

Comp512, Spring 2006
Lab 2 – Building SSA
Due March 6, 2006

The purpose of this lab is to give the student practical experience building and debugging an SSA implementation.

Your SSA implementation should consist of three main steps: data-flow analysis, phi-node insertion, and renaming. This writeup should serve as a guide to the specific data structures you should use. The lecture notes should give enough information to build SSA, but you should use the data structures found in `/home/comp512/iloc/include/SSA.h`, as this will greatly simplify your lab. All of the data structures described herein are further elaborated on in the `SSA_Chapter1.ps` file in the `iloc/docs` directory. This file is the first chapter out of our SSA implementation, and it describes all of the data structures in `SSA.h`. You should use this document as your guide to building SSA and use the chapter from our implementation to supplement this spec – remember that the chapter touches on some irrelevant topics that your specialized SSA builder will not need to concern itself with.

1 Data-flow analysis

You calculated liveness in your first lab, so the only other analysis you'll need is dominance. Note that `SSA.h` includes a data structure for liveness information; you are free to ignore it.

The dominator calculation, as defined by Cooper *et al.* (go to the 512 website for the paper) can be made using an iterative scheme, much like the liveness calculation from the last lab.

This formulation, suggested by Hecht and Ullman, lets the compiler compute DOM sets as a forward data-flow problem. Given a CFG, $G = (N, E, n_0)$, where N is a set of nodes, E is a set of directed edges, and n_0 is the designated entry node for the CFG, the following data-flow equations define the DOM sets:

$$\text{DOM}(n_0) = \{n_0\}$$
$$\text{DOM}(n) = \left(\bigcap_{p \in \text{preds}(n)} \text{DOM}(p) \right) \cup \{n\}$$

We assume that the nodes are numbered in postorder, and that *preds* is a relation defined over E that maps a node to its predecessors in G .

As the paper points out, a naive implementation of the above equations requires an unworkable amount of time to complete. By following the algorithm in the paper, however, your implementation will be competitive with any other implementation out there.

The data structures you need are laid out in the paper. For the next two steps, you'll need to use our data structures, but for this step, you can roll your own – the information will only be used by you internally, and it will be thrown away after the renaming step.

2 Placing phi-nodes

In the second step for building SSA, you'll need to insert phi-nodes. The phi-node data structure is as follows:

```
typedef struct phi_node Phi_Node;
struct phi_node
{
    Phi_Node *next;
    Unsigned_Int2 old_name;
    Unsigned_Int2 new_name;
    Unsigned_Int2 def_type; /* values defined in Shared.h */
    Boolean useless;
    Unsigned_Int2 parms[1];
};
```

You should store phi-nodes in an array indexed, as usual, by a block's pre-order index. Use the name given in the header file: *SSA_phi_nodes*. Each entry in *SSA_phi_nodes* will be a linked list of *Phi_Node* structures – thus the *next* pointer. For each phi-node, you'll need to keep track of its original name (*old_name*) and its SSA name (*new_name*). You'll need to keep track of the type of the phi-node (integer, float, *etc.*) in the *def_type* field. You can ignore the *useless* field; we use this in a separate pass over the code to determine if the algorithm put in a phi-node we didn't need to, but your lab will not require this additional work. Finally, the *parms* array will hold the parameters of the phi-node. This array works like the *arguments* array in operations: it is sized when you allocate the data structure. For example, if you want to insert a phi-node into a block with *n* predecessors, you might use the following command:

```
new_phi_node = Arena_GetMemClear(phi_node_memory_bank,
                                sizeof(Phi_Node) + ((n-1)*sizeof(Unsigned_Int2)));
```

Of course, the above should be a macro... Notice that we zero-out the data structure. This is mostly to ensure that all of the phi-node parameters start out

at zero. It's possible (probable) that a value will not be defined down all paths in a program, and this does not necessarily mean that the code is wrong, so we have to represent it somehow. Our method (and yours, too) is to reserve zero to show any uninitialized values.

3 Renaming

In the renaming step, you'll need to set up a number of variables defined in `SSA.h`, so be careful in your implementation.

In class, we added subscripts to the original names as we renamed them in SSA form, but this was merely for clarity of exposition. For ILOC, you'll rename registers starting at one (don't use zero; as we explained above, we use zero for undefined phi-node parameters). Remember that these renamings are done in place; you'll overwrite the names in the `cfg` with SSA names. To get out of SSA, we'll need to have a mapping from original names back to SSA names, and for this, we use `SSA_name_map`. This is an array indexed by SSA name (since they're just integers), where each index is the original name that SSA name replaced.

Probably the most seemingly complex data structure is the `SSA_edge_maps` data structure. Consult the `SSA_Chapter1.ps` file in the `spring01/docs` directory for a full explanation – it does a fine job of explaining this necessary data structure. You can ignore the `parm_map` data structure; it had to be made global for our register allocator, but it's not something you need to worry about.

The `SSA_def_count` variable is analagous to `register_count`, which you used to build liveness. It holds the highest SSA name plus one. To compute this, you will need to walk over the code, counting the number of definitions (including phi-nodes).

Some of the optimizations require us to be able to map back to a block or an instruction where the SSA name was defined. To do this, as you create new SSA names, write down in the `SSA_block_map` and `SSA_inst_map` arrays the current block and instruction you're in. Both of these arrays, of course, are indexed by SSA name.

It is imperative that, as the last thing you do in your SSA builder, you set the `SSA_CFG_renumbered` variable. This tells all the rest of the library code that you've modified the CFG.

One of the things that SSA form gives us is a compact set of def-use or use-def chains. Obviously, which of these a particular optimization needs is peculiar to that optimization. Thus, you only need implement what your other optimization is going to need.

Of course, if you really want to exercise your SSA builder, you can plug it into an installed optimization by simply relinking. We'll give details later in the newsgroup, but to do this, you'll have to name your SSA builder `SSA_Build` and match the parameter list in the header file. (This doesn't mean that you'll

have to build all of the flavors of SSA to do this; I'll explain in the newsgroup how to fool the calling code.)

You will not be responsible for converting out of SSA. In theory, if you build all of the data structures correctly, you'll be able to use *SSA_Replace_Phi_Nodes* and *SSA_Restore* on your converted code. Again, we'll elaborate any problems in the newsgroup.