator_CalcPostDom`, calculates the post-dominator tree and post-dominance frontiers for the CFG and puts this information in each block's `post_dom_node` field. We will not show the body yet; for the reasons for this, consult Chapter 3.

Chapter 2

Algorithmic Implementation

Lengauer and Tarjan provide an implementation of the equations from the last chapter. Their implementation is broken up into the following three steps:

1. build the DFS tree
2. compute the semidominators of each block
3. compute dominators in two steps:
   (a) make an initial guess at the dominators
   (b) define the dominators based on the initial guess

As we noted in the previous chapter, we split the definition of the relationship between dominators and semidominators into two sections so that we can calculate the first part during the computation of the semidominators.

There are a number of data structures which will get built during these computations. Lengauer and Tarjan show all of these at the beginning of the discussion on their implementation. We have chosen to instead elaborate each of the data structures in the section in which we introduce it.

2.1 Building the DFS tree

The first step in the implementation is building the DFS tree. This requires a straightforward depth-first walk over the CFG to connect the edges of the DFS tree. We will store the tree in the **DFS_tree_parents** array as shown in Figure 1.2, on page 3.

⟨Internal global variables 8a⟩ ≡
```
static Unsigned_Int *DFS_tree_parents;
```
◇

Macro defined by scraps 8a, 9ad, 10b, 12, 13c, 25a.
Macro referenced in scrap 7a.

⟨Allocate and initialize arrays 8b⟩ ≡
```
DFS_tree_parents = (Unsigned_Int *)Arena_GetMem(local_memory_bank,
                                                sizeof(Unsigned_Int) * (block_count+1));
```
◇

Macro defined by scraps 8b, 9b, 10c, 13ad.
Macro referenced in scrap 10a.

Note that the parser has already numbered the blocks in both postorder and preorder (or depth-first order). Thus, we could rely on the already-calculated depth-first indices as our depth-first numbering for finding dominators. However, the same code we are creating to find dominators will also be used to find post-dominators (see Chapter 3 for more on this). Since the shared code does not build a depth-first ordering starting from the end block,[1] we could not reuse the dominator code if it relied on the shared code's calculations. Thus, we will write the code to perform the exact same calculations for both dominators and post-dominators, and, for this step only, we will consequently do slightly more work for the calculation of dominators than is strictly necessary.

Here, instead of relying on the `preorder_block_list` array provided by the shared code, we build our own mapping from the depth-first index to each block.

⟨Internal global variables 9a⟩ ≡
```
static Block **DFS_block_list;
◇
```
Macro defined by scraps 8a, 9ad, 10b, 12, 13c, 25a.
Macro referenced in scrap 7a.

⟨Allocate and initialize arrays 9b⟩ ≡
```
DFS_block_list = (Block **)Arena_GetMem(local_memory_bank,
                                        sizeof(Block *) * (block_count+1));
◇
```
Macro defined by scraps 8b, 9b, 10c, 13ad.
Macro referenced in scrap 10a.

Likewise, we will need to store in each block its depth-first index to provide the mapping the other way. For this, we will utilize the extension field in the `Block` data structure.

⟨Internal data structures 9c⟩ ≡
```
typedef struct block_extension {
    Unsigned_Int DFS_name;
};
◇
```
Macro defined by scraps 9c, 14a.
Macro referenced in scrap 7a.

We will need a global counter to keep track of the next name to assign a block during the walk. It is just as easy to make this variable global as to pass it into the recursive routine, so that is what we have done.

⟨Internal global variables 9d⟩ ≡
```
static Unsigned_Int next_DFS_name = 1;
◇
```
Macro defined by scraps 8a, 9ad, 10b, 12, 13c, 25a.
Macro referenced in scrap 7a.

This section can be broken down into two steps: initializing the variables necessary to build the DFS treeand hold the information related to dominators, and calling the recursive routine which will perform the walk. The initialization includes setting up special values for `start_block` and resetting the `visited` flags for the blocks.

Note: We add one to `block_count` because block numbers start at one and C arrays are zero-based. Thus, by allocating the array to be one larger than we need, we can index it directly by the block number. We will do this for all similar arrays.

[1] A depth-first numbering starting from the end block and going backwards through the CFG is different from a *postorder* walk starting from the CFG's start block.

⟨Build the DFS tree and initialize variables 10a⟩ ≡

```
⟨Allocate and initialize arrays 8b, ... ⟩
⟨Store original block_extension values 10d⟩
⟨Set up the initial values for start_block 11a⟩
Block_Clear_Visited_Flags();

build_DFS_tree(start_block, local_memory_bank);
◇
```

Macro referenced in scrap 7b.

As we perform the depth-first walk over the graph to build the DFS tree, we will be allocating new `block_extension` structures for each block to hold that block's depth-first index. However, this is a module which is supposed to be used in the shared-code library. When the entry points in this module are called, we have no way of telling if the calling routine has already allocated its own `block_extension` structures. Thus, before we can begin building the DFS tree, we need to store away the pointers to all of the already-existent `block_extension` structures. We will create an array called `old_extension` for this purpose. The following scraps of code show how we can maintain the original data while still using the `block_extension` field for our own purposes.

⟨Internal global variables 10b⟩ ≡

```
Block_Extension **old_extension;
◇
```

Macro defined by scraps 8a, 9ad, 10b, 12, 13c, 25a.
Macro referenced in scrap 7a.

⟨Allocate and initialize arrays 10c⟩ ≡

```
old_extension = (Block_Extension **)Arena_GetMem(local_memory_bank,
                                                 sizeof(Block_Extension *) * (block_count+1));
◇
```

Macro defined by scraps 8b, 9b, 10c, 13ad.
Macro referenced in scrap 10a.

⟨Store original `block_extension` values 10d⟩ ≡

```
{
    Block *block_runner;

    ForAllBlocks(block_runner)
        old_extension[block_runner->preorder_index] = block_runner->block_extension;
}◇
```

Macro referenced in scrap 10a.

⟨Restore original `block_extension` values 10e⟩ ≡

```
{
    Block *block_runner;

    ForAllBlocks(block_runner)
        block_runner->block_extension = old_extension[block_runner->preorder_index];
}◇
```

Macro referenced in scrap 7b.

The root of the CFG, obviously, has no parent in the DFS tree. Thus, in order to start off the recursive walk, we need to initialize all of the values pertaining to `start_block`.

⟨Set up the initial values for `start_block` 11a⟩ ≡

```
start_block->block_extension = (Block_Extension *) Arena_GetMem(local_memory_bank,
                                                                sizeof(Block_Extension));
start_block->block_extension->DFS_name = next_DFS_name;
DFS_tree_parents[next_DFS_name] = 0;
DFS_block_list[next_DFS_name] = start_block;
next_DFS_name++;
```

◇

Macro referenced in scrap 10a.

As a convenience, we will also initialize the `semidominator` and `path_minimums` global arrays in the following routine. These arrays will be defined when we explain their use in later sections.

This routine is a straightforward recursive routine which walks over the `CFG`. For each block, we need to record its depth-first index, initialize its fields in our arrays, and recur on its children.

⟨Internal function bodies 11b⟩ ≡

```
static Void build_DFS_tree(Block *current_block, Arena local_memory_bank)
{
    Unsigned_Int current_block_num = current_block->block_extension->DFS_name;
    Edge *successor;

    current_block->visited = TRUE;
    ⟨Initialize array entries for semidominator calculation 13b, ... ⟩
    Block_ForAllSuccs(successor, current_block)
    {
        Block *successor_block = successor->succ;
        Unsigned_Int succ_block_num = successor_block->block_extension->DFS_name;

        if (!successor_block->visited)
        {
            ⟨Set up initial values for successor_block 11c⟩
            build_DFS_tree(successor_block, local_memory_bank);
        }
    }
} /* build_DFS_tree */
```

◇

Macro defined by scraps 11b, 19ab, 26a, 35b.
Macro referenced in scrap 7a.

For each of the current block's successors which we have not yet reached in our walk, we need to "attach" it to its parent in the DFS tree and set up our mappings by recording the successor's DFS number in the successor's `block_extension` field and entering the pointer to the successor in the `DFS_block_list` array.

⟨Set up initial values for `successor_block` 11c⟩ ≡

```
DFS_tree_parents[next_DFS_name] = current_block_num;
successor_block->block_extension = (Block_Extension *) Arena_GetMem(local_memory_bank,
                                                                    sizeof(Block_Extension));
successor_block->block_extension->DFS_name = next_DFS_name;
DFS_block_list[next_DFS_name] = successor_block;
next_DFS_name++;
```

◇

Macro referenced in scrap 11b.

2.2 Calculating semidominators and implicitly defining dominators

As we stated in the introduction to this chapter, the second and the first half of the third steps are carried out together. Although there is no obvious reason from the discussion in the first chapter why the dominance calculation should be broken up, Lengauer and Tarjan split this process into two steps so that steps 2 and 3*a* might be run together. This is an implementation issue; this configuration allows us to process the information as we develop it, instead of storing it for a subsequent pass over the DFS tree.

This section of the implementation will run through the blocks in reverse depth-first order. The reason for this is straightforward. Recall the first definition of semidominators – the semidominator of a block b is the lowest-numbered block such that any block lying on a path between the two blocks has a larger depth-first index than the block in question. This naturally splits the CFG predecessors of a block into two groups.

The first group is the set of predecessors whose depth-first index is less than b. Obviously, these blocks have no intervening blocks between themselves and b and, thus, are candidates.

The second group is computed from the set of predecessors whose depth-first index is greater than b. Because we iterate over the blocks in reverse depth-first order, these blocks will already have had their semidominators calculated. Our goal is to find the minimum semidominator of these predecessors and their ancestors in the DFS tree. Our iteration order tells us that these higher-numbered predecessor blocks of b will either lie on (and below) the DFS tree branch containing b, or some other DFS tree branch. In either case, we will find the minimum semidominator along each path up the DFS tree from each of b's predecessors, stopping when we reach a node whose depth-first index is less than (or equal to) b. We rely on careful initialization of the `semidominator` array to know when to stop this search, as well as path compression to avoid traversing these subtrees multiple times (see below).

The semidominator of b, then, is the minimum of these two sets.

Taking an example from our running example from chapter 1, let us consider Figure 1.3 on page 4, and imagine that we are looking for the semidominator for node 5. This node has two predecessors, nodes 4 and 10. Node 4 immediately goes into the first set. Now, we need to accumulate all of the semidominators of the nodes between node 10 and node 10's semidominator. We search up this branch of the DFS tree, recording the semidominators for nodes 9 and 8, and stopping at node 1, since it is lower than 5. The second set of semidominator candidates for 5, then, contains the node 1, and so the full set of candidates is 1 and 4. Taking the minimum, we find that the semidominator for node 5 is node 1.

In practice, we do not build these two groups separately. Instead, we rely on our initialization of the `semidominator` array to simplify the processing. Remember that in Step one we initialized the fields in this array to each block's depth-first index. This allows us to loop over the predecessors of our block b using the same comparator. Initially, b's entry in `semidominator` is b itself. All of the predecessors of b whose depth-first index is less than b will still have their initial `semidominator` value, which is less than b. Likewise, all of the predecessors of b whose depth-first index is greater than b will already have their semidominator calculated into their respective entry in `semidominator`.

Lengauer and Tarjan show that the blocks which are not immediate predecessors of the current block are, in fact, roots of trees in a forest based on the DFS tree. This forest consists of trees whose root is a node which has not been processed, but whose children (if any) have already been examined. In the beginning, all nodes are unconnected, so they are all roots of trees in the forest. Each time a node is processed, we add it to its DFS tree parent's tree.

As we build up the trees, we will also keep track of the minimum semidominator from each node in the tree to the root of that tree. By computing this information incrementally and storing it, we will reduce the running time of the algorithm. Although this requires a small amount of extra storage, we should make that cost up at runtime, since we do not have to recompute it each time we need it. Likewise, we will update the information incrementally (*i.e.,* on demand) which should speed up the algorithm, since we will never compute values which are not needed.

We will require a list to hold the semidominators, as well as one to hold the dominators. We will implement these as arrays indexed by block number. Of course, the `semidominator` array gets initialized while we are building the DFS tree. We will create the `dominator` array in this step, but we do not need to initialize its fields, as we will simply overwrite each field in these steps.

⟨Internal global variables 12⟩ ≡
```
static Unsigned_Int *semidominator;
static Unsigned_Int *dominator;
```
◇

Macro defined by scraps 8a, 9ad, 10b, 12, 13c, 25a.
Macro referenced in scrap 7a.

⟨Allocate and initialize arrays 13a⟩ ≡
```
semidominator = (Unsigned_Int *)Arena_GetMem(local_memory_bank,
                                             sizeof(Unsigned_Int) * (block_count+1));
dominator = (Unsigned_Int *)Arena_GetMem(local_memory_bank,
                                         sizeof(Unsigned_Int) * (block_count+1));
dominator[0] = semidominator[0] = 0; /* pedantic */
◇
```
Macro defined by scraps 8b, 9b, 10c, 13ad.
Macro referenced in scrap 10a.

⟨Initialize array entries for semidominator calculation 13b⟩ ≡
```
semidominator[current_block_num] = current_block_num;
◇
```
Macro defined by scraps 13be.
Macro referenced in scrap 11b.

We will store information about the DFS tree forest using two arrays, again indexed by block number. The first array, `ancestor`, encapsulates the forest of trees from the DFS tree. We will initialize each field in this array to zero. An `ancestor` entry which is zero corresponds to a block which is a root of a tree in the forest. Each time a block is processed, we will insert the name of its DFS tree parent into its field in `ancestor`. The second array will be used to store the minimum semidominators along all of the paths in the forest.[2] Each entry in this array will be initialized with its index, much the same way the `semidominator` array is initialized, and for the same reason. Lengauer and Tarjan call this array `label`. but we refer to it as `path_minimums`, since this more accurately reflects its function.

Note: We use `Arena_GetMemClear` to allocate the `ancestor` array because we rely it its fields to all be initialized to zero.

⟨Internal global variables 13c⟩ ≡
```
static Unsigned_Int *ancestor;
static Unsigned_Int *path_minimums;
◇
```
Macro defined by scraps 8a, 9ad, 10b, 12, 13c, 25a.
Macro referenced in scrap 7a.

⟨Allocate and initialize arrays 13d⟩ ≡
```
ancestor = (Unsigned_Int *)Arena_GetMemClear(local_memory_bank,
                                             sizeof(Unsigned_Int) * (block_count+1));
path_minimums = (Unsigned_Int *)Arena_GetMem(local_memory_bank,
                                             sizeof(Unsigned_Int) * (block_count+1));
path_minimums[0] = 0; /* pedantic */
◇
```
Macro defined by scraps 8b, 9b, 10c, 13ad.
Macro referenced in scrap 10a.

⟨Initialize array entries for semidominator calculation 13e⟩ ≡
```
path_minimums[current_block_num] = current_block_num;
◇
```
Macro defined by scraps 13be.
Macro referenced in scrap 11b.

We will need one further array to compute dominators from semidominators. Recall that our definition of dominators is recursive, since we cannot always compute a block's dominator until some of that block's ancestor in the DFS tree have been computed. Thus, we need, for each block, a list of those blocks which are still waiting to finish their computation. Since we cannot know beforehand how large each of these lists is going to be, each entry in the `not_yet_done` array is a linked list of pointers to these unfinished blocks.

[2] In reality, this array stores the block whose semidominator is minimum, not the semidominator itself.

⟨Internal data structures 14a⟩ ≡

```
typedef struct linked_list_node LL_Node;
struct linked_list_node {
    Unsigned_Int block_num;
    LL_Node *next;
};
◇
```

Macro defined by scraps 9c, 14a.
Macro referenced in scrap 7a.

Note: As usual, it will save some time if we can assume that the entries in this array start out as `NULL` pointers, so we use `Arena_GetMemClear` to accomplish this quickly.

⟨Define local variables for Steps 2 and 3a 14b⟩ ≡

```
LL_Node **not_yet_done = (LL_Node **)Arena_GetMemClear(local_memory_bank,
                                                sizeof(LL_Node *) * (block_count+1));
◇
```

Macro defined by scraps 14bd.
Macro referenced in scrap 14c.

These data definitions lead us to the discussion of the algorithm for Steps 2 and 3. As we said above, we will be iterating through the blocks in reverse depth-first order. At each block, we will look at the forests built for all of its predecessors and find the minimum semidominator of all of these trees. We then add the current block to the forest, making its immediate ancestor in the DFS tree the root of a tree. Remember that we will have to postpone the initial guess at a block's dominator until its ancestors are computed. Each time we add a block to the forest, we can take the ancestor of that block and calculate the dominators for all blocks waiting on that ancestor.

The scrap containing these steps is below. Consult Figure 2.1 for a comparison of how this code matches the paper's pseudo code.

⟨Calculate semidominators and implicitly define immediate dominators 14c⟩ ≡

```
{
    ⟨Define local variables for Steps 2 and 3a 14b, ... ⟩

    ⟨For each block b in the CFG (in reverse depth-first order) 14e⟩
    {
        ⟨Find the semidominator for each of b's CFG predecessors 15a⟩
        ⟨Put off calculating b's dominator until we have added b's semidominator to the forest 16b⟩
        ⟨Add b to the forest 16a⟩
        ⟨Calculate dominators for those blocks who had to wait until b was added to the forest 17a⟩
    }
}
◇
```

Macro referenced in scrap 7b.

Remember that we are referring to the blocks by their depth-first index. Thus, to run through all of the blocks, we need only define a counter and set up a normal `for`-loop. Of course, the root block does not have a semidominator, so we only need to loop until we hit this first block.

⟨Define local variables for Steps 2 and 3a 14d⟩ ≡

```
Unsigned_Int b; /* represents the current block */
◇
```

Macro defined by scraps 14bd.
Macro referenced in scrap 14c.

⟨For each block b in the CFG (in reverse depth-first order) 14e⟩ ≡

```
for (b=block_count; b>1; b--)
◇
```

Macro referenced in scrap 14c.

for $i := n$ **by** -1 **until** 2 **do**	*<For each block b in the* CFG *(in reverse depth-first order)>*
$w := vertex(i)$;	*<b is the current block>*
for each v *in* $pred(w)$ **do** $u :=$ EVAL(v);	*<Find the semidominator for each of b's* CFG *predecessors>*
if $semi(u) < semi(w)$ **then** $semi(w) := semi(u)$	*<Find the minimum of these semidominators>*
add w *to* $bucket(vertex(semi(w)))$;	*<Put off calculating b's dominator until we have added b's semidominator to the forest>*
LINK$(parent(w), w)$;	*<Add b to the forest>*
for each v *in* $bucket(parent(w))$ **do** *delete* v *from* $bucket(parent(w))$;	*<Calculate dominators for those blocks who had to wait until b was added to the forest>*
$u :=$ EVAL(v);	*<Find the minimum semidominator between this block and the root of its tree>*
$dom(v) :=$ **if** $semi(u) < semi(v)$ **then** u **else** $parent(w)$;	*<Make a guess at the dominator>*

Figure 2.1: Lengauer and Tarjan's pseudo-code compared to our pseudo-code for finding semidominators

The semidominator of b is simply the minimum of the predecessors' semidominators as stored in the `semidominator` array.[3]

Finding each semidominator should be a simple matter of looking up each predecessor's value in the `path_minimums` array. Recall, however, that one of the advantages of this algorithm is that we do not perform path compression and recompute the minimum semidominators of a path in a tree until required. Thus, instead of a simple array-lookup, we need to call the function `EVAL`, which will perform the path compression and return the block in the path with the lowest semidominator. We will elaborate this function in Section 2.4.

⟨Find the semidominator for each of b's CFG predecessors 15a⟩ ≡

```
{
    Block *curr_block = DFS_block_list[b];
    Edge *pred_runner;

    Block_ForAllPreds(pred_runner, curr_block)
        ⟨Find the minimum of the predecessor semidominators 15b⟩
}
◇
```

Macro referenced in scrap 14c.

Of course, the quickest way to find the minimum semidominator is to compare each one with our current value.

⟨Find the minimum of the predecessor semidominators 15b⟩ ≡

```
{
    Unsigned_Int pred_num = pred_runner->pred->block_extension->DFS_name;
    Unsigned_Int next_semi = semidominator[EVAL(pred_num)];

    if (next_semi < semidominator[b])
        semidominator[b] = next_semi;
}
◇
```

Macro referenced in scrap 15a.

[3] Remember that we initialized the `path_minimums` array such that the ancestors in the DFS tree of b (which have not yet been calculated) have their own value listed as their semidominator, in the same way that we initialized the `semidominator` array.

Having found b's semidominator, we now have to add b to the forest. Remember that we do not do any path compression until there is a demand for it, so adjusting trees in the forest simply becomes a matter of inserting b's DFS tree parent into b's slot in the `ancestor` array.

⟨Add b to the forest 16a⟩ ≡

```
ancestor[b] = DFS_tree_parents[b];
```

◇

Macro referenced in scrap 14c.

At this point in the iteration, we cannot yet make a guess at b's dominator. Recall that the first half of the definition of dominators states that the dominator of some block b is b's semidominator if it is the minimum semidominator along the path in the DFS tree between b and its semidominator. We know that we have calculated all of the semidominators along this path when we reach b's semidominator itself. That is, we know that we have calculated the semidominators along this path when b's semidominator becomes a root of a tree containing b in the DFS tree forest. Thus, we add b to its semidominator's list of blocks waiting to complete the first-guess computation of their dominators. We will compute the first guess at b's dominator when we reach a block whose ancestor in the DFS tree (and, thus, a root of the tree in the forest) is b's semidominator.

⟨Put off calculating b's dominator until we have added b's semidominator to the forest 16b⟩ ≡

```
{
    LL_Node *new_node = (LL_Node *)Arena_GetMem(local_memory_bank, sizeof(LL_Node));
    Unsigned_Int semidominator_of_b = semidominator[b];

    new_node->block_num = b;
    new_node->next = not_yet_done[semidominator_of_b];
    not_yet_done[semidominator_of_b] = new_node;
}
```

◇

Macro referenced in scrap 14c.

As we said above, a block b has to wait to have its dominator calculated until all of the semidominators of the blocks on the path in the DFS tree between b and $semi(b)$ have been calculated. We know that this calculation is complete when the tree containing b is connected to $semi(b)$. Thus, the last step in each iteration is to make a first guess at the dominators of those blocks whose path to their semidominator is complete. Remember, however, that the definition for dominators says that we need to know the lowest semidominator of the blocks along the path in the DFS tree connecting b and $semi(b)$, not including $semi(b)$ itself. Thus, at this point in the calculations, we do not clear *b's* list of blocks, but, rather, b's *parent's* list of pending blocks.[4]

[4] Of course, it should be obvious that b's list of blocks has *already* been taken care of at this point.

⟨Calculate dominators for those blocks who had to wait until b was added to the forest 17a⟩ ≡

```
{
    Unsigned_Int parent = DFS_tree_parents[b];
    LL_Node *first_in_list;

    first_in_list = not_yet_done[parent];
    not_yet_done[parent];

    while(first_in_list)
    {
        Unsigned_Int next_block = first_in_list->block_num;
        Unsigned_Int semidominator_min;

        ⟨Find the minimum semidominator between this block and the root of its tree 17b⟩
        ⟨Make a guess at next_block's dominator 18a⟩
        first_in_list = first_in_list->next;
    }
}
◇
```

Macro referenced in scrap 14c.

The definition of dominators says that we need to compare the semidominator of **next_block** against the minimum semidominator of the blocks along the path in the DFS tree between **next_block** and its semidominator (**parent**). Therefore, before we can find a **next_block**'s dominator, we need to call **EVAL** to give us this minimum value.

⟨Find the minimum semidominator between this block and the root of its tree 17b⟩ ≡

```
semidominator_min = EVAL(next_block);
◇
```

Macro referenced in scrap 17a.

Finally, we can evaluate the first half of the dominance equation. This single line of code, on page 129 of the paper, can be somewhat confusing.

The dominance formula for some block b_1 and the block b_2, which has the minimum semidominator along the path in the DFS tree between b_1 and $semi(b_1)$ looks like:

$$idom(b_1) = \left\{ \begin{array}{ll} sdom(b_1) & \text{if } sdom(b_1) = sdom(b_2) \\ idom(b_2) & \text{otherwise} \end{array} \right\}.$$

The line of pseudo code Lengauer and Tarjan use to calculate the first line of this equation is:

$$dom(b_1) = \textbf{if } semi(b_2) < semi(b_1) \textbf{ then } b_2 \quad \textbf{else } parent(w)$$

The way *we* write the code, using the paper's notation, is:

$$dom(b_1) = \textbf{if } semi(b_1) == semi(b_2) \textbf{ then } semi(b_1) \quad \textbf{else } b_2$$

Although the two different scraps of code are equivalent, the second more accurately reflects the actual dominance formula. The first scrap of code checks for inequality instead of equality between $semi(b_2)$ and $semi(b_1)$.[5] Since our code matches the authors' formula and bases the conditional on the equality of $semi(b_1)$ and $semi(b_2)$ instead of inequality between them, the result code for the **then** and the **else** cases are switched. If the two semidominators are unequal, we will insert b_2 into $dom(b_1)$. We do this simply as a convenience for calculating step four. On the other hand, if the two semidominators are equal,

[5] Since $semi(b_2)$ will never be lower on the DFS tree than $semi(b_1)$, $semi(b_2)$ can never be *greater than* $semi(b_1)$, so checking that $semi(b_2)$ is less than $semi(b_1)$ is sufficient to show inequality.

then we insert $semi(b_1)$, not $parent(w)$. These are the same value; when we processed b_1, we put it into its semidominator's **not_yet_done** list, which is the same list (*i.e.,* the list attached to $parent(w)$) that we are now emptying.

⟨Make a guess at **next_block**'s dominator 18a⟩ ≡

```
if (semidominator[next_block] == semidominator[semidominator_min])
    dominator[next_block] = semidominator[next_block];
else
    dominator[next_block] = semidominator_min;
```

◇

Macro referenced in scrap 17a.

2.3 Explicitly defining dominators

At this point in the calculation, we have calculated the set of semidominators and implicitly defined the dominators by applying the first half of the dominance formula. The entry for a block in the **dominator** array contains either the block's actual dominator or the block whose dominator is also that block's dominator.

We iterate forward through the blocks for this part of the calculation. This is so that in the case that we have to take the dominator of **dominator[b]**, it will already have been calculated. Notice that we start at block number 2 instead of 1, since the starting block has no dominator.

⟨Explicitly define immediate dominators 18b⟩ ≡

```
{
    Unsigned_Int i;

    dominator[1] = 0;
    for(i=2; i<=block_count; i++)
        if (dominator[i] != semidominator[i])
            dominator[i] = dominator[dominator[i]];
}
```

◇

Macro referenced in scrap 7b.

2.4 Finding the minimum semidominator along a path in the DFS tree (EVAL)

The last thing we must do in this chapter is to explain and provide the code for the **EVAL** routine. This routine maintains the data structures representing the forest of trees from the DFS tree which we build during steps 2 and 3.

This forest requires two data structures, one to represent the actual set of trees (**ancestor**), and one to represent for each block, b, the block whose semidominator is lowest along the path in the tree from b to the tree's root (**path_minimums**).

It is the value from the **path_minimums** array which **EVAL** returns. The logic for this routine is simple: if the passed-in block b is a root of the tree in the forest (*i.e.,* if **ancestor[**b**]** $==0$), then we know that the entry in **path_minimums** for b is b itself (because we initialized it this way). Otherwise, we (possibly) need to form path compression on b. After the path compression, the value in the **path_minimums** array is the block which has the smallest semidominator on the path in the DFS tree between b and $semi(b)$. In the first case, the paper's code simply returns b instead of doing a useless array lookup. We have deviated from this for purposes of clarity. Our implementation of **EVAL** always returns the value in the **path_minimums** array, once all path compression and recalculation of minimum semidominators along the path have been performed.

⟨Internal function bodies 19a⟩ ≡
```
static Unsigned_Int EVAL(Unsigned_Int block_num)
{
    if (ancestor[block_num])
        Compress(block_num);

    return path_minimums[block_num];
} /* EVAL */
```
◇

Macro defined by scraps 11b, 19ab, 26a, 35b.
Macro referenced in scrap 7a.

Path compression requires a recursive walk from a block to the root of its tree in the forest. Once we find the root, we will write its index into the `ancestor` entry for each of the blocks in the path.

We also keep track of the block along the path which has the lowest semidominator. Remember, however, that the list of blocks considered for this minimum does not include the root of the tree. Because of this, we only perform path compression for blocks which are more than one block away from a tree's root (*i.e.,* blocks which are not immediate descendants of the root block).[6]

⟨Internal function bodies 19b⟩ ≡
```
static Void Compress(Unsigned_Int block_num)
{
    if (⟨The path from block_num to the root is longer than one block 19c⟩)
    {
        Unsigned_Int parent = ancestor[block_num];

        Compress(parent);
        ⟨Update block_num's path_minimums entry 19d⟩
        ⟨Compress block_num's path 20a⟩
    }
    /* else no compression necessary */
} /* Compress */
```
◇

Macro defined by scraps 11b, 19ab, 26a, 35b.
Macro referenced in scrap 7a.

Because a root's immediate descendants already point to the root, and because the values in `path_minimums` do not take into account the root's semidominator, we need only perform path compression on blocks whose parent is not the root. Remember that we identify tree-roots by a zero in those blocks' `ancestor` entries.

⟨The path from `block_num` to the root is longer than one block 19c⟩ ≡
```
ancestor[ancestor[block_num]] != 0
```
◇

Macro referenced in scrap 19b.

Since this is a recursive routine, by the time we get to the code to update `block_num`'s values, `block_num`'s parent has been fully updated and contains the block whose semidominator is a minimum from the parent to the root of the tree. Thus, we need only compare the value in `block_num`'s `path_minimums` entry against its ancestor.

⟨Update `block_num`'s `path_minimums` entry 19d⟩ ≡
```
if (semidominator[path_minimums[parent]] < semidominator[path_minimums[block_num]])
    path_minimums[block_num] = path_minimums[parent];
```
◇

Macro referenced in scrap 19b.

[6] The immediate descendants of the root, of course, *already* have the correct values – that is, the descendant of a root in the forest is the only block to be considered in the path, so our initialization of `path_minimums` has already set up the correct value.

Likewise, the (path-compressed) ancestor of `block_num` (that is, the root of the tree containing `block_num`) is the same as `block_num`'s parent.

⟨Compress `block_num`'s path 20a⟩ ≡

```
ancestor[block_num] = ancestor[parent];
◇
```

Macro referenced in scrap 19b.

Step four is the final step in the calculation of dominators. We are not finished, however. We will use the calculation of dominators to calculate dominance frontiers. For more on this, see Chapter 4. Also, we need to convert the information we have just calculated into a form which the data structures in the shared library expect. This conversion is shown in the next section.

2.5 Changing Data Structures

The data structures in our shared code are slightly different from the ones which we have used up to this point. The main difference between the two is that this algorithm produces only the immediate dominator of each block. On the other hand, we will need to store the entire dominator tree; instead of computing each block's immediate parent in the dominator tree, this is somewhat like computing a block's immediate children in the dominator tree. Once we have calculated immediate dominators, however, we need only walk over the CFG, connecting children to their parents. We will do all of the translation into our shared data structures at this time.

The nice thing about this step, however, is that we can rely on all of the facts of the DFS tree when building these structures. Each block's `DomNode` data structure, for example, must be allocated before we build the tree. But if we simply iterate through the blocks in depth-first order, we know that we will always allocate the structure before we ever have need to add anything to it.[7]

⟨Rewrite into `shared` data structures 20b⟩ ≡

```
{
    Unsigned_Int i;

    for(i=1; i<= block_count; i++)
    {
        Block *block = DFS_block_list[i];
        Block *parent = (i==1) ? NULL : DFS_block_list[dominator[i]];

        block->dom_node = (DomNode *)Arena_GetMemClear(permanent_memory_bank,
                                                       sizeof(DomNode));
        block->dom_node->parent = parent;
        ⟨Add this block into its parent's list of dominator-tree children 21⟩
    }
}
◇
```

Macro referenced in scrap 7b.

Adding a linked list node takes just enough work that we separated it out into its own scrap to avoid obfuscating the surrounding code.

[7] Of course, we could do this walk and data structure allocation inside the loop which explicitly calculates dominators. We separate it out only for aesthetics.

⟨Add this block into its parent's list of dominator-tree children 21⟩ ≡

```
if (parent)
{
    Block_List_Node *temp = (Block_List_Node *)Arena_GetMem(permanent_memory_bank,
                                                             sizeof(Block_List_Node));

    temp->block = block;
    temp->next = parent->dom_node->children; /* safe, b/c initialized to NULL */
    parent->dom_node->children = temp;
}
```

◇

Macro referenced in scrap 20b.

Chapter 3

Calculating Post-Dominators

As we said in the first chapter, we use forward dominance information for such things as building SSA form. We will require backwards dominance information for such things as computing control dependence. This information will be used in optimization passes like the dead code remover, **dead** [3].

To calculate post-dominators, we need to perform the same algorithm which we applied to compute dominators, except that we do everything backwards. That is, instead of performing a depth-first search starting at the head of the CFG (`start_block`), we start the search at the bottom of the CFG (`end_block`). Likewise, where we iterate through a block's successors in the dominator algorithm, we will iterate through a block's predecessors in the post-dominator algorithm.

This loose formulation suggests an implementation solution: instead of writing essentially the same code twice, we will instead write a UNIX **awk** script which will create a new file whose contents are built by taking the file with the code to build dominators and textually substituting differences between the two algorithms.

The code for post-dominators can be built by first nuwebbing this file, and then executing the following command:

```
awk -f Post_Dominators.awk Dominators.c >! Post_Dominators.c
```

This command will be executed automatically by the makefile (the makefile is elaborated in Appendix A, on page 28).

"Post_Dominators.awk" 22 ≡
    {⟨Substitution commands 23a, … ⟩}
    ◇

The substitutions are shown in Figure 3.1.

`Dominator_CalcDom`	$\Longrightarrow$	`Dominator_CalcPostDom`
`Block_ForAllSuccs`	$\Longrightarrow$	`Block_ForAllPreds`
`Block_ForAllPreds`	$\Longrightarrow$	`Block_ForAllSuccs`
`->succ`	$\Longrightarrow$	`->pred`
`->pred`	$\Longrightarrow$	`->succ`
`->dom_node`	$\Longrightarrow$	`->post_dom_node`
`start_block`	$\Longrightarrow$	`end_block`

Figure 3.1: Lexical substitutions required to change the code to compute backward dominance information instead of forward dominance information

Notice that we include C's arrow "`->`" in the substitution of structure fields so that we do not accidentally overwrite some other variable name.

Note: The **awk** global-substitution command (`gsub`) works as follows: It takes three parameters. The first is a regular expression representing the target. The second parameter is a regular expression representing the replacement string. The third parameter, which is optional, represents the part of each input line on which to perform the substitution. The value `$0` tells `gsub` to perform the substitution over the entire line. While it is an optional parameter, the substitutions are not cumulative if it is not included.

⟨Substitution commands 23a⟩ ≡

```
gsub("->dom_node", "->post_dom_node", $0);
gsub("start_block", "end_block", $0);
gsub("Dominator_CalcDom", "Dominator_CalcPostDom", $0);
◇
```

Macro defined by scraps 23abcd.
Macro referenced in scrap 22.

Notice that we have to replace the macros `Block_ForAllSuccs` and `Block_ForAllPreds` with each other. This requires the use of a temporary string and an extra replacement, much like the use of a temporary variable when swapping values in code.

⟨Substitution commands 23b⟩ ≡

```
temp_string = "A_VeRy_LoNg_StRiNg_WhIcH_Is_UnLiKeLy_To_ApPeAr_ElSeWhErE";
gsub("Block_ForAllSuccs", temp_string, $0);
gsub("Block_ForAllPreds", "Block_ForAllSuccs", $0);
gsub(temp_string, "Block_ForAllPreds", $0);
◇
```

Macro defined by scraps 23abcd.
Macro referenced in scrap 22.

We have to perform a similar conversion for the `pred` and `succ` fields of `Edge`s.

⟨Substitution commands 23c⟩ ≡

```
gsub("->succ", temp_string, $0);
gsub("->pred", "->succ", $0);
gsub(temp_string, "->pred", $0);
◇
```

Macro defined by scraps 23abcd.
Macro referenced in scrap 22.

Having done all of the substitutions, the line must be printed back out. The output routine in **awk** looks and acts very similar to C's `printf` procedure.

⟨Substitution commands 23d⟩ ≡

```
printf "%s\n", $0;
◇
```

Macro defined by scraps 23abcd.
Macro referenced in scrap 22.

With these substitutions, our new entry point is called `Dominator_CalcPostDom`. This entry point calculates the post-dominator tree and post-dominance frontiers for the CFG and puts this information in each block's `post_dom_node` field.

Chapter 4

Calculating Dominance Frontiers

The calculation of dominance frontiers is critical to the ϕ-nodeplacement phase of SSA construction. We take our implementation from Cytron *et al.* [4].

Before we begin, we need to describe some terms. First, for this section of the document, when we use the term dominator, we are referring to *any* of a block's ancestors in the dominator tree. Further, we need to differentiate dominance from *strict* dominance. Recall that a block b_1 is said to dominate a second block b_2 if there is no path from the starting block to b_2 which does not pass through b_1. A block is also said to dominate itself. *Strict* dominance, then, is a dominance relationship between the two blocks where b_1 does not equal b_2.

The *dominance frontier* of a block b_1 is the set of blocks such that, for each block b_2 in the set, b_1 dominates a predecessor of b_2 but does not *strictly* dominate b_2. To say it another way, b_2 is in b_1's dominance frontier if it has a predecessor which is dominated by b_1 but is not itself dominated by b_1.

Figure 4.1 shows a section of a CFG representing a simple `if` statement. Control flows out of the branching block into either of the two successors, with control flowing back together in the subsequent block. Looking at our definition of dominance frontiers, we can see that block 2 dominates block 5 but does not dominate block 6 (there is a path from block 1 to block 6 which avoids block 2). Thus, block 6 is in block 2's dominance frontier.

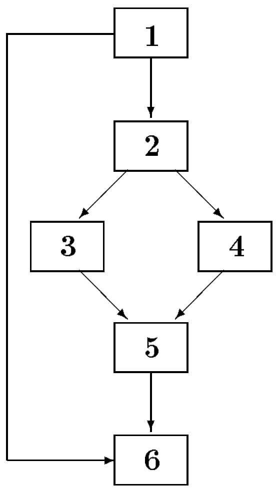


Figure 4.1: Simple dominance frontier example

Cytron's illustration at the end of section 1.1 of Chapter 1 gives us insight into how we will calculate dominance frontiers. If we were to use the first definition we gave, we would have to search the entire graph to find each block's dominance frontier, producing a quadratic algorithm. Instead, we will redefine a block's dominance frontier as the union of two different sets.

The first of these two sets is comprised of the children of a block which that block does not dominate. The only way a block would not dominate one of its children is if that child had another incoming edge. In Cytron's example, light would flow along this other edge, illuminating the child even while its parent is dark.[1]

The second set of blocks in a particular block's dominance frontier results from contributions from that block's children in the dominator tree. Specifically, we take the union of the dominance frontiers of all of a block's dominator-tree children as candidates. From this set, we examine each member to see if its immediate dominator is our target block. If not, then it is a member of the block's dominance frontier. Again using Cytron's model, imagine putting one's finger over the target block's node. All of its descendants in the dominator tree will go dark, but the blocks in those children's dominance frontiers will stay illuminated – *unless those blocks are dominated by the target block*. Indeed, the only way for one of these blocks to be dominated by the target block is if it is *immediately* dominated by the target block, since the block is not dominated by any of the target block's dominator-tree children.

The algorithm, then, for building dominance frontiers is as follows: We will perform a bottom-up traversal of the dominator tree. For each block, we will calculate which of its CFG successors are in its dominance frontier. Then we will search the dominance frontiers of each of its dominator-tree children, putting each member which is not dominated by the current block in the current block's dominance-frontier set.

At each iteration, we will build up the current block's dominance frontier (*i.e.,* both of the above-mentioned sets) in a `SparseSet`. After the dominance frontier has been fully calculated, we will transfer the information from the `SparseSet` to our `DomNode` structure. We have defined this as a recursive routine so that a node's dominator-tree children will all have been processed prior to that node being processed. Because the routine is recursive, we need to allocate the `SparseSet` as a global variable so that we can reuse it at each iteration, but only allocate the memory once.

⟨Internal global variables 25a⟩ ≡

```
SparseSet frontier_set;
◇
```

Macro defined by scraps 8a, 9ad, 10b, 12, 13c, 25a.
Macro referenced in scrap 7a.

⟨Calculate dominance frontiers 25b⟩ ≡

```
frontier_set = SparseSet_Create(local_memory_bank, block_count+1);
calculate_dominance_frontiers(start_block, permanent_memory_bank);
◇
```

Macro referenced in scrap 7b.

Note: We assume that we do this step *after* we have allocated the `DomNode` data structures for each block and have filled in these data structures to represent the dominator tree.

Note: We pass in the `permanent_memory_bank` arena because we will be adding linked-list nodes onto each block.

[1] Remember that a block dominates itself, so the dominance-frontier definition holds.

⟨Internal function bodies 26a⟩ ≡

```
static Void calculate_dominance_frontiers(Block *current_block,
                                          Arena permanent_memory_bank)
{
    Unsigned_Int current_block_num = current_block->block_extension->DFS_name;

    ⟨Build up current_block's children's dominance frontiers 26b⟩
    SparseSet_Clear(frontier_set);
    ⟨Look at current_block's CFG children 26c⟩
    ⟨Look at the dominance-frontier sets of current_block's children 27a⟩
    ⟨Convert the SparseSet into our CFG data structures 27b⟩
} /* calculate_dominance_frontiers */
◇
```

Macro defined by scraps 11b, 19ab, 26a, 35b.
Macro referenced in scrap 7a.

Before we can calculate `current_block`'s dominance frontier, we need to build all of its dominator-tree children's dominance frontiers. We do this by recursively calling `calculate_dominance_frontiers`.

⟨Build up current_block's children's dominance frontiers 26b⟩ ≡

```
{
    Block_List_Node *child_runner;

    Dominator_ForChildren(child_runner, current_block->dom_node)
        calculate_dominance_frontiers(child_runner->block, permanent_memory_bank);
}
◇
```

Macro referenced in scrap 26a.

Once dominance frontiers of the children of `current_block` have been built, we need to build the two sets necessary to compute `current_block`'s frontier set. The first of these sets is computed by iterating over the CFG successors of `current_block`, adding any block which is not dominated by `current_block`.

⟨Look at current_block's CFG children 26c⟩ ≡

```
{
    Edge *succ_runner;

    Block_ForAllSuccs(succ_runner, current_block)
    {
        Unsigned_Int succ_block_num = succ_runner->succ->block_extension->DFS_name;

        if (dominator[succ_block_num] != current_block_num)
            SparseSet_Insert(frontier_set, succ_block_num);
    }
}
◇
```

Macro referenced in scrap 26a.

The second set we need to build is constructed from the union of all of `current_block`'s dominator-tree children's frontier sets. Any member of this union which is not dominated by `current_block` gets added to `frontier_set`.

⟨Look at the dominance-frontier sets of current_block's children 27a⟩ ≡

```
{
    Block_List_Node *child_runner;

    Dominator_ForChildren(child_runner, current_block->dom_node)
    {
        Block_List_Node *frontier_runner;

        Dominator_ForFrontier(frontier_runner, child_runner->block->dom_node)
        {
            Unsigned_Int frontier_block_num =
                        frontier_runner->block->block_extension->DFS_name;

            if (dominator[frontier_block_num] != current_block_num)
                SparseSet_Insert(frontier_set, frontier_block_num);
        }
    }
}
```

◇

Macro referenced in scrap 26a.

The final step in each iteration of building a block's dominance frontier is to dump the information in **frontier_set** into a linked list of blocks, which is the data structure specified by **Parser.h**.

⟨Convert the SparseSet into our CFG data structures 27b⟩ ≡

```
{
    Unsigned_Int set_runner;

    SparseSet_ForAll(set_runner, frontier_set)
    {
        Block_List_Node *temp = (Block_List_Node *)Arena_GetMem(permanent_memory_bank,
                                                        sizeof(Block_List_Node));

        temp->block = DFS_block_list[set_runner];
        temp->next = current_block->dom_node->frontier;
        current_block->dom_node->frontier = temp;
    }
}
```

◇

Macro referenced in scrap 26a.

Appendix A

Makefile

The makefile for this code is important, since it will build the post-dominance code from the dominance code. Since this code is intended to be part of our shared code, we envision this makefile being called as a subprocess of the makefile which builds all of the rest of the shared code. Many of the variables which we need will be passed in to us.

"Dominators_Makefile" 28a ≡
⟨Commands to build the dominator object files 28b, ... ⟩
◇

To begin, if `Dominators.c` is out of date, we need to build it.

The variable `ENV_FLAG` is used to specify to the compiler to produce either code with debugger symbols or optimized code. Other makefiles use the `DEBUG_FLAG` variable to control the insertion of debugging code. Since we do not have any of those constructs in this code, we will ignore this flag. Thus, the only flags we need to pass to the C compiler will be the location of the include files and the `ENV_FLAG`.

⟨Commands to build the dominator object files 28b⟩ ≡

```
Dominators.c: Dominators.w
        nuweb Dominators.w
```
◇

Macro defined by scraps 28bc, 29abc.
Macro referenced in scrap 28a.

Likewise, if `Post_Dominators.c` is out of date, we need to build it.

Note: We use "`>!`" to force output redirection and overwriting when the file already exists. It is unclear that this is portable. However, this is the cleanest way to circumvent the `noclobber` shell variable.

⟨Commands to build the dominator object files 28c⟩ ≡

```
Post_Dominators.c: Dominators.c Post_Dominators.awk
        if test -f Post_Dominators.c; then \
        awk -f Post_Dominators.awk Dominators.c >! Post_Dominators.c; \
        else \
        awk -f Post_Dominators.awk Dominators.c > Post_Dominators.c; \
        fi
```
◇

Macro defined by scraps 28bc, 29abc.
Macro referenced in scrap 28a.

Of course, we cannot build the code for post-dominators if the `awk` script we use to build the file is out of date.

⟨Commands to build the dominator object files 29a⟩ ≡

```
Post_Dominators.awk: Dominators.w
        nuweb Dominators.w
◇
```

Macro defined by scraps 28bc, 29abc.
Macro referenced in scrap 28a.

The above rules build each piece of code separately. Usually, we will require that everything be built. Thus, we will use `make`'s "`all`" identifier to do this. Notice that we do not have to actually give this identifier a command. Both object files will get built according to the commands associated with their identifiers.

⟨Commands to build the dominator object files 29b⟩ ≡

```
all: Dominators.o Post_Dominators.o
◇
```

Macro defined by scraps 28bc, 29abc.
Macro referenced in scrap 28a.

Finally, if we want to create the document from this web file, we will invoke the following command.

⟨Commands to build the dominator object files 29c⟩ ≡

```
Dominators.aux<: Dominators.tex
        latex Dominators
        nuweb Dominators
        latex Dominators

Dominators.bbl: Dominators.aux
        bibtex Dominators
        latex Dominators

Dominators.tex: Dominators.w
        nuweb Dominators.w

Dominators.dvi: Dominators.tex Dominators.aux Dominators.bbl
        latex Dominators.tex

view: Dominators.dvi
        xdvi -margins 1 -S 30 Dominators.dvi
◇
```

Macro defined by scraps 28bc, 29abc.
Macro referenced in scrap 28a.

Appendix B

Function Prototypes

This appendix will elaborate the function prototypes used both internally and provided for external use.

B.1 External function prototypes

This module provides two external entry points. The first, `Dominator_CalcDom`, calculates forward dominance information. The second entry point, `Dominator_CalcPostDom`, calculates backwards dominance information. The user of this module sends in an arena from which we can allocate the dominator data structures.

However, because these function prototypes are already elaborated in `shared.h`, we will not define them as an actual scrap.

extern Void Dominator_CalcDom(Arena permanent_memory_bank);
extern Void Dominator_CalcPostDom(Arena permanent_memory_bank);

B.2 Internal function prototypes

Despite the usual web style of inlining as much code as possible, it is sometimes necessary or valuable to proceduralize certain sections. A prime example of when this is necessary is for the use of recursion. This code makes use of three internal recursive routines, `build_DFS_tree`, which does a depth-first search over the CFG and builds the DFS tree, `calculate_dominance_frontiers`, which builds the lists of blocks in the dominance frontiers of other blocks in the CFG, and `Compress`, the routine from Lengauer and Tarjan which performs path compression in the forest during steps 2 and $3a$.

⟨Internal function prototypes 30⟩ ≡

```
static Void build_DFS_tree(Block *current_block, Arena local_memory_bank);

static Void calculate_dominance_frontiers(Block *current_block,
                                          Arena permanent_memory_bank);

static Void Compress(Unsigned_Int block_num);
```
◇

Macro defined by scraps 30, 31ab, 33b.
Macro referenced in scrap 7a.

Another example of when we may choose to separate code into its own function is when, for purposes of clarity, we are trying to match a paper's implementation. The `EVAL` function is an example of this case. This function mimics the functionality provided by Lengauer and Tarjan's function of the same name. That is, for the block passed in (`block_num`), it returns the minimum semidominator along the path in the DFS tree between `block_num` and its semidominator or `block_num` itself if `block_num`'s semidominator has not yet been computed.

⟨Internal function prototypes 31a⟩ ≡

```
static Unsigned_Int EVAL(Unsigned_Int block_num);
```

◇

Macro defined by scraps 30, 31ab, 33b.
Macro referenced in scrap 7a.

Finally, we have the prototype for the debugging routine used by `Dominator_Dump`.

⟨Internal function prototypes 31b⟩ ≡

```
static Void DumpTree(Block * root, Boolean domflag, Unsigned_Int depth);
```

◇

Macro defined by scraps 30, 31ab, 33b.
Macro referenced in scrap 7a.

Appendix C

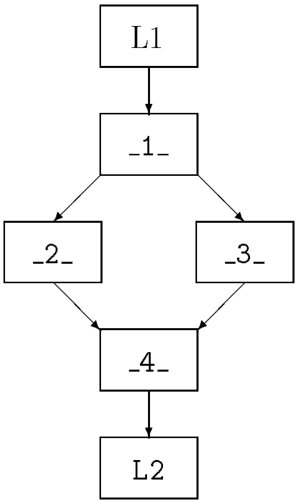
Debugging Routines

This chapter shows the debugging routines we used while developing this code. We divided this appendix section into two parts. The first part shows the routines we used during debugging of this routine. We also have an external entry point, `Dominator_Dump`, which is the routine which was created with the first incarnation of our dominator-calculation implementation, and is intended to be accessed by users of this module.

C.1 Internal debugging routines

We created a single external entry point to dump all of the dominator information. It prints out the information in a tabular format. Although this should be a function returning `Void`, we make it an integer so that the C compiler will not complain that we have not made it visible through the use of a header file.

The output from this routine is shown in Figure C.1.



NAME(DFS #)	PARENT	SEMI	DOM	ANC	PATH_MIN
L1(1)	ERR-0	L1-1	ERR-0	ERR-0	L1-1
_1_(2)	L1-1	L1-1	L1-1	L1-1	_1_-2
_3_(3)	_1_-2	_1_-2	_1_-2	_1_-2	_3_-3
_4_(4)	_3_-3	_1_-2	_1_-2	L1-1	_1_-2
L2(5)	_4_-4	L1-1	L1-1	L1-1	L2-5
_2_(6)	_1_-2	_1_-2	_1_-2	_1_-2	_2_-6

Figure C.1: Output from `Dominator_dump_info`

This figure shows a simple CFG and the table that would be produced by `Dominator_CalcDom`. The table shows both block names and DFS tree indices for each block listed. In the first column, the names and indices are separated by parentheses, while in the remaining columns they are separated by a dash (this saves one character's worth of space). In all but the first column, the name is concatenated to `MAX` characters in order to fit into the columns. The first column lists the blocks in DFS order. The remaining columns correspond to the entries in the following arrays (respectively):

1. DFS_tree_parents
2. semidominator
3. dominator
4. ancestor
5. path_minimums

The string ERR is used to indicate values out of range. The number following this dash is the incorrect integer value. Note that this string will always occur for the start block in the PARENT, DOM, and ANC fields, as those entries are initialized to zero (and blocks are numbered starting at one).

⟨Externally-visible function bodies 33a⟩ ≡

```
Int Dominator_dump_info(Void)
{
    Unsigned_Int i;

    printf("NAME(PRE_ORD)\t\tPARENT\tSEMI\tDOM\tANC\tPATH_MIN\n");

    for(i=1;i<= block_count; i++)
    {
        ⟨Print each field of debug output 33c⟩
    }
    printf("\n\n");
} /* Dominator_dump_info */
◇
```

Macro defined by scraps 7b, 33a, 34ab, 35a.
Macro referenced in scrap 7a.

To print each field of the debug table, we will need to perform almost exactly the same set of calls. As such, we will define a macro which will act like a function call so that we do not have to have multiple copies of the printing steps for each field.

⟨Internal function prototypes 33b⟩ ≡

```
static Char *strtrunc(Char *input_string);
#define PRINT_NEXT_FIELD(field) \
        printf("%s-%d\t", (field > 0 && field <= block_count) ? \
            strtrunc(Expr_Get_String(DFS_block_list[field]->labels->label)) : \
            "ERR", \
            field)
◇
```

Macro defined by scraps 30, 31ab, 33b.
Macro referenced in scrap 7a.

Notice that the first column is slightly different, so we cannot use the above macro.

⟨Print each field of debug output 33c⟩ ≡

```
printf("%s(%d)\t\t", (i==0) ? "NONE" :
                                Expr_Get_String(DFS_block_list[i]->labels->label),
                        i);
PRINT_NEXT_FIELD(DFS_tree_parents[i]);
PRINT_NEXT_FIELD(semidominator[i]);
PRINT_NEXT_FIELD(dominator[i]);
PRINT_NEXT_FIELD(ancestor[i]);
PRINT_NEXT_FIELD(path_minimums[i]);
printf("\n");
◇
```

Macro referenced in scrap 33a.

Since Dominator_dump_info prints its information in a tabular format, we need to truncate expressions to fit.

⟨Externally-visible function bodies 34a⟩ ≡
```
static Char *strtrunc(Char *input_string)
{
#define MAX 4
    static char too_long[MAX+1];
    Unsigned_Int i;

    for(i=0; i<MAX; i++)
    {
        too_long[i] = input_string[i];
        if (!too_long[i])
            break;
    }

    if (i==MAX)
        too_long[i] = '\0';

    return too_long;

} /* strtrunc */
```
◇

Macro defined by scraps 7b, 33a, 34ab, 35a.
Macro referenced in scrap 7a.

Finally, since debugging recursive routines can be tedious interactively, we used this routine to print out the DFS tree after it had been computed.

⟨Externally-visible function bodies 34b⟩ ≡
```
static Int print_dfs_tree(Void)
{
    Unsigned_Int i;

    printf("PREORD\tDFS\tLABEL\tPARENT_LABEL\n");

    for(i=2; i<= block_count; i++)
    {
        printf("%d\t%d\t%s\t%s\n", i, DFS_tree_parents[i],
                Expr_Get_String(DFS_block_list[i]->labels->label),
                Expr_Get_String(DFS_block_list[DFS_tree_parents[i]]->labels->label));
    }
    printf("\n\n");
} /* print_dfs_tree */
```
◇

Macro defined by scraps 7b, 33a, 34ab, 35a.
Macro referenced in scrap 7a.

C.2 External debugging routines

This section shows the code for the entry point Dominator_Dump, which is useful to those including this module in their code. This routine prints out the information stored in each block's dom_node or post_dom_node structures, depending on the value of domflag. If domflag's value is TRUE, dominance information is printed. If it's value is FALSE, post-dominance information is printed.[1]

[1] This code was originally written by Chris Vick.

⟨Externally-visible function bodies 35a⟩ ≡

```
Void Dominator_Dump(Block * root, Boolean domflag)
{
  Unsigned_Int    depth=0;

  Block_Clear_Visited_Flags();
  DumpTree(root,domflag,depth);  /* dump the dominator information */
} /* Dominator_Dump */
◇
```

Macro defined by scraps 7b, 33a, 34ab, 35a.
Macro referenced in scrap 7a.

We perform a recursive walk over the dominator tree, printing out each block's values as we reach it. The `depth` variable allows us to keep track of how many levels deep we are in the walk so that we can print the information out with proper indentation.

⟨Internal function bodies 35b⟩ ≡

```
static Void DumpTree(Block * root, Boolean print_forward_info, Unsigned_Int depth)
{
    Block_List_Node *child, *front;

    if((root == NULL) || (root->visited == TRUE)) return;
    depth = depth + 1;
    fprintf(stderr,"\n");
    ⟨Insert indentation 36a⟩
    fprintf(stderr,"Block Index = %d\n",root->preorder_index);
    root->visited = TRUE;
    if(print_forward_info)
    {
        front = root->dom_node->frontier;
        ⟨Insert indentation 36a⟩
        fprintf(stderr,"Dominance Frontier = ");
        ⟨Print the values in the frontier list 36c⟩
        child = root->dom_node->children;
        ⟨Recur over the children in the dominator tree 36b⟩
    }
    else
    {
        front = root->post_dom_node->frontier;
        ⟨Insert indentation 36a⟩
        fprintf(stderr,"PostDominance Frontier = ");
        ⟨Print the values in the frontier list 36c⟩
        child = root->post_dom_node->children;
        ⟨Recur over the children in the dominator tree 36b⟩
    }
} /* DumpTree */

◇
```

Macro defined by scraps 11b, 19ab, 26a, 35b.
Macro referenced in scrap 7a.

We pulled the following three scraps out because they can be reused for printing both forward and backwards dominance information.

⟨Insert indentation 36a⟩ ≡

```
{
    Int i;

    for(i = 0; i <= depth; i++)
        fprintf(stderr,"\t");
}◇
```

Macro referenced in scrap 35b.

⟨Recur over the children in the dominator tree 36b⟩ ≡

```
while(child != NULL)
{
    DumpTree(child->block, print_forward_info, depth);
    child = child->next;
}
◇
```

Macro referenced in scrap 35b.

⟨Print the values in the frontier list 36c⟩ ≡

```
while(front != NULL)
{
    fprintf(stderr," %d,",front->block->preorder_index);
    front = front->next;
}
◇
```

Macro referenced in scrap 35b.

Appendix D

Memory Use Diagram

Figure D.1 shows our standard memory diagram. This diagram is much simpler than for other webs in the Massively Scalar Compiler, since we really do not use very much memory to calculate dominators.

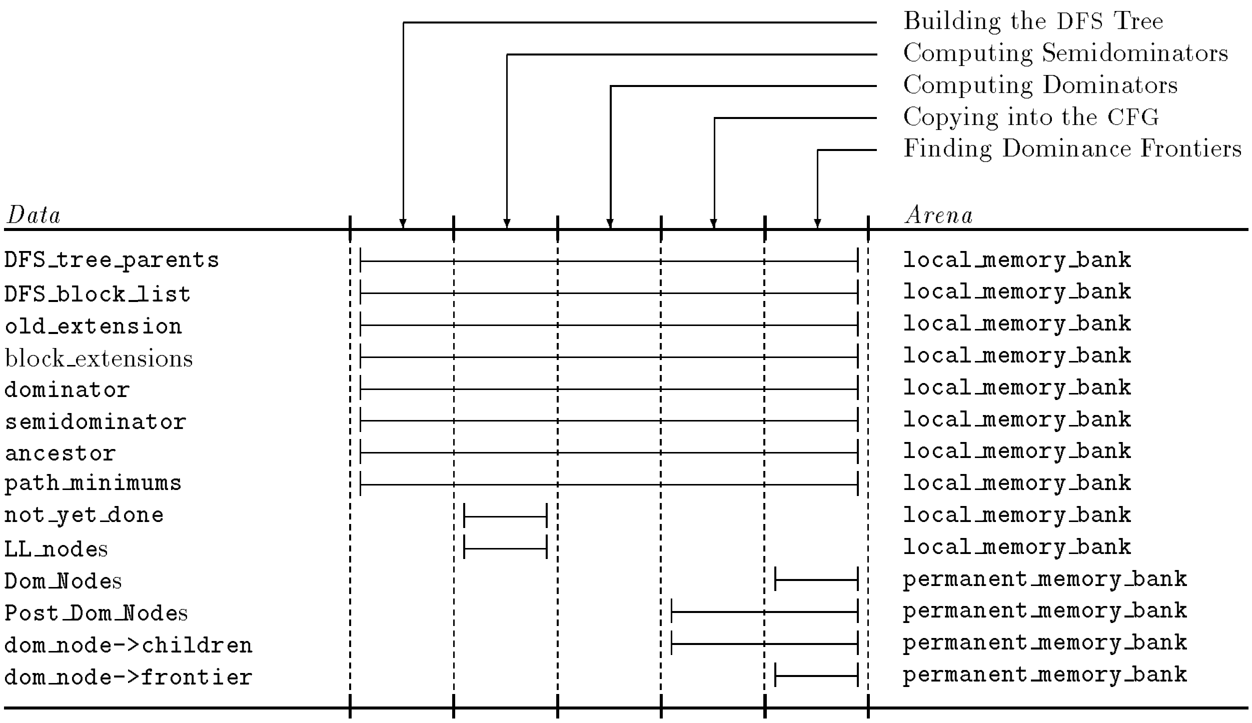


Figure D.1: Memory-use diagram for calculating dominators

Appendix E

Indices

E.1 Files

E.2 Macros

E.3 Identifiers

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Preston Briggs. The massively scalar compiler project. Technical Report ONR grant N00014-91-J-1989, Rice University, July 1995. Not yet available.

[3] Preston Briggs, Rob Shillner, and Taylor Simpson. Dead code elimination. Technical Report ONR grant N00014-91-J-1989, Rice University, June 1993. Available for anonymous ftp.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global flow problems. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 207–217, Boston, Massachusetts, October 1973.

[6] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.