SOFTWARE—PRACTICE AND EXPERIENCE

Softw. Pract. Exper. 2001; **4**:1–10   *Prepared using* speauth.cls *[Version: 2000/03/06 v2.1]*

A Simple, Fast Dominance Algorithm

Keith D. Cooper,
Timothy J. Harvey* and
Ken Kennedy

Rice University, Houston, TX

SUMMARY

The problem of finding the dominators in a control-flow graph has a long history in the literature. The original algorithms suffered from a large asymptotic complexity but were easy to understand. Subsequent work improved the time bound, but generally sacrificed both simplicity and ease of implementation. This paper returns to a simple formulation of dominance as a global data-flow problem. Some insights into the nature of dominance lead to an implementation of an $O(N^2)$ algorithm that runs faster, in practice, than the classic Lengauer-Tarjan algorithm, which has a timebound of $O(E * log(N))$. We compare the algorithm to Lengauer-Tarjan because it is the best known and most widely used of the fast algorithms for dominance. Working from the same implementation insights, we also rederive (from earlier work on control dependence by Ferrante, *et al.*) a method

Contract/grant sponsor: This research was supported, in part, by DARPA through USAFRL contract F30602-97-2-298, and the State of Texas through its Advanced Technology Program, grant number 3604-0122-1999.
*Correspondence to:
Tim Harvey
6100 South Main St., MS 132
Houston, TX, USA 77005
harv@cs.rice.edu

Copyright © 2001 John Wiley & Sons, Ltd.

for calculating dominance frontiers that we show is faster than the original algorithm by Cytron, *et al.* The aim of this paper is *not* to present a new algorithm, but, rather, to make an argument based on empirical evidence that algorithms with discouraging asymptotic complexities can be faster in practice than those more commonly employed. We show that, in some cases, careful engineering of simple algorithms can overcome theoretical advantages, even when problems grow beyond realistic sizes. Further, we argue that the algorithms presented herein are intuitive and easily implemented, making them excellent teaching tools.

KEY WORDS: Dominators, Dominance Frontiers

Introduction

The advent of static single assignment form (SSA) has rekindled interest in dominance and related concepts [13]. New algorithms for several problems in optimization and code generation have built on dominance [8, 12, 25, 27]. In this paper, we re-examine the formulation of dominance as a forward data-flow problem [4, 5, 19]. We present several insights that lead to a simple, general, and efficient implementation in an iterative data-flow framework. The resulting algorithm, an iterative solver that uses our representation for dominance information, is significantly faster than the Lengauer-Tarjan algorithm on graphs of a size normally encountered by a compiler—less than one thousand nodes. As an integral part of the process, our iterative solver computes immediate dominators for each node in the graph, eliminating one problem with previous iterative formulations. We also show that a natural extension of

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

these ideas leads to an efficient algorithm for computing the dominance frontiers used in the SSA-construction algorithm.

Allen, in her work on control-flow analysis, formulated the dominance computation as a global data-flow problem [4]. In a 1972 paper with Cocke, she showed an iterative algorithm to solve these equations [5]. Hecht and Ullman the showed that a reverse postorder iterative scheme solves these equations in a single pass over the CFG for reducible graphs [19]. The simple, intuitive nature of the iterative formulation makes it attractive for teaching and for implementing. Its simplicity leads to a high degree of confidence in the implementation's correctness. The prime result of this paper is that, with the right data structure, the iterative data-flow framework for dominance is faster than the well-known Lengauer-Tarjan algorithm on graphs that arise in real programs.

In practice, both of these algorithms are fast. In our experiments, they process from $50,000$ to $200,000$ control-flow graph (CFG) nodes per second. While Lengauer-Tarjan has faster asymptotic complexity, it requires unreasonably large CFGs—on the order of $30,000$ nodes—before this asymptotic advantage catches up with a well-engineered iterative scheme. Since the iterative algorithm is simpler, easier to understand, easier to implement, and faster in practice, it should be the technique of choice for computing dominators on CFGs.

The dominance problem is an excellent example of the need to balance theory with practice. Ever since Lowry and Medlock's $O(N^4)$ algorithm appeared in 1969 [23], researchers have steadily improved the time bound for this problem [7, 10, 17, 19, 22, 26, 29]. However, our results suggest that these improvements in asymptotic complexity may not help on realistically-

Copyright © 2001 John Wiley & Sons, Ltd.

sized examples, and that careful engineering makes the iterative scheme the clear method of choice.

History

Prosser introduced the notion of dominance in a 1959 paper on the analysis of flow diagrams, defining it as follows:

> *We say box* i *dominates box* j *if every path (leading from input to output through the diagram) which passes through box* j *must also pass through box* i. *Thus box* i *dominates box* j *if box* j *is subordinate to box* i *in the program [26].*

He used dominance to prove the safety of code reordering operations, but he did not explain the algorithm to compute dominance from his connectivity matrix.

Ten years later, Lowry and Medlock sketched an algorithm to compute dominators [23]. In essence, their algorithm considers all of the paths from the entry node to each node, b, and successively removes nodes from a path. If the removal of some node causes b to be unreachable, that node is in b's dominator set. Clearly, this algorithm is at least quadratic in the number of nodes, although the actual complexity would depend heavily on the implementation[†].

The data-flow approach to computing dominance begins with Allen's 1970 paper, where she proposed a set of data-flow equations for the problem [4]. Two years later, Allen and

[†]Lowry and Medlock do not give enough details to assess accurately the complexity of their algorithm, but Alstrup *et al.* claim that it has an asymptotic complexity of N^4, where N is the number of nodes in the graph [7].

Copyright © 2001 John Wiley & Sons, Ltd.

Cocke showed an iterative algorithm for solving these equations and gave its complexity as $O(N^2)$ [5]. In 1975, Hecht and Ullman published an analysis of iterative algorithms using reverse postorder traversals. They showed that the dominance equations can be solved in linear time on reducible graphs [19]. They restrict their algorithm to reducible graphs so that they can achieve the desired time bound, even though iterating to a fixed point would generalize the algorithm (but not the time bound) to handle irreducible graphs. Both Hecht's book and Aho and Ullman's "dragon" book present the iterative algorithm for dominance without restricting it to reducible graphs [18, 3]. Unfortunately, Aho and Ullman mistakenly credit the algorithm to Purdom and Moore [24], rather than to Allen and Cocke.

Aho and Ullman approached the problem from another direction [2]. Their algorithm takes the graph and successively removes nodes. Any nodes in the entire graph (rather than a single path) that cannot subsequently be reached are dominated by the removed node. This algorithm works in quadratic time, in the number of nodes. With Hopcroft, they improved this time bound to $O(E \log E)$ for reducible graphs, where E is the number of edges in the graph, by using an efficient method of finding ancestors in trees [1].

Purdom and Moore, in the same year, proposed a similar algorithm, which first builds a tree from the graph. Dominators are then found by successively removing each node from the graph and noting which children of that tree node can no longer be reached in the graph. This algorithm again requires quadratic time to complete [24].

In 1974, Tarjan proposed an algorithm that uses depth-first search and union-find to achieve an asymptotic complexity of $O(N \log N + E)$ [29]. Five years later, Lengauer and Tarjan built on this work to produce an algorithm with almost linear complexity [22]. Both algorithms rely

Copyright © 2001 John Wiley & Sons, Ltd.

on the observation that a node's dominator must be above it in the depth-first spanning tree. This gives an initial guess at the dominator, which is corrected in a second pass over the nodes. The algorithm relies on the efficiency of union-find to determine its time bound.

In 1985, Harel published an algorithm built on Lengauer-Tarjan that computes immediate dominators in linear time. He improved the time bound by speeding up the union-find operations with a technique from Gabow and Tarjan in that same year [16]. Harel's explanation, however, was subsequently found to be incomplete. In 1999, Alstrup *et al.* published a simpler method based on Harel's initial work that achieves a theoretical linear-time complexity [7]. The authors posit that the actual complexity of the algorithm, using practical data structures, is $O(E+N \log\log\log N)$, where N is the number of nodes in the graph, and E is the number of edges. The paper does not provide any experimental data that shows the algorithm's measured behavior versus Lengauer-Tarjan.

In 1998, Buchsbaum *et al.* presented a linear-time algorithm based on Lengauer-Tarjan [10, 11]. Their algorithm is essentially a divide-and-conquer algorithm that groups the bottom nodes of the depth-first search tree into *microtrees*. By solving the local problem for the microtrees, they can perform the subsequent union-find operations needed for Lengauer-Tarjan in linear time. Thus, their algorithm has better asymptotic complexity than Lengauer-Tarjan. However, they state that their algorithm runs ten to twenty percent slower than Lengauer-Tarjan on "real flowgraphs" [11]. As an interesting aside, their analysis suggests that, for many classes of graphs, Lengauer-Tarjan also runs in linear time.

This paper compares an iterative scheme for finding dominators against Lengauer-Tarjan. This comparison is appropriate for several reasons. First, Lengauer-Tarjan is the best known

Copyright © 2001 John Wiley & Sons, Ltd.

and most widely implemented of the fast dominator algorithms. Comparing our work with Lengauer-Tarjan provides meaningful information to the many people who have implemented that algorithm. Second, the primary result of this paper is that a simple iterative scheme for this problem can be engineered to outrun the more complicated techniques built on union-find—despite the higher asymptotic complexity of the iterative data-flow approach. From this perspective, Lengauer-Tarjan is an appropriate comparison; the other algorithms expend additional effort to speed up the union-find operations when, in fact, the asymptotic advantage from union-find does not show up until the problem size becomes unrealistically large. Third, while Alstrup *et al.* and Buchsbaum *et al.* both have lower asymptotic complexity than Lengauer-Tarjan, the papers provide no evidence that this theoretical advantage translates into faster running times. Indeed, Buchsbaum *et al.* show that their algorithm tends to run slower than Lengauer-Tarjan [11]. Finally, the analysis presented by Buchsbaum *et al.* suggests that, for many graphs, Lengauer-Tarjan also has linear-time behavior.

The Data-flow Approach

To compute dominance information with data-flow techniques, we can pose the problem as a set of data-flow equations and solve them with a reverse-postorder iterative algorithm. This approach builds on the well-understood principles of iterative data-flow analysis to guarantee termination and correctness and to provide insight into the algorithm's asymptotic complexity.

This section presents the iterative data-flow solver for dominance. It discusses the properties of the algorithm that we can derive from the theory of iterative data-flow analysis. Finally, it shows how simple insights into the nature of dominance information and a carefully engineered

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

data structure lead to a more efficient implementation—one that competes favorably with Lengauer-Tarjan.

To compute dominance information, the compiler can annotate each node in the CFG with a DOM set.

DOM(b): A node n in the CFG dominates b if n lies on every path from the entry node of the CFG to b. DOM(b) contains every node n that dominates b. For $x, y \in$ DOM(b), either $x \in$ DOM(y) or $y \in$ DOM(x). By definition, for any node b, $b \in$ DOM(b).

While DOM(b) contains every node that dominates b, it is often useful to know b's *immediate dominator*. Intuitively, b's immediate dominator is the node $n \in$ DOM(b) which is closest to b. Typically, the compiler captures this information in a set IDOM(b).[‡]

IDOM(b): For a node b, the set IDOM(b) contains exactly one node, the *immediate dominator* of b. If n is b's immediate dominator, then every node in $\{$DOM(b)$ - b\}$ is also in DOM(n).

This formulation lets the compiler compute DOM sets as a forward data-flow problem [5, 19]. 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:

$$\text{DOM}(n_0) = \{n_0\}$$

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

[‡]Since IDOM(b) always has exactly one member, we could describe it as a function on DOM(b). For consistency in the paper, we write it as a set; the implementation realizes it as a trivial function of DOM(b).

Copyright © 2001 John Wiley & Sons, Ltd.

```
for all nodes, n
    DOM[n] ← {1 ... N}
Changed ← true
while (Changed)
    Changed ← false
    for all nodes, n, in reverse postorder
        new_set ← (⋂_{p∈preds(n)} DOM[p]) ∪ {n}
        if (new_set ≠ DOM[n])
            DOM[n] ← new_set
            Changed ← true
```

Figure 1. The Iterative Dominator Algorithm

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.

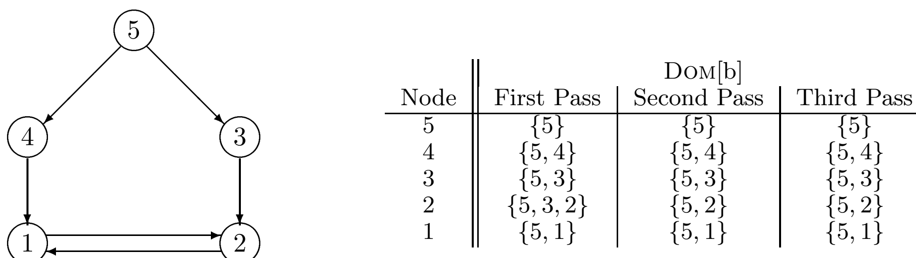
To solve these equations, the compiler can use an iterative algorithm, as shown in Figure 1. Correct initialization is important; the algorithm must either initialize each DOM set to include all the nodes, or it must exclude uninitialized sets from the intersection. Both Allen and Cocke [5] and Hecht [18, pp. 179–180] show similar formulations.

Properties of the Iterative Framework

This algorithm produces correct results because the equations for DOM, as shown above, form a distributive data-flow framework as defined by Kam and Ullman [20]. Thus, we know that the iterative algorithm will discover the maximal fixed-point solution and halt. Since the framework is distributive, we know that the maximal fixed-point solution is identical to the meet-over-all paths solution—which matches the definition of DOM.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls



Node	DOM[b] First Pass	Second Pass	Third Pass
5	{5}	{5}	{5}
4	{5, 4}	{5, 4}	{5, 4}
3	{5, 3}	{5, 3}	{5, 3}
2	{5, 3, 2}	{5, 2}	{5, 2}
1	{5, 1}	{5, 1}	{5, 1}

Figure 2. Computing Dominators on an Irreducible Graph

The equations for DOM are simple enough that they form a *rapid* framework [20]. Thus, an iterative algorithm that traverses the graph in reverse postorder will halt in no more than $\mathsf{d}(G)+3$ passes, where $\mathsf{d}(G)$ is the loop connectedness of the graph. For a reducible graph, $\mathsf{d}(G)$ is independent of the depth-first spanning tree used to generate the postorder numbering.[§] With an irreducible graph, $\mathsf{d}(G)$ depends on the specific depth-first spanning tree used to compute reverse postorder. Thus, on an irreducible graph, the running time of the algorithm depends on the traversal order.

An Example

Figure 2 shows a small irreducible graph, along with the results of the iterative dominators computation. Each node is labelled with its postorder number, and we will refer to each node by that number. The right side shows the sets computed on each iteration over the graph. After the first iteration, node 2 has the wrong dominator set because the algorithm has not processed node 1. Since node 1 is one of node 2's predecessors, the intersection that computes

[§]Two studies suggest that, even before the advent of "structured programming," most procedures had reducible CFGs [5] and that $\mathsf{d}(G)$ is typically ≤ 3 for CFGs [21].

Copyright © 2001 John Wiley & Sons, Ltd.

DOM[2] uses the initial value for DOM[1], and overestimates DOM[2] as DOM[3] $\cup \{2\}$. On the second iteration, both of node 2's predecessors have been processed, and their intersection produces the correct dominator set for node 2. The third iteration changes no sets, and the algorithm halts.

Engineering the Data Structures

This iterative algorithm is both easy to understand and easy to implement. However, it is impractically slow. A version that uses distinct bit-vector DOM sets at each node is up to 900 times slower than our implementation of the Lengauer-Tarjan algorithm. With large graphs, considerable time is wasted performing intersections on sparsely populated sets and copying these sets from one node to another. When we tried to reduce the time necessary to do the intersections by substituting *SparseSet*s for the bit vectors [9], the increased memory requirements again made the algorithm impractical.

To improve the iterative algorithm's performance, we need a memory-efficient data structure that supports a fast intersection. Keeping the DOM sets in a consistent order is one way to speed up intersection. If we think of the set as a list, and require that the union always add to the end of the list, then the DOM sets will reflect the order in which nodes are added. With this order, if DOM$(a) \cap$ DOM$(b) \neq \emptyset$, then the resulting set is a prefix of both DOM(a) and DOM(b).

This observation lets us implement intersection as a forward walk through the ordered sets, performing a pairwise comparison of elements. If the elements agree, that node is copied into the result set, and the comparison moves to the next element. When the elements disagree or

Copyright © 2001 John Wiley & Sons, Ltd.

the end of the sets is reached, the intersection terminates, and the current node is added as the last element of its own dominator set.

To improve memory efficiency, we rely on a subtle property of these ordered DOM sets. Notice that, for all nodes except n_0

$$\text{DOM}(b) \;=\; \{b\} \;\cup\; \text{IDOM}(b) \;\cup\; \text{IDOM}(\text{IDOM}(b)) \cdots \{n_0\}$$

This suggests a relationship between the ordered DOM set and an auxiliary data structure called the *dominator tree*. It contains the nodes of the CFG, with edges that reflect dominance. In the dominator tree, each node is a child of its immediate dominator in the CFG.

DOM(b) contains exactly those nodes on a path through the dominator tree from the entry node n_0 to b. Our ordered intersection operator creates DOM(b) in precisely the order of that path. The first element of DOM(b) is n_0. The last element of DOM(b) is b. The penultimate element of DOM(b) is b's immediate dominator—the node in DOM(b) that is closest to b. Thus, with these ordered DOM sets, we can read IDOM directly from the sets.

This relationship between the DOM sets and the dominator tree suggests an alternative data structure. Rather than keeping distinct DOM sets, the algorithm can represent the dominator tree and read the DOM sets from the tree. The algorithm keeps a single array, *doms*, for the whole CFG, indexed by node. For a node b, we represent b's inclusion in DOM(b) implicitly. The entry *doms*(b) holds IDOM(b). The entry *doms*(*doms*(b)) holds the next entry, which is IDOM(IDOM(b)). By walking the *doms* array, starting at b, we can reconstruct both the path through the dominator tree from b to n_0 and b's DOM set.

To use this representation, the algorithm must perform intersection by starting at the back of the DOM set and moving toward the front—the opposite direction from our earlier description.

Copyright © 2001 John Wiley & Sons, Ltd.
Prepared using speauth.cls

```
for all nodes, b /* initialize the dominators array */
    doms[b] ← Undefined
doms[start_node] ← start_node
Changed ← true
while (Changed)
    Changed ← false
    for all nodes, b, in reverse postorder (except start_node)
        new_idom ← first (processed) predecessor of b /* (pick one) */
        for all other predecessors, p, of b
            if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
                new_idom ← intersect(p, new_idom)
        if doms[b] ≠ new_idom
            doms[b] ← new_idom
            Changed ← true

function intersect(b1, b2) returns node
    finger1 ← b1
    finger2 ← b2
    while (finger1 ≠ finger2)
        while (finger1 < finger2)
            finger1 = doms[finger1]
        while (finger2 < finger1)
            finger2 = doms[finger2]
    return finger1
```

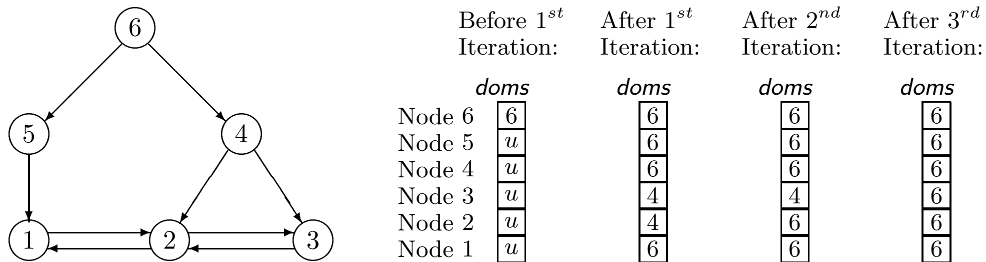
Figure 3. The Engineered Algorithm

This reverses the sense and termination condition of the intersections: under this scheme, we move backwards through the lists, comparing elements until they are the *same*.

Figure 3 shows the code for the iterative algorithm with these improvements. The intersection routine appears at the bottom of the figure. It implements a “two-finger” algorithm – one can imagine a finger pointing to each dominator set, each finger moving independently as the comparisons dictate. In this case, the comparisons are on postorder numbers; for each intersection, we start the two fingers at the ends of the two sets, and, until the fingers point to the same postorder number, we move the finger pointing to the smaller number back one element. Remember that nodes higher in the dominator tree have *higher* postorder numbers,

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls



	Before 1st Iteration: *doms*	After 1st Iteration: *doms*	After 2nd Iteration: *doms*	After 3rd Iteration: *doms*
Node 6	6	6	6	6
Node 5	u	6	6	6
Node 4	u	6	6	6
Node 3	u	4	4	6
Node 2	u	4	6	6
Node 1	u	6	6	6

Figure 4. An Example with the Engineered Algorithm

which is why *intersect* moves the finger whose value is less than the other finger's. When the two fingers point at the same element, *intersect* returns that element. The set resulting from the intersection begins with the returned element and chains its way up the *doms* array to the entry node.

This scheme has several advantages. It saves space by sharing representations—IDOM(b) occurs once, rather than once in each DOM set that contains it. It saves time by avoiding the cost of allocating and initializing separate DOM sets for each node. It avoids data movement: with separate sets, each element in the result set of an intersection was copied into that set; with the *doms* array, none of those elements are copied. Finally, it explicitly represents IDOM.

An Example

Figure 4 shows a somewhat more complex example. This graph requires four iterations instead of three, since its loop connectedness, $\mathsf{d}(G)$, is one greater than the example of Figure 2. The right side of the figure shows the contents of the *doms* array at each stage of the algorithm. For brevity, we have omitted the fourth iteration, where *doms* does not change.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

The *doms* array is indexed by node name, and all entries are initialized with a recognizable value to indicate that they are not yet computed. At each node, b, we intersect the dominator sets of b's predecessors. At node 2, in the first iteration, we call the *intersect* routine with 3 and 4. An intuitive way to view the *intersect* routine is to imagine that it walks up the dominator tree from two different nodes until a common parent is reached. Thus, *intersect* sets $finger1$ to 3 and $finger2$ to 4. Since $finger1$ is less than $finger2$ (remember that we are using postorder numbers as node names), we set $finger1$ equal to *doms*[$finger1$], which is 4. (This moved $finger1$ up the dominator tree to its parent.) The two fingers now point to the same node, so *intersect* returns that node, which is 4. Next, we look at 2's last predecessor, node 1, but, since its dominator has not yet been calculated, we skip it and record *new_idom* (in this case, 4) in *doms*[2].

Of course, node 2's dominator is not node 4, but this is only the result of the first iteration. The second iteration is more interesting. The first intersection at node 2 compares 3 and 4, and produces 4. The second intersection compares 1 and 4, and produces 6, which becomes the final value for 2. It takes one more iteration for 6 to filter over to node 3. The algorithm then makes one final pass (not shown) to discover that none of the information changes.

Complexity Analysis

Traversing the graph to compute the reverse postorder sequence takes $\mathbf{O}(N)$ time. The resulting sequence has N elements. The traversal that computes DOM and IDOM visits each node. At each node, it performs a set of pairwise intersections over the incoming edges. (The

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

unions have been made implicit in the data structure.) Taken over the entire traversal, this is $\mathbf{O}(E)$ intersections that require time proportional to the size of the DOM sets they consume. Thus, the total cost per iteration is $\mathbf{O}(N + E \cdot D)$ where D is the size of the largest DOM set. The number of iterations depends on the shape of the graph. Kam and Ullman showed that the iterative algorithm will halt in no more than $\mathsf{d}(G)+3$ iterations. Of course, this is an upper bound, and the loop connectedness of an irreducible graph is a function of the order in which the depth-first search computes the reverse postorder numbering.

Dominance Frontiers

The other part of dominance that plays an important part in the SSA construction is the calculation of dominance frontiers for each node in the CFG. Cytron *et al.* define the dominance frontier of a node, b, as:

> *...the set of all* CFG *nodes, y, such that b dominates a predecessor of y but does not strictly dominate y [13].*

Dominance frontiers have applications to algorithms other than SSA, as well. For example, finding postdominance frontiers is an efficient method of computing control dependence, a critical analysis for automatic parallelization [6].

Cytron *et al.* propose finding the dominance frontier set for each node in a two step manner. They begin by walking over the dominator tree in a bottom-up traversal. At each node, b, they add to b's dominance-frontier set any CFG successors not dominated by b. They then traverse

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

the dominance-frontier sets of b's dominator-tree children – each member of these frontiers that is not dominated by b is copied into b's dominance frontier.

We approach the problem from the opposite direction, based on three observations. First, nodes in a dominance frontier represent join points in the graph, nodes into which control flows from multiple predecessors. Second, the predecessors of any join point, j, must have j in their respective dominance-frontier sets, unless the predecessor dominates j. This is a direct result of the definition of dominance frontiers, above. Finally, the dominators of j's predecessors must themselves have j in their dominance-frontier sets unless they also dominate j.

These observations lead to a simple algorithm.[¶] First, we identify each join point, j – any node with more than one incoming edge is a join point. We then examine each predecessor, p, of j and walk up the dominator tree starting at p. We stop the walk when we reach j's immediate dominator – j is in the dominance frontier of each of the nodes in the walk, except for j's immediate dominator. Intuitively, all of the rest of j's dominators are shared by j's predecessors as well. Since they dominate j, they will not have j in their dominance frontiers. The pseudo code is given in Figure 5.

There is a small amount of bookkeeping not shown; specifically, any j should be added to a node's dominance frontier only once, but the data structure used for the dominance frontier sets will dictate the amount of additional work necessary. In our implementation, we use linked lists for dominance-frontier sets, so we keep a *SparseSet* [9] to restrict multiple entries – when a

[¶]This algorithm first appeared in a paper by Ferrante *et al.* in the context of control dependence [14]. We believe that its value as a method of computing dominance frontiers has not been studied prior to this work.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
for all nodes, b
    if the number of predecessors of b ≥ 2
        for all predecessors, p, of b
            runner ← p
            while runner ≠ doms[b]
                add b to runner's dominance frontier set
                runner = doms[runner]
```

Figure 5. The Dominance-Frontier Algorithm

join point is entered into a node's dominance-frontier set, we put that node into the *SparseSet*, and, before a join point is entered into a node's dominance-frontier set, we check to see if that node is already in the *SparseSet*.

Complexity Analysis

Traversing the CFG requires $\mathbf{O}(N)$ time. If each node in the graph were a join point, we would have to do at least $N \times 2$ walks up the dominator tree giving us a quadratic timebound. But remember that the walks always stop as early as possible. That is, we only touch a node, n, if the join point belongs in the dominance frontier of n. Thus, the number of nodes touched is equal to the sum of the sizes of all of the dominance-frontier sets. This sum can be quadratic in the number of nodes, but we contend that the sets cannot be built any more efficiently. In other words, we do no more work than is required.

As we shall see, this approach tends to run faster than Cytron *et al.*'s algorithm in practice, almost certainly for two reasons. First, the iterative algorithm has already built the dominator tree. Second, the algorithm uses no more comparisons than are strictly necessary.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Experiments

The iterative formulation of dominators, with the improvements that we have described, is both simple and practical. To show this, we implemented both our algorithm and the Lengauer-Tarjan algorithm (with path compression but not balancing) in C in our research compiler and ran a series of experiments to compare their behavior. With our algorithm, we built the dominance-frontier calculation as described in the previous section. Alongside our Lengauer-Tarjan implementation, we built the dominance-frontier calculation as described by Cytron *et al.* [13]. The timing results for our algorithm include the cost of computing the reverse postorder numbers, although in neither case do we include the cost of building the dominator tree – it is a natural byproduct of our algorithm, and so we felt it an unfair test to include the time to build the tree in the Lengauer-Tarjan/Cytron *et al.* implementation.

One of the first problems that we encountered was the size of our test codes. They were too small to provide reasonable timing measurements. All the experiments were run on a 300 MHz Sun Ultra10 under the Solaris operating system. On this machine, the `clock()` function has a granularity of only one hundredth of a second.[‖] Our standard test suite contains 169 Fortran routines taken from the SPEC benchmarks and from Forsythe, Malcolm, and Moler's book on numerical methods [15]. The largest CFG in the suite is from `field`, with 744 basic blocks.[**] On this file, the timer only measures one hundredth of a second of CPU time to compute

[‖] The manual page for `clock()` says that the time returned is in microseconds; however, we found that in practice, the amount of CPU time is reported only down to the hundredth of a second.

[**] This number includes empty basic blocks put in to split critical edges, a transformation often done to enable or simplify optimizations.

Copyright © 2001 John Wiley & Sons, Ltd.

dominators using the iterative algorithm. The vast majority of the codes in our test suite registered zero time to compute dominators.

This result is significant in its own right. On real programs, both of the algorithms ran so fast that their speed is not a critical component of compilation time. Again, this suggests that the compiler writer should choose the algorithm that is easiest to understand and to implement.

We performed two sets of experiments to better understand the tradeoff of complexity with runtime. In the first experiment, we modified the dominator calculation to run multiple times for each CFG, carefully making sure that each iteration did the same amount of work as a single, isolated run. In the second experiment, we created artificial graphs of a size which would register reliably on our timer.

Our Test Suite

For our first experiment, we modified each of the two dominator algorithms to iterate over the same graph multiple times and recorded the total time. To ensure that the smaller graphs – those of fewer than twenty-five nodes – register on our timer, we build the dominator information 10,000 times on each example. To allow for comparison, all graphs were run that many times.

We ran the two algorithms on an unloaded 300 MHz Sun Ultra 10 with 256 megabytes of RAM. To adjust for interference, we ran each graph through both algorithms ten times and recorded the lowest run time.[†] For each graph, we measured the time to compute

[†]Choosing the lowest time, instead of, as is often done, the average time, makes sense when comparing deterministic algorithms. Measurements on real machines, necessary to show that the theory is working in

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Number of Nodes	*Iterative Algorithm* Dominance Dom	DF	Postdominance Dom	DF	*Lengauer-Tarjan/Cytron et al.* Dominance Dom	DF	Postdominance Dom	DF
> 400	3148	1446	2753	1416	7332	2241	6845	1921
201–400	1551	716	1486	674	3315	1043	3108	883
101–200	711	309	600	295	1486	446	1392	388
51–100	289	160	297	151	744	219	700	191
26–50	156	86	165	94	418	119	412	99
<= 25	49	26	52	25	140	32	134	26

Average times by graph size, measured in $\frac{1}{100}$*'s of a second*

Table I. Runtimes for 10,000 Runs of Our Fortran Test Suite, aggregated by Graph Size

dominators and the time to compute dominance frontiers. We also measured the time to compute postdominators and postdominance frontiers.[‡] The postdominance computation has a different behavior, because the reversed CFG has a different shape. This gives us additional insight into the behavior of the algorithms.

In CFGs generated from real-world codes, it is more common to encounter irreducible graphs when calculating postdominance information. Broadly speaking, irreducible loops are caused by multiple-entry loops. While few modern languages allow a jump into the middle of a loop, almost all languages allow a jump out of the middle of the loop, and real-world codes tend to do this with some frequency. When we walk backwards through the CFG, jumps *out* of a loop become jumps *into* a loop, and irreducibility results.

practice, will have variance due to external factors, such as context switching. These factors can only *increase* the running time. Thus, the most accurate measure of a deterministic algorithm's running time is the one that includes the lowest amount of irrelevant work.

[‡]Postdominators are dominators on the reversed CFG.

Copyright © 2001 John Wiley & Sons, Ltd.

The timing results are shown in Table I. For the dominance calculation, the iterative algorithm runs about 2.5 times faster than Lengauer-Tarjan, on average. The improvement slowly decreases as the number of blocks increases. This is what we would expect: the Lengauer and Tarjan algorithm has a greater startup cost, which gets amortized in larger graphs. Of course, the Lengauer-Tarjan results should catch up quickly based on the relative asymptotic complexity. That this is not the case argues strongly that real-world codes have low connectivity of irreducible loops and their shapes allow for comparatively fast intersections.

For computing dominance frontiers, the times begin to diverge as the number of blocks increases. It appears that the advantage of our algorithm over Cytron *et al.*'s algorithm increases as the CFG gets larger, ultimately resulting in an approximately 30% speedup for the largest graphs. We believe this is because, in general, larger graphs have more complicated control flow. The amount of work done by the Cytron *et al.* algorithm grows as a function of the size of the dominance-frontier sets, whereas our formulation grows with the size of the CFG.

Larger Graphs

We believe that the sizeable improvement in running time only tells part of the story. While the advantage in asymptotic complexity of Lengauer-Tarjan should give it better running times over the iterative algorithm, we need to ask the question of when the asymptotic advantage takes over. To answer this, we built huge graphs which, as we will show, provide an insight into the value of the iterative algorithm.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Building Random Graphs

To obtain appropriate CFGs, we had to design a mechanism that generates large random CFGs.[§] Since we were primarily interested in understanding the behavior of the algorithms on programs, as opposed to arbitrary graphs, we measured the characteristics of the CFGs of the programs in our test suite and used these statistics to generate random graphs with similar properties.

Our test suite contains, in total, 11,644 blocks and 16,494 edges. Eleven percent of the edges are back edges. Sixty-one percent of the blocks have only one outgoing edge, and fifty-five percent of the blocks have only one incoming edge. Blocks with two incoming or outgoing edges were thirty-four percent and forty-three percent of the total, respectively. The remaining incoming and outgoing edges were grouped in sets of three or more per block.

Using these measurements, we built a program that performs a preorder walk over an imaginary graph. As the "walk" progresses, it instantiates nodes as it reaches them. It starts by creating a single, initial block, n_0. It randomly determines the number of edges coming out of that block, based on the statistics from our test suite, and instantiates those edges. Next, it walks those edges in a recursive depth-first search. Each time it traverses an edge, it creates a new block to serve as its sink, determines the number and kind of successor edges that the new block should have, and instantiates them. It continues in this fashion until it has generated

[§]Note that this analysis concerns only the *structure* of the CFG, so the basic blocks in our random graphs contain nothing except branching operations.

Copyright © 2001 John Wiley & Sons, Ltd.

the desired number of nodes. At that point, there will be a number of edges that do not yet have a sink. It connects those edges to blocks that already exist.

Creating back edges is problematic. The classic method used to identify a back edge has an implicit assumption that the entire graph has been defined [28]. Since the random graph builder does not have access to the entire graph, it cannot tell, for certain, whether an edge will be a back edge in the completed graph. Thus, it cannot add the back edges as it builds the rest of the graph. Instead, the graph builder builds a queue of all the edges that it intends to be back edges. It does not traverse these edges in the depth-first search. Instead, it waits until all of the nodes have been instantiated and then processes the prospective back edges on the queue, connecting them to appropriate nodes.

The resulting graphs matched the characteristics of the test suite codes almost exactly, with two variations, back edges and loop connectedness. Because the graph builder must add edges before the graph is fully defined, it cannot know the total number of successors that a node will have. This uncertainty shows up in two ways. First, the generated graphs have fewer back edges than the CFGs in the test suite—eight percent versus eleven percent. Second, the loop connectedness of the generated graphs is higher than that of the CFGs in the test suite—slightly over three versus 1.11 for the test suite.

These differences have an important implication for measuring the running time of our algorithm versus Lengauer-Tarjan. The complexity of our algorithm depends on the loop connectedness of the graph, since it uses Kam and Ullman's formulation of the iterative algorithm. The Lengauer-Tarjan algorithm always requires two passes, independent of the

Copyright © 2001 John Wiley & Sons, Ltd.

	Iterative Algorithm				*Lengauer-Tarjan/Cytron et al.*			
	Dominance		Postdominance		Dominance		Postdominance	
	Dom	DF	Dom	DF	Dom	DF	Dom	DF
Average (.01-secs)	39.72	18.91	38.68	12.63	41.80	25.21	39.75	18.85
Standard Deviation	2.30	1.86	2.12	0.58	0.92	2.88	0.78	0.51

Table II. Runtime Statistics for the 100-graph Test Suite

graph's structure. Thus, the higher value for $\mathsf{d}(G)$ should favor Lengauer-Tarjan, at our algorithm's expense.

Measurements on Large Random Graphs

Because graphs with 500 nodes barely registered on our system's timer, we needed graphs that were at least an order of magnitude larger. To fill this need, we used the graph builder to create a test suite of 100 graphs of $30,000$ nodes each.

Table II shows the results of our timing experiments for the test suite of 100 large, random graphs. It reports the numbers for dominance and postdominance, under the iterative scheme and under the Lengauer-Tarjan algorithm. The first row shows the average solution time, across all 100 graphs. The second row shows the standard deviation in that time, across all 100 graphs. Within each category, two numbers appear. The column labelled Dom reports the time for the dominator calculation, while the column labelled DF reports the time for the dominance-frontier calculation.

If we add the times to compute dominators and dominance frontiers, our algorithms run about 12% faster than Lengauer-Tarjan. The main advantage is from the faster dominance-frontier calculation, which is about 25% faster on the forward problem, and about 33% faster on the backward problem. We conjectured that the asymptotic differences between the dominance

Copyright © 2001 John Wiley & Sons, Ltd.

algorithms would shift the advantage from the iterative algorithm to the Lengauer-Tarjan, and this graph suggests that this point is near 30,000 nodes; however, computing the crossover is complicated by the variability of the shape of the graph. For example, on a CFG with 30,000 nodes in a straight line—each node has only one parent and only one child—the iterative scheme takes only four hundredths of a second, while the Lengauer-Tarjan algorithm takes its standard 0.4 seconds. The conclusion to be drawn from this experiment is nonetheless clear: the iterative algorithm is competitive with Lengauer-Tarjan even on unrealistically large graphs.

Summary and Conclusions

In this paper, we have presented a technique for computing dominators on a control-flow graph. The algorithm builds on the well-developed and well-understood theory of iterative data-flow analysis. By relying on some fundamental insights into the nature of the dominance information, we were able to engineer a representation for the dominator sets that saves on both space and time. The resulting algorithm is 2.5 times faster than the classic Lengauer-Tarjan algorithm on real programs. On control-flow graphs of 30,000 nodes—a factor of almost 40 times larger than the largest CFG in the SPEC benchmarks—our method and Lengauer-Tarjan take, essentially, the same time.

Our simple iterative technique is faster than the Lengauer-Tarjan algorithm. Its simplicity makes it easy to implement, easy to understand, and easy to teach. It eliminates many of the costs that occur in an implementation with separate sets—in particular, much of the allocation, initialization, and data movement. This simple, fast technique should be the method of choice for implementing dominators – not only is development time very short, but the compiler writer

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls

can have a high degree of confidence in the implementation's correctness. The algorithm for the dominance frontier calculation is not only simpler but also faster than the original algorithm by Cytron *et al.*

ACKNOWLEDGEMENTS

Several people provided insight, support, and suggestions that improved this work. Fran Allen helped us track down the origins of the iterative approach to dominance. Preston Briggs made insightful suggestions on both the exposition in general and the experimental results in particular. Steve Reeves and Linda Torczon participated in many of the discussions. The members of the Massively Scalar Compiler Project provided us with support and a system in which to conduct these experiments. To all these people go our heartfelt thanks.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. In *STOC: ACM Symposium on Theory of Computing (STOC)* , 1973. **check this**
2. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, 1972.
3. A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.
4. F. E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, July 1970. *Proceedings of a Symposium on Compiler Optimization.*
5. F. E. Allen and J. Cocke. Graph-theoretic constructs for program flow analysis. Technical Report RC 3923 (17789), IBM Thomas J. Watson Research Center, July 1972.
6. R. Allen and K. Kennedy. *Advanced Compilation for Vector and Parallel Computers.* Morgan-Kaufmann, 2001.
7. S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM J. Comput.*, 28(6):2117–2132, June 1999.
8. P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software–Practice and Experience*, 00(00), June 1997. Also available as a Technical Report From Center for Research on Parallel Computation, Rice University, number 95517-S.
9. P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1–4):59–69, March–December 1993.
10. A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 279–288, 1998.
11. A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–1296, Nov. 1998.
12. M. Carroll and B. G. Ryder. An incremental algorithm for software analysis. In *Proceedings of the SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Software Development Environments,* SIGPLAN Notices 22(1), pages 171–179, Jan. 1987.

Copyright © 2001 John Wiley & Sons, Ltd.
Prepared using speauth.cls

13. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.
14. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
15. G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1977.
16. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30:209–221, 1985.
17. D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 185–194, May 1985.
18. M. S. Hecht. *Flow Analysis of Computer Programs*. Programming Languages Series. Elsevier North-Holland, Inc., 52 Vanderbilt Avenue, New York, NY 10017, 1977.
19. M. S. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM J. Comput.*, 4(4):519–532, Dec. 1975.
20. J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, Jan. 1976.
21. D. E. Knuth. An empirical study of Fortran programs. *Software – Practice and Experience*, 1:105–133, 1971.
22. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):115–120, July 1979.
23. E. Lowry and C. Medlock. Object code optimization. *Communications of the ACM*, pages 13–22, Jan. 1969.
24. J. Paul W. Purdom and E. F. Moore. Immediate predominators in a directed graph. *Communications of the ACM*, 15(8):777–778, Aug. 1972.
25. K. Pingali and G. Bilardi. Optimal control dependence computation and the Roman chariots problem. *ACM Transactions on Programming Languages and Systems*, 19(3):462–491, May 1997.
26. R. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138. Spartan Books, NY, USA, Dec. 1959.
27. P. H. Sweany and S. J. Beaty. Dominator-path scheduling—a global scheduling method. *SIGMICRO Newsletter*, 23(12):260–263, Dec. 1992. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*.
28. R. E. Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.*, 3(1):62–89, Mar. 1974.
29. R. E. Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9:355–365, 1974.

Copyright © 2001 John Wiley & Sons, Ltd.

Prepared using speauth.cls