Validating Dominator Trees for a Fast, Verified Dominance Test

Sandrine Blazy, Delphine Demange, David Pichardie

Validating Dominator Trees for a Fast, Verified Dominance Test*

Sandrine Blazy[1], Delphine Demange[1], David Pichardie[2]

[1] Université Rennes 1 – IRISA – Inria
[2] ENS Rennes – IRISA – Inria

Abstract The problem of computing dominators in a control flow graph is central to numerous modern compiler optimizations. Many efficient algorithms have been proposed in the litterature, but mechanizing the correctness of the most sophisticated algorithms is still considered as too hard problems, and to this date, verified compilers use less optimized implementations. In contrast, production compilers, like GCC or LLVM, implement the classic, efficient Lengauer-Tarjan algorithm [12], to compute dominator trees. And subsequent optimization phases can then determine whether a CFG node dominates another node in constant time by using their respective depth-first search numbers in the dominator tree. In this work, we aim at integrating such techniques in verified compilers. We present a formally verified validator of untrusted dominator trees, on top of which we implement and prove correct a fast dominance test following these principles. We conduct our formal development in the Coq proof assistant, and integrate it in the middle-end of the CompCertSSA verified compiler. We also provide experimental results showing performance improvement over previous formalizations.

1 Introduction and Related Work

Given a control flow graph (CFG) with a single entry node, computing dominators consists in determining, for each node in the graph, the set of nodes that dominate it. Informally, a node d dominates another node n if d belongs to every path from the entry node to n. The problem of computing dominators is ubiquitous in computer science, and occurs in applications ranging from program optimization, to circuit testing, analysis of component systems, and worst-case execution time estimation.

Since 1972, this problem has been extensively studied. Many algorithms have been proposed, trading-off ease of implementation and efficiency. The natural formulation of the problem as data-flow equations is due to Allen and Cocke [1]. It can be directly implemented using an iterative Kildall algorithm, but suffers, in this case, from a quadratic asymptotic complexity. Cooper et al. [4] present another iterative solution for this equation system, based on a more compact

* This work was supported by Agence Nationale de la Recherche, grant number ANR-14-CE28-0004 DISCOVER.

representation of dominator sets (only the immediate dominator, i.e. the closest dominator, is computed for each node), and a careful implementation, leading to better performance in practice, despite the same worst-case bound time as [1]. To date, the most popular algorithm remains the one by Lengauer-Tarjan [12], which, as Cooper et al. algorithm, computes a compact representation of the dominance relation (namely the *dominator tree*). But this sophisticated algorithm relies on depth-first search (DFS) spanning tree of the CFG with elaborate path compression and tree balancing techniques to achieve a stunning near-linear complexity. We refer the interested reader to [16] for a more complete survey of the numerous algorithms proposed so far in the literature, and to [10] for a thorough experimental study comparing the leading algorithms.

We consider the problem of dominators in the specific context of compilation, where dominators allow, for instance, the implementation of a variety of powerful and efficient program optimizations (e.g. loops optimization or global code motion), and the construction of the SSA form [5], an intermediate representation of code that is specially tailored towards program optimization. Production compilers, like GCC or LLVM, implement the classic, efficient Lengauer-Tarjan algorithm [12], to compute dominator trees. Subsequent optimization phases can then determine whether a node dominates another node in constant time by using their respective DFS traversal numbers in the dominator tree.

Specifically, the present work is part of a compiler verification effort, where an (optimizing) compiler must be formally proved to preserve the program behaviors along the compilation chain, i.e. the generated code behaves as prescribed by the semantics of the source program, if any. In this context, correctly implementing a time- and space-efficient dominator algorithm is not sufficient; one has to formally prove its correctness. We are not aware of any formal verification of the dominator problem outside of the field of compiler verification. Further, faced with this technical difficulty, existing verified compilers either ignore dominators, or implement simplified and under-optimized versions of dominator algorithms.

For instance, the CompCert C compiler [14,13] is not based on any SSA form for performing code optimization, and no global optimization uses explicit dominance information. The CompCertSSA project extends the CompCert compiler with an SSA-based middle-end. The SSA generation algorithm [2] is proved by *a posteriori* validation of an external checker. Although we prove that the checker ensures the strictness of the SSA generated function (that is, each variable use is dominated by its definition), the checker implementation (a simple, non-iterative, CFG traversal) and soundness proof do not rely on the computation of dominators. The only phase of the CompCertSSA middle-end that depends on such a computation (that we would like to be efficient) is a common sub-expression elimination (CSE) optimization based on Global Value Numbering (GVN). It discovers equivalence classes between program variables, where variables belonging to the same class are supposed to evaluate to the same value. Its implementation, presented in [7], closely follows the choices made in production compilers, and performs some dominance test requests to make sure that the chosen representative of a variable class dominates the definition point of that variable. To date, this dominance test was implemented (and proved directly) with a simple Allen and Cocke algorithm, using a Kildall workset algorithm, thus impacting the performance of our middle-end.

Another SSA-based verified compiler is Vellvm. Zhao et al. [20,18] formalize the LLVM SSA intermediate form and its generation algorithm in Coq. Their work follows closely the LLVM design and their verified transformation can be run inside the LLVM platform itself. Zhao et al. [19] formalize in Coq a fast dominance computation based on the Cooper et al. algorithm [4], but their algorithm is, for verification purposes, a simplified version of the initial algorithm. This is a non-trivial formalization work that also proves in Coq the completeness of the dominance relation computation, an interesting and difficult problem in itself. However, this work does not focus on compilation time. Other CPS or ANF-based verified compilers for functional languages [3,6] implement simple optimizations that do not require dominance information, although their (unverified) peers, like MLton, benefit from dominators for, e.g. contification [8] for inter-procedural optimization.

Facing the conceptual complexity of the most clever variants for computing dominators, there has been a growing interest in proposing ways of checking their results. Georgiadis et al. [11] propose a linear-time checker of the dominator tree, based on the notions of headers and loop nesting forests. Georgiadis et al. [9] propose a linear-time certifying algorithm, producing a certificate (a preorder of the vertices of the dominator tree, with a so-called property *low-high*), that helps simplifying the checking process. Despite that checking the low-high property on the certificate is straightforward, and easily implemented in linear time, linking the low-high property back to the immediate dominance relation (via the concepts of strongly independent spanning trees) remains quite involved. As a matter of fact, to date, these two recent, sophisticated algorithms are still out of the reach of mechanized developments.

In our context of verified compilation, we need two things: compute efficiently the dominance relation, and represent this relation in a compact way, so that the dominance test can be implemented efficiently. Note however that mechanically verifying (or validating) the dominator tree remains, for the time being, unessential. Hence, we believe that the technique used in GCC and LLVM, i.e. computing a dominator tree using Lengauer-Tarjan's algorithm, and then fast-checking dominance with an ancestor test in the DFS numbering of the dominator tree, provides a, perhaps more modest, yet viable, trade-off between efficiency and verifiability. We argue that this technique can also be applied to verified compilers, by relying on an *a posteriori* validation approach. We present a formally verified validator of untrusted dominator trees, on top of which we implement and prove a fast dominance test that follows these principles.

Contributions. After recalling the technical background on dominators and the main algorithms (Section 2), we present the following contributions.

- A new, simple and verified validator for the dominance relation (Section 3), which leads to a formally verified implementation of a dominance test technique used in production compilers. The heart of the validator algorithm is our own contribution but it is mixed with well-known graph algorithms for fast ancestor checking. This paper presents, to our knowledge, the first verification of these kinds of techniques.

- Empirical evidence that this technique allows, in practice, a non-negligible performance gain, even in the context of verified compilers (Section 4).
- The integration of this dominance computation and dominance test within the CompCertSSA verified compiler. Our formal development and experiments are available online at http://www.irisa.fr/celtique/ext/ssa_dom/.

2 Technical Background and Overview of Algorithms

In this section, we recall the technical background on dominance, together with standard techniques to compute this relation. We also present how dominance and dominance testing is implemented in modern compilers.

2.1 Definitions

A control flow graph $G = (N, E, e)$ is defined as an oriented graph, i.e. a set of nodes N, a set of edges E, and a distinguished entry node $e \in N$ (that is not the successor of any other node). In the following, we depict an edge connecting node $i \in N$ to $j \in N$ by $i \to j$.

Definition 1 (Dominance relation). *A node d dominates a node n if n is reachable from the graph entry node and if any path from the entry point to n contains d. If $d \neq n$, the dominance is said to be strict.*

For every node n (except the entry e), the set $sdom(n)$ of nodes that strictly dominate n, contains a node $idom(n)$ that is dominated by every other nodes in $sdom(n)$ [12]: the *immediate dominator*. As an extra important corollary, the immediate dominance relation can be represented as a tree [12].

Definition 2 (Dominator tree). *The dominator tree of a CFG is a tree whose nodes are the nodes of the CFG, and where the children of a node are all the nodes that it immediately dominates.*

Figure 1 shows an example of a CFG and its dominator tree. For instance, in the CFG, node 10 is dominated by node 15, since all paths from the entry node 17 to node 10 must go through 15. Hence, in the dominator tree, node 10 must have node 15 as an ancestor. However, 15 is not the immediate dominator of node 10, it is node 13: indeed, in the set $sdom(10) = \{17, 16, 15, 14, 13\}$, node 13 is the one dominated by every other node in the set. Hence, node 10 is a child of node 13 in the dominator tree.

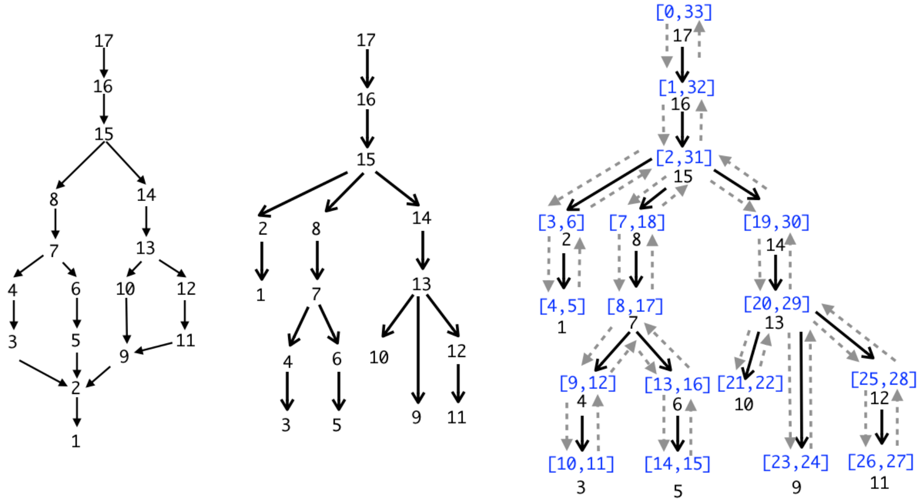

Figure 1. Left: example CFG, with entry point 17, and nodes ordered in reverse postorder (on the left). Center: its dominator tree (where, if i has j as a child, then i is the immediate dominator of j). Right: dominator tree (solid arrows), annotated with a DFS traversal (dotted arrows), and its corresponding DFS intervals (see section 2.3).

2.2 Standard Techniques for Computing Dominance

Allen-Cocke (AC) standard data-flow analysis [1]. The AC algorithm is based on the following fixpoint characterization of dominance.

$$dom(n) = \{n\} \cup \bigcap_{n' \to n} dom(n') \tag{1}$$

Intuitively, it captures that every strict dominator of a node n must also dominate every n's predecessors in the CFG. Such fixpoint equation can be solved using a workset fixpoint iteration *à la* Kildall. As is typical for forward data-flow problems, the fixpoint resolution is speeded up if at each workset iteration we choose the node with the lowest rank with respect to a reverse postorder ordering on the CFG (a node is visited before any of its successor nodes has been visited, except when the successor is reached by a back edge). A direct implementation is quadratic in the number of nodes, or $O(|N|^2)$.

Cooper-Harvey-Kennedy (CHK) algorithm [4]. The CHK dominance computation improves Allen-Cocke data-flow approach using the following properties. First, dominator sets can be characterized by the immediate dominator table.

$$\forall n, \exists k, dom(n) = \{n, idom(n), \ldots, idom^k(n)\} \tag{2}$$

CHK can be understood as a variation of the previous approach where dominance sets are implicitly represented by the immediate dominator tree. Using reverse postorder ordering, by noticing that $\forall n, n \prec_{\text{rpo}} idom(n)$, set intersection can be performed in an efficient way because if $dom(a) \cap dom(b) \neq \emptyset$, then the resulting set is a prefix of both $dom(a)$ and $dom(b)$ [4]. This algorithm performs better in practice than AC, but follows the same $O(|N|^2)$ asymptotic time complexity.

Lengauer-Tarjan (LT) algorithm [12]. Modern compilers implement dominance using the LT algorithm. It uses depth-first search and union-find data structures to achieve an asymptotic complexity of $O(|N|log|N|+|E|)$. It relies on the subtle notion of *semi-dominator* which provides a convenient intermediate step in the dominators computation. An amortized quasi-linear complexity can be obtained using path compression but it does not seem to be implemented in practice.

2.3 Modern Implementation of Dominance Test in Compilers

As explained above, modern compilers such as GCC or LLVM implement dominance following the LT algorithm. Once they obtain a dominator tree (as shown in Figure 1), they pre-process it to obtain a constant-time dominance test. The dominance between two nodes d and n can be determined by testing if the node d is an ancestor of n in the dominator tree. For instance, in Figure 1, node 15 dominates node 10 because there is an upward path from 10 to 15 in the tree.

This test can be performed in constant time thanks to a linear pre-computation (on the $|N| - 1$ edges of the dominator tree). For each node, one computes a *depth-first search interval* $I(n) = [d(n), f(n)]$ where $d(n)$ is the discovery time of node n during the traversal (the first time n exists in the DFS stack) and $f(n)$ is the finishing time (the time where all sons of n have been processed) [17]. In a direct acyclic graph, d is an ancestor of n if and only if $I(n) \subseteq I(d)$. Figure 1 shows, on the right, the results of such an interval computation: intervals bounds are determined according to the starting and ending time clocks when depth-first traversing the tree. There, the fact that 15 dominates node 10 is obtained by observing that interval $I(10) = [21, 22]$ is included in $I(15) = [2, 31]$.

As a result of this pre-computation with complexity $O(|N|log|N| + |E|)$, a constant time dominance can be obtained by storing the intervals information in adequate data structures.

3 Validator and Proof of Dominance Test

Our formalization is done on top of an abstract notion of CFG. Such a graph is defined as follows by an entry node and a set of edges.

```
Variable entry : node.
Variable cfg : node → node → Prop.
```

In the sequel, `reached : node→Prop` is a predicate characterizing the set of nodes that are reachable, via `cfg`, from the node `entry`, and `dom : node→node→Prop` denotes the dominance relation that is defined using a standard definition of CFG paths.

In this section, we assume that an external tool computes a list `dt_edges` that contains the reversed edges of the candidate dominator tree (i.e. the pair (i,j) represents that, in the candidate dominator tree, i is a child of j, or that j immediately dominates i).

```
Variable dt_edges : list (node * node).
```

We then validate this list and build a dominance test, implemented by the function test_dom : node→node→bool that satisfies the following theorem:

```
    forall i j, reached j → test_dom i j = true → dom i j.
```

In the rest of this section, we proceed in three steps. First, we give a dominator map D : node→node (extracted from dt_edges), a specification that entails dominance. Then, we provide an efficient procedure to test whether a node is a descendant of another in the dominator tree (encoded morally in D). This procedure is used twice: for checking that D meets its specification, and in the final implementation of the dominance test, test_dom.

3.1 Validation of Dominator Tree

In this section, we assume a dominator map D:node→node that provides an (immediate) dominator candidate for each node. We will explain in Section 3.4 how we build D from the list dt_edges. We provide a formal specification for D and prove it entails dominance. Note that we do not prove that it implies immediate domination, as this is not required in our final soundness theorem[3].

The specification, inspired from Equations (1) and (2), is defined as follows.

```
Record D_spec := { D_entry : D entry = entry;
                   D_cfg   : forall i j,  cfg i j → Dstar i (D j) }.
```

where (Dstar i j) holds whenever j is of the form D^k(i), for some k. Formally:

```
Inductive Dstar : node → node → Prop :=
| D_refl : forall i, Dstar i i
| D_trans: forall i j, Dstar i j → Dstar i (D j).
```

We then prove, quite straightforwardly, that D_spec implies dominance by induction on the definition of predicate reached.

```
Theorem D_spec_correct : D_spec →
  forall i j, reached i → Dstar i j → dom j i.
```

Hence, we can validate the map D if we manage to check that it satisfies the specification D_spec. Interestingly, we need an executable version of the Dstar relation for two distinct usages. First we want to validate D_spec on D. Second, we want to implement a dominance test using Dstar.

[3] Such a property would be required to prove *completeness*: if a node d dominates a node n then the dominance test on (d, n) should succeed. To our experience in verified compilation, we never make usage of such a completeness property. The property holds, but we do not need to prove it in Coq.

3.2 Ancestor Test in the Dominator Tree

In this section we assume an acyclic oriented graph[4], defined by an entry node and a map, `sons`, from nodes to the list of their successors[5]. We will later relate this graph with our dominator tree.

```
Variable entry: node.                 (* entry node *)
Variable sons : PTree.t (list node).  (* adjacency map *)
```

As outlined in Section 2.3, the ancestor test consists in performing a depth-first traversal of the graph, starting from `entry`, and using a traversal clock, that increases each time a node is encountered (by visiting it or by marking it). We compute for each node n, an interval $I(n) = [d(n), f(n)]$ where $d(n)$ is the value of the clock when node n was first encountered, and $f(n)$ is the value of the clock when all successors of n have been processed. If the graph is acyclic and each node is reachable from `entry`, we can use these intervals to perform efficient ancestor tests [17]: there exists a path from n to m in the graph if and only if $I(m) \subseteq I(n)$. For our purpose, we only need to prove that this condition is sufficient. We define intervals, intervals inclusion and our efficient ancestor test in an interval map as follows.[6]

```
Record itv := { pre: Z; post: Z }.
Definition itv_Incl (i1 i2:itv) : Prop :=
  i2.(pre) <= i1.(pre) /\ i1.(post) <= i2.(post).
Definition is_ancestor (itvm: PTree.t itv) (n1 n2:node) : bool :=
  match itvm!n1, itvm!n2 with
    | Some i1, Some i2 ⇒ itv_Incl i2 i1
    | _, _ ⇒ false
  end.
```

Now, to state the correctness of our interval computation, we specify a notion of ancestor called `InSubTree`. A node r is an ancestor of n (or equivalently n belongs to a subtree whose root is r) if $n = r$ or there exists a successor s of n such that s is an ancestor of n.

```
Inductive InSubTree (r:node) : node → Prop :=
|InSubTree_root: InSubTree r r
|InSubTree_sons: forall n s, InSubTree s n→In s (sons r)→InSubTree r n.
```

The interval map is computed by the function `build_itv` that performs the recursive DFS traversal of the graph, accumulating in a record of type `state`, the current interval map, and the current time clock.

```
Record state := { itvm: PTree.t itv;   (* the interval map *)
                  next: Z              (* the current time *) }.
```

[4] Not to be confused with the control flow graph here.

[5] `PTree` is a dictionary implementation using Patricia trees provided in CompCert. Type (`Ptree.t a`) denotes an associative, partial map with keys of type `positive` – binary encoding of strictly positive integers – with associated data of type `a`. In this paper, types `node` and `positive` are synonyms.

[6] We write `m!n` the lookup of a key `n` in a map `m`.

Note that, to ensure termination of build_itv_rec, we use a fuel auxiliary argument, i.e. a natural number counter decreasing at each recursive call. The fuel argument is useful not only to avoid proving termination, but also, and more crucially, to get a useful induction principle on the next inductive predicate.

```
Variable fuel: nat.
Definition build_itv (entry:node) : option state :=  build_itv_rec
      entry   (* start traversing the graph at entry node *)
      {| itvm := PTree.empty _; next := 0 |} (* initial state *)
      fuel.   (* initial fuel *)
Fixpoint build_itv_rec (n:node) (st:state) (fuel:nat) : option state :=
  match fuel with
    | O ⇒ None                  (* no more fuel, abort computation *)
    | S fuel ⇒
      let pre_n := st.(next) in  (* current time when we reach node n *)
      match fold_left (fun ost s ⇒  (* we process each successor *)
                          match ost with
                            | None ⇒ None
                            | Some st ⇒ build_itv_rec s st fuel
                          end)
                      (sons n)
                      (Some {| itvm := st.(itvm); next := st.(next)+1 |})
      with
      | None ⇒ None
      | Some st ⇒ (* if no fuel error occurred, we extract st.(next) *)
                 (* to build the last component of n's interval  *)
        let itv_n := {| pre := pre_n; post := st.(next) |} in
      Some {| itvm := PTree.set n itv_n st.(itvm); next := st.(next)+1 |}
      end
  end.
```

The correctness theorem of build_itv states that, in the resulting interval map st.itvm, interval inclusion implies an ancestor relationship in the tree.

```
Theorem build_itv_correct :
  NoRepetTreeN entry (S fuel) →
  forall st, build_itv = Some st →
  forall n1 n2 itv1 itv2,
    st.(itvm)!n1 = Some itv1 → st.(itvm)!n2 = Some itv2 →
    itv_Incl itv1 itv2 → InSubTree n2 n1.
```

As can be seen, this theorem is proved under the hypothesis that the graph is well-formed, namely that it does not contain duplicates or crossing edges, as expressed by predicate NoRepetTreeN, whose formal definition is the following.

```
Inductive NoRepetTreeN (r:node) : nat → Prop :=
| NoRepetTreeN0: NoRepetTreeN r O
| NoRepetTreeN_sons: forall k,
  (forall s,                                    (* sons are well formed *)
          In s (sons r) → NoRepetTreeN s k) →
  (forall s,              (* r does not appear in any of its subtrees *)
          In s (sons r) → ¬ InSubTree s r) →
  (forall s1 s2 n,                     (* r’s subtrees do not intersect *)
           In s1 (sons r) → InSubTree s1 n →
           In s2 (sons r) → InSubTree s2 n → s1=s2) →
  (list_norepet (sons r)) →           (* r’s sons don’t have duplicates *)
  NoRepetTreeN r (S k).
```

The definition of NoRepetTreeN is *staged*, i.e. indexed by a natural number. This level in the definition (that coincides with the height of the tree under consideration) provides a nice induction principle when combined with the fuel argument of function build_itv. Without such a trick, Coq does not generate a useful induction principle.

We prove build_itv_rec correctness using several auxiliary invariants, notably that the clocks are monotonic, that computed intervals are never empty, and that in a given subtree, computed intervals are included in the interval of the root of the subtree.

3.3 Well-formed Graph Construction

This section explains how we relate the list dt_edges that contains the edges of the dominator tree, with the immediate dominator map D we use in Section 3.1, and the graph representation used in Section 3.2. We not only build a map of successors, but also check sufficient conditions enforcing the NoRepetTreeN property presented previously.

Starting from the list dt_edges, we straightforwardly build a map D from nodes to their immediate dominator candidate with the function make_D_fun of type make_D_fun (dt_edges:list (node*node)) : node → node. If a node is not in the association list dt_edges, its (correct) immediate dominator is set to itself.

In a similar way to the construction of the candidate dominator tree from dt_edges, we also define the function build_succs of type

```
build_succs (dt_edges:list (node * node)): option (PTree.t (list node))
```

that performs a reverse topological sort to build a map that associates to each node the (candidate) list of immediately dominated nodes. Function build_succs somewhat builds the inverse of the map D. Its correctness theorem states that the output successor tree, if any, is well-formed.

```
Theorem build_succs_no_repet : forall dt_edges sn,
  build_succs dt_edges = Some sn →
  NoRepetTreeN (sons sn) entry (S fuel).
```

This theorem follows from the checks performed during the computation of build_succs. Indeed, in its signature, the option type of the result represents a validation failure.

```
Theorem build_succs_correct_tree : forall dt_edges sn,
  make_D_fun dt_edges = D → build_succs dt_edges = Some sn →
  forall i j, In j (sons sn(i)) → D j = i.
```

```
Definition compute_test_dom (f: function) :
                                    option (node → node → bool) :=
 (* external computation in OCaml, using Lengauer-Tarjan algorithm *)
 let dt_edges : list (node*node) := extern_d f in
 (* immediate dominator map computation from dt_edges *)
 let D : node → node := make_D_fun dt_edges in
 (* successors tree computation from dt_edges *)
 match build_succs (entry f) l with
  | Some sn ⇒
    (* interval computation from successor tree *)
   match build_itv_map (entry f) sn with (* build_itv(entry f)(sons sn)*)
        | Some itvm ⇒
          (* fast ancestor test using interval, see section 3.2  *)
          let td := (fun i j ⇒ is_ancestor itvm (D j) i) in
          (* immediate dominator tree validation using ancestor test *)
          if (check_D_eq f D td) then Some (is_ancestor itvm) else None
        | None ⇒ None
    end
  | None ⇒ None
 end.
```

Figure 2. Dominance test construction

During the traversal of dt_edges, we check that it contains no edge of the form (n,n), and that, when processing an edge (n,d), i.e. adding n to the list of successors of node d, node d was already seen (i.e. is already a key in the tree), and that node n has not yet been seen. Hence, to be accepted by the validator, the provided list dt_edges should be topologically sorted, and by the same validation, we ensure there is no loop in the graph. For further details, we refer the reader to the formal development available online.

3.4 Final Construction

The final dominance test computation is given in Figure 2. It takes as input a program represented by its CFG (more precisely any function of this program) and combines the various functions presented earlier. It is proved correct with the following theorem.

```
Theorem dom_test_correct : forall f test_dom,
   compute_test_dom f = Some test_dom →
   forall i j, reached f j → test_dom i j = true → dom f i j.
```

We now discuss its asymptotic complexity. If N denotes the number of nodes in the CFG, and E the number of edges, then the asymptotic complexity of this computation is as follows.

- The list dt_edges has length $N - 1$ (every node, except the entry, has a unique immediate dominator).

- The map `make_D_fun dt_edges` is computed with 1 traversal of `dt_edges` and 1 map update is performed at each step. The overall complexity is $O(N \log N)$.
- `build_succs` is computed with one traversal of `dt_edges` and several set and map updates are performed at each step. The overall complexity is $O(N \log N)$.
- Intervals are built with a traversal of a graph with N nodes and $N-1$ edges (this is a tree). At each step, some map updates are performed. The overall complexity is $O(N \log N)$.
- One ancestor test requires two map lookups and some integer comparisons. Each integer[7]is between 0 and $2N-1$. The overall complexity of an ancestor test is $O(\log(N))$.
- Dominance tree validation requires, for each edge in the CFG, one ancestor test and some map lookups. The overall complexity of this step is $O(E \log(N))$.

Overall, the dominance tree pre-computation follows an asymptotic complexity of $3O(N \log(N)) + O(E \log(N)) = O(E \log(N))$ ($N \leq E$ as all nodes are reachable from the entry) and the generated dominance test requires $O(\log(N))$ computations.

As will be explained in the next section, we also provide a *native* version of the implementation, that uses native integers for graph nodes and interval bounds. It does not improve the asymptotic time complexity of the whole dominance test construction, but it enables a constant time dominance test since interval lookup is as fast as an array access and interval test inclusion requires four comparisons between native integers.

4 Experimental Results

Implementations. We compare experimentally the following dominance tests:

I-CHK This is the implementation of the CHK algorithm available from the Vellvm project [19]. To be able to plug it inside our middle-end, we have performed the slightest adaptation possible (essentially by-passing the abstract data-type of `atoms` and making them be bare `positive`; these are used to define program points in the CFG). We have kept the choices of data-structures used in their available development.

I-ACZ This is the implementation of the AC algorithm available from the Vellvm project [19]. We performed the same adaptations as in I-CHK.

I-AC This is the implementation of the AC algorithm initially available in CompCertSSA [2,7]. The implementation uses a classical Kildall workset algorithm for solving the data-flow equations, and its correctness is proved directly (no *a posteriori* validator). The CFG is stored in a `PTree` mapping to every node in the graph, the list of its successors. The Kildall solver, taken directly from CompCert, uses a Coq implementation of a heap data structure

[7] Recall that we rely on the standard implementation of positive integers, `positive`.

Program	LoC
bzip2	7007
hmmer	13458
mcf	2697
raytracer	2867
nsichneu	4033
papabench	4763
lzss	3063
lzw	2629
oggenc	58463
spass	82103

Cat. ID	#nodes	#func
(1)	$[500; 1k[$	229
(2)	$[1k; 2.5k[$	155
(3)	$[2.5k; 4k[$	66
(4)	$[4k; 8k[$	49
(5)	$[8k; 22k[$	42
(6)	$[22k; 40k[$	20
(7)	$[40k; 50k[$	13
(8)	$[50k; max]$	8

Table 1. Benchmarks description (lines of codes and categories of function sizes)

(splay tree). Dominator sets are implemented using `PTree` while I-ACZ [19] uses unsorted lists.

I-DT This is the implementation presented in the previous section of this paper. Its correctness is partly validated, partly verified. The CFG and dominance test computation use `PTree`. The external computation of the dominator tree is done in OCaml, using the LT algorithm on arrays, for more efficiency.

I-DT-NAT Same as I-DT, but in native mode. The CFG of the function and the dominance test computation use Patricia trees [15] on OCaml integers, and the pre-computed dominance test is stored in an OCaml array to allow for a constant time access. Dominator test is constant time.

I-LLVM This is the OCaml-only version of the algorithm presented in the previous section, without *a posteriori* validation. The CFG is stored in a `PTree`. The dominator tree is computed using mutable OCaml data-structures (arrays and stacks), for more efficiency. Interval are stored in a mutable array but since graph nodes are encoded as Coq binary numbers (`positive`), dominator test is $O(\log |N|)$ instead of constant time.

I-LLVM-NAT Same as I-LLVM, except that the CFG of the function and the dominance test computation use arrays. Dominator test is constant time. This provides a lower bound of the performance we could aim to achieve.

I-CHK, I-ACZ, and I-AC represent the state-of-the-art of mechanically verified dominance test (with a direct correctness proof and no extra validator).

Benchmarks. We plug each implementation in our SSA middle-end by extracting its Coq implementation into OCaml code, and running it on some realistic C program benchmarks, described in Table 1, taken from the CompCert test suite, the SPEC2006 benchmarks and WCET-related reference benchmarks. These represent around 192,600 lines of C code, each program ranging from thousands of lines of C code, to tens of thousands.

To evaluate the scalability of the implementations in extreme conditions, we force the compiler to always inline functions with a CFG size below 1000 nodes. We classify some of our results by categories of function size, i.e. the number of CFG nodes of its SSA form (see Table 1). We also present some global results, categorized by programs. Experiments were run on a MacBook OSX 10.8.5, 2.3GHz Intel Core i7, 8GB 1600MHz DDR3.

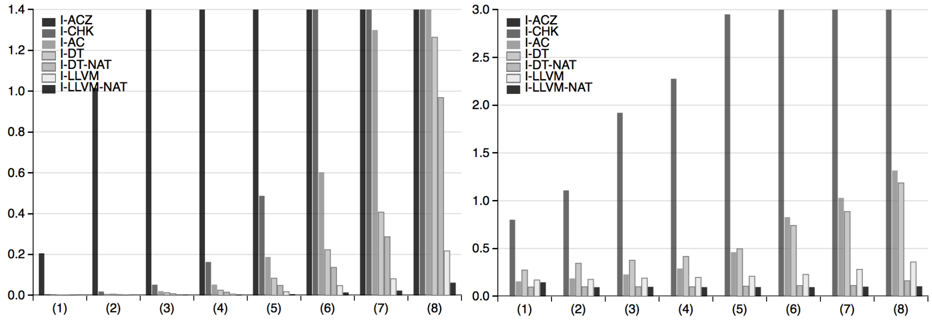

Figure 3. Building (left) and using (right) times, by function size (the average time for each category). For I-ACZ, we set a time-out of 2 seconds for building the dominance test of one program function. Because of these time-outs, we do not show using times for I-ACZ (in practice, they are similar to, or higher than the ones for I-CHK).

Measures and results. To evaluate and compare the dominance test implementations, we measure both the building time of the dominance test, and its practical cost in time, when using it.[8]Results are presented in Figure 3 and Figure 4.

The building time of the dominance test is the time, in seconds, required to compute the function `test_dom: node` → `node` → `bool`. For I-DT, and I-DT-NAT, this includes the validation time. As for the using time of dominance tests, we measure for each function the time, in micro-seconds, spent in executing dominance test requests. To avoid glitches in the measures of so small values, we performed 5 times the measures for all tests, and kept the lowest value. The collection of dominance tests is the same for all implementations, these are the ones required by our GVN-based CSE optimization (see Section 1). It is worth noting that our GVN-CSE performs the exact same set of requests to the dominance test, independently of the implementation that is used. Additionally, on this set of dominance test requests, we have checked that the various implementations were returning the same verdict (thus establishing their relative correctness and completeness one to each other).

In Figure 3 and Figure 4, we observe that I-DT performs significantly better than I-AC, I-ACZ, I-CHK for the building time. Setting I-DT to native mode (I-DT-NAT) improves performance. Comparing building times for I-LLVM against I-DT, and I-LLVM-NAT against I-DT-NAT gives a good estimate of the cost of the validator. In terms of using time, we do not observe so much a big difference between I-DT and I-AC. This is as expected, given that both implementations require `PTree` accesses. I-ACZ and I-CHK provide much slower dominance test because dominator sets are represented by lists. In Figure 3, for using time, we observe a constant time for I-DT-NAT and I-LLVM-NAT thanks to arrays accesses. I-DT and I-LLVM are slower due to `PTrees` accesses.

[8] The impact of building and using the dominance test is currently negligible compared to the whole compilation time, as, currently, certain compiler passes (such as the SSA deconstruction) would need performance improvement.

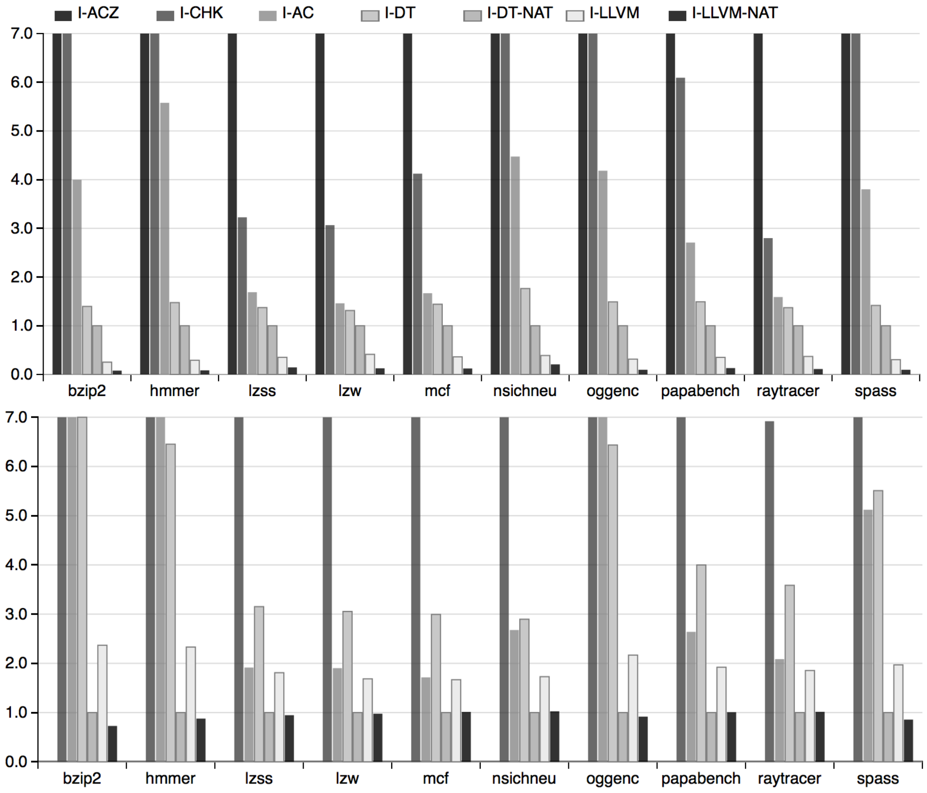

Figure 4. Total building (top) and using (bottom) time overheads, relatively to the I-DT-NATIVE implementation, classified by programs.

5 Conclusion and Perspectives

We have described a new verified validator for the dominance relation. It is able to validate the state-of-the-art dominance construction by Lengauer and Tarjan combined with an ancestor test in the dominator tree candidate. This technique, borrowed from (un-verified) production compilers like GCC and LLVM, brings an important speedup compared to previous verified algorithms [19,2]. Using native data-structures after extraction, it builds a constant-time dominance test similar, in terms of efficiency, to the non-verified test.

In terms of program optimization, this dominance test already provides a strong support (i.e. we are able to perform efficient dominance test on the CFG on demand), and we already leverage this tool in our GVN-based CSE. This important building block could help us implement other powerful optimizations such as loop invariant code motion. However, the most efficient implementation of natural loops detection rely on iteration startegies on the dominator tree itself. In this case, the dominance checking is no longer sufficient, and one may have to investigate the mechanized verification of certifying algorithms for the dominator tree, such as the linear-time certifying algorithm by Georgiadis et al. [9].

References

1. F. E. Allen and J. Cocke. Graph Theoretic Constructs For Program Control Flow Analysis. Technical report, IBM T.J. Watson Research Center, 1972.
2. G. Barthe, D. Demange, and D. Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM TOPLAS*, 36(1):4:1–4:35, March 2014.
3. A. Chlipala. A verified compiler for an impure functional language. In *POPL'10*, pages 93–106. ACM, 2010.
4. K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Technical report, Rice University, 2006.
5. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.
6. Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In *LPAR'07*, LNCS, pages 211–225. Springer-Verlag, 2007.
7. D. Demange, L. Stefanesco, and D. Pichardie. Verifying Fast and Sparse SSA-based Optimizations in Coq. In *Proc. of CC'15*, volume 9031 of *LNCS*, pages 233–252, 2015.
8. M. Fluet and S. Weeks. Contification Using Dominators. In *Proc. of ICFP'01*, pages 2–13. ACM, 2001.
9. L. Georgiadis, L. Laura, N. Parotsidis, and R.E. Tarjan. Dominator certification and independent spanning trees: An experimental study. In *Experimental Algorithms*, volume 7933 of *LNCS*, pages 284–295. Springer, 2013.
10. L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications*, 10(1):69–94, 2006.
11. L. Georgiadis and R.E. Tarjan. Dominator tree verification and vertex-disjoint paths. In *Proc. of SODA '05*, pages 433–442. ACM, 2005.
12. T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM TOPLAS*, 1(1):121–141, 1979.
13. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
14. X. Leroy. A formally verified compiler back-end. *JAR*, 43(4):363–446, 2009.
15. C. Okasaki and A. Gill. Fast mergeable integer maps. In *In Workshop on ML*, pages 77–86, 1998.
16. N. Parotsidis and L. Georgiadis. Dominators in Directed Graphs: A Survey of Recent Results, Applications, and Open Problems. In *2nd International Symposium on Computing In Informatics And Mathematics (ISCIM 13)*, volume 1, pages 15–20. Epoka University, 2013.
17. R.L. Rivest T.H. Cormen, C.E. Leiserson and C. Stein. *Introduction to Algorithms, third edition.* 2009.
18. J. Zhao, S. Nagarakatte, M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *PLDI'13*, pages 175–186. ACM, 2013.
19. J. Zhao and S. Zdancewic. Mechanized verification of computing dominators for formalizing compilers. In *Proc. of CPP'12*, pages 27–42. Springer, 2012.
20. J. Zhao, S. Zdancewic, S. Nagarakatte, and M. Martin. Formalizing the LLVM intermediate representation for verified program transformation. In *POPL'12*, pages 427–440. ACM, 2012.