A Practical Interprocedural Dominance Algorithm

BJORN DE SUTTER, LUDO VAN PUT, and KOEN DE BOSSCHERE
Ghent University

Existing algorithms for computing dominators are formulated for control flow graphs of single procedures. With the rise of computing power, and the viability of whole-program analyses and optimizations, there is a growing need to extend the dominator computation algorithms to context-sensitive interprocedural dominators. Because the transitive reduction of the interprocedural dominator graph is not a tree, as in the intraprocedural case, it is not possible to extend existing algorithms directly. In this article, we propose a new algorithm for computing interprocedural dominators. Although the theoretical complexity of this new algorithm is as high as that of a straightforward iterative solution of the data flow equations, our experimental evaluation demonstrates that the algorithm is practically viable, even for programs consisting of several hundred thousands of basic blocks.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*; E.1 [**Data Structures**]: *Graphs and networks*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms, path and circuit problems*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Interprocedural control flow graph, dominators, interprocedural analysis

ACM Reference Format:

De Sutter, B., Van Put, L., and De Bosschere, K. 2007. A practical interprocedural dominance algorithm. ACM Trans. Program. Lang. Syst. 29, 4, Article 19 (August 2007), 44 pages. DOI = 10.1145/1255450.1255452 http://doi.acm.org/10.1145/1255450.1255452

1. INTRODUCTION

The dominator relation plays an important role in the theory and practice of compilers. It has led, among other things, to the identification of natural loops

While doing the research presented in this article, Bjorn De Sutter was supported by the Fund for Scientific Research-Belgium-Flanders (FWO) as a Postdoctoral Research Fellow. Ludo Van Put was supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research was partially supported by Ghent University, by the European Network of Excellence HiPEAC, and by the European Integrated Project SARC.
Authors' address: Electronics and Information Systems Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: {brdsutte,lvanput,kdb}@elis.ugent.be.

[Lowry and Medlock 1969] and to the efficient construction of static single assignment representations [Cytron et al. 1991]. The computation of dominators has been the subject of many papers. The fastest algorithm nowadays runs in linear time [Alstrup et al. 1999]. The most widely used implementation is based on the Lengauer and Tarjan [1979] algorithm, which runs in near linear time.

So far, the computation of dominators has always been formulated for single procedures, for which the computation is applied on their control flow graph (CFG). CFGs are directed graphs in which all paths are valid. As compiler optimizations are mostly performed intraprocedurally, there has for a long time been little need to extend the computation of dominators to whole programs or, in other words, to interprocedural dominators.

With ever-rising computing power whole-program analyses have become viable during the last decade however, and many useful applications have been found. These include, among others, whole-program optimization [Wall 1986; Muth et al. 2001; De Sutter et al. 2005; Triantafyllis et al. 2006] and automated software verification [Agrawal 1999]. For example, verifying the sequence in which procedures can be called in a program is a typical subject of software verification. To enable new whole-program analyses and transformations, or to improve or extend existing ones, there exists a growing need to extend the computation of intraprocedural dominators to its interprocedural counterpart. Unfortunately, all efficient intraprocedural dominator computation algorithms exploit the fact that the transitive reduction of the intraprocedural dominator graph is a tree [Allen and Cocke 1972]. Because this is not the case for interprocedural dominators, those algorithms cannot be extended directly.

This article presents a practical algorithm to compute interprocedural dominators for whole programs. This algorithm borrows ideas from the efficient dominator set representation proposed by Cooper et al. [2001] to compute intraprocedural dominators, but those ideas are adapted to fit the properties of the interprocedural dominator relationship. The remainder of the article is structured as follows. Section 2 discusses related work. Interprocedural dominators and their properties are discussed in Section 3. A simple, but inefficient dataflow algorithm to compute interprocedural dominators is presented in Section 4. Section 5 presents a preorder context-sensitive depth-first-traversal ordering of a program's basic blocks, on which our new algorithm will be based. A base version of that new algorithm is introduced in detail in Section 6. Important optimizations to the base version of the algorithm are presented in Section 7, after which the optimized algorithm is evaluated in Section 8. Finally, conclusions are drawn in Section 9.

2. RELATED WORK

The original formulation of the dominator relation dates back to the work by Prosser [1959]. This relation identifies, for each node in a directed graph, the nodes that must be traversed when starting from the root of the graph, to reach that node.

This relation has been extremely useful in the domain of program analysis and code optimization. With the dominator information, natural loops can

be identified [Lowry and Medlock 1969]. During code motion, the dominator information can indicate potential locations to which code can be moved such that the code will certainly be executed [Allen and Cocke 1972]. With the advent of static single assignment code representation forms [Cytron et al. 1991], the dominator relation has again received increased attention. During the past decades, the computation of the dominator relation has been a hot topic and, as can be seen from recent papers [Ramalingam 2002; Georgiadis and Tarjan 2004; Georgiadis et al. 2004], the problem still attracts the attention of the research community.

Lowry and Medlock [1969] are acknowledged for proposing the first algorithm for the calculation of the dominator relation. In their algorithm an arbitrary path to a node K is considered and from this path a node is removed repeatedly when it is discovered that another path reaches node K without going through the node on the initial path. The remaining nodes are K's dominators.

Later, Allen [1970] provided a data-flow solution. Her algorithm was developed in the context of graph intervals. This work was extended by Allen and Cocke [1972]. Aho and Ullman [1977] provide a complete description of the data-flow solution. Purdom and Moore [1972] published another algorithm, in which they repeatedly remove single nodes from the original graph and perform a reachability analysis on the thus created graphs. For each of these graphs, the node that was removed dominates those nodes that become unreachable because of its removal.

The best known work on the calculation of the dominator relation is the work by Lengauer and Tarjan [1979]. Their algorithm is also the most widely used in the compiler community, although there are asymptotically faster algorithms available that run in linear time [Harel 1985; Alstrup et al. 1999]. The Lengauer-Tarjan algorithm is better understood, however, and has a clear implementation.

Recently, Cooper et al. [2001] suggested that the data-flow solution for finding dominators does not perform worse than near-linear time algorithms if the underlying data structures are carefully engineered. In their paper, Cooper et al. propose to model all dominator relations of a program during the fixed-point computations with a tree in which each node's parent is its estimated immediate dominator. Thus, they avoid the need to store whole dominator sets for each node. Also, computing an intersection of the estimated dominator sets of two nodes does not require copying sets of nodes. Instead, the intersection computation is limited to finding common ancestors in the estimated dominator tree. Cooper et al. [2001] claim that their data-flow implementation outperforms the Lengauer-Tarjan algorithm for real-world control flow graphs that were generated from existing Fortran programs and contain up to 744 basic blocks. Furthermore, they found that both algorithms perform equally well for "unrealistically large graphs" that were artificially generated and contained up to 30,000 nodes.

Due to this renewed interest in the data-flow algorithm, Georgiadis et al. [2004] carried out detailed measurements to compare Cooper's algorithm with two versions of the Lengauer-Tarjan algorithm and one new algorithm. The authors concluded that for real-life procedures, the performance of the algorithms

is similar and that no algorithm is clearly superior over the other ones. For artificial graphs of up to several hundred thousands of basic blocks, the Lengauer-Tarjan and derived algorithms are superior.

By comparison, we will evaluate our interprocedural dominator computation algorithm on real-life programs of up to several hundred thousand blocks.

Several applications are being conceived that can exploit interprocedural dominator information. One example comes from the field of coverage testing. Agrawal [1999] proposes to use interprocedural dominator information to reduce the minimal set of instructions of a program that needs to be executed in order to guarantee that all instructions in the program will have been executed. The author suggests using interprocedural dominator information by modifying the data-flow algorithm. He does not describe these modifications, however, nor does he use the interprocedural dominator information because he believes this computation to be too expensive. His motivation comes from the high asymptotic complexity of the data-flow algorithm.

3. INTERPROCEDURAL DOMINATORS

This section first discusses the interprocedural control flow graph (ICFG) to represent a whole program, and the valid paths contained in it. We then define a context-sensitive, interprocedural dominator relation on the ICFG and explore some of the differences with the intraprocedural dominator relation. To avoid confusion, we will use the term traditional dominator relation to indicate the intraprocedural dominator relation.

3.1 The ICFG and Valid Paths

3.1.1 *Informal Description.* The ICFG of a program is a graph representing the potential control flow in the program. The nodes in the ICFG are the program's basic blocks, and the edges model potential control flow paths. In this article, we will refer to the nodes by using their number. A useful numbering scheme for the nodes will be explained later. Edges are referred to as $head \rightarrow tail$, in which $head$ and $tail$ are node numbers.

To model intraprocedural control flow, the ICFG contains the same edges as the ordinary control flow graphs of the procedures in a program. On top of these intraprocedural edges, two types of interprocedural edges model interprocedural control flow transfers. Figure 1 depicts their use. The *call edge* $10 \rightarrow 4$ models the procedure call from procedure S to procedure T by connecting the *call-site* to the *entry point* of procedure T. The *return edge* $6 \rightarrow 11$ models the corresponding return from T to S by connecting the *exit block* 6 to the *return block* 11, which corresponds with call-site 10.

In this article, we require that each procedure has a unique exit block, because this facilitates our reasoning, and because it allows for a more efficient implementation. Although real procedures may contain multiple exit points, it is trivial to add a virtual unique exit block to their graphs.

Furthermore, we require that each procedure has a unique entry point. This facilitates both the clear presentation and the efficient implementation of the

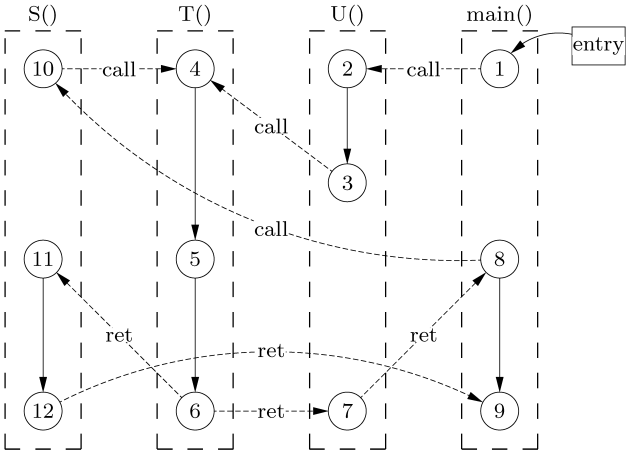
```
void S() {
  10: T();
  11: ...;
  12: return;
}
```

```
void T() {
   4: ...
   5: ...
   6: return;
}
```

```
void U() {
   2: ...
   3: T();
   7: return;
}
```

```
void main() {
   1: U();
   8: if (....) S();
   9: return;
}
```

(a) The relevant parts of an example program, in which all the basic blocks are labeled with their numbers.



(b) The corresponding ICFG.

Fig. 1. An example program, and the corresponding ICFG containing call and return edges. As in the other graphs in this article, interprocedural edges are dashed.

algorithms we propose. This requirement does not prohibit us from applying the proposed algorithms to programs that contain multiple-entry procedures, such as some Fortran programs, as we can easily split such multiple-entry procedures into multiple single-entry procedures by inserting the necessary virtual nodes and the appropriate edges. Virtual nodes and corresponding edges are necessary to handle a broad range of programs anyway, as many programs contain interprocedural gotos. This happens in manually-written assembler code that is linked into compiled programs from the standard system libraries, or in procedures on which compilers have applied tail-call optimizations. To model indirect procedure calls, for which the targets are not always known conservatively, as well as other anomalous control flow such as the standard C procedures `longjmp()` and `setjmp()`, additional virtual nodes and edges can be added to an ICFG as well. As all these virtual nodes and edges enable one to treat such anomalous control flow as normal, they are not relevant to the discussion in this article. We refer to Muth et al. [2001] for more details. Here, it suffices to note that adding virtual edges and nodes to the ICFG can never result in additional dominators being found. Instead, dominator sets can only become smaller. Therefore the addition of such edges and nodes is conservative in the context of dominator computations.

It is clear that when a procedure is entered through a specific call edge, it will be exited through the corresponding return edge.[1] In other words, some execution paths in the ICFG are invalid. For example, the execution path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 11$ in Figure 1 is invalid. In general, program analyses become more precise if they only consider valid paths. This is also the case when interprocedural dominators are computed. In Figure 1 block 11 is dominated by block 10 because block 11 cannot be executed without block 10 being executed first. Had a dominator computation considered the invalid path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 11$ as valid, block 10 would not have been considered a dominator of block 11, because the latter would have been considered reachable from the unique entry point of the program without going through block 10.

3.1.2 *Formal Description*

Definition 3.1. A directed graph $G = (V, E)$ is composed of a set of nodes $V \triangleq \{v_1, v_2, \ldots, v_n\}$ and a set of directed edges $E \triangleq \{e_1, e_2, \ldots, e_m\}$ that each connect node $head(e_i) \in V$ to node $tail(e_i) \in V$.

Definition 3.2. The incoming edges of node v in graph G, noted $in_G(v)$, are defined by $in_G(v) \triangleq \{e \mid e \in E : tail(e) = v\}$.

Definition 3.3. The outgoing edges of node v in graph G, noted $out_G(v)$, are defined by $out_G(v) \triangleq \{e \mid e \in E : head(e) = v\}$.

In the remainder of this article, we will abuse notation by applying operations whose domain is a set of single elements such as nodes or edges to sets of those elements as well. In that case, the result is the union of the operation applied on the elements of the operand set. For example, $tail(\{e_1, e_2, \ldots\}) \triangleq \bigcup_{e_i \in \{e_1, e_2, \ldots\}} \{tail(e_i)\}$.

Definition 3.4. The predecessor nodes $pred_G(v)$ of a node v are hence defined by $pred_G(v) \triangleq head(in_G(v))$.

Definition 3.5. The set of ancestor nodes $anc_G(v)$ of v, including v itself, is defined by

$$w \in anc_G(v) \Leftrightarrow \begin{cases} w = v, \\ w \in pred_G(v), \\ w \in anc_G(pred_G(v)). \end{cases} \tag{1}$$

Definition 3.6. A *path* in a graph $G = (V, E)$ is a sequence of edges $(e_0, \ldots, e_m)$ such that $\forall i.0 \leq i < m : tail(e_i) = head(e_{i+1})$.

Definition 3.7. Let r and q be different elements in V, $C \subseteq E$ be the set of all call edges, $R \subseteq E$ the set of all return edges, ϕ a bijection in $C \rightarrow R$,

[1]For procedures from which control never returns, such as the C-library procedure `exit`, the incoming call edges do not have corresponding outgoing edges. We will neglect this case for the sake of clarity. In practice, it is trivial to deal with. It suffices to add a test to line 13 of the numbering algorithm of Figure 7 to check whether a corresponding edge exists.

and λ a function in $V \to (2^V \setminus \emptyset)$. An *interprocedural control flow graph* $G = (V, E, C, R, r, q, \phi, \lambda)$ of a program is a directed graph in which nodes represent basic blocks of the program and edges represent possible control flow in the program. r is the unique program entry and q is the artificially inserted unique program exit. ϕ maps call edges to their corresponding return edges. λ maps each node v to the procedure $\lambda(v)$ in which the node is located. As such, a procedure represents a set of basic blocks.

We will call $\phi(e)$ the *corresponding edge* of edge e, and e the corresponding edge of $\phi(e)$. Obviously, an edge can only be interprocedural if the edge is in C or R. More formally, the ICFG has to satisfy the following property.

Property 3.8. $\forall e.e \in E \setminus (C \cup R) : \lambda(head(e)) = \lambda(tail(e))$.

The requirements of each procedure having a unique entry and a unique exit node can be formalized as follows:

Property 3.9. $\forall e_1, e_2 \in C : \lambda(tail(e_1)) = \lambda(tail(e_2)) \Rightarrow tail(e_1) = tail(e_2)$.

Property 3.10. $\forall e_1, e_2 \in R : \lambda(head(e_1)) = \lambda(head(e_2)) \Rightarrow head(e_1) = head(e_2)$.

Definition 3.11. The set of *associated heads* of an edge e, denoted $ahead(e)$, is defined by

$$ahead(e) \triangleq \begin{cases} \{head(e)\} & \text{if } e \notin R, \\ \{head(e), head(\phi^{-1}(e))\} & \text{if } e \in R. \end{cases}$$

The sets $ahead(e)$ will be needed later in the article to introduce context-sensitivity in the data-flow equations of dominators. For a return edge e, $ahead(e)$ consists of the corresponding call-site and the exit block of the callee of the call.

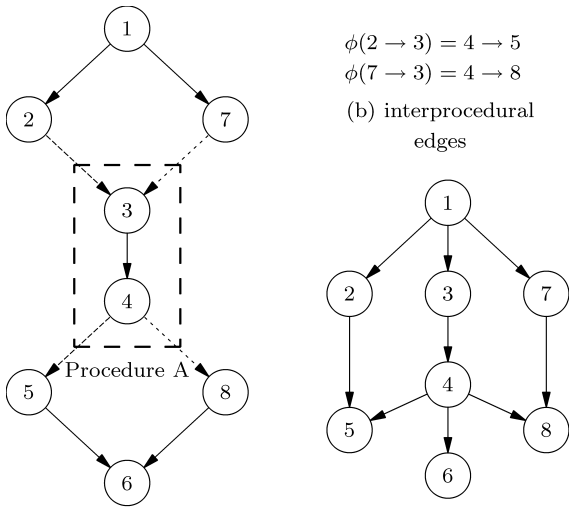
Definition 3.12. A *full valid path* in an ICFG $G = (V, E, C, R, r, q, \phi, \lambda)$ is a path $(e_0, \ldots, e_m)$ with $head(e_0) = r$ and $tail(e_m) = q$ where

$$\begin{aligned} &\forall i.0 < i \leq m : e_i \in R \Rightarrow \exists\, j.0 \leq j < i : \\ &(e_j = \phi^{-1}(e_i) \wedge (\forall l.j < l < i : e_l \in R \Rightarrow \exists\, k.j < k < l : e_k = \phi^{-1}(e_l)) \\ &\quad \wedge (\forall l.l > i : e_l \in R \Rightarrow \exists\, k.k < j \vee i < k < l : e_k = \phi^{-1}(e_l))). \end{aligned}$$

This definition reflects the fact that a procedure A that is called from within a procedure B must return before the calling instance of procedure B can return. The definition also implies that an exit node can only be reached through a call edge that corresponds to one of the exit node's successor return edges.

Definition 3.13. A *valid path* in an ICFG $G = (V, E, C, R, r, q, \phi, \lambda)$ is a path for which there exists a full valid path that contains this path. In other words, any subsequence of a full valid path is a valid path.

In the remainder of the article we will use the term path when we mean a valid path and no confusion is possible. With the above definitions, we can define the dominator relation in an ICFG.



(a) An example ICFG (c) the dominator graph

Fig. 2. An example ICFG with two procedures; procedure A contains the nodes 3 and 4, the function ϕ and the corresponding minimal dominator graph.

Definition 3.14. Node v dominates node w in an ICFG $G = (V, E, C, R, r, q, \phi, \lambda)$ if every valid path from r to w passes through v. We write $v \,\mathsf{D}\, w$. The so-called dominator set $dom(w)$ of nodes dominating node w in graph $G = (V, E, C, R, r, q, \phi, \lambda)$ is defined by

$$dom(w) \triangleq \{v \mid v \in V \wedge v \,\mathsf{D}\, w\}.$$

The definition of postdominance is analogous to that of dominance. The traditional dominator relation is usually represented as a graph of which the nodes are the nodes of the ICFG, and in which directed edges connect each node to the nodes it dominates. It has been shown that the transitive reduction of the traditional dominator graph is a tree [Allen and Cocke 1972].

The interprocedural dominator relation can also be represented by a graph. Its transitive reduction is a directed acyclic graph, but it is not necessarily a tree. The example ICFG and its transitively reduced dominator graph in Figure 2 illustrate this. We will call the graph representation of the dominator relation the *dominator graph D*. Its transitive reduction is called the *minimal dominator graph M*.

THEOREM 3.15. *The dominator graph is an acyclic graph.*

PROOF. Suppose that the dominator graph contains a cycle $v \,\mathsf{D}\, w$ and $w \,\mathsf{D}\, v$ for a pair of nodes v and w, with $v \neq w$. By definition, $v \,\mathsf{D}\, w$ implies that there is a path from r to w, that passes through v before it passes through w. In turn, this implies that there is a path that reaches v before it reaches w, which contradicts $w \,\mathsf{D}\, v$. Hence the dominator graph cannot contain cycles. □

THEOREM 3.16. $v \,\mathsf{D}\, w \wedge u \,\mathsf{D}\, w \not\Rightarrow v \,\mathsf{D}\, u \vee u \,\mathsf{D}\, v$.

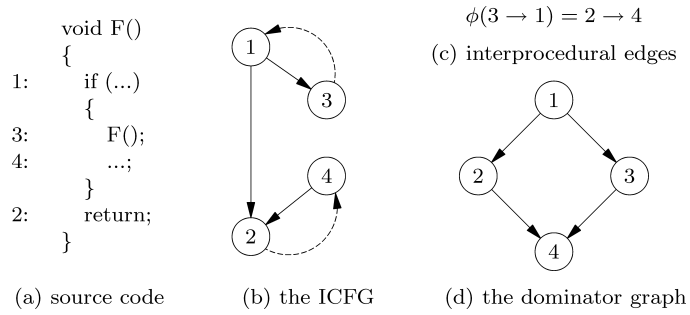


Fig. 3. A recursive procedure on the left, its ICFG in the middle, and its ϕ-function and minimal dominator graph on the right.

PROOF. We present two examples. The first can be found in the ICFG depicted in Figure 2. There, nodes 4 and 7, which are from different procedures, both dominate node 8, even though neither of the two dominates the other. To show that the two nodes u and v need not be from different procedures, we have depicted a recursive procedure in Figure 3. In its ICFG, both nodes 2 and 3 dominate node 4, as node 4 could only be reached after at least one recursive call was made, in block 3, and returned from, in block 2. □

This theorem contrasts sharply with the well-known theorem for traditional dominators that states that $v \,\mathsf{D}\, w \wedge u \,\mathsf{D}\, w \Rightarrow v \,\mathsf{D}\, u \vee u \,\mathsf{D}\, v$.

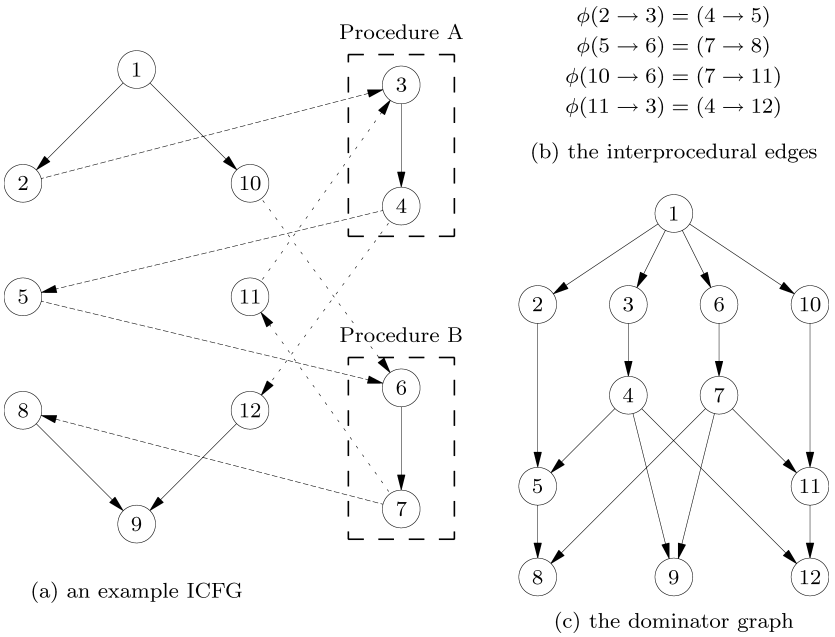
As a consequence of the latter theorem for traditional dominators, each node u in a procedure (except for the entry node) has a unique traditional *immediate dominator*, which is defined as that dominator of u that is executed last of all u's dominators on any path to u. Because each node has a unique traditional immediate dominator, the reduced traditional dominator graph is a tree, in which each node's immediate dominator is its sole predecessor.

In the interprocedural case, some nodes do not have a unique dominator that is the last executed dominator on all paths to them. An example of this is given in Figure 4. In the ICFG on the left of this figure, both nodes 4 and 7 dominate node 9, as can be seen in the corresponding minimal dominator graph on the right. In the path $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9$, node 7 is the last dominator of node 9, executed. On the path $1 \to 10 \to 6 \to 7 \to 11 \to 3 \to 4 \to 12 \to 9$, node 4 is the last dominator of node 9, executed.

Consequently, introducing the notion of interprocedural immediate dominators is useless. Because the Lengauer-Tarjan dominator computation algorithm relies on the existence of immediate dominators, this algorithm cannot be extended to compute interprocedural dominators. This is by far the most important practical consequence of Theorem 3.16.

4. A DATA-FLOW SOLUTION

In this section, we introduce the data-flow equations that need to be solved in order to compute interprocedural dominators. First, the difference with the data-flow equations of traditional dominators is discussed. Then an iterative algorithm to solve the interprocedural data-flow equations is presented. This



(a) an example ICFG

(b) the interprocedural edges

(c) the dominator graph

Fig. 4. An ICFG on which the immediate dominance relation cannot be uniquely defined; the minimal dominance graph; and the ϕ function.

simple, but inefficient algorithm will later serve as the basis for a much more efficient algorithm.

In the traditional dominator computation, a maximal fixed-point solution for the following data-flow equation needs to be found for each node v in G:

$$dom_G(v) = \left(\bigcap_{w \in pred_G(v)} dom_G(w) \right) \cup \{v\}, \tag{2}$$

$$= \left(\bigcap_{e \in in_G(v)} dom_G(head(e)) \right) \cup \{v\}. \tag{3}$$

These equations reflect that a block's traditional dominators are the common dominators of its predecessors in the graph, and the block itself. As we have seen in the example graph of Figure 2, this property does not always hold for interprocedural dominators. For example, it does not hold for nodes at the tail of return edges. In Figure 2, node 8 is dominated by nodes 4 and 7, even though node 7 does not dominate node 4, which is node 8's only predecessor in the graph.

In general, the dominators of a block at the tail of a return edge cannot be prescribed solely in terms of the block's predecessor nodes in the ICFG. Fundamentally, this follows from the fact that the exit node at the head of the return edge may be executable in contexts other than that of the corresponding call-site. For this reason, we need a context-sensitive version of the given

```
Algorithm DataFlow()
 1:  dom[r] ← {r}
 2:  forall nodes v ∈ V \ {r}
 3:      dom[v] ← V
 4:  changed ← true
 5:  while changed
 6:      changed ← false
 7:      forall nodes v ∈ V
 8:          new_dom_set ← (⋂_{e ∈ in_G(v)} dom_G(ahead(e))) ∪ {v}
 9:          if dom[v] ≠ new_dom_set
10:              dom[v] ← new_dom_set
11:              changed ← true
```

Fig. 5. An iterative algorithm that directly solves the context-sensitive data-flow equations.

equations:

$$dom_G(v) = \left(\bigcap_{e \in in_G(v)} dom_G(ahead(e)) \right) \cup \{v\}. \tag{4}$$

To find the maximal fixed-point solution of the context-sensitive data-flow equations for every node, the simple iterative algorithm in Figure 5 can be used. In this algorithm, a node's dominator set can change at most $O(|V|)$ times, since it can only get smaller with each iteration. Moreover, there are $|V|$ nodes. Hence $O(|V|^2)$ assignments can take place on line 10. In the worst case, each such change requires visiting $|V|$ blocks in one iteration of the while loop. Hence line 8 is executed at most $O(|V|^3)$ times.

Performing a single intersection or comparison operation also requires $O(|V|)$ time per intersection, as we can use sorted lists or a bit-vector of length $|V|$ to represent the sets. On average $|E|/|V|$ intersections are needed per node. The asymptotic complexity of the algorithm is therefore $O(|V|^3|E|)$.

The space required is $O(|V|^2)$, as we at most need to store $|V|$ dominator sets containing at most $|V|$ nodes each.

At this point, we can show why we require procedures to have a single entry point. Part of an ICFG containing an exemplary multiple-entry procedure is depicted in Figure 6. In this graph, there is only one execution path to reach node 5: $1 \to 2 \to 3 \to 4 \to 5$. Clearly, node 2 dominates node 5. However, node 2 does not dominate node 4, because node 4 can also be reached through $6 \to 3 \to 4$. Hence in this case Equation 4 does not hold, as $dom(5) \neq dom(ahead(4 \to 5)) \cup \{5\}$.

Fundamentally, the problem is that whether or not some node in a multiple-entry procedure will definitely be executed before the procedure is exited depends on the calling context of that procedure. Hence the dominators of a return node such as node 5 in the example cannot be described solely in terms of two other independent dominators sets anymore.

While it is possible to adapt the data-flow equation in order to deal with multiple-entry procedures, this would lead to very complex equations, and consequently to a very error-prone implementation. In practice, it is simpler to split a multiple-entry procedure into two or more single-entry procedures. For the

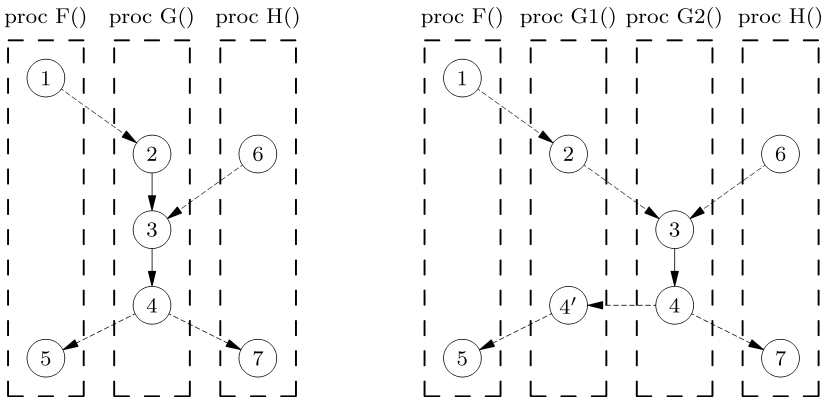


Fig. 6. In the left ICFG, we have three procedures of which the middle one has two entry points. In the right ICFG, that multiple-entry procedure is split into two procedures.

example ICFG on the left of Figure 6, the corresponding ICFG with split procedures is depicted on the right of that figure. In the right ICFG, the new, virtual node $4'$ enables the use of Equation 4 to define all dominator sets. $ahead(4' \rightarrow 5)$ adds all nodes in $dom(1)$ and in $dom(4')$ to $dom(5)$, and $ahead(4 \rightarrow 4')$ puts node 2 in $dom(4')$, and thus in $dom(5)$. Without node $4'$, node 2 would not have been added to $dom(5)$, as it is not present in $dom(4)$.

Converting multiple-entry procedures to single-entry ones is trivial. Because the solution of the dominator problem with algorithm *DataFlow*() is path-insensitive, this conversion leaves the dominator relation and the computed results unchanged.

5. CONTEXT-SENSITIVE DEPTH-FIRST TRAVERSAL

Our improved dominator algorithm will use information produced by a node numbering scheme that is best described as a preorder context-sensitive depth-first traversal (CSDFT). In this section, we present this CSDFT. In addition, we prove some important properties of the CSDFT that will enable a number of important optimizations on our algorithm.

Figure 7 depicts a nonrecursive algorithm to compute a preorder CSDFT numbering. The numbering is done in preorder, as a node is numbered on line 8 before its successors are traversed. The CSDFT basically traverses the nodes in the order a traditional DFT would traverse them. In this context-sensitive version however, return edges are not traversed until their corresponding call edges have been traversed. Note that the strange-looking condition tested on line 20 can evaluate to true for recursive procedures.

An example of the resulting numbering is depicted in Figure 1. Note that when the traversal arrives at node 10, there is no unvisited successor node left. But since the edge $10 \rightarrow 4$ is a call edge, the traversal continues at the return node (11) corresponding with call-site 10.

THEOREM 5.1. *The algorithm CSDFT terminates.*

Algorithm $CSDFT$

```
 0:  forall nodes v
 1:      DFT(v) ← −1
 2:  DFTstack ← ∅
 3:  DFTnumber ← 0
 4:  Push(DFTstack,r)
 5:  while DFTstack ≠ ∅
 6:      v ← Pop(DFTstack)
 7:      if DFT(v) ≠ −1 continue
 8:      DFT(v) ← DFTnumber
 9:      DFTnumber ← DFTnumber + 1
10:      forall edges e ∈ out_G(v)
11:          if e ∉ (C ∪ R)
12:              if DFT(tail(e)) = −1
13:                  Push(DFTstack, tail(e))
14:          else if e ∈ C
15:              if DFT(tail(e)) = −1
16:                  Push(DFTstack, tail(e))
17:              else if DFT(head(φ(e))) ≠ −1 ∧ DFT(tail(φ(e))) = −1
18:                  Push(DFTstack, tail(φ(e)))
19:          else if e ∈ R
20:              if DFT(head(φ^−1(e))) ≠ −1 ∧ DFT(tail(e)) = −1
21:                  Push(DFTstack, tail(e))
```

Fig. 7. Algorithm to compute preorder context-sensitive depth-first-traversal numbering for an ICFG $G = (V, E, C, R, r, q, \phi, \lambda)$.

PROOF. Each iteration of the while loop can only push a limited number of nodes onto the stack (lines 13, 16, 18, and 21), after one node has been popped (line 6) and numbered (line 8). This numbering can happen at most once per node because of the test on line 7. Hence the number of pushes performed is finite, as is the number of pops, and the number of iterations. □

LEMMA 5.2. *When CSDFT has ended,*

$$\forall e.e \in R : head(e) \text{ and } head(\phi^{-1}(e)) \text{ are numbered} \Rightarrow tail(e) \text{ is numbered.}$$

PROOF. Suppose $head(e)$ was numbered during iteration i of the while loop. At that time, either $head(\phi^{-1}(e))$ was already numbered during some previous iteration $j < i$, or it was not.

In the former case, node $tail(e)$ will be pushed onto the stack on line 21 in iteration i. Thus, it will be numbered when it is popped from the stack.

In the latter case, if $head(\phi^{-1}(e))$ is numbered during a later iteration $k > i$, $tail(e)$ will be pushed on the stack on line 18 in iteration k, and hence be numbered when it is popped from the stack. □

THEOREM 5.3. *The algorithm CSDFT numbers all nodes that are reachable through valid paths.*

PROOF. Suppose a node v is reachable through the valid path $P = (e_0, \ldots, e_m)$. We will prove by induction that $tail(e_m)$ will be numbered. To that extent, we will first prove that for any e_i in the path P:

$$(\forall j.0 \leq j \leq i : head(e_j) \text{ is numbered}) \Rightarrow tail(e_i) \text{ is numbered.}$$

In other words, if the heads of all edges of a prefix of P are numbered, so will the tail of the last edge in the prefix.

There are three possible types of edges for e_i. If $e_i \in C$, it is clear that when $head(e_i)$ was numbered on line 8 of the algorithm, $tail(e_i)$ would either be found to be already numbered (line 15), or it would be pushed on the stack (line 16), after which it will definitely be numbered. The same reasoning holds for the intraprocedural edges $e_i \in E \setminus (C \cup R)$, that are handled on lines 12–13.

For edges $e_i \in R$, the definition of full valid paths implies that $\exists\, j.0 \leq j < i : e_j = \phi^{-1}(e_i)$. When both $head(e_i)$ and $head(\phi^{-1}(e_i))$ are numbered, Lemma 5.2 states that $tail(e_i)$ will be numbered as well.

Since $head(e_0) = r$, $head(e_0)$ will definitely be numbered. By induction, all edges e_i will have their tail numbered, including $tail(e_m)$. □

LEMMA 5.4. *When CSDFT has ended,*

$$\forall v.v \in (V \setminus \{r\}) : \exists w.w \in pred_G(v) : \mathit{DFT}(w) < \mathit{DFT}(v).$$

PROOF. On lines 13, 16, and 21 of algorithm *CSDFT*, unnumbered nodes $tail(e)$ are pushed on the stack for which the predecessor $head(e)$ was already numbered on line 8. On line 18, $tail(\phi(e))$ is only pushed on the stack if $head(\phi(e))$ is numbered. Hence any push of a node, and by consequence that node's numbering, is only performed after at least one predecessor node was already numbered. □

LEMMA 5.5. *When CSDFT has ended,*

$$\forall e.e \in R : \max_{x \in ahead(e)} \mathit{DFT}(x) < \mathit{DFT}(tail(e)).$$

PROOF. Since both the pushes on line 18 and line 21 of such a $tail(e)$ are performed if and only if the two nodes in $ahead(e)$ are already numbered, $tail(e)$ for edges $e \in R$ will be numbered later than the nodes in $ahead(e)$. Hence their DFT-number will be higher. □

THEOREM 5.6. *When CSDFT has ended,*

$$\forall v, w.v, w \in V : v \,\mathsf{D}\, w \Rightarrow \mathit{DFT}(v) \leq \mathit{DFT}(w).$$

PROOF. First, we note that whenever a node w gets numbered during the execution of the CSDFT algorithm, there is at least one valid path to w that only goes through nodes that have already been numbered. To prove this, let us consider all nodes that are not yet numbered by the time node w gets numbered. These nodes either have not been put on the stack yet, or they have been put on the stack, but have not yet been popped. In both cases, they cannot have affected the numbering performed by the algorithm until node w got numbered. In other words, on a reduced graph G' that consists of only those nodes in V that have already been numbered and of only those edges in E that have been traversed during this numbering, the same numbering would have been applied to those nodes. Theorem 5.3 thus implies that in that reduced graph G', there exists a

valid path to node w. As this path is also present in G, and only includes nodes that have already been numbered, there exists at least one valid path to w in G, which we call P_{num}, that only goes through nodes that have already been numbered.

Since any dominator v of w needs to occur on all valid paths to w, it also needs to occur on P_{num}. As such, any proper dominator v of w must have been numbered before w itself is numbered. And since the numbers assigned during CSDFT only increase, any dominator v of w satisfies $DFT(v) \leq DFT(w)$. □

6. A CONSTRAINT-BASED ALGORITHM

The iterative data-flow solution presented in Figure 5 is both easy to understand and easy to implement. In fact, it doesn't differ much from the simplest implementation of a traditional dominator computation [Allen and Cocke 1972].

However, just like Cooper et al. [2001] noted for the intraprocedural version, the base iterative algorithm is very slow. When bit-vector representations are used to represent the dominator sets, the amount of memory required makes the algorithm impractical. For large programs, with several hundred thousand basic blocks, the required amount of memory even poses a problem when sparse set representations are used based on, for example, sorted linked lists. Moreover, with such sparse set representations large amounts of time are wasted on performing intersections on the sparsely populated sets, and on copying sets from one node to the other:

To overcome these problems, we propose a more efficient algorithm that is based on constraint solving and an efficient graph representation of set constraints. The most practical properties of this graph representation are that:

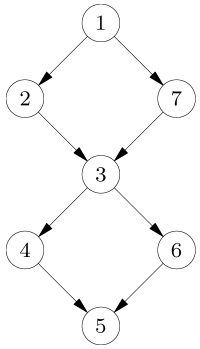
(1) the graph requires little memory;
(2) it enables efficient intersection computation for Equation 4;
(3) it does not require copying dominator sets;
(4) it enables several optimizations to the base algorithm.

In the remainder of this section, we present a base, suboptimal version of our constraint-based algorithm, and prove its correctness. This will enable us to focus on the basic concepts of the algorithm, instead of losing ourselves in smaller, less fundamental optimizations. Such optimizations are discussed in Section 7.

6.1 Dominator Set Constraints

Conceptually, our constraint-based algorithm starts with a set of rather loose constraints on dominator set, which can be derived directly from the CSDFT ordering of a program's nodes. All these constraints will be of the form

$$dom_G(v) \subseteq \left(\bigcup_{\{p_0,\cdots,p_n\} \subseteq V} dom_G(p_i) \right) \cup \{v\}. \tag{5}$$



$$dom(1) \subseteq \{1\}$$
$$dom(2) \subseteq dom(1) \cup \{2\}$$
$$dom(3) \subseteq dom(2) \cup \{3\}$$
$$dom(4) \subseteq dom(3) \cup \{4\}$$
$$dom(5) \subseteq dom(4) \cup \{5\}$$
$$dom(6) \subseteq dom(3) \cup \{6\}$$
$$dom(7) \subseteq dom(1) \cup \{7\}$$

Fig. 8. A small ICFG on the left, and the initial constraints derived from it on the right.

This set of constraints will then be transformed iteratively by applying Equation 4. Each time the equation is applied for a node v, a new stricter constraint of the same form is derived for v using previously derived constraints. When the algorithm finishes, the constructed constraints are met by the exact dominator sets. Moreover, the remaining constraints explicitly define the dominator sets.

To illustrate this concept, we will first apply it to the simple example ICFG depicted on the left of Figure 8. On the right, seven very simple initial constraints are stated, one for each node in the graph. For the top node of the graph, the constraint is trivial. For the other six nodes, the constraints state that the dominators of a node, excluding the node itself, should dominate one of its predecessors. As such, these initial constraints are looser than the general requirement stated in Equation 4. Consequently, these constraints are met by the solution to the dominator problem. Moreover, because of Lemmas 5.4 and 5.5, these initial constraints can be constructed such that each dominator set appearing on the right-hand side of a constraint is of a node that is numbered lower than the node appearing on the left hand side of the constraint. The initial set of constraints is hence acyclic.

Using Equation 4, stricter constraints can now be constructed. For example, Equation 4 is applied for node 5 as follows:

$$\begin{aligned} dom(5) &= (dom(4) \cap dom(6)) \cup \{5\} \\ &\subseteq ((dom(3) \cup \{4\}) \cap (dom(3) \cup \{6\})) \cup \{5\} \\ &\subseteq ((dom(3) \cap dom(3)) \cup (dom(3) \cap \{6\}) \cup (\{4\} \cap dom(3)) \cup (\{4\} \cap \{6\})) \cup \{5\} \\ &\subseteq dom(3) \cup \{5\}. \end{aligned}$$

In the first step of this derivation, the dominator sets on the right hand side of the equation are replaced by their upper bounds as stated by the existing constraints. In the second step, distributivity is applied. In the last step, Theorem 5.6 is used to deduct that $dom(3) \cap \{4\}$ is empty. The resulting new constraint on node 5 is stricter than its original constraint, which is hence replaced by the new one. When we perform a similar derivation for node 3, we end up with the

set of constraints

$$\begin{aligned}
dom(1) &\subseteq \{1\},\\
dom(2) &\subseteq dom(1) \cup \{2\},\\
dom(3) &\subseteq dom(1) \cup \{3\},\\
dom(4) &\subseteq dom(3) \cup \{4\},\\
dom(5) &\subseteq dom(3) \cup \{5\},\\
dom(6) &\subseteq dom(3) \cup \{6\},\\
dom(7) &\subseteq dom(1) \cup \{7\},
\end{aligned}$$

which can no longer be made stricter. By construction, this final set of constraints is still acyclic. Consequently, the following corresponding equations uniquely define the sets $dom_G(v)$:

$$\begin{aligned}
dom(1) &= \{1\},\\
dom(2) &= dom(1) \cup \{2\},\\
dom(3) &= dom(1) \cup \{3\},\\
dom(4) &= dom(3) \cup \{4\},\\
dom(5) &= dom(3) \cup \{5\},\\
dom(6) &= dom(3) \cup \{6\},\\
dom(7) &= dom(1) \cup \{7\}.
\end{aligned}$$

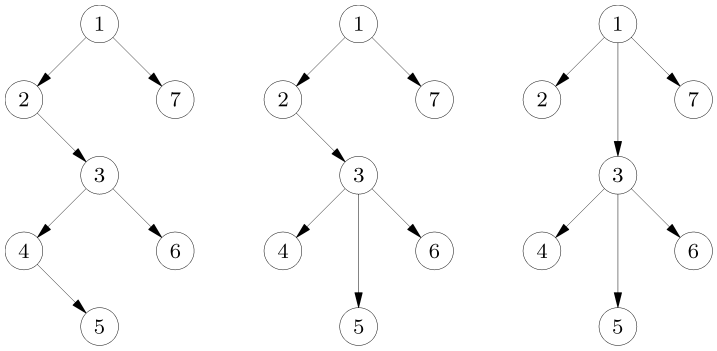
As we have only replaced "$\subseteq$" by "$=$" in these constraints, the sets thus defined are the maximal solution meeting all constraints. And because we have applied Equation 4 on all nodes v until we could not find stricter constraints, the dominator sets thus defined also meet Equation 4. Indeed, with these "$=$" constraints for $dom(4)$ and $dom(6)$, the derivation for $dom(5)$ can be repeated with "$=$" instead of "$\subseteq$". These final "$=$"-constraints therefore define a fixed-point solution for Equation 4. Consequently, the derived constraints define the maximal fixed-point solution to Equation 4.

The strength of this computation originates from the fact that, in the above derivation of $dom(5)$, we did not need to enumerate the nodes in $dom(4)$ and $dom(6)$. Instead, we computed the intersection $dom(4) \cap dom(6)$ using properties of the CSDFT ordering. In order to efficiently exploit these properties during the computation of the intersection of Equation 4, we will model all constraints with one big graph.

6.2 The Dominator Constraint Graph

In the dominator constraint graph C, all constraints of the form of Equation 5 are represented by directed edges from nodes p_i to node v. In other words, the constraint given by Equation 5 is equivalent to $pred_C(v) = \{p_0, \ldots, p_n\}$. As such, the graph C models the following constraints on the nodes v of a program:

$$\forall v.v \in V : dom_G(v) \subseteq \left(\bigcup_{p \in pred_C(v)} dom_G(p) \right) \cup \{v\} = anc_C(v). \qquad (6)$$



(a) initial C (b) C after node 5 has been recomputed (c) C after node 3 has been recomputed

Fig. 9. Three consecutive versions of the constraint graphs C of the ICFG G from Figure 8.

As we will show later, this equation is an invariant of our algorithm. Consequently, the algorithm will terminate when no stricter constraints can be generated than the ones already present in the graph. This means that the algorithm finishes when the following property holds:

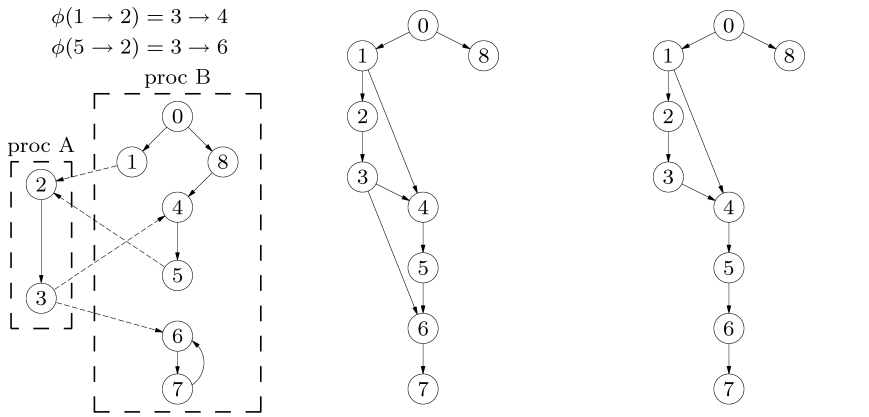
$$\forall v.v \in V : dom_G(v) = \left(\bigcup_{p \in pred_C(v)} dom_G(p) \right) \cup \{v\} = anc_C(v). \tag{7}$$

The final graph C that is produced by our algorithm will be equivalent to the minimal dominator graph M, for which the following, very similar, property holds:

$$\forall v.v \in V : dom_G(v) = \left(\bigcup_{p \in pred_M(v)} dom_G(p) \right) \cup \{v\} = anc_M(v). \tag{8}$$

For the initial constraints on the right of Figure 8, the corresponding graph C is depicted in Figure 9(a). Now instead of rewriting Equation 4 as in the derivation of the new constraint for $dom(5)$ in Section 6.1, we will rewrite the intersection $dom(4) \cap dom(6)$ by computing $anc_C(4) \cap anc_C(6)$ and by selecting a set of nodes from that intersection of which the ancestors equal the intersection. In the example, this results in the set $\{3\}$, with $anc_C(3) = anc_C(4) \cap anc_C(6)$. The replacement of the old constraint on $dom(5)$ by the new constraint $dom(5) \subseteq dom(3) \cup \{5\}$ is then reflected by redrawing the graph such that $pred_C(5) = \{3\}$ as in Figure 9(b). After a new constraint for $dom(3)$ has been derived similarly, we obtain the final C from Figure 9(c), which in this case equals the minimal dominator graph M.

It is important to note that the set of nodes that will become the new $pred_C(v)$ for node v for which a new constraint is derived, needs to meet more requirements than simply including a descendant of all nodes in the computed intersection. For example, consider the ICFG in Figure 10(a) and its corresponding initial C in Figure 10(b). How this initial graph is obtained is discussed in Section 6.3. When we derive a new constraint for node 6, we



(a) the ICFG G (b) the initial C (c) the ICFG after node 6 was recomputed incorrectly

Fig. 10. On the left, an example ICFG G in which procedure A calls procedure B at two call-sites, namely nodes 1 and 5. In the middle, the initial graph C is depicted, and on the right the adapted graph after the constraint for $dom(6)$ was recomputed incorrectly as $dom(6) \subseteq dom(5) \cup \{6\}$.

might compute the intersection $anc_C(ahead(7 \to 6)) \cap anc_C(ahead(3 \to 6)) = anc_C(7) \cap (anc_C(5) \cup anc_C(3)) = \{0, 1, 2, 3, 4, 5\}$. In the current graph, $anc_C(5)$ equals $\{0, 1, 2, 3, 4, 5\}$, so we might want to set $pred_C(6)$ to $\{5\}$, thus modeling a new constraint $dom(6) \subseteq dom(5) \cup \{6\}$ as reflected in the updated constraint graph in Figure 10(c).

However, this is not the constraint we would have derived by applying Equation 4 on existing constraints. With those, we would have made the following derivation:

$$\begin{aligned} dom(6) &= (dom(7) \cap (dom(5) \cup dom(3))) \cup \{6\}, \\ &\subseteq ((dom(6) \cup \{7\}) \cap (dom(5) \cup dom(3))) \cup \{6\}, \\ &\subseteq (dom(6) \cap (dom(5) \cup dom(3))) \cup \{6\}, \\ &\subseteq ((dom(5) \cup dom(3) \cup \{6\}) \cap (dom(5) \cup dom(3))) \cup \{6\}, \\ &\subseteq ((dom(5) \cup dom(3)) \cap (dom(5) \cup dom(3))) \cup \{6\}, \\ &\subseteq (dom(5) \cup dom(3)) \cup \{6\}. \end{aligned}$$

Clearly the constraint $dom(6) \subseteq dom(5) \cup \{6\}$ is stricter than $dom(6) \subseteq dom(5) \cup dom(3) \cup \{6\}$. In fact, the former constraint is too strict, and incorrect, as it is obvious from the ICFG that node 3 does dominate node 6, while it does not dominate node 5. The former constraint is too strict because, in order to derive such a constraint, we need to assume that $dom(3) \subseteq dom(5)$, as it is only under that assumption that the above derivation of $dom(6)$ can be continued to result in $dom(6) \subseteq dom(5) \cup \{6\}$. Now while the current graph C in Figure 10(b) suggests that indeed $dom(3) \subseteq dom(5)$, as there is a path from node 3 to node 5 via node 4, this suggestion cannot be backed up with already derived constraints. To the contrary, if we would have computed a new constraint for $dom(4)$ before recomputing the constraint on $dom(6)$, the graph C

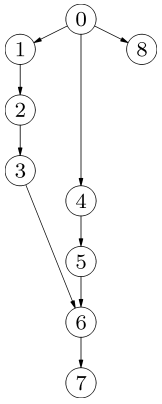


Fig. 11. The constraint graph C when the constraint on $dom(4)$ has been recomputed correctly as $dom(4) \subseteq dom(0) \cup \{4\}$, and the constraint on $dom(6)$ has not yet been recomputed.

would have looked as depicted in Figure 11. This graph does not at all suggest that $dom(3) \subseteq dom(5)$.

Note that because $dom(4)$ does not appear in the above derivation, this derivation does not depend on whether or not the constraint for $dom(4)$ was recomputed first. Between the graphs in Figure 10(b) and in Figure 11, the set of nodes in the intersection $anc_C(ahead(7 \rightarrow 6)) \cap anc_C(ahead(3 \rightarrow 6))$ does not change either, as it still equals $anc_C(7) \cap (anc_C(5) \cup anc_C(3)) = \{0, 1, 2, 3, 4, 5\}$. On this graph, however, $anc_C(5) \neq \{0, 1, 2, 3, 4, 5\}$. Instead $anc_C(5) \cup anc_C(3) = \{0, 1, 2, 3, 4, 5\}$, as we derived by rewriting the equations.

Now while it may seem that the constraint we derived incorrectly at the top of this page was only obtained because we computed the constraints of $dom(4)$ and $dom(6)$ in the wrong order, this is not true. There exist graphs, that are too large and too complex to illustrate in this article, in which any order of constraint computation on the graph C goes wrong if no special precautions are taken on how to select nodes from the computed intersection to become a node's new predecessors in C. What these precautions should be in order to ensure that no incorrect constraints are ever derived when computing the intersection of Equation 4, is discussed more formally in Section 6.5. But first we formally discuss the construction of the initial constraint graph C and the basics of our constraint-based algorithm.

6.3 Initializing the Dominator Constraint Graph

To start our dominator computation algorithm, we need to initialize the graph C with a set of constraints on dominator sets that we can derive from the program. These initial constraints must meet two requirements. Most importantly, they need to be met by the correct solution to the dominator problem. Hence the initial graph will need to respect the invariant stated in Equation 6. Furthermore, these constraints should be such that the initial constraint graph C is acyclic.

For deriving such initial constraints from the program, we will exploit the property that $(dom_G(v) \setminus \{v\}) \subseteq dom_G(ahead(e))$ for each of v's incoming edges

e, and in particular for that incoming edge e_{min} with the lowest numbered head. Because of Lemmas 5.4 and 5.5, and Theorem 5.6, initializing $pred_C(v)$ to $ahead(e_{min})$ of that edge e_{min}, meets the aforementioned requirements on the initial constraints. Concretely, we can initialize the graph by assigning a set $init(v)$ to $pred_C(v)$ for each v, that is computed as follows.

First, we need to determine v's predecessor with the smallest number. For edges other than return edges, this node's number is given by the value $min_{\not R}(v)$:

$$min_{\not R}(v) \triangleq \begin{cases} min_{e \in (in_G(v) \setminus R)}\, \boldsymbol{DFT}(head(e)) & \text{if } (in_G(v) \setminus R) \neq \emptyset, \\ \infty & \text{otherwise.} \end{cases}$$

For return edges, that can only be reached after their corresponding call-sites have been reached, we look for the corresponding call-site with the lowest number, which is captured in the value $amin_R(v)$:

$$amin_R(v) \triangleq \begin{cases} min_{e \in (in_G(v) \cap R)}\, \boldsymbol{DFT}(head(\phi^{-1}(e))) & \text{if } (in_G(v) \cap R) \neq \emptyset, \\ \infty & \text{otherwise.} \end{cases}$$

Of the two computed numbers, the smallest will be put in the set $init(v)$. Now if the smallest number corresponds to a return edge, we not only need to add the call-site, but we also need to add the exit-node at the head of the return edge. This is captured in the set $min_R(v)$:

$$min_R(v) \triangleq head(\{e \mid e \in (in_G(v) \cap R) \wedge \boldsymbol{DFT}(head(\phi(e))) = amin_R(v)\}).$$

Thus, all nodes are initialized with

$$init(v) \triangleq \begin{cases} \{\boldsymbol{DFT}^{-1}(min_{\not R}(v))\} & \text{if } min_{\not R}(v) \leq amin_R(v), \\ \{\boldsymbol{DFT}^{-1}(amin_R(v))\} \cup min_R(v) & \text{otherwise.} \end{cases}$$

Finally, we should note that it is possible that $min_{\not R}(v) = amin_R(v)$. This occurs for example, with conditional procedure calls, for which both a pair of call and return edges, and a fall-through path, connect the call-site to the return node. In such cases, we prefer to go with $min_{\not R}(v)$, as this results in the strictest initial constraint.

Figure 10(a) shows an ICFG of which the initial constraint graph C is depicted in Figure 10(b). For example, the initial constraint for node 4 is computed as follows:

$$\begin{aligned} min_{\not R}(4) &= \min_{e \in \{8 \to 4\}} \boldsymbol{DFT}(head(e)) = 8, \\ amin_R(4) &= \min_{e \in \{3 \to 4\}} \boldsymbol{DFT}(head(\phi(e))) = 1, \\ min_R(4) &= head(\{e \mid e \in \{3 \to 4\} \wedge \boldsymbol{DFT}(head(\phi(e))) = amin_R(4)\}) = \{3\}. \end{aligned}$$

Because $amin_R(4) < min_{\not R}(4)$, the initial constraint for node 4 becomes $dom_G(4) \subseteq dom_G(1) \cup dom_G(3) \cup \{4\}$. This is modeled in C with the edges $1 \to 4$ and $3 \to 4$.

```
Algorithm PracticalDomCompBase
 0:  forall nodes v ∈ V
 1:      pred_C[v] ← init(v)
 2:  changed ← true
 3:  while changed
 4:      changed ← false
 5:      forall nodes v ∈ V in pre-order CSDFT
 6:          new_pred_set ← CompConstraint(v)
 7:          if new_pred_set ≠ pred_C[v]
 8:              pred_C[v] ← new_pred_set
 9:              changed ← true
10:  reduce(C)
```

Fig. 12. The base version of our practical algorithm to compute interprocedural dominators.

6.4 The Basic Algorithm

Figure 12 depicts the base version of the constraint-based dominator computation algorithm, in which C is first initialized, and then iteratively updated with the function *CompConstraint*. Conceptually, *CompConstraint* takes Equation 4, in which all sets $dom_G(ahead(e))$ are substituted using their corresponding existing constraints, and generates a new constraint for $dom_G(v)$. Formally, the function $CompConstraint(v)$ is defined by the following equation:

$$\bigcap_{e \in in_G(v)} anc_C(ahead(e)) = \bigcup_{p \in CompConstraint(v)} anc_C(p). \tag{9}$$

Assuming the invariant of Equation 6 holds before *CompConstraint* is called, this implicit definition of *CompConstraint* results in a new constraint as follows:

$$\begin{aligned}
dom_G(v) &= \left(\bigcap_{e \in in_G(v)} dom_G(ahead(e)) \right) \cup \{v\} \text{ (because of Eq. 4)}, \\
&\subseteq \left(\bigcap_{e \in in_G(v)} anc_C(ahead(e)) \right) \cup \{v\} \text{ (because of Eq. 6)}, \\
&\subseteq \left(\bigcup_{p \in CompConstraint(v)} anc_C(p) \right) \cup \{v\} \text{ (because of Eq. 9)}, \\
&\subseteq \left(\bigcup_{p \in pred_C(v)} anc_C(p) \right) \cup \{v\} \text{ (after the assignment on line 8)}.
\end{aligned}$$

If the computed set $CompConstraint(v)$ differs from v's current set of predecessors $pred_C(v)$, the graph is updated accordingly on line 8 of the algorithm.

The above derivation proves that, given that the invariant of Equation 6 holds before the computation of $CompConstraint(v)$, the invariant holds at least for the node v of which $pred_C(v)$ is updated with the assignment on line 8.

This does not prove that Equation 6 is an invariant for all nodes in G however. Changing the predecessors of v on line 8 of the algorithm not only changes $anc_C(v)$, but potentially it also changes the sets $anc_C(w)$ of descendants w of

```
Algorithm CompConstraint(v)
0:  s ← some edge in in_G(v)
1:  new_pred_set ← ahead(s)
2:  forall edges e ∈ (in_G(v) \ {s})
3:      new_pred_set ← Intersect(new_pred_set, ahead(e))
4:  return new_pred_set
```

Fig. 13. The computation of a new constraint, based on the iterative pair-wise intersection over all edges coming into node v.

v in C. Hence the assignment on line 8 might invalidate Equation 6 for such nodes w.

In order to prohibit this invalidation from happening, each newly computed constraint needs to be such that no future assignment to a set $pred_C[v']$ of any (other) node v' will ever be able to invalidate the invariant for node v. If this is the case, it is guaranteed that the final solution meets all constraints modeled in C at any time during the computations. To achieve this, we need to impose additional restrictions on constraints generated by *CompConstraint*(v).

In our algorithm, we will ensure this by requiring, and guaranteeing, that a computed set of nodes *CompConstraint*(v) $= \{p_0, \ldots, p_n\}$ is computed completely independently of the sets $anc_C(p_i)$. When this requirement is met, no future change to any such set $anc_C(p_i)$ can invalidate the invariant.[2] We call this requirement the *independent constraint requirement*. If this requirement is met, any newly derived constraint will be based on constraints that have been proved to be correct earlier, but it will not be based on other, accidental properties of the current graph C that may later prove to be invalid. In the next section, we present a *CompConstraint* that meets the independent constraint requirement.

6.5 Construction of New Constraints

CompConstraint's main job is to compute a set that meets Equation 9. As such, it must rewrite the intersection of Equation 4 as a union. One way to do so would be to generate the sets anc_C, compute the set that constitutes the intersection in a first step, and then rewrite this set as a union of ancestor sets in a second step. In practice, however, this two-step approach would be a very time-consuming operation.

Instead, we have developed an efficient algorithm that combines both steps without needing to explicitly compute the intersection. This algorithm, which performs iterative pair-wise computations on all elements in $ahead(in_G(v))$, is presented in Figures 13 and 14. Figure 13 shows the outer loop that iterates over the elements in $ahead(in_G(v))$, and Figure 14 displays the actual pair-wise computation that in essence computes the intersection of the two sets of ancestors of its two arguments. But instead of returning the intersection itself, this computation returns a set of nodes of which the ancestors form the

[2]While there might exist more relaxed sufficient restrictions on *CompConstraint*(v) that, for example, make its computations depend on the descendants w of v, we believe that the implemention of such restrictions will be very inefficient and difficult in practice, if at all possible.

```
Algorithm Intersect(s1, s2)
 0:  visited ← pred ← ∅
 1:  to_visit ← s1
 2:  unmark all nodes in C
 3:  iteratively mark all nodes in anc_C(s2)
 4:  while to_visit ≠ ∅
 5:      v ← one element from to_visit
 6:      to_visit ← to_visit \ {v}
 7:      visited ← visited ∪ {v}
 8:      if (v is marked)
 9:          pred ← pred ∪ {v}
10:      else
11:          to_visit ← to_visit ∪ (pred_C[v] \ visited)
12:  return pred
```

Fig. 14. The pair-wise intersection computation to compute the new predecessors of a node.

intersection. Indeed, algorithm *Intersect* is implicitly defined such that

$$anc_C(Intersect(s_1, s_2)) = anc_C(s_1) \cap anc_C(s_2).$$

As the ancestor relation is transitive, the resulting set $CompConstraint(v)$ obviously satisfies Equation 9.

First, *Intersect* marks s_2's ancestors in C on line 3 of this algorithm.[3] Then the algorithm iteratively traverses C in an upwards direction, starting from the nodes in s_1. This traversal ends when a marked node is visited. Because any node in C can have multiple predecessors, this algorithm might have to traverse multiple paths. Each of these paths needs to be traversed until a marked node is reached, and all of these marked nodes need to be included in the result of $Intersect(s_1, s_2)$.

This implementation of *Intersect* clearly meets the independent constraint requirement stated in the previous section, as none of the elements in $anc_C(v)$ of nodes v that end up in the final result *pred* (on line 9) are traversed. In other words, the constraint generated by *Intersect* is independent of the ancestor sets of elements in the resulting *pred* set.

It is important to observe that the thus computed set $CompConstraint(v)$ is not necessarily the smallest set that satisfies Equation 9 in the current graph C. It may in fact happen that $CompConstraint(v)$ includes two nodes p_1 and p_2 of which $p_1 \in anc_C(p_2)$, as was the case with nodes 3 and 5 when the constraint for $dom(6)$ was recomputed before recomputing that of $dom(4)$ in Section 6.2. In such a case, node p_1 can clearly be omitted from $CompConstraint(v)$ without violating Equation 9. This removal violates the independent constraint requirement, so it can lead to incorrect solutions, as demonstrated in Section 6.2.

A direct consequence of this observation is that the derived graph C will usually not equal the minimal dominator graph M. Instead C will only be an

[3]Note that unmarking all nodes in a graph, as on line 2 of algorithm *Intersect*, is a constant time operation if we use an integer attribute for marking nodes. For example, if the attribute is $mark(v)$, then $IsMarked(v) \triangleq mark(v) = global\_marking\_number$ and $Mark(v) \triangleq mark(v) \leftarrow global\_marking\_number$. Unmarking all nodes then simply consists of incrementing *global_marking_number* and checking for overflow.

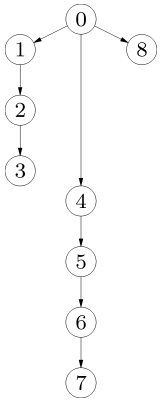


Fig. 15. The constraint graph C of the ICFG G, which is derived from the incorrect graph in Figure 10(c) after we have also recomputed the constraint for $dom(4)$.

approximation of M that still needs to be reduced to obtain M after its iterative redrawing has converged. In our basic algorithm *PracticalDomCompBase*, this reduction is added on line 10.

Now one might think that the independent constraint requirement is not fundamental. After all, the invariant of our algorithm and the resulting independent constraint requirement have so far only been used to prove our algorithm correct. They have not been set forward as necessary conditions, so why don't we just base our correctness proof on other invariants?

In practice, it is problematic if Equation 6 does not hold invariably during the computations. For example, it is impossible to later include node 3 in the ancestors of node 6 once the incorrect constraint on $dom(6)$ was computed in Section 6.2 and the graph C was adapted incorrectly. The reason is that, once the constraint for $dom(4)$ has been recomputed and the graph updated accordingly, as in Figure 15, *CompConstraint*(6) will be computed as *Intersect*($\{7\}, \{3, 5\}$), in which node 3 is no longer an ancestor of node 7. Consequently, node 3 can never again become an ancestor of node 6.

While developing our algorithm, we have tried many ways to reduce the size of the computed *CompConstraint*() sets without breaking the invariant. Some of the techniques we tried involved backtracking mechanisms that allow speculating which nodes may be omitted from *CompConstraint*() sets. None of these techniques proved to be worthwhile, however, mainly because speculated decisions often propagate very far into the computations before being detected as incorrect.

Fortunately, the sets $pred_C(v)$ in our algorithm still remain much smaller than the dominator sets computed in the data-flow algorithm, even though we do not compute the minimal dominator graph M directly. Hence our new algorithm will still prove to be much more efficient.

To finish this discussion of our implementation of the intersection operation, we should note that the traversal of the graph in *Intersect* is very similar to the *two-finger algorithm* described by Cooper et al. [2001]. The latter is used as a meet operator for traditional dominator computation, where one uses two

fingers to point at two nodes in the intermediate dominator tree and then moves the fingers upwards until a common ancestor of the two nodes is found. The difference with our algorithm is that we require more fingers, as our contraint graph is not a tree.

6.6 Algorithm Termination

Now that we have presented the base algorithm *PracticalDomCompBase* in detail; we will discuss its termination. Because a formal proof of the monotonicity and termination of the algorithm is very cumbersome, we will only outline some of the arguments.

6.6.1 *Monotonicity.* Just like the data-flow algorithm, our constraint-based algorithm operates on sets that model dominator sets. In the constraint-based algorithm, these are the $anc_C(v)$ sets. It is obvious that the sets $anc_C(v)$ can only shrink with every application of *CompConstraint*(v), just like the dominator sets only shrink in the data-flow algorithm. The sets $anc_C(v)$ converge toward their lower bound set by Equation 6.

6.6.2 *Termination.* Because the stop criterion of our constraint-based algorithm is based on a comparison of sets $pred_C(v)$ instead of sets $anc_C(v)$, the monotonicity of the $anc_C(v)$ sets by itself does not guarantee termination.

We note that the function *Intersect*(s_1, s_2) does not depend on the way $anc_C(s_2)$ is computed. It hence does not depend on the precise set $pred_C(s_2)$. Therefore the computations in *CompConstraint*(v) only depend on ancestors sets, and on at most two specific predecessor sets, namely those of $ahead(s)$ of the edge s that is chosen on line 0 of the code depicted in Figure 13.

We can easily impose the restriction that the selection of the edge s should be deterministic (which is trivially so when implemented in a deterministic programming language), such that $\max(DFT(ahead(s))) < DFT(v)$. This is possible because of Lemmas 5.4 and 5.5. Once the $anc_C()$ sets have become fixed, the computation of a set *CompConstraint*(v) only depends on sets $pred_C(w)$ of nodes w of which $DFT(w) < DFT(v)$.

If the main while loop in algorithm *PracticalDomCompBase* iterates over the nodes in preorder CSDFT, this means that as soon as all $anc_C()$ sets have reached their fixed point, the $pred_C()$ sets will become fixed as well, thus guaranteeing termination of the algorithm. At that point, the constraints are still acyclic, hence they define the correct dominator sets directly, as indicated for the example in Section 6.1.

6.7 Complexity

Our constraint-based algorithm does not reduce the worst-case running time complexity of the dominator computation. The algorithm is derived from the data-flow implementation, and no underlying assumptions to calculate the worst-case time complexity have changed: each of the $|V|$ sets $anc_C(v_i)$ can be made smaller at most $|V|$ times. Since at least one $anc_C(v_i)$ becomes smaller with every iteration over all $|V|$ nodes in the outer loop of the algorithm, at most $|V|^3$ invocations of *CompConstraint*() can be required. Executions of

CompConstraint(v_i) on average will need to invoke *Intersect*() $|E|/|V|$ times. And each computation of *Intersect*() requires marking at most $|V|$ nodes. Hence the theoretic time complexity is still $O(|V|^3|E|)$. Furthermore, the space complexity remains $O(|V|^2)$. This follows from the assumption that the number of call-sites in a program is linear to the number of basic blocks $|V|$, and from the fact that each of the $|V|$ nodes in G theoretically can have all the procedure exit blocks as predecessors.

Even though the theoretic complexities are identical for both algorithms, the constraint-based algorithm will prove to be much faster than the data-flow algorithm. This is particularly so when the optimizations discussed in the next section are applied to the base algorithm.

7. FURTHER OPTIMIZATIONS

In this section, we present a number of important optimizations to the base algorithm.

7.1 Optimizing the Intersection—Part 1

When the preorder CSDFT numbering is used, we observe that when a node v is removed from the *to_visit* set on line 6 of the algorithm *Intersect*, the nodes added on line 11 will all have smaller DFT-numbers. In other words, during the execution of this part of the algorithm, the value $\min_{x \in (to\_visit \cup visited)} DFT(x)$ only decreases. Furthermore, no node v with $DFT(v) < \min_{x \in (to\_visit \cup visited)} DFT(x)$ will ever be added to *pred* on line 9 of the algorithm. Having nodes marked with numbers lower than $\min_{x \in (to\_visit \cup visited)}$ is therefore of no use during the algorithm.

This can be exploited because the iterative marking of $anc_C(s_2)$ also iterates over ever decreasing numbers. Instead of immediately marking all nodes in $anc_C(s_2)$, it initially suffices to mark all nodes in $anc_C(s_2)$ that have numbers higher than $\min_{x \in s_2} DFT(x)$. Later on, additional marking can be performed whenever $\min_{x \in (to\_visit \cup visited)}$ changes. Often at least parts of the original marking can thus be avoided. The new, optimized algorithm is depicted in Figure 16.

To demonstrate how this optimization works, consider how we apply the intersection computation on node 5 of the ICFG in Figure 8. For this node the algorithm invokes *Intersect*({4}, {6}) and on line 3 of that computation nodes 6, 3, 2, and 1 are marked as ancestors of node 6 in the original constraint graph depicted in Figure 9(a). In the optimized version, *CSDFTIntersect*({4}, {6}) first assigns the value 4 to *min_dft* on line 3. Then on line 4, no nodes are initially marked at all. Node 4 is taken from the *to_visit* set on line 6 and added to the *visited* set on line 7. Then the nodes in $pred_C[4]$ are considerd on line 12, and *min_dft* is updated to $\min(4, 3) = 3$, and node 3 is marked on line 13. On line 14, node 3 is added to *to_visit*, and in the next iteration the loop is exited after node 3 is added to the *pred* set. So in this optimized version, only node 3 needs to be marked.

In this example, the optimization might seem somewhat of an overkill, because the initial savings due to the reduced initial marking seem minimal and the inner while loop of the computation has become significantly more complex.

Algorithm $CSDFTIntersect(s_1, s_2)$

```
 0:   visited ← pred ← to_visit ← ∅
 1:   to_visit ← s1
 2:   unmark all nodes in C
 3:   min_dft ← min_{x∈to_visit} DFT(x)
 4:   iteratively mark all nodes w in anc_C(s2) for which DFT(w) ≥ min_dft
 5:   while to_visit ≠ ∅
 6:       v ← one element from to_visit
 7:       to_visit ← to_visit \ {v}
 8:       visited ← visited ∪ {v}
 9:       if (v is marked)
10:           pred ← pred ∪ {v}
11:       else
12:           min_dft ← min (min_dft, min_{x∈pred_C[v]} DFT(x))
13:           continue marking nodes w in anc_C(s2) for which DFT(w) ≥ min_dft
14:           to_visit ← to_visit ∪ (pred_C[v] \ visited)
15:   return pred
```

Fig. 16. The CSDFT-optimized algorithm to compute the pair-wise intersection. The dots on the left mark lines that have changed compared to the original code in Figure 14.

On large programs, however, the initial savings become much larger, especially because the initial marking does not pollute the cache with nodes that will not later be visited in the inner loop. Furthermore, the average number of executed iterations of the while loop in *CSDFTIntersect* increases much more slowly as programs become larger, than the increase in the average number of nodes in ancestor sets. So for large programs, the cost of the complicated while loop is very small compared to the cost of the initial marking in the original *Intersect*() algorithm. Slowing down the while loop, even considerably, is therefore more than compensated by the speedup of the initial marking.

As we mentioned in Section 6.5, our intersection computation borrows ideas from the two-finger algorithm described by Cooper et al. [2001]. With this optimization, the resemblance becomes total. Just as in the algorithm by Cooper et al. [2001], fingers are moved conditionally, when the position of the other fingers indicates it may be useful.

7.2 Optimizing the Intersection—Part 2

Another optimization of the intersection computation relates to leaf procedures, and what we will call pseudo-leaf procedures. To illustrate this, consider the example ICFG in Figure 17 and suppose we need to compute *CompConstraint*(3). During this computation, *CSDFTIntersect*({2}, {7}) will be invoked, in which the initial marking on line 4 would need to mark nodes 7, 6, 5, and 4.

In this example, marking nodes 5 and 6 is clearly useless. Because procedure G() is a leaf procedure whose nodes all have higher numbers than node 3 for which we are computing the intersection, we know beforehand that no dominators of node 3 will be found during the traversal of nodes in procedure G(). Hence the marking of nodes 5 and 6 could have been skipped.

In general, a necessary condition to skip marking the nodes in a procedure F during the computation of *CompConstraint*(v) is the existence of at least one

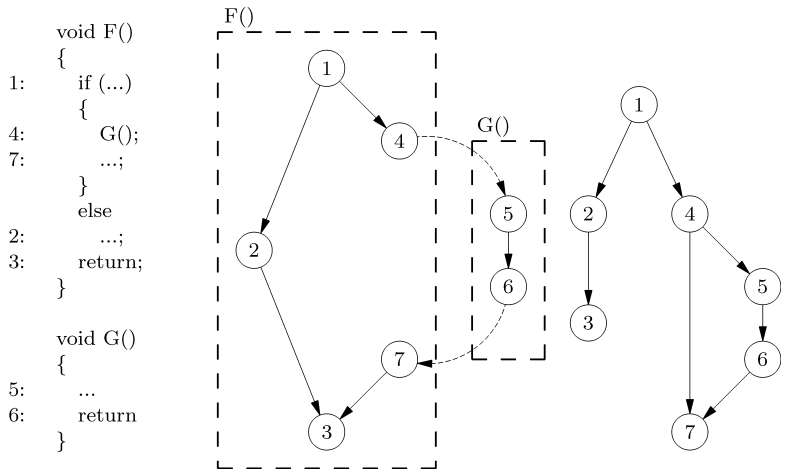


Fig. 17. On the left, the relevant parts of the source code of two procedures. The labels identify their preorder CS-DFT numbers. In the middle, the corresponding ICFG is depicted, and on the right the initial constraint graph.

path through F (including its callees) on which all nodes are numbered higher than v. Because testing this condition is complex and very time-consuming, we do not compute this necessary condition, but instead fall back on any of the following, much simpler sufficient conditions (in increasing order of complexity):

—there are no nodes in F or any of the callees in its call chain that are numbered lower than v,

—F does have callees with lower numbered nodes, but there exists an execution path through F on which no such callees are called,

—F does have callees with lower numbered nodes; those callees are called on every path through F, but there exist paths in those callees that do not pass through nodes with those lower numbers.

From a dominator computation point of view, procedures that meet any of these conditions can be treated as if they were leaf procedures, hence we call them pseudo-leaf procedures. In practice, we can precompute for each pseudo-leaf procedure the lowest numbered node that will certainly be executed when the procedure is invoked. This can be done with varying levels of complexity, depending on which of the above sufficient conditions one is willing to consider. In our implementation, we opted for the former two conditions, because computing the latter consumed more additional execution time than it saved by speeding up the intersection computations.

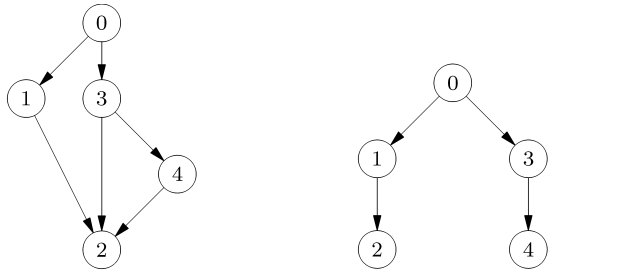
Once this precomputation is done, all nodes v in the graph are given a new attribute, say $lowest\_callee\_dft(v)$. For nodes that are not exit blocks of pseudo-leaf procedures, this attribute is set to -1. For exit nodes of pseudo-leaf procedures, this attribute is set to the precomputed number of their procedure.

With this new attribute, the algorithm $CSDFTIntersect(s_1, s_2)$ is adapted to the version in Figure 18. Note that it now takes an additional argument, namely the node n, for which the computations are being performed.

Algorithm $CSDFTIntersectSkip(s_1, s_2, n)$

```
 0:  visited ← pred ← to_visit ← ∅
 1:  to_visit ← s1
 2:  unmark all nodes in C
 3:  min_dft ← min_{x∈to_visit} DFT(x)
 4:  iteratively mark all nodes w in anc_C(s2) for which DFT(w) ≥ min_dft \
 5:      and for which lowest_callee_dft(w) < DFT(n)
 6:  while to_visit ≠ ∅
 7:      v ← one element from to_visit
 8:      to_visit ← to_visit \ {v}
 9:      visited ← visited ∪ {v}
10:      if (v is marked)
11:          pred ← pred ∪ {v}
12:      else
13:          min_dft ← min (min_dft, min_{x∈pred_C[v]} DFT(x))
14:          continue marking nodes w in anc_C(s2) for which DFT(w) ≥ min_dft \
15:                  and for which lowest_callee_dft(w) < DFT(n)
16:          to_visit ← to_visit ∪ (pred_C[v] \ visited)
17:  return pred
```

Fig. 18. The CSDFT-optimized algorithm to compute the pair-wise intersection that skips the unnecessary marking of nodes in pseudo-leaf procedures.



(a) a small ICFG G (b) the initial constraint graph C

Fig. 19. On the right a small ICFG, on the left its initial constraint graph.

In this new version, the iterative marking of nodes on lines 4 and 14 is now limited by the additional conditions on lines 5 and 15. During that iterative marking, no exit nodes of leaf or pseudo-leaf procedures with a sufficiently high attribute *lowest_callee_dft*(w) will be traversed.

7.3 Eliminating Redundant Intersections

Besides optimizing the intersection computation itself, we can also minimize the number of pair-wise intersections that needs to be performed. When we look back at the intersection algorithm, we can observe that it is of no use to include an edge $e \in in_G(v)$ in the intersection if there exists another edge $e' \in in_G(v)$ such that $ahead(e') \subseteq anc_C(ahead(e')) \subseteq anc_C(ahead(e))$. In that case, the final result of the meet operation will remain unchanged whether or not we included the edge e.

As an example, consider the simple example graph in Figure 19(a), and its correponding initial constraint graph in Figure 19(b). When a new constraint for

node 2 needs to be computed, we essentially compute $anc_C(1) \cap anc_C(3) \cap anc_C(4)$, by means of *CSDFTIntersectSkip*(*CSDFTIntersectSkip*({1}, {3}, 2), {4}, 2). Now from the constraint graph C, it is obvious that $anc_C(3) \cap anc_C(4) = anc_C(3)$. So in fact, we only need to compute *CSDFTIntersectSkip*({1}, {3}, 2). In the constraint graph of this example, node 1 is rather close to nodes 3 and 4, so the one invocation of *CSDFTIntersectSkip*() that we can avoid by first comparing $anc_C(3)$ to $anc_C(4)$ will not save us very much computation time. On larger graphs, however, it can be worthwhile to avoid redundant computations by first performing some additional tests.

To test which edges in $in_G(v)$ can be skipped, we can first iteratively mark all proper descendants in C of each node in $ahead(in_G(v))$. All edges e for which the nodes in $ahead(e)$ got marked can then be skipped in the meet operation of the original algorithm. However, simply iterating over a node's descendants in an (approximate) minimal dominator graph is a rather expensive operation on average. So with this method, we would spend a lot of time in the tests.

Fortunately, even though the minimal dominator graph is not a tree but a graph, it usually looks much more like a tree than like an inverted tree because most nonleaf nodes have fewer predecessors than successors. Consequently, iterating over, or marking, all ancestors of a set of nodes is on average much cheaper than iterating over, or marking, all its descendants.

So to avoid the high cost of iterating over all proper descendants of all nodes in $ahead(in_G(v))$, we can first mark all nodes $w \in anc_C(ahead(e))$ for which $DFT(w) > \min_{x \in ahead(in_G(v))} DFT(x)$, which can be done very efficiently. Thereafter, we only need to unmark the proper descendants of nodes in $ahead(in_G(v))$ that were previously marked, which is a much smaller set of descendants.

Furthermore, we have observed that this optimization should be applied only for nodes v that have a large number of incoming edges, and that have no incoming return edges. Otherwise, the additional cost of finding the edges to skip is not compensated by the avoided invocations of *CSDFTIntersectSkip*().

The resulting optimized algorithm *CompConstraintSkip*(v) is depicted in Figure 20. The bottom of the figure depicts the pseudo-code for the auxiliary function *ComputeEdgesToBeIntersected*, in which $desc^+(v)$ denotes the set of proper descendants of node v; the set of descendants excluding v itself.

It is important to note that this optimization does not break the independent constraint requirement. When it is invoked, *CompConstraintSkip*() only exploits properties of the current graph C to optimize its computations. The resulting set does not change however.

7.4 A Work-List Algorithm

The most fundamental optimization to the base algorithm involves the addition of a work-list. With a work-list, much fewer nodes need to be visited during later iterations of the while loop in algorithm *PracticalDomCompBase*, which allows for a considerable speedup.

In order to see how we can get to a work-list algorithm, we need to study the consequences of the assignment on line 8 of algorithm *PracticalDomCompBase* (see Figure 12). Every time $pred_C[v]$ gets assigned on line 8, the ancestor set

```
Algorithm CompConstraintSkip(v)

•  0:  S ← ComputeEdgesToBeIntersected(v)
•  1:  s ← some edge in S
   2:  new_pred_set ← ahead(s)
   3:  forall edges e ∈ (S \ {s})
   4:      new_pred_set ← CSDFTIntersectSkip(new_pred_set, ahead(s), v)
   5:  return new_pred_set

Function ComputeEdgesToBeIntersected(v)

   6:  if v has enough predecessors
   7:      unmark all nodes w in V
   8:      mark all nodes w in anc_C(ahead(in_G(v))) for which \
   9:        DFT(w) > min_{x∈ahead(in_G(v))} DFT(x)
  10:      unmark all nodes w in desc_C^+(ahead(in_G(v)))
  11:      return in_G(v) \ {e | e ∈ E ∧ all nodes in ahead(e) are unmarked}
  12:   else
  13:      return in_G(v)
```

Fig. 20. The optimized computation of a new constraint in which unnecessary intersections are skipped.

$anc_C(v)$ of node v shrinks or remains identical. In other words, the possibly empty set $removed(v) \triangleq anc_C^-(v) \setminus anc_C^+(v)$ is removed from $anc_C(v)$. Here, the superscript $+$ is used to denote a set immediately after an assignment on line 8 of the algorithm, while a superscript $-$ denotes a set just prior to such an assignment. Implicitly, the sets $anc_C(w)$ of nodes $w \in desc_C(v)$ are reduced as well. Obviously, this implicit reduction can only involve nodes in $removed(v)$.

Conceptually, removing the nodes $removed(v)$ from $anc_C(v)$ corresponds to v's constraint being made stricter. As a consequence of this operation, new opportunities might be created to make other constraints stricter as well, either during this iteration of the while loop, or during the next iteration.

Hence to come to a work-list algorithm, we need to answer the following question:

> When we have made the constraints for all nodes as strict as possible, except for the constraint of one node v, how might replacing that constraint create new opportunities to restrict other constraints? In other words, if we replace the set $pred_C^-(v)$ by $pred_C^+(v)$, which nodes could require the recomputation of $CompConstraint(v)$?

Algorithm *PracticalDomCompBase* is implemented as if the best answer to this question is the set V of all nodes: whenever a set $pred_C(v)$ changes in one iteration of the while loop, all sets $pred_C(w)$ for all nodes $w \in V$ will be recomputed in the next iteration of the while loop. This section presents a conservative, but more aggressive answer to the above question, and an efficient method to compute the answer on the fly.

Obviously, the replacement of $pred_C^-(v)$ by $pred_C^+(v)$, and the corresponding removal of $removed(v)$ from $anc_C(v)$, can only influence the computation of $CompConstraint(w)$ for nodes w that can be reached from v in the ICFG. A

Algorithm $PracticalDomCompWorkList()$

```
 0:  forall nodes v ∈ V
 1:      pred_C[v] ← init(v)
 2:      smallest_changed[v] ← ∞
 3:  changed ← true
 4:  need_to_redo = V
 5:  while changed
 6:      changed ← false
 7:      forall nodes v ∈ need_to_redo in pre-order CSDFT order
 8:          need_to_redo ← need_to_redo \ {v}
 9:          new_pred_set ← CompConstraint(v)
10:          if new_pred_set ≠ pred_C[v]
11:              unmark all nodes in V
12:              iteratively mark all nodes in anc_C(v) \ anc_C(new_pred_set)
13:              forall marked nodes w
14:                  smallest_change[w] ← min(smallest_change[w], DFT(v))
15:              pred_C[v] ← new_pred_set
16:              changed ← true
17:      forall nodes v ∈ V
18:          if smallest_change[v] ≠ ∞
19:              forall nodes w ∈ succ_C(v)
20:                  if smallest_change[v] ≤ max_pred(w)
21:                      need_to_redo ← need_to_redo ∪ {w}
22:          smallest_change[v] ← ∞
23:  reduce(C)
```

Fig. 21. The version of our practical algorithm to compute interprocedural dominators that avoids unnecessary recomputations of $pred_C[v]$ by using a work list.

necessary condition for this to hold is that

$$DFT(v) \leq max\_pred(w) \triangleq \max_{x \in ahead(in_G(w))} DFT(x).$$

As $max\_pred(w)$ can be precomputed for every node w before the main loop in *PracticalDomCompBase* starts, testing this condition during the algorithm is very cheap.

Furthermore, the removal of $removed(v)$ from $anc_C^-(v)$ can only result in the need to remove nodes from $anc_C^+(w)$ when $removed(v) \cap anc_C^+(w) \neq \emptyset$. In a slightly different form, it is required that $w \in desc_C(removed(v))$.

Figure 21 depicts algorithm *PracticalDomCompWorkList*, in which these two requirements are implemented. In this algorithm, the variable *need_to_redo* holds the set of nodes for which recomputation is required because their corresponding constraint can still potentially be made stricter.

The array *smallest_change* is used to store the value $min_{\{v|w \in removed(v)\}} DFT(v)$ for each node w, for each iteration of the while loop. These values are computed during each iteration of the while loop by updating them on line 14 each time a node w is removed from a set $anc_C(v)$.

At the end of each iteration of the while loop, the stored values are used on line 20 of the new algorithm to verify whether the two necessary conditions hold for a node w. This verification can be limited to the successors in C of nodes v who's value $smallest\_change[v]$ was set, because it is through these nodes,

according to Equation 7, that the nodes v may influence the ancestor sets of other nodes.

To end the discussion of this optimization, we would like to note that the proposed transformation of the base algorithm into a work-list algorithm can also be applied to the traditional dominator algorithm proposed by Cooper et al. [2001]. We have not yet studied the potential of this optimization in that context.

7.5 Exploiting A Priori Known Information

A final optimization to our algorithm exploits the fact that we can estimate the strictest constraints a priori. More precisely, for nodes with only one incoming edge, we don't need to recompute anything during the iterative computation of the constraint graph C. For such nodes v, $pred_C(v) = ahead(in_G(v))$ during the whole computation. To implement this in algorithm *PracticalDomCompWorkList*, it suffices to tag these nodes, and to never add them to *need_to_redo*. As this is trivial, we do not depict the corresponding pseudo-code.

We should note however that it may still be necessary to remove elements from $pred_C(v)$ of such nodes during the final reduction of C on line 23 of algorithm *PracticalDomCompWorkList*. This is necessary, for example, for nodes v whose only incoming edge is a return edge. In such cases, $ahead(in_G(v))$ contains two nodes, of which one can dominate the other. Whether or not this is the case only becomes known at the end of the while-loop. If that is the case, the transitive reduction will remove one of the nodes from $pred_C(v)$.

8. EXPERIMENTAL EVALUATION

To evaluate the constraint-based algorithm, we have implemented it, together with the data-flow algorithm, in Diablo [De Bus et al. 2004], a framework for link-time program rewriting, and applied it on a number of real-life programs covering a broad range in program sizes.

8.1 The Benchmarks

The benchmark programs and their most important properties are presented in Table I. All programs were compiled on a Gentoo Linux x86 system with GCC 3.3.x compilers, and statically linked against the glibc standard system libraries.

With the exception of the gcc benchmarks,[4] the ratio edges/blocks in the programs ICFG is pretty invariant. The most obvious reason for the increased ratio is found in the number of switch edges in the program. As can be seen from Table I, the fraction of edges that represents cases in a switch statement is much higher in the gcc benchmarks than in any other benchmark. As can be expected, this property will heavily influence the computation time.

In order to facilitate the interpretation of the execution time and memory requirements of the algorithms, Tables II and III present six properties of the

[4]The smallest of the two gcc benchmarks is the reduced version included in the SPECint2000 benchmark suite, the other version is distributed on http://gcc.gnu.org.

Table I.
Our set of benchmark programs, with the numbers of nodes and edges in their ICFGs, their ratio, and the fraction of edges that originates from switch statements.

Benchmark	Description	nr of basic Blocks	nr of Edges	Edges/ Block	Switch Edges/ Edges
gzip (spec2000)	compression program	22673	36395	1.61	0.02
vortex (spec2000)	OO database	41739	68926	1.65	0.02
eon (spec2000)	probabilistic ray tracer	55064	91786	1.67	0.01
blackbox	window manager	62687	101025	1.61	0.01
xfreecell	X11 card game	65607	106257	1.62	0.02
gs	PostScript interpreter	75294	121598	1.61	0.03
povray	ray tracer	85120	141814	1.67	0.04
gcc (spec2000)	compiler	108115	192779	1.78	0.07
mplayer	media player	128923	210726	1.63	0.02
postgres	database server	145065	245477	1.69	0.04
gtk-pixbuf-demo	demo GTK-toolkit	176625	292849	1.66	0.03
mysqld	database server	184740	309579	1.68	0.02
gcc 3.3.5	compiler	200666	384673	1.92	0.13
gtk-demo	demo GTK-toolkit	206241	342871	1.66	0.03
vim	editor	206903	350197	1.69	0.04
qt-addressbook	addressbook	259556	435428	1.68	0.03
qt-demo	demo QT-toolkit	328637	555803	1.69	0.02
gimp	image manipulation	370261	627549	1.69	0.02
qt-designer	IDE	442858	756734	1.71	0.02
lyx	WYSIWYG word processor	451102	761147	1.69	0.02
linux 2.4.25	operating system kernel	786218	1294837	1.65	0.03

Table II.
The distribution of the in-degree in the minimal dominator graph. The last column shows the maximum in-degree for each benchmark.

	in-degree											
Benchmarks	1	2	3	4	5	6	7	8	9	10	>10	max
gzip (spec2000)	20470	2165	32	2								4
vortex (spec2000)	36411	5280	41	2	0	1						6
eon (spec2000)	47552	7356	108	18	7	5	5	0	3			9
blackbox	53634	8938	87	10	7	1						6
xfreecell	55862	9593	134	5	3	2	2	1				8
gs	66449	8793	46	2								4
povray	72822	12178	96	6	4	3	2					7
gcc (spec2000)	93429	14386	206	47	14	12	6	7	1	2	1	12
mplayer	114359	14465	80	11	2	1						6
postgres	108917	35730	269	76	38	14	10	4	1	1		10
gtk-pixbuf-demo	143877	32540	164	25	9	1	3	0	0	1	1	11
mysqld	144703	39571	317	79	31	20	8	4	2			9
gcc 3.3.5	166596	33566	321	95	35	13	11	8	9	6	1	11
gtk-demo	165190	40756	202	41	28	7	6	3	1	1	2	16
vim	168318	38325	184	46	17	5	3					7
qt-addressbook	209726	49177	420	120	52	26	18	4	4	1	3	13
qt-demo	259488	68235	565	163	82	37	26	11	8	4	13	18
gimp	279178	90357	435	119	79	38	19	15	3	2	11	32
qt-designer	346058	95391	769	272	155	83	46	24	15	12	27	23
lyx	358268	91483	752	276	125	73	40	25	21	10	21	17
linux 2.4.25	652546	132819	623	159	37	15	4	0	3	1	3	11

minimal dominator graphs of these programs. For each benchmark, Table II presents the histogram of the in-degrees of the nodes in the minimal dominator graph. It can be seen that for larger programs, the in-degrees can become quite large. For gimp, for example, a maximal in-degree of 32 means that one basic block is dominated by at least 32 procedure exit nodes that do not dominate each other. In other words, at least 32 procedures are executed before that basic

Table III.
The average minimum depth and maximum depth of a node in the minimal dominator graph (the minimal and maximal distance to the root), the average number of paths leading from the root to a node in the minimal dominator graph, the average number of dominators per node, and the average number of edges per node in the minimal dominator graph.

Benchmark	Avg Min Depth	Avg Max Depth	Avg nr of Paths	Avg nr of Dominators	Edges in M/Block
gzip (spec2000)	30	56	252	82	1.10
vortex (spec2000)	27	144	807	180	1.13
eon (spec2000)	24	64	1614	95	1.14
blackbox	25	52	287	84	1.26
xfreecell	30	52	739	72	1.15
gs	21	30	76	38	1.12
povray	28	61	556	87	1.15
gcc (spec2000)	27	182	1196	234	1.14
mplayer	35	82	598	103	1.11
postgres	25	48	851	92	1.25
gtk-pixbuf-demo	28	39	166	52	1.19
mysqld	29	59	900	99	1.22
gcc 3.3.5	31	139	11046	229	1.17
gtk-demo	28	38	185	53	1.20
vim	29	40	191	57	1.19
qt-addressbook	29	46	588	75	1.20
qt-demo	28	49	1301	92	1.22
gimp	28	42	270	61	1.25
qt-designer	29	71	5140	160	1.23
lyx	31	89	2626	168	1.21
linux 2.4.25	12	22	59	34	1.17

block is executed, but there is no fixed order in which any pair out of those 32 procedures is executed. Clearly, one expects that higher in-degrees occurring in a graph corresponds to more paths being traversed in the constraint graph C during the intersection computations.

Table III presents five more statistics on the minimal dominator graphs of the benchmark programs. These are the average minimum depth of a node in the graph, the average maximal depth, the average number of paths leading to a node, the average cardinality of the dominator sets of all nodes, and the ratio between the number of edges and the number of nodes in the graph. In each case, one expects that larger numbers indicate that more work will need to be done during the iterative computation of this graph.

We observe that the variation on the average minimal depth is relatively small. This follows from the fact that all programs are linked against the same standard library, and that the first code executed in all programs consists of initialization code of that library. The other numbers in the table depend more on the actual program itself, and for those numbers, huge variations are observed.

It is important to note that the Linux kernel is a special case in this table. We have included this benchmark because it is the largest program we could find that is handled correctly with Diablo [Chanet et al. 2006]; but to make it this large, we had to configure the kernel to include all possible drivers. Most, if not all, of the driver code is called through function-pointers that are stored in

large tables. In Diablo, such calls are modeled by calls to a so-called *unknown node* [Muth et al. 2001], which in turn calls all procedures for which the address is either stored in the statically allocated data or computed in the code. Because, calls to this unknown node happen quite early in the program, most, if not all, drivers are considered reachable very early in the program. As a result, the minimal dominator graph has a very small average depth.

It is difficult to tell how more precise models of indirect control flow, that, for example, use type information, would influence the properties in Table III and the computation of dominators. On the one hand, the average depth of the nodes in the minimal dominator graph would certainly increase, thus slowing down the dominator computation. On the other hand, fewer blocks would be dominated by this one low-numbered unknown node; thus fewer paths to the unknown node would need to be traversed during the computation of intersections in our algorithm.

8.2 Execution Time

Because some of the programs are very large, one cannot expect that all computed data will fit into a processor's cache. For the larger programs, cache behavior will deteriorate and thus increase execution time. It hence does not suffice to look at clock-wall execution times to obtain insights in the practical complexity of our algorithm. For that reason this section presents both execution times and executed instruction counts. The execution times were measured by means of the standard C-library procedure clock(), and the instruction counts were obtained using the performance counters of the processor in our evaluation system.

For these experiments, each algorithm was executed five times on each benchmark, on an otherwise unloaded system comprising a hyper-threaded 3.4GHz Intel Pentium IV processor with a 16KB L1 data cache, a 1MB unified L2 cache, and 4GB of memory connected to a 400MHz front side bus. This system runs Gentoo Linux based on a 2.6.10 Linux kernel. We present the fastest of the five runs. We believe this to be the best solution, because the algorithms are deterministic, and because longer execution times can hence only be caused by interference with other coincidental processes running on the evaluation machine.

The numeric results of our experiments are presented in Tables IV and V, while Figure 22 presents a graphical representation. Note that the scales of the axes of this chart are logarithmic.

Most importantly, the constraint-based algorithm is an order of magnitude faster than the data-flow algorithm. The speedup varies between factors 11.88 and 58.99, averaging at a factor of 23.99. There seems to be no correlation between the obtained speedup and the size of the benchmark programs.

We believe that the execution times of our new algorithm show that interprocedural dominator computation has become practically viable even for programs of up to several hundred thousand basic blocks. Obviously, execution times on the order of tens of seconds are not viable in traditional compilers. In whole-program analyses and optimizations, however, such times are often acceptable.

Table IV.
The number of executed instructions, the running time, and the maximal memory consumption for both the improved algorithm and the data-flow algorithm, as well as their ratios.

	Constraint-Based Algorithm			Data-Flow Algorithm			Data-Flow / Constraint-Based		
Benchmark	Instr. ($x10^9$)	Execution Time (sec)	Memory (MiB)	Instr. ($x10^9$)	Execution Time (sec)	Memory (MiB)	Instr.	Execution Time	Memory
gzip (spec2000)	0.11	0.17	1.60	2.72	2.02	15.37	24.28	11.88	9.63
vortex (spec2000)	0.56	0.63	3.00	14.26	18.87	58.69	25.32	29.95	19.53
eon (spec2000)	0.60	0.75	3.95	16.70	16.00	42.34	27.76	21.33	10.72
blackbox	0.51	0.82	4.41	16.09	16.96	42.46	31.30	20.68	9.63
xfreecell	0.35	0.55	4.63	11.00	8.24	39.20	31.29	14.98	8.48
gs	0.29	0.51	5.24	5.00	6.45	24.94	17.45	12.65	4.76
povray	0.72	1.05	6.05	21.69	20.45	61.54	30.24	19.48	10.17
gcc (spec2000)	1.88	2.39	7.73	87.29	50.21	486.62	46.44	21.01	62.93
mplayer	0.79	1.20	9.00	28.90	29.25	105.36	36.44	24.38	11.71
postgres	2.52	4.19	10.72	47.29	60.63	107.28	18.74	14.47	10.00
gtk-pixbuf-demo	1.38	2.44	12.61	27.05	56.72	76.24	19.55	23.25	6.05
mysqld	2.38	3.78	13.43	83.42	149.17	146.11	35.08	39.46	10.88
gcc 3.3.5	3.72	5.13	14.50	246.01	174.07	504.31	66.22	33.93	34.77
gtk-demo	1.90	3.48	14.82	38.24	89.26	90.78	20.15	25.65	6.13
vim	1.78	2.85	15.22	31.94	48.62	97.42	17.91	17.06	6.40
qt-addressbook	3.43	5.66	18.63	81.63	156.36	158.18	23.79	27.63	8.49
qt-demo	6.16	12.14	23.75	125.84	261.28	248.30	20.42	21.52	10.45
gimp	5.82	10.83	26.90	104.96	309.45	188.81	18.04	28.57	7.02
qt-designer	18.56	42.53	32.91	277.03	575.31	554.93	14.92	13.53	16.86
lyx	15.52	24.06	33.71	282.90	563.36	595.54	18.23	23.41	17.67
linux 2.4.25	6.98	8.86	54.39	74.76	522.63	229.64	10.72	58.99	4.22

Table V.
The number of cache misses per executed instruction and the execution time per instruction, for both the improved algorithm and the data flow algorithm.

	Constraint-Based		Data-Flow	
Benchmark	L2 cache misses/100 ins	sec/Gins	L2 cache misses/100 ins	sec/Gins
gzip (spec2000)	1.21	1.52	0.21	0.74
vortex (spec2000)	0.68	1.12	0.74	1.32
eon (spec2000)	0.81	1.25	0.49	0.96
blackbox	1.33	1.60	0.58	1.05
xfreecell	1.40	1.56	0.30	0.75
gs	1.89	1.78	0.78	1.29
povray	1.02	1.46	0.50	0.94
gcc (spec2000)	0.64	1.27	0.21	0.58
mplayer	1.24	1.51	0.52	1.01
postgres	1.27	1.66	0.88	1.28
gtk-pixbuf-demo	1.58	1.76	1.59	2.10
mysqld	1.23	1.59	1.40	1.79
gcc 3.3.5	0.85	1.38	0.33	0.71
gtk-demo	1.67	1.83	1.81	2.33
vim	1.35	1.60	1.07	1.52
qt-addressbook	1.29	1.65	1.49	1.92
qt-demo	1.71	1.97	1.66	2.08
gimp	1.70	1.86	2.49	2.95
qt-designer	2.09	2.29	1.71	2.08
lyx	1.25	1.55	1.58	1.99
linux 2.4.25	1.30	1.27	6.94	6.99

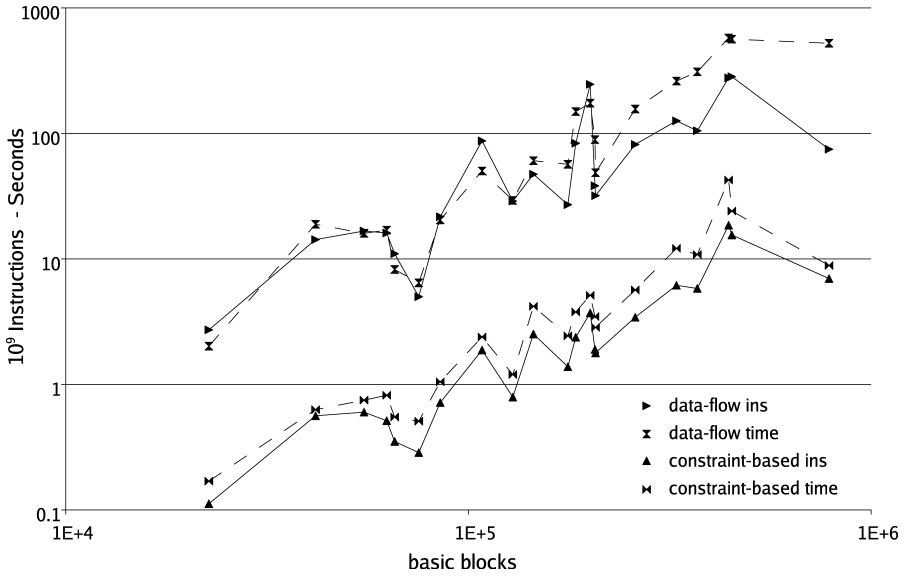


Fig. 22. The number of executed instructions and the running time for the data-flow algorithm and the improved algorithm.

When we look at the number of executed instructions, the constraint-based algorithm is between 10.72 times and 66.22 times lower than the data-flow algorithm. On average, the data-flow algorithm executes 26.39 times more instructions than the constrained-based algorithm. In the case of executed instructions, the ratio does tend to become smaller with increasing program size. In terms of executed instructions, the data-flow algorithm in practice hence seems to scale somewhat better than our constraint-based algorithm.

However, the data-flow algorithm requires much more memory (see Section 8.4). Consequently it suffers more from deteriorating cache behavior when the programs become larger. This can be seen in the widening gap between the number of executed instructions and the execution time of the data-flow algorithm in Figure 22. No such widening is apparent for the constraint-based algorithm, as can be seen by the numbers in Table V. First, these numbers show the strong relationship between the number of cache misses and the part of the execution time that is not accounted for by just the number of executed instructions. Furthermore, these numbers show that, whereas the number of cache misses clearly increases with the program size for the data-flow algorithm, this is not the case for the constraint-based algorithm. We can conclude that in terms of execution time our constraint-based algorithm scales at least as well as the data-flow algorithm. To a large extent, this is due to the that fact the cache behavior scales better.

Besides being an order of magnitude faster, the execution time of the constraint-based algorithm also seems more predictable, as the peaks and lows in its timing results are less pronounced.

With respect to those peaks and lows in the execution times, it is clear that our algorithm is sensitive to the properties of paths in the minimal dominator

graph. As we explained in Section 8.1, the computation of the intersections of ancestor sets on average becomes more expensive in our algorithm, with growing values of the statistics presented in Table III. Benchmarks such as gs, gtk-pixbuf-demo, gtk-demo, vim, and Linux, which constitute the lows in the execution time charts, have the flattest minimal dominator graphs, whereas gcc, qt-designer, and lyx, which all have deeper graphs or relatively more paths, constitute the main peaks.

Most often, there is a close correlation between these properties and the average size of dominator sets. As a result, the peaks and lows in the execution times of both algorithms are highly correlated as well. That said, it does seem that the constraint-based algorithm is more sensitive to the number of paths in the minimal dominator graph. For example, qt-designer and lyx are comparable in size, and the statistical properties of their minimal dominator graphs are very similar as well, with the exception of the average number of paths from the root node of the minimal dominator graph to other nodes. For qt-designer, it is 5140, which is almost twice as high as that of lyx, which is 2626. For the data-flow algorithm, this difference does not result in different execution times. For the constraint-based algorithm, it does make a big difference, however, as the algorithm requires almost two times more computation time for qt-designer. Most of this additional computation time is not due to an increase in executed instructions however. Instead, it results from deteriorated cache behavior, as can be seen in Table V. Unfortunately, we cannot conclude from this that there is a direct link between cache behavior and dominator graph properties. Like qt-designer and lyx, the two gcc versions also have a high number of paths in the minimal dominator graph, but for both gcc versions, the number of cache misses is very low. In summary, we can assume that there exists a relationship between minimal dominator graph properties and execution time, but we cannot be conclusive on the exact nature of this relationship.

8.3 Optimizations

To assess the contribution of the different optimizations discussed in Section 7 to the overall speedup obtained with the constraint-based algorithm, we have measured the incremental speedups obtained by enabling the optimizations one after the other. The results of these experiments are shown in Figure 23. Each single block in the bars indicates the difference in speedup obtained (compared to the data-flow algorithm) with and without the additional optimization enabled.

The large white blocks indicate that the base version of our constraint-based algorithm already improves the execution significantly. On average, the base constraint-based algorithm is already 7 times faster than the data-flow version.

When the intersection computation is first optimized by incrementally marking nodes in the dominator constraint graph, as discussed in Section 7.1, the speedup on average increases to over 9. On average, this optimization is therefore rather marginal. But on some programs, such as vortex and mplayer, it does contribute significantly to the overal speedup obtained with the fully optimized algorithm.

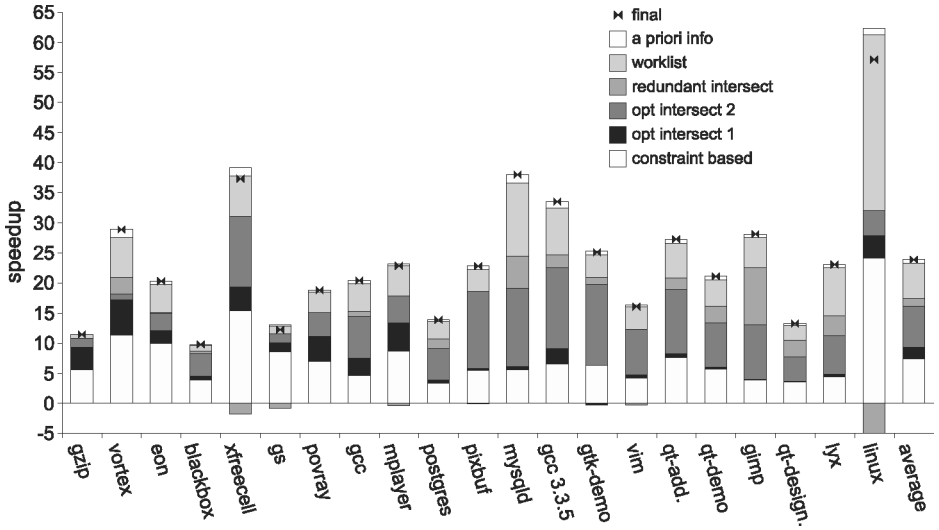


Fig. 23. The speedup in execution time for the optimizations, applied incrementally in the order as layed out in Section 7. The speedup is expressed in factors of the execution time of the data-flow algorithm. Negative numbers indicate that the execution time increased when the optimization was added. The final speedup is shown using a bullet.

The second intersection optimization, based on the notion of leaf procedures (see Section 7.2) is more successful than the first. In fact, this optimization is the major contributor to the maximal speedup obtained over the data-flow algorithm. With this optimization enabled, the average speedup becomes 16.2.

Avoiding intersection operations altogether, as explained in Section 7.3, provides a minor contribution to the final speedup. In some cases it even slows down the algorithm, as is the case for the Linux kernel. For other programs, such as gimp, however, this optimization is a large contributor to the total speedup obtained. As indicated in Section 7.3, this optimization is applied on nodes with a lot of incoming edges in the ICFG. To determine a threshold, we performed a number of experiments, ranging the number of incoming edges threshold from 3 to 1000. A threshold of 200 proved to give the best results on average. Unfortunately, we found no threshold at which the computation for all programs was improved while still obtaining a significant average speedup. While we believe that our simple heuristic based on a number of incoming edges threshold can probably be improved by making it depend on other properties of the ICFG in combination with the CSDFT numbering, we have not yet found such heuristics.

The second most successful optimization to our constraint-based algorithm is the move to a work-list (see Section 7.4). This optimization gives an additional speedup of 5.85 on average. It is by far the most successful optimization for Linux.

Finally, the last optimization for nodes with only one predecessor in the ICFG (Section 7.5) speeds up the algorithm by only a minimal factor. It is the only optimization that is not a major contributor in any of the benchmarks. All other

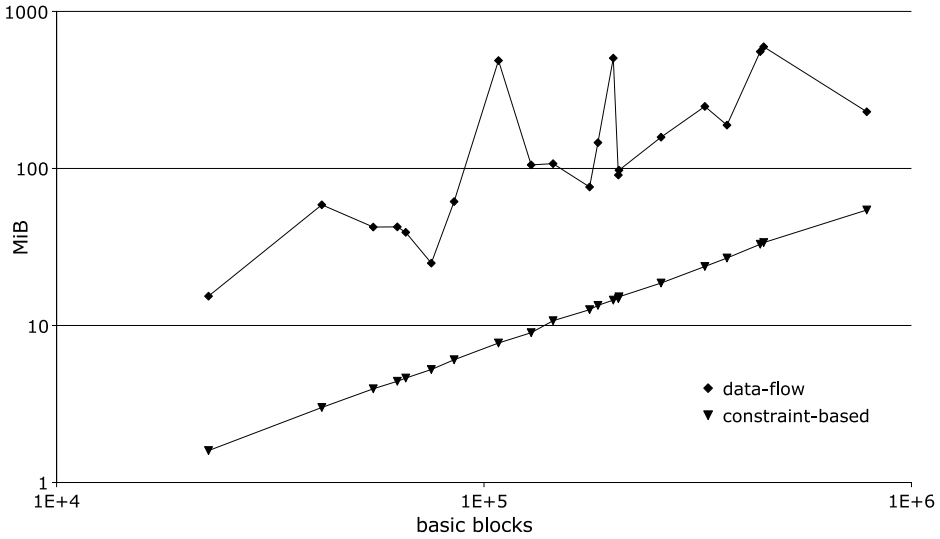


Fig. 24. The maximum memory usage (in megabytes) for the data-flow algorithm and the new algorithm.

optimizations play an important role in speeding up the computation for at least one benchmark.

8.4 Memory Consumption

Besides the execution times, we have also measured the required amount of heap memory of both the data-flow and the optimized algorithm. This was done by instrumenting the standard C-libray malloc(), calloc(), free(), and realloc() routines. The results of this experiment are shown in Figure 24 on a chart of which the axes have logarithmic scales.

First, we should note that the memory requirements of the data-flow algorithm are highly correlated to its execution times, as can be seen by comparing the peaks and lows of the curves in Figures 22 and 24. This should not come as a surprise, since almost all execution time in the data-flow algorithm is spent walking and copying the memory allocated to store the dominator sets.

This situation is completely different with our constraint-based algorithm. For this algorithm, the memory complexity is clearly linear in practice. This corresponds with the in-degree histograms being very skewed towards 1 and 2, and the fact that the ratio between the number of edges in a program's minimal dominator graph and the number of basic blocks in the program itself varies very little. Indeed, the latter ratio only ranges from 1.10 to 1.26 (see Table III for our entire benchmark suite. So far, we have found no theoretical arguments based on, for example, software complexity measures to explain this behavior.

All in all, our constraint-based algorithm on average requires 13.64 times less memory than the data-flow solution, ranging from 4.22 times to 62.93 times less.

9. CONCLUSION

In this article, we have shown that the interprocedural dominance relation has other properties than the traditional, intraprocedural dominance relation. As a consequence, existing work for the intraprocedural case cannot be directly extended for the interprocedural case.

We have presented a new, constraint-based algorithm for the computation of interprocedural dominators. This practical algorithm achieves its low computation time because it operates on efficient data structures that exploit a number of properties of a preorder context-sensitive depth-first basic blocks ordering.

Most importantly, the presented algorithm is an order of magnitude faster than the iterative data-flow solution. Even though the theoretical time complexity of the constraint-based algorithm is not better than that of the data-flow solution, the observed execution times for real-life programs of up to several hundred thousand basic blocks show that the computation of interprocedural dominators has become practically viable. In practice, the memory consumption of the presented algorithm is linear. On average, it is an order of magnitude smaller than the memory consumption of a straightforward iterative data-flow solution.

REFERENCES

AGRAWAL, H. 1999. Efficient coverage testing using global dominator graphs. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*. 11–20.

AHO, A. V. AND ULLMAN, J. D. 1977. *Principles of Compiler Design*. Addison-Wesley, Reading, MA.

ALLEN, F. E. 1970. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*. ACM Press, New York, 1–19.

ALLEN, F. E. AND COCKE, J. 1972. Graph-theoretic constructs for program control flow analysis. Tech. Rep. RC 3923, IBM T.J. Watson Research Center.

ALSTRUP, S., HAREL, D., LAURIDSEN, P. W., AND THORUP, M. 1999. Dominators in linear time. *SIAM J. Comput. 28*, 6, 2117–2132.

CHANET, D., DE SUTTER, B., DE BUS, B., VAN PUT, L., AND DE BOSSCHERE, K. 2007. Automated reduction of the memory footprint of the linux kernel. *ACM Trans. Embedded Comp. Syst. 6*, 1 (Feb.).

COOPER, K. D., HARVEY, T. J., AND KENNEDY, K. 2001. A simple, fast dominance algorithm. Available on-line at: http://www.hipersoft.rice.edu/grads/publications/dom14.pdf.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst. 13*, 4, 451–490.

DE BUS, B., DE SUTTER, B., VAN PUT, L., CHANET, D., AND DE BOSSCHERE, K. 2004. Link-time optimization of ARM binaries. *SIGPLAN Notices 39*, 7, 211–220.

DE SUTTER, B., DE BUS, B., AND DE BOSSCHERE, K. 2005. Link-time binary rewriting techniques for program compaction. *ACM Trans. Prog. Lang. Syst. 27*, 5 (Sept.), 882–945.

GEORGIADIS, L. AND TARJAN, R. E. 2004. Finding dominators revisited: extended abstract. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete Algorithms*. 869–878.

GEORGIADIS, L., WERNECK, R., TARJAN, R., TRIANTAFYLLIS, S., AND AUGUST, D. 2004. Finding dominators in practice. *Lecture Notes in Computer Science 3221*, 677–688.

HAREL, D. 1985. A linear algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*. ACM Press, 185–194.

LENGAUER, T. AND TARJAN, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1, 121–141.

LOWRY, E. S. AND MEDLOCK, C. W. 1969. Object code optimization. *Comm. ACM 12*, 1, 13–22.

MUTH, R., DEBRAY, S. K., WATTERSON, S. A., AND DE BOSSCHERE, K. 2001. alto: a link-time optimizer for the compaq alpha. *Software—Practice and Experience 31*, 1, 67–101.

PROSSER, R. T. 1959. Applications of Boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*. Spartan Books, New York, 133–138.

PURDOM, P. W. AND MOORE, E. F. 1972. Immediate predominators in a directed graph [H]. *Comm. ACM 15*, 8, 777–778.

RAMALINGAM, G. 2002. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst. 24*, 5, 455–490.

TRIANTAFYLLIS, S., BRIDGES, M., RAMAN, E., OTTONI, G., AND AUGUST, D. 2006. A framework for unrestricted whole-program optimization. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 61–71.

WALL, D. W. 1986. Global register allocation at link time. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN Symposium on Compiler construction*. ACM Press, New York, NY, 264–275.

Received April 2005; revised July 2006; accepted August 2006