Abstract—We present a new algorithm for computing interprocedural dominators. The algorithm identifies a set of special nodes, which are the only ones that can have interprocedural dominance edges, and extends the intraprocedural dominator trees by deriving those edges. The computation of the dominators of each node is independent of the computation of any other node, and therefore can be done on demand for each node as required. For the same reason, the algorithm lends itself naturally to parallelization. The algorithm has been implemented, and is shown to be practical for large programs. Because of its cooperative caching behavior, the algorithm gains a large performance boost when running on parallel hardware. We also present an efficient way of extending the algorithm for computing interprocedural dominance frontiers and control dependence.

I. INTRODUCTION

A crucial issue in program analysis is what control paths the program can take. A fundamental concept underlying this issue is that of dominance. In a single function, a node \( u \) in the control-flow graph (CFG) is said to dominate another node \( v \) if every path from the entry of the function to \( v \) must first pass through \( u \). The dominance relation and its derivatives, such as the control-dependence relation, are used in compiler optimization [1] as well as applications such as program slicing [2]. The computation of the popular static single assignment (SSA) form requires the notion of dominance, and the efficient algorithm for computing it [3] introduced the notion of dominance frontiers, which are also useful for computing control dependence.

The above definition of dominance was stated for the intraprocedural case, and most applications are also intraprocedural. There have been various attempts to generalize such intraprocedural applications to the interprocedural case. Examples are program testing [4], optimization [5], and code analyses such as program slicing [2]. In each case, ad-hoc techniques had to be developed to support the interprocedural case. This is due, in part, to the lack of an efficient algorithm for computing interprocedural dominance. The definition of dominance can be extended to the interprocedural case, but it needs to consider only paths that preserve the semantics of function calls, which cannot be nested improperly. (The following description is mostly based on Sharir and Pnueli’s classic paper [6].) If a function \( f \) calls (directly or indirectly) a function \( g \), then \( f \) cannot return before \( g \) has returned. Function calls and returns can therefore be paired on any valid path,\(^\dagger\) so that they nest properly. In addition, any valid path from the initial

\(^\dagger\)In the sequel, anywhere we use the term “path” we refer to an interprocedural path, unless stated otherwise.
node may have unmatched calls (allowing, for example, for program termination by methods such as System.exit()), but cannot have unmatched returns. A balanced path is defined to be a valid path that has no unmatched calls or returns. Interprocedural dominance is defined with respect to valid paths only. For example, in the program of Figure 1 there is an invalid path that passes through the call to $f_1$ in main (line 6) and returns to $f_2$ after the call to $f_1$ on line 14. It this path is considered in the dominance computation, it would prevent the discovery of correct relationships; for example, the fact that that the conditional on line 5 dominates the entry of $f_2$.

In the intraprocedural case, each node $v$ in the CFG except the first has a unique immediate dominator, which is a strict dominator of $v$ and is dominated by every other strict dominator of $v$. (A strict dominator of $v$ is a dominator $u \neq v$ of $v$.) The simplest representation for the intraprocedural domination relation [7] is therefore a tree in which the parent of each node $v$ is its immediate dominator, denoted $\text{idom}(v)$. However, in the interprocedural case there is no unique immediate dominator [8]. For example, in the program of Figure 1, the if $(c)$ conditional inside $f_2$ (line 13) is clearly an (intraprocedural) immediate dominator of the exit of $f_2$. The fact that the exit of function $f_3$ is also an immediate dominator of the exit of $f_2$ is less obvious. This is due to the fact that it is impossible to exit $f_2$ without going through $f_3$; if $f_2$ does not call $f_3$ directly, it calls $f_1$, and it is impossible to exit this call without going through $f_3$, either from the call in $f_2$ or the one in $f_1$. Neither if $(c)$ nor the exit of $f_3$ dominates the other; it is possible to go through the conditional without calling $f_3$, and it is possible to call $f_3$ without calling $f_2$ at all.

This example shows that the representation of the dominance relation in the interprocedural case needs to be a graph. We generalize $\text{idom}(v)$ to be the set of all nodes that strictly dominate $v$ but do not strictly dominate any other strict dominator of $v$. The graph consisting of all edges $(u,v)$ where $u \in \text{idom}(v)$ is a DAG with the same property enjoyed by the intraprocedural dominator tree; each path in the graph represents a dominance relationship. The algorithm presented in this paper computes the interprocedural dominance graph, an over-approximation to this graph; it preserves the same property, but may have some additional edges that summarize some interprocedural edges already in the graph. These extra edges can easily be removed by a post-processing stage at the end of the algorithm.

The algorithm starts with a forest consisting of the intraprocedural dominator trees for each function. It then adds interprocedural dominance edges for functions having a single call site, and for all returns. The main part of the algorithm traces all dominators for a special set of nodes, called interprocedural dominance candidates, for which new interprocedural dominators could be discovered. It does this by following previously-discovered immediate dominators; but when it reaches an interprocedural dominance candidate, it follows its predecessors in the control-flow graph in order to find dominators that have not yet been discovered. A set of blocking nodes is associated with each such search in order to prevent the algorithm from following circular paths.

The main part of the algorithm iterates over all interprocedural dominance candidates to compute the dominator set of each. The computations of these dominator sets are independent of each other, and can be carried out in any order, or in parallel. The only interaction between these computations is an optimization where the results of one computation can be used to save part of the computation for another; however, the algorithm is correct regardless of how many of these optimizations are performed. In fact, these optimizations turn out to be particularly useful when the computations for different interprocedural dominance candidates are intertwined; this can be simulated in sequential code, but gives an immediate boost the more parallel threads are available.

Furthermore, the function that computes dominators for an interprocedural dominance candidate can be called for any node (and is called recursively in this way). It can therefore be used to compute the dominators of any node on demand, and the results can be recorded in order to speed up later dominance computations.

The contributions of this paper include (1) a provably-correct efficient parallel algorithm for computing interprocedural dominance; (2) an empirical evaluation showing significant speedup with more hardware threads; (3) an efficient way of computing interprocedural dominance frontiers and control dependence.

### II. RELATED WORK

A straightforward way to compute all the nodes dominated by some node $u$ is to compute reachability of nodes in the graph after removing $u$ from it; those that were originally reachable but are no longer reachable are exactly those dominated by $u$ [9]. This simple algorithm always achieves its asymptotic complexity, which is $O(mn)$ for a graph with $n$ nodes and $m$ edges. It is also applicable to the interprocedural case, provided reachability is computed for valid interprocedural paths. Lengauer and Tarjan [10] provide a widely used algorithm for the intraprocedural case in time almost linear in $n$. This was followed by several linear-time algorithms. Georgiadis et al. [11] compared the leading algorithms and found them to have similar performance, with the best being the Lengauer-Tarjan algorithm and a hybrid presented in that paper.

De Sutter et al. [8] present several versions of an interprocedural dominance algorithm, each more efficient but more complicated than its predecessor, and compare their performance. They report "practically viable" computation times "even for programs of up to several hundred thousand basic blocks." The authors claim that "the Lengauer-Tarjan dominator computation algorithm...cannot be extended to compute interprocedural dominators." Their algorithm therefore computes all dominance relationships from scratch. Because this algorithm is based on abstract interpretation with chaotic iteration, it is inherently sequential and needs to compute the full dominance relation even when information is needed only for a small set of nodes. In contrast, our algorithm starts with the intraprocedural dominator trees computed by Lengauer-Tarjan or any other interprocedural algorithm, and extends these to the full interprocedural relation. Because it works independently for each interprocedural dominance candidate, it can be called on-demand instead of having to compute the
full relation all at once, and it can also be run in parallel. In fact, as demonstrated in Section VI, parallelization provides a significant performance boost to our algorithm.

## III. Computing Dominance

The input to our algorithm is an interprocedural control-flow graph (ICFG) similar to the one used in the call-string approach of Sharir and Pnueli [6] and that of De Sutter et al. [8]. This graph consists of the standard control-flow graphs of each procedure, where each procedure has a unique entry node and a unique exit node, and each call site is represented by two nodes, a call node and a resume node. The call node is connected by a call edge to the entry of the called procedure, and the resume node has an incoming return edge from the exit of the called procedure. In addition, the call edge of each call site is connected by a regular intraprocedural edge to the resume node of the same site. This implies that entry nodes may have multiple incoming call edges, and exit nodes may have multiple outgoing return edges. We assume that there is a unique main procedure, which is the entry point to the whole program. The ICFG representing the program of Figure 1 appears in Figure 2.

The algorithm computes the interprocedural dominance graph (IDG) [8] to represent the interprocedural dominance relation. The IDG is a natural extension of the dominator tree; in fact, it contains the dominator trees for the procedures in the program as subgraphs, with one important exception: a call node may not have an edge connecting it to the following resume node, if the latter is not reachable from the former; this can happen when the called function never returns.

The top-level algorithm appears as Algorithm 1; it builds the IDG incrementally from the intraprocedural dominator trees. At all times during the computation, the IDG represents correct domination relationships; but it will only be complete at the end. The initial IDG consists of the intraprocedural dominator trees without unreachable call-to-resume edges (step 1). The algorithm then adds edges that require only local analysis of the ICFG. If a function has a single call site, that site dominates the function entry (step 2); and every function exit node dominates all the resume nodes it can return to (step 3). These two steps consist of copying edges from the ICFG to the IDG. Finally, the algorithm adds derived edges that require global analysis (step 4). In principle, this step could be done for every node in the graph, but there are only certain nodes whose immediate-dominator set can be extended in this step; we call these interprocedural dominance candidates. This definition uses the concept of (intraprocedural) dominance frontiers, defined by Cytron et al. [3].

**Definition 1:** A node \( v \) is an interprocedural dominance candidate if: (1) \( v \) is a function entry with more then one

---

```
Algorithm 1 The interprocedural dominator algorithm. Input: an ICFG on a set \( V \) of nodes; output: the corresponding IDG.

1) [Add intraprocedural edges] Compute the intraprocedural dominator trees; using context-sensitive DFS [8] remove all call-to-resume edges where the resume node is unreachable from the call node; combine the trees to form the initial IDG.
2) [Add call edges] For every function that has a single call site, add to the IDG an edge connecting the call node to the function entry node.
3) [Add return edges] Add to the IDG edges connecting each function exit node to every resume node it can return to.
4) [Add derived edges] For every interprocedural dominance candidate \( v \in IDC \):
   a) Let \( S = \text{dominators}(v, \emptyset) \setminus \{v\} \).
   b) Set \( \text{idom}(v) \) to be the set of all elements \( u \in S \) such that \( u \) does not have an IDG-successor \( u' \in S \).
   c) Add to the IDG an edge \( (u, v) \) for each \( u \in \text{idom}(v) \).
```

---

```
Algorithm 2 Compute the dominator set for a node \( v \) given a blocker set \( B \).

\text{dominators}(v, B):

1: if \( v \in IDC \) then
2: if a previous call \( \text{dominators}(v, B') \) where \( B' \subseteq B \) returned \( R \) then
3: return \( R \)
4: end if
5: \( R \leftarrow V \)
6: Let \( B' = B \cup \{v\} \)
7: for every ICFG-predecessor \( r \) of \( v \) such that there is no path in the IDG from an element of \( B' \) to \( r \) do
8: Let \( S = \text{dominators}(r, B') \)
9: \( R \leftarrow R \cap S \)
10: end for
11: return \( R \cup \{v\} \)
12: else
13: \( P \leftarrow \{v\} \)
14: for every IDG-predecessor \( d \) of \( v \) do
15: \( P \leftarrow P \cup \text{dominators}(d, B) \)
16: end for
17: return \( P \)
18: end if
```

---

The interprocedural control-flow graph for the example of Figure 1. The solid arrows denote intraprocedural data flow, and dashed arrows denote call and return edges.
predecessor in the ICFG; (2) \( v \) is in the intraprocedural dominance frontier of a resume node; or (3) \( v \) is in the intraprocedural dominance frontier of another intraprocedural dominance candidate.

Dominance frontiers can be computed efficiently [3]; once these have been computed, Definition 1 yields a linear-time algorithm for computing the set of all intraprocedural dominance candidates, which we denote by IDC.

The dominators function used in step 4a of Algorithm 1 appears as Algorithm 2. It computes interprocedural dominators for its input \( v \), which is usually (but not necessarily) an interprocedural dominance candidate. It follows existing dominance edges in the IDG, but treats interprocedural dominance candidates in a special way. Whenever it reaches one of those, the algorithm computes the intersection of the dominators of its predecessors in the ICFG. This computation must ignore cycles in the ICFG, and so it uses a blocking set \( B \) of nodes that have already been visited during this search. Line 2 of the algorithm is an optimization that prevents repeated computations based on a cache of previous calls to the dominators function. Without this optimization the algorithm is correct but much less efficient.

Up to this point, we have assumed that each call site uniquely identifies the called function. This is not the case in object-oriented languages, in which the code executed for a specific method call depends on the type of the target of the invocation. Using static analysis (including pointer analysis) it is possible in many cases to narrow the possible implementations that can be called. Because our implementation (see Section VI) is based on the WALA analysis infrastructure, which performs this analysis, it is able to make use of this information to provide better accuracy in the computation of interprocedural dominance.

We extended the algorithm to support polymorphism by representing each polymorphic call in the ICFG as a series of tests on the dynamic type of the target, using all possible implementing classes as determined by the best approximation available. The new nodes added for this representation do not appear in the original ICFG, and can be removed by a later post-processing step if required. The same technique can be used to handle function pointers in languages that offer them.

We have proved that the IDG computed by the algorithm correctly represents the interprocedural dominance relationships in the program. The proof appears in the appendix.

IV. EXAMPLE

For the example program of Figure 1, the ICFG appears in Figure 2, and the final IDG appears at the bottom of Figure 1, which shows interprocedural dominance candidates with thick borders. These consist of the entries of \( f1 \) and \( f3 \), each of which has multiple call sites, the exit of \( f2 \), which is in the dominance frontier of both resume nodes inside \( f2 \), and the call to \( f4 \) inside \( f1 \), which is in the dominance frontier of three resume nodes inside \( f1 \). IDG edges that are part of the intraprocedural dominator trees are shown in Figure 1 using full thin arrows; these are added in step 1 of Algorithm 1. Call and return edges added in steps 2 and 3 are dashed. Finally, the derived edges added in step 4 are shown as thick full arrows, and are numbered according to the order in which they are added to the IDG. Edge number 1 was added during the processing of IDC node \( f1\text{-entry} \); edge number 2 was added while processing \( f3\text{-entry} \); edge number 3 was due to \( f2\text{-exit} \); and the two edges marked 4 were added due to \( f4\text{-call} \).

Figure 3 shows three steps in the computation of the IDG in the example program of Figure 1. Nodes visited by the dominators function are shown shaded; the background of blockers is hatched; and X marks designate ICFG-predecessors that were skipped in line 7 of Algorithm 2 because they were reachable from blockers in the IDG.

\footnote{See http://wala.sourceforge.net.}
Consider first the treatment of f1-entry (top of Figure 3). It has three ICFG predecessors, f1-call1, f1-call2, and f1-call3. The last two are already reachable from f1-entry in the IDG, and so the result only contains f1-call1 and its single IDG-ancestor, namely, main-entry, which are the two elements of the set S in step 4a of Algorithm 1. Because f1-call1 is reachable from main-entry in the IDG, only the former is in idom(f1-entry), and edge number 1 is added to the IDG.

We now skip to the computation of the dominators of f2-exit (middle of Figure 3). It has two ICFG predecessors, f3-resume2 and f1-resume3. Suppose that the first recursive call is dominators(f3-resume2, {f2-exit}). The nodes visited during this call are shown with their right halves shaded. This call follows IDG edges backward to f3-call2, if (c), f2-entry, f2-call, if (b), if (a), and f1-entry. This is an ICFG node for which idom has already been computed, and so the algorithm will add f1-call1 and main-entry to the result of the first recursive call. From f3-resume2, the algorithm also follows the interprocedural IDG edge to f3-exit, then to f3-entry and, following edge number 2, which was added during the handling of f3-entry (not shown in the detailed example), to if (a), which has already been visited.

The second recursive call is dominators(f1-resume3, {f2-exit}). The nodes visited during this call are shown with their left halves shaded. This call follows IDG edges to f1-call3, and continues in the same way as the previous call to if (c) until it reaches main-entry; here, again, we employ the optimization of using the previously-computed IDG edges for f1-entry. From f1-resume3, the algorithm also follows the interprocedural IDG edge to f1-exit, then to f4-resume, and f4-call. This requires a recursive call dominators(f4-call, {f2-exit, f1-resume3}). This call is shown at the bottom of Figure 3. The ICFG predecessors of f4-call are f1-resume2, f2-resume, and f3-resume1. The first is reachable in the IDG from the blocker f4-call, and the second from f2-exit. Therefore the algorithm follows IDG edges from f3-resume1 to f3-exit, f3-entry and if (a), from which it continues as before; from f3-resume1 it also follows to f3-call1 and if (b), continuing as before. All these are added to the result of the second recursive call. The four new nodes added by this recursive call are shown with lighter shading on their left sides.

The minimal nodes in the intersection of the two calls are if (c) and f3-exit. The IDG already contains an edge from the former to f2-exit, but edge number 3 is added to the IDG as a result of this call.

V. COMPUTING INTERPROCEDURAL DOMINANCE FRONTIERS

Using the interprocedural dominators graph, it is possible to compute interprocedural dominance frontiers, in the spirit of Cytron et al. [3]. The definition of interprocedural dominance frontiers immediately extends to the interprocedural case. The algorithm to compute interprocedural dominance frontiers is similar to the intraprocedural algorithm [3]. The intra-procedural algorithm is given in an optimized form that uses immediate dominators instead of the dominance relation. The basic (unoptimized) algorithm is easily generalized to interprocedural dominance frontiers, but the corresponding generalization of the optimization fails in the interprocedural case, unless some redundant edges are added to the IDG. These are interprocedural edges that summarize paths in the IDG, and are likely to be beneficial in any dominance search on the IDG.

The algorithm computes dominance frontiers bottom up for nodes in the IDG, so that the frontiers for all successors of a node are known when frontiers for that node are computed. The frontiers for a node x come from two sources: the first contains immediate ICFG-predecessors of x that are not dominated by x; the second contains nodes that are further away in the ICFG, and whose computation requires propagation on the ICFG.

The optimized algorithm replaces domination checks in the basic algorithm by checking whether the candidate frontier node is in idom(x). This is always correct for frontiers of the first kind, but requires the additional IDG edges in the second case.

We denote the set of all interprocedural dominance frontiers of a node x by DF(x). This set is composed of the two subsets mentioned above, DF_{local}(x) and DF_{up}(x).

Definition 2: The local dominance frontier of x, denoted DF_{local}(x), contains all ICFG-predecessors y of x such that y does not dominate x. The derived dominance frontier of x, denoted DF_{up}(x), contains all nodes y that are in DF(z) for some IDG-successor z of x such that x does not dominate y.

It can be proven that for each ICFG node x, DF(x) = DF_{local}(x) ∪ DF_{up}(x). This shows that the algorithm that computes dominance frontiers bottom up on the IDG using the sets DF_{local} and DF_{up} is correct.

The optimized algorithm requires an extended form of the IDG, which adds some interprocedural edges. This form, which will be denoted EIDG, is computed by Algorithm 1, except that in steps 4b and 4c the set idom(v) is replaced by a larger set I(v), computed as the set of all elements u ∈ S such that u does not have an IDG-successor u′ ∈ S such that u is not an interprocedural dominator of u′. The emphasized addition will allow I(v) to contain interprocedural edges that are not in idom(v), but not any edges that can be inferred from these using interprocedural reasoning. For example, consider the following program:

```c
1 void g() {
2   if (a) { g1(); g2(); g3(); }
3   else { g1(); g3(); }
4   s;
5 }
```

In this case, the exit of g1 dominates s; however, the exit of g1 is not in idom(s), since that edge can be inferred from the fact that the exit of g1 dominates the exit of g3, which in turn dominates s. This edge will be added to the extended IDG. However, edges from internal nodes of g1 to s will not be added, since these can be inferred intraprocedurally. This change makes EIDG larger than the original IDG, but could speed up both the computation of domination relations, as well
as the computation of dominance frontiers. We believe that the effects of this trade-off will be beneficial in most cases.

The optimized algorithm uses the following modified definitions of the auxiliary sets, which refer to EIDG edges rather than full dominances:

**Definition 3:** The optimized local dominance frontier of \( x \), denoted \( \text{DF}_{\text{opt}}(x) \), contains all ICFG-predecessors \( y \) of \( x \) such that \( y \not\in I(x) \). The optimized derived dominance frontier \( x \), denoted \( \text{DF}_{\text{opt}}(x) \), contains all nodes \( y \) that are in \( \text{DF}(z) \) for some EIDG-successor \( z \) of \( x \) such that \( y \) is not an EIDG-predecessor of \( x \).

To understand why the additional edges are necessary, consider again the example of function \( g \) above. The statement \( s \) is in the \( \text{DF}_{\text{local}} \) set of the call to \( g3 \) on line 3, because \( s \) is an CFG-successor of the call, but the call does not dominate it. The set \( \text{DF}_{\text{opt}} \) of the call to \( g3 \) is empty, because it has no IDG-successors. Therefore, its dominance frontier is just \( s \). Moving now to the exit of \( g1 \), which is one of the IDG-predecessors of the call to \( g3 \), we find that its \( \text{DF}_{\text{opt}} \) set does not contain \( s \), because it dominates \( s \). However, as explained above, this fact is not expressed as an IDG edge. The optimized algorithm will therefore consider \( s \) to be in the dominance frontier of the exit of \( g1 \), unless this edge is added to EIDG. It can be shown that \( \text{DF}(x) = \text{DF}_{\text{local}}(x) \cup \text{DF}_{\text{opt}}(x) \).

As in the intraprocedural case, dominance frontiers can be used to efficiently compute both control dependence, as well as the correct placement of \( \phi \) functions for the interprocedural SSA form.

### VI. Evaluation

We tested the algorithm on several Java projects. Our major findings are:

- The order in which calls are made to the dominators function, and the resulting behavior of the cache, have a profound effect on overall performance.
- A sequential depth-first strategy, even with some heuristic ordering of nodes, gives the worst results.
- Running multiple threads with yield points dramatically improves performance, even on a single CPU.
- Additional hardware threads provide a significant boost in performance.

Table I summarizes the applications on which we tested our algorithm; several come from the same project. All are open-source projects. RemoteSearchable, PorterStemmer, and IndexReader are part of Lucene (http://lucene.apache.org/core), a text search engine library; HSQLDB (http://hsqldb.org) and Derby (http://db.apache.org/derby) are relational database engines; Cassandra (http://cassandra.apache.org) is a no-SQL distributed DBMS; and dump, decorate, and optimize are part of Bloat (http://sourceforge.net/projects/javabloat), a Java bytecode optimizer and class rewriter.

Because the analysis includes libraries in jar files, we have no measure of the lines of codes; the more relevant statistic to the computation of dominators is in any case the number of basic blocks. In addition, the table lists the total number of methods and the number of IDC nodes in each application. The last number is also the number of basic blocks that contain IDC nodes, since there can be at most one IDC node in a basic block. As can be seen from Table I, this number is between 2–89% of the total number of basic blocks, and is close to 30% for the two largest applications. This is significant because the crucial factor in the performance of the algorithm is the number of IDC nodes.

The implementation we tested is different in one important respect from the description of the algorithm given above. Instead of searching for all blocked paths on line 7 of Algorithm 2, we continue the recursive calls. Whenever such a call encounters a blocker, it returns an empty set, as a marker of failure. Such failures are ignored in the final result. It is not clear that this strategy is better than the original one, and we intend to explore this (and other optimizations) in the future.

Tables I and II show the running times of the algorithm on the test projects, in various configurations. The first configuration, labeled “1 Thread,” gives the time for a single-threaded run, with some heuristics for ordering nodes. In three cases, the algorithm did not finish after an hour; these are marked with dashes in this column. The other columns give the results for an algorithm that creates one thread per IDC node, where each thread yields in the beginning of every dominators call, running with 1, 2, 4, or 8 hardware threads on an Intel i7-2720QM 2.20GHz CPU that has 4 cores and 8 hardware threads. Table I shows average results for one or more runs; Table II shows the smallest and largest results of four sets of runs for each multi-threaded configuration of the three largest projects. Figure 4 shows the speedup of the (average) multi-threaded results in graph form. Times shown are only for the computation of the interprocedural dominators algorithm, excluding the time required to compute the ICFG and intraprocedural dominators and dominance frontiers.

The most striking conclusion from Table II is the large variability in the run times. This is due to the strong interdependence between various threads, because of the cache optimization of line 2 of Algorithm 2. When running with a single thread that explores the graph in a depth-first manner, there are not enough opportunities for this optimization to take effect, and performance can be considerably worse than when running multiple threads on a single processor, as can be seen by comparing the “1 Thread” and “1 CPU” columns of Table I. This effect is particularly strong for the larger
programs. However, the variability was never so large for the multi-threaded version to approach the single-threaded performance for these cases; while the single-threaded version never terminated in under an hour for the three largest projects, the multi-threaded version running on a single CPU always finished within a few minutes at most.

This dependence on the order in which IDC nodes are processed also affects the speedup results. In some cases, the speedup is larger than the ratio of the number of processors; this is due to the fact that more processors enable more IDC nodes to be processed in parallel, which enriches the cache and makes the optimization more effective.

VII. CONCLUSIONS

We have presented a new algorithm to compute interprocedural domination, which extends the intraprocedural dominator trees. The algorithm can be executed incrementally and in parallel, and has been demonstrated to have practically-useful performance on large Java programs. Interprocedural applications [4], [5], [2], [12], [13], [14], [15], usually treat global variables by passing them to all procedures where they are set or used, including all intermediate procedures in a calling chain. Using our interprocedural domininance graph and the interprocedural dominance frontiers, it is possible to compute reaching definitions across procedures, and create an interprocedural SSA representation with interprocedural \( \phi \) functions that jump across call chains. For example, if \( f(x) \) sets a global variable \( x \), then calls \( g(x) \), which does not refer to \( x \) but calls \( h(x) \) which does, there is no need to pass \( x \) as a parameter; instead, \( f(x) \) sets an instance, say \( x3 \), and \( h(x) \) refers to \( x3 \) directly, using a \( \phi \) function if \( x \) may have other sources in \( h(x) \). This technique can significantly decrease the number of parameters required by other representations in languages such as C, C++, and COBOL. The benefits of a limited interprocedural SSA are shown by Calman and Zhou [5]; the full interprocedural SSA, based on interprocedural domination, can improve upon these results.

Similarly, interprocedural slicing [2] can benefit from this representation. Using the interprocedural control and data dependence relations, the intraprocedural slicing algorithm, which uses reachability over these two relations, immediately extends to the interprocedural case. To this will be added the efficiency of the direct data-dependence links that bypass some procedure calls. We intend to implement this interprocedural representation and explore its uses along these lines.

REFERENCES


APPENDIX

CORRECTNESS PROOFS

A. Preliminaries

Before proving the correctness theorem, we need to define some basic concepts.

The dominators function of Algorithm 2 computes a partial domination relationship, relative to a set $B$ of blocker nodes.

Definition 4: The partial domination set $p\text{dom}(v, B)$ is the set of all nodes $u$ such that every path from the initial node to $v$ that does not include any element of $B$ contains $u$.

We will show that the call dominators($v, B$) returns precisely the set $p\text{dom}(v, B)$. Obviously, $p\text{dom}(v, \emptyset)$ is the set of all dominators of $v$.

For every valid path $p$, we define summarize($p$) to be the path $p'$ derived from $p$ by replacing every balanced path from a call to the corresponding return with a single call-to-resume edge. We call such a path a summary path. On a summary path, any edge from a call node to a function entry, or from a function exit to a resume node, is therefore unmatched on the path.

In general, it is impossible to splice two interprocedural paths, since calls in the first might not match returns in the second. However, replacing a balanced path is always possible, as is splicing a suffix that has no unmatched returns:

Lemma 1: If $p_1$ is a valid path that contains a balanced sub-path $p_1'$ from $u$ to $v$, and $p_2$ is another valid path that contains a balanced sub-path $p_2'$ from $u$ to $v$, then replacing $p_1'$ by $p_2'$ in $p_1$ results in a valid path. Also, a suffix $p_1''$ of $p_1$ that has no unmatched returns (but may have unmatched calls) and starts at a node $u$ can replace any suffix $p_2''$ of $p_2$ that starts at $u$ to yield a valid path.

We prove the correctness of a simpler version of the algorithm, which does not contain the optimization of line 2 (that is, with lines 2–4 removed); all references to the dominators function in this appendix refer to the unoptimized version. The proof of correctness of the optimization is not shown here for lack of space.

Theorem 2 (correctness): There is a path from $u$ to $v$ in the IDG computed by Algorithm 1 with the unoptimized version of the dominators function of Algorithm 2 on a given program iff $u$ dominates $v$ interprocedurally in the program.

B. Soundness

We first prove the soundness of the representation: if there is a path from $u$ to $v$ in the IDG, then $u$ dominates $v$. By the transitivity of the dominance relationship, it is sufficient to show this property for every IDG edge. The proof will show by induction that each edge added is sound, assuming all previous edges are.

All edges added in step 1 are sound because intraprocedural dominance does not change in an interprocedural context (except, as mentioned above, for calls that can be shown never to terminate). If a function has a single call site, it dominates its entry (step 2), and return edges (step 3) are sound because there is no way to get to a resume node without passing through the exit of the called procedure.

Step 4 is not so straightforward, since it performs global analysis. The following lemma is the crucial step in the soundness proof for this case.

Lemma 3: Suppose the call dominators($v, B$) is performed with a partial IDG that is sound, and there is no path in the IDG from any element of $B$ to $v$. If the set returned by that call includes the node $u$, then $u \in p\text{dom}(v, B)$.

Note that the lemma allows for a partial IDG that contains redundant edges; this is crucial for the soundness of the parallel algorithm. Such redundant edges might be added in step 4b due to IDG edges that have not yet been computed.

Proof: Consider the set of recursive calls to the dominators function that resulted in $u$ being returned from the call dominators($v, B$). We prove the lemma by induction on the number of times that $u$ was returned in line 11 of Algorithm 2 within a series of recursive calls all of which returned $u$; call this number $N(e)$ where $c$ is the first call in the series. (It is possible that $u$ was returned by some call but was not propagated upwards, because it was removed in the intersection on line 9; such calls are not counted.)

If $N(c) = 0$, then $u$ could only have been returned in line 17. In all these calls, the condition on line 1 was therefore false, and all recursive calls were made from line 15. The first parameter was therefore some ancestor of $v$ in the IDG, and because we assumed that the IDG is sound on entry to the top-level call, $u$ is a dominator of $v$ and must be on every valid path from the initial node to $v$, whether or not it contains elements from $B$.

Assume now that $N(e) > 0$. Denote by $c_1$ the last call to return of all those counted in $N(e)$. All containing calls were not counted in $N(e)$, and therefore must have returned from line 17, implying that the call $c_1$, and all containing recursive calls (which also returned $u$) were made from line 15. All these calls pass $B$ unchanged, and so $c_1$ therefore had the form dominators($w, B$). As in the base case, because all recursive calls containing $c_1$ followed IDG edges, $w$ must dominate $v$.

Let $p$ be some valid path from the initial node to $v$ that does not contain any elements of $B$. Because $w$ dominates $v$, it must appear somewhere on $p$. If $w$ is the initial node, so is $u$, and it appears on $p$. Otherwise, let $x$ be the predecessor of the first appearance of $w$ on $p$. The path $p'$ from the initial node to $x$, being a prefix of $p$, does not contain any element of $B$; in addition, since $x$ precedes the first occurrence of $w$ on $p$, $p'$ does not contain $w$ either. Therefore, no element in $B \cup \{w\}$ dominates $x$, and because the IDG is assumed to be sound, there is no path in the IDG from any element of $B \cup \{w\}$ to $x$. The computation of $c_1$ therefore included a call dominators($x, B \cup \{w\}$), to be denoted $c_2$. Since $c_1$ returned $u$ as an intersection (line 9) that included the result of $c_2$, the result of $c_2$ must have included $u$ as well. The inductive hypothesis applies to $c_2$, since its computation does not include $c_1$, which was counted in $N(e)$, and so $N(e_2) < N(e)$. It follows that $u$ must be on $p'$, and therefore on $p$ as well.

Suppose now that the edge $(u, v)$ was added to the IDG in step 4c. This implies that $u$ was returned by the call
dominators $(v, \emptyset)$ in step 4a. We use Lemma 3 inductively on the number of iterations through step 4 to show that the IDG remains valid after each iteration. The base case follows from the soundness of the preceding steps in the algorithm. For the inductive step, Lemma 3 applied to the call dominators $(v, \emptyset)$ shows that every valid path from the initial node to $v$ (which vacuously does not contain any elements of the empty set) must contain $u$. Thus, the validity of the IDG is maintained until the end of Algorithm 1.

Corollary 4: Suppose the algorithm performed the call dominators $(v, B)$. If the set returned by that call included the node $v$, then $v \in \text{pdom}(v, B)$.

The proof is based on the inductive argument above, which shows that the soundness of the IDG is maintained after each addition. It is easy to check that the other condition of Lemma 3 is also true for every call.

C. Completeness

Turning now to completeness, we want to prove that if $u$ dominates $v$ in the program, there is a path from $u$ to $v$ in the IDG returned by Algorithm 1. Because of the complexity of the proof, we break it into several lemmas. We start with some technicalities.

Definition 5: A dominance-frontier chain from $u$ to $v$ is a sequence of nodes $d_1, \ldots, d_n$ such that $d_1$ belongs to the dominance frontier of $u$, each $d_{i+1}$ belongs to the dominance frontier of $d_i$, and $d_n$ dominates $v$.

According to Definition 1, if $u$ is an interprocedural dominance candidate, so are all the nodes in the chain. The following lemma is a simple consequence of the definition of dominance frontiers.

Lemma 5: If a node $v$ is reachable from another node $u$ in the same function but is not dominated by it, then on every path from $u$ to $v$ there is a dominance-frontier chain from $u$ to $v$.

Lemma 6: Let $p$ and $p'$ be two summary paths, and let $c$ and $c'$ be the last interprocedural dominance candidates on $p$ and $p'$, respectively. If both paths have a common node following the last occurrence of $c$ on $p$ and $c'$ on $p'$, then $c = c'$.

Proof: Let $m$ be the first common node on both paths following the last occurrence of $c$ on $p$ and $c'$ on $p'$. The proof of this lemma is tedious but not difficult. It is done by cases, depending on how the two paths enter the function containing $m$. Here we show the case where $c, c'$, and $m$ are all in the same function, whose entry node we denote by $e$. If there is a balanced path from $e$ to $m$ that does not contain $c$, then $m$ is reachable from $c$ but not dominated by it, and by Lemma 5 there is a dominance-frontier chain $d_1, \ldots, d_n$ from $c$ to $m$. In particular, $d_n$ must appear on $p$, since it dominates $m$. But $d_n$ follows $c$ on $p$, contradicting the assumption that $c$ was the last IDC element on $p$. Therefore, every balanced path from $e$ to $m$ must contain $c$, and a similar argument shows that every such path must contain $c'$ as well. It follows that both $c$ and $c'$ dominate $m$ interprocedurally. If $c \neq c'$, one must dominate the other, which must then appear last on every path to $m$, contradicting the assumption that each is the last IDC element on its path. Therefore $c = c'$.

The other cases are proved similarly, focusing on the first nodes of $p$ and $p'$ to enter the function of $m$.

Lemma 7 proves completeness for the cases covered by steps 1–3 of Algorithm 1. Lemma 8 is the counterpart of Lemma 3 for the completeness of step 4.

Lemma 7: If $u$ is a dominator of $v$ such that there are no IDC nodes on any cycle-free summary path from $u$ to $v$ (except perhaps for $u$), then $v$ is reachable from $u$ in the IDG after step 3 of Algorithm 1.

Proof: If $u = v$ the conclusion is trivially true. Assume therefore that $u \neq v$. The proof proceeds by induction on the maximum number of dominators of $v$ on any valid path from $u$ to $v$ (exclusive). In the base case, there are no dominators of $v$ inside any valid path from $u$ to $v$. If $v$ is a function entry node, it must have a single call site, otherwise it would be in IDC. This call site is a dominator of $v$, and so must be identical to $u$. Step 2 of Algorithm 1 will then add the edge $(u, v)$ to the IDG.

Otherwise, $v$ is not a function entry node. Let $e$ be the entry node of the function $f$ immediately containing $v$; $e$ dominates $v$, and therefore $u$ must follow $e$ on every valid path from the initial node to $v$ (since we assumed there are no other dominators on any valid path from $u$ to $v$). Assume by contradiction that $v$ is not reachable from $u$ in the IDG after step 3. There must be at least one unbalanced interprocedural path $p$ from $u$ to $v$, otherwise $v$ would be reachable from $u$ after step 1. This path cannot contain $e$ (which dominates $v$), so it must enter $f$ via some return-to-resume edge. Let $r$ be the call site on the last such edge on the path. According to the assumption that there are no other dominators of $v$ on the path, the resume node $r$ does not dominate $v$, and therefore there must be some other valid path from the initial node to $v$ that does not go through $r$. These paths must join at some node $w$ inside the function $f$, and so $w$ is reachable from $r$ but is not dominated by it, while both are in the same function. By Lemma 5, there is a dominance-frontier chain from $r$ to $w$; thus the path $p$ contains IDC nodes that are not inside any balanced sub-path. This contradicts the assumption of the lemma, concluding the base case.

For the inductive step, let $p$ be an any valid path from the initial node to $v$, and denote the function immediately containing $v$ by $f$. Since $u$ dominates $v$, $u$ must appear somewhere on $p$. Suppose first that $u$ appears on $p$ before the last occurrence of the entry $e$ of $f$. This entry is unmatched on $p$, and must be the last entry on the summarize($p$). It will therefore remain on that summary path even after all cycles are removed, and because we assumed that there are no IDC nodes on any cycle-free summary path, $e$ must have a single call site $c$. Therefore, Step 2 must have added to the IDG an edge from $e$ to $c$. Because $e$ dominates $v$ intraprocedurally, step 1 added a path between them, completing an IDG path from $c$ to $v$. (This path cannot contain any edges that were removed in step 1, because we know that $v$ is reachable from the initial node.) The prefix of $p$ that reaches $c$ has fewer dominators of $v$ than the full $p$, since it does not contain $e$. By the induction hypothesis, $c$ is reachable in the IDG from $u$, completing an IDG path from $u$ to $v$.

If $u = e$, it dominates $v$ intraprocedurally and, as above, we are done. Suppose now that $u$ appears on $p$ following the
last occurrence of $e$, and assume by contradiction that there is no IDG-path from $u$ to $v$ after step 3. In particular, $u$ does not dominate $v$ intraprocedurally, so there must exist a path $q$ from $e$ to $v$ such that summarize($q$) does not contain $u$. As above, on the way from the last occurrence of $u$ to $v$ on $q$ there is an unmatched return-resume edge $(t, r)$, with the following part of the path being balanced. The edge $(t, r)$ will therefore remain on the summary path even after all cycles are removed. If $r$ dominates $v$ intraprocedurally, there must be an IDG-path from $r$ to $v$ after step 1. Otherwise, there is some other valid path from the initial node to $v$ that does not go through $r$, and, as above, that path must join with $q$ inside the function $f$, resulting in a contradiction to the assumption of the lemma. This establishes the fact that there is an IDG-path from $r$ to $v$. The IDG edge $(t, r)$ was added in step 3. Any valid path from the initial node to $t$ has fewer dominators of $t$ than the maximum number of dominators of $v$ on a path to $v$, since any dominator of $t$ also dominates $v$, and every path to $v$ includes $r$, which also dominates $v$, following $t$. Therefore the induction hypothesis applies, and there is an IDG-path from $u$ to $t$, completing the path to $v$. □

Lemma 8: Let $B \subseteq IDC$. If $u \in \text{pdom}(v, B)$, then the result of the call dominators($v, B$) will contain $u$.

Proof: The result of the dominators function always contains $v$, so if $u = v$, we are done. Assume therefore that $u \neq v$.

The proof proceeds by induction on the maximum number of IDC nodes (counting $v$ but not $u$) on any summary path from $u$ to $v$ that passes through $u$ exactly once. In the base case, there are no IDC nodes on any cycle-free summary path from $u$ to $v$. According to Lemma 7, there will be an IDG-path from $u$ to $v$ after step 3 of Algorithm 1. This path contains no IDC nodes, and therefore the call dominators($v, B$) will include $u$ through a series of recursive calls on line 15 of Algorithm 2.

Assume now that there is at least one IDC node on some cycle-free summary path $q = \text{summarize}(p)$ from $u$ to $v$ that passes through $u$ exactly once. If $v \notin IDC$, let $c$ be the last interprocedural dominance candidate on $q$; by Lemma 6, $c$ is the last element of IDC on every summary path from $u$ to $v$. We will show that there is an IDG-path from $c$ to $v$. Consider the suffix $\hat{q}$ of $q$ that starts at $c$, and the corresponding suffix $\hat{p}$ of $p$; $\hat{q}$ contains no IDC nodes except for $c$. It may include some unmatched returns, followed by some unmatched calls. Each call site on $\hat{q}$ is the only call site for that function, otherwise the function entry would have been in IDC, and therefore there is an IDG-edge from each call site to the corresponding function entry. Every function entry node dominates all nodes inside the function (intraprocedurally), and so an IDG-path from the entry to the next call site on the path was created by step 1 of Algorithm 1, and similarly for the path from the entry of the function containing $v$ to $v$. Each resume node must dominate the next return node on the path, since otherwise, by Lemma 5 there would be an IDC node between them, contradicting the assumption that $c$ is the last IDC node on $q$. Therefore there is an IDG-path from each resume node to the next call on $\hat{q}$, and similarly for the last resume node and the first call (or $v$ if there is no call) and for $c$ and the first return, call, or $v$ as the case may be. Each return node is connected in the IDG to the corresponding resume node in step 3, completing the IDG-path from $c$ to $v$. This path contains no IDC nodes, and Algorithm 1 will therefore follow this path with recursive calls on line 15, at which point it will call dominators($c, B$).

If $v \in IDC$, take $c = v$, and the same call is performed. We need to show that the result of this call will contain $u$. In order to invoke the inductive hypothesis, we need to show that every valid path from the initial node to any predecessor of $c$ that does not include any element of $B \cup \{c\}$ contains $u$. It will be sufficient to show that each path to $c$ that does not pass through any element of $B$ contains $u$, since any allowed path to a predecessor of $c$ can be extended to such a path.

Assume first that $c$ is the entry node of some function. Because $c$ is an unmatched function entry, $\hat{q}$ (and also $\hat{p}$) do not contain any unmatched returns, and by Lemma 1, every valid path from the initial node to $c$ (in particular those that contain no element of $B$) can be extended with $\hat{p}$. This yields a valid path to $v$ that does not include any element of $\hat{B}$, and by the assumption of the lemma it must contain $u$. But $\hat{p}$, as part of $p$, does not contain $u$ (note that if $c = u$ there are no further IDC nodes on the path to $v$; this was handled by the base case). Therefore, $u$ must be on the path before $c$, as required.

If $c$ is not a function entry, but $\hat{p}$ does not contain any unmatched returns, the same argument applies. If $\hat{p}$ does contain unmatched returns, we assume by contradiction that there is a valid path $p′$ from the initial node to $c$ that contains neither elements from $\hat{B}$ nor $u$. Denoting the entry node of the function containing $c$ by $e$, we note that both $p$ and $p′$ must pass through $c$. The suffix $\hat{p}$ must contain a return node $r$ of that function. We now build a new path $p''$ that consists of the prefix of $p$ that reaches $e$, followed by the path on $p′$ from $e$ to $c$, followed by $\hat{p}$. This is a valid path because the only difference it has from $p$ is the balanced part from $e$ to $c$, which is taken from $p′$. The suffix of $p''$ from $e$ to $v$ does not contain $u$, because we assumed that $p''$ does not include it, and neither does $\hat{p}$. It must be on $p''$ because of the assumption of the lemma, and must therefore precede $e$ on $p''$. On the suffix of $p''$ from the last occurrence of $u$ to $v$, $c$ appears inside a balanced call, and will therefore not be part of the corresponding summary path. We can repeat this process to eliminate any series of occurrences of $c$ on the path, resulting in a summary path whose last IDC node is not $c$, contradicting the fact that $c$ must be last on any such path. Therefore $p′$ must contain $u$.

Let $r$ be an ICFG-predecessor of $c$. We have shown that $r$ satisfies the condition of the lemma, and (cycle-free) summary paths to $r$ have fewer IDC elements than (cycle-free) summary paths to $v$, since they do not contain $c$. By the inductive hypothesis, the result of the call dominators($r, B \cup \{c\}$) will contain $u$. Since this is true for every predecessor, $u$ will be in the intersection of all results on line 9 and will be returned from the enclosing call. □

Proof of Theorem 2: Corollary 4 and Lemma 8 together show that dominators($v, B$) = pdom($v, B$). In particular, by taking $B = \emptyset$ it follows that the call dominators($v, \emptyset$) in step 4 of Algorithm 1 will return the set of all dominators of $v$. The correctness of step 4 follows directly from the definition of the domination relation. □