Finding Dominators

CSE 501 Spring 2013 Compilers Assignment 1

Andre Baixo, Jacob Nelson

April 17, 2013

This writeup describes our optimizer implementation for Assignment 1.

1 Implementation

Our optimizer implements the dominance algorithm described in [1].

Programs are processed in four phases. In the first phase, we parse the textual representation of the intermediate language output from the Start front-end and create an object for each element found. Non-executable elements like the *method* construct are stored separately from the executable instructions.

In the second phase, we form the instructions into basic blocks. To do this, we use the *method* elements in the input to partition the set of instructions; then we iterate over the instructions in each method's set and divide them into basic blocks based on branch (and other) instructions as well as branch targets. We add CFG edges due to program order in this phase.

In the third phase, we finish forming the control flow graph by adding edges between the basic blocks based on branch targets. We store both successor and predecessor edges to make traversal cheaper.

Finally, we compute dominators for each method. The first step here is to compute a topological order over the basic blocks in the method. We do this by running a depth-first search over the CFG. Once we have that order, we execute the dominance algorithm, iteratively refining each block's immediate dominator until nothing changes. When that is complete, we store the immediate dominators in the blocks and add the dominator tree edges to the CFG.

Our optimizer writes two forms of output. First, it dumps the all the requested information about blocks, instructions, and dominators to a file. Second, it writes the CFGs in GraphViz (dot) format.

The optimizer may be run by executing the run.sh script in the `assignment1` directory. We have included precompiled intermediate languages files for all examples in the distribution. Running the script will create separate output files in the `examples` directory for each method of each example; dump information has the extension `.txt`, and CFG output has the extension `.dot`. For example, the resuls for the GCD method in `gdc.dart` can be found in `examples/gcd.dart.il-main-info.txt`. The GraphViz files can be turned into PDF with a command like `dot -Tpdf file.dot > file.pdf`.

Our optimizer is implemented in Ruby, so no separate build step is required. Ruby version 1.9 is required.

2 Examples

We implemented three additional examples to better understand our implementation. The first is `examples/hammock.il`, a simple dual hammock shown in Figure 1. The second is `examples/weave.il`, a more complicated set of woven branches shown in Figure 2. Both of these examples provide a simple verification of our dominator construction.

The third example is a script that generates nested *if* constructs of different sizes. The Ruby file `examples/nested-if.rb` takes two parameters: a total if-construct count, and a nesting depth. The nesting depth controls how many levels of nested if constructs are generated. Figure 3 shows a CFG from a 10-node, singly-nested example; figure 4 shows a 10-node, 5-nested example, and figure 5 shows a 10-node, 10-nested example.

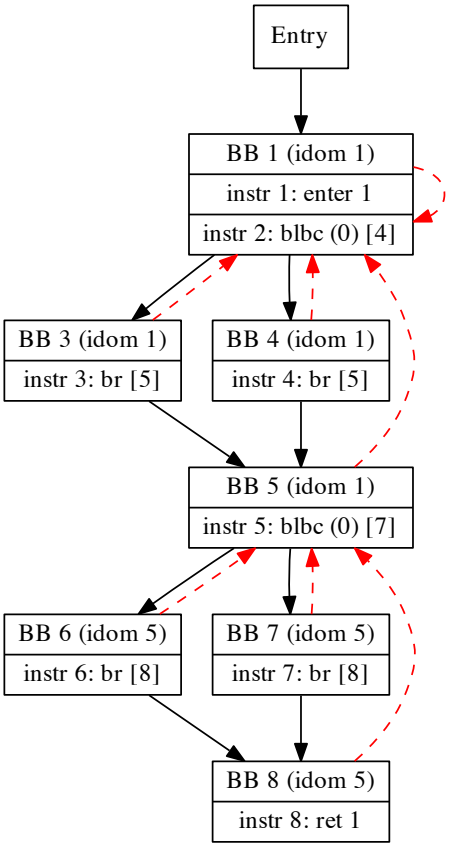

Figure 1: CFG from hammock example. Red edges show dominator relationships.

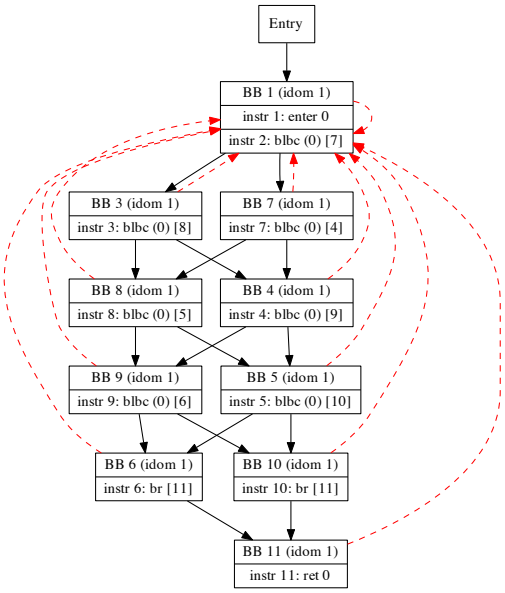

Figure 2: CFG from weave example. Red edges show dominator relationships.

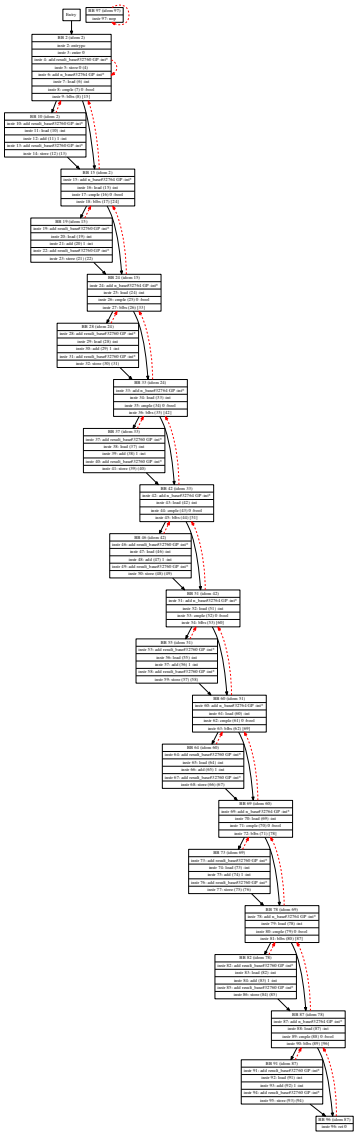

Figure 3: CFG from 10-if singly-nested example. Red edges show dominator relationships.

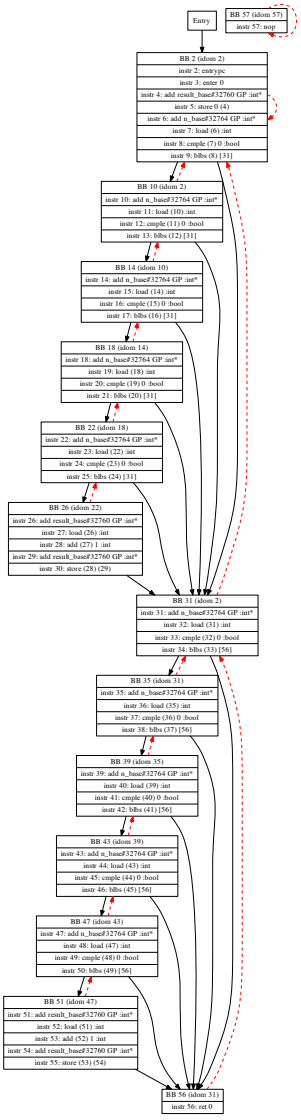

Figure 4: CFG from 10-if 5-nested example. Red edges show dominator relationships.

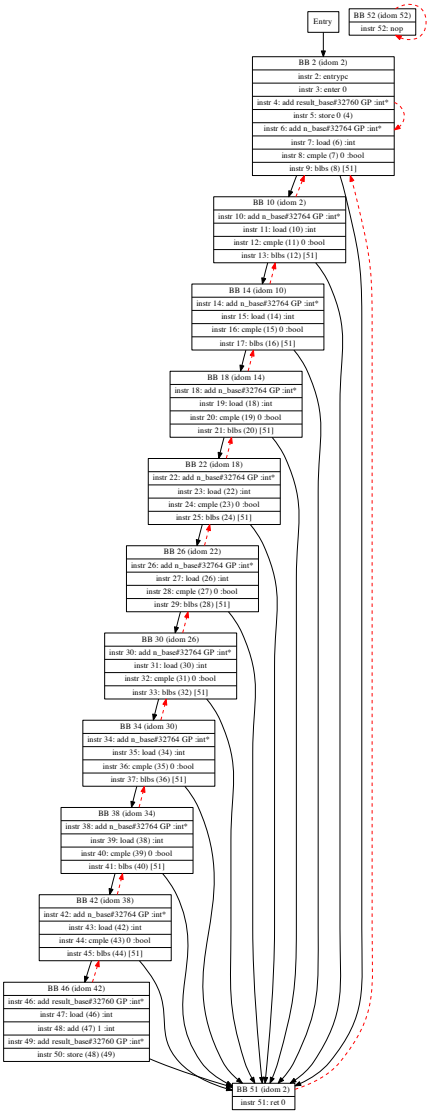

Figure 5: CFG from 10-if 10-nested example. Red edges show dominator relationships.

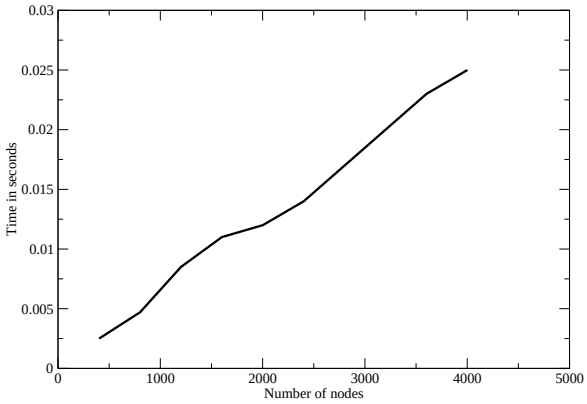

Figure 6: Relationship between CFG node count and runtime in nested if example with nesting depth of 1.

This lets us generate different families of programs with different numbers of nodes and edges, as well as different patterns of CFG connectivity. In Section 3, we use this script to explore how the structure of the CFG affects runtime.

3 Results

We explored the performance of our implementation in three ways. For all our experiments, we ran on an Ivy Bridge-based MacBook Air with 8GB of RAM.

First, we processed all the example programs provided. The longest-running method was the main method from `mmm.dart`, which ran for about 200 microseconds. The examples can be run by executing the `assignment1/run.sh` script. This

Since this time is too short to make meaningful comparisons, we then used our nested-if generator to create a set of larger programs with different CFG shapes. Unfortunately we were hampered by our implementation choices: by depending on the Ruby VM stack for computing a topological order, we were limited to CFGs whose depth was on the order of 2000 nodes. This could be easily avoided by modifying the code to use an external stack for the traversal, but since we expect that real programs are rarely nested this deep, we chose not to do this.

The first experiment we ran with the nested if generator

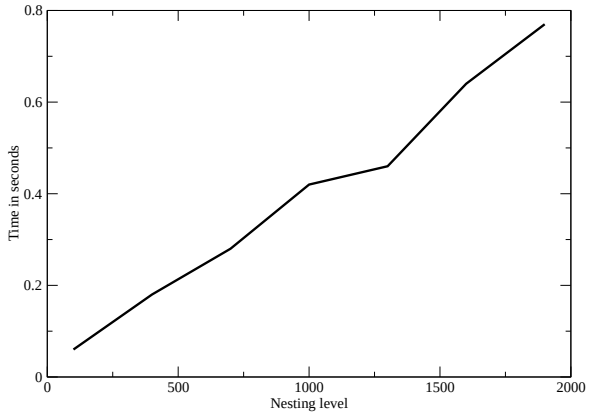

Figure 7: Relationship between nesting depth and runtime in nested if example.

was intended to show the cost of scaling the size of the CFG. In this experiment, we varied the number of nodes and edges together while keeping the nesting depth fixed at 1. Figure 6 shows the result. We find that runtime varies linearly with the number of nodes and edges.

The second experiment we ran with the nested if generator was intended to show that varying the structure of the graph given a constant node and edge count has a significant impact runtime. In this experiment, we held the node count fixed at 4000 and varied the depth of the nested if blocks. Figure 7 shows the result. We find that runtime varies linearly with the depth of nesting. This shows that for the dominance algorithm we implemented, it is insufficient to describe its performance only in terms of nodes and edges—the structure of the CFG has a significant effect on its performance.

4 Conclusion

We implemented a dominance algorithm for the Start intermediate language. We found that the algorithm performs well for our example programs, and that the structure of the CFG has a significant effect on the performance of dominance detection.

References

[1] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm.