An Efficient SSA-Based Algorithm for Complete Global Value Numbering

Jiu-Tao Nie and Xu Cheng

Peking University, Beijing, China
{njt,chengxu}@mprc.pku.edu.cn

Abstract. Global value numbering (GVN) is an important static analysis technique both for optimizing compilers and program verification tools. Existing complete GVN algorithms discovering all Herbrand equivalences are all inefficient. One reason of this is the intrinsic exponential complexity of the problem, but in practice, since the exponential case is quite rare, the more important reason is the huge data structures annotated to every program point and slow abstract evaluations on them site by site. In this paper, we present an SSA-based algorithm for complete GVN, which uses just one global graph to represent all equivalences at different program points and performs fast abstract evaluations on it. This can be achieved because in SSA form, interferences among equivalence relations at different program points can be entirely resolved with dominance information. We implement the new algorithm in GCC. The average proportion of execution time of the new algorithm in the total compilation time is only 0.36%. To the best of our knowledge, this is the first practical complete GVN algorithm.

1 Introduction

Global value numbering (GVN) is an important static analysis technique. It detects equivalences of program expressions, which have a variety of applications. Optimizing compilers use this information to detect and eliminate semantic redundant computations [2,14,16,11,15], and useless branches. Program verification tools use it to verify program assertions. Translation validation tools use it to check the validation of program transformations [10], such as the correctness of an optimizer, by discovering equivalences of different programs.

Since checking general equivalence of program expressions is an undecidable problem even when all conditionals are treated as non-deterministic [12], most GVN algorithms treat both all conditionals as non-deterministic and all operators as uninterpreted. The equivalence relation with these restrictions is called *Herbrand equivalence* [13]. The GVN algorithms that can discover all Herbrand equivalences are referred to as *complete* GVN algorithms.

Unfortunately, existing efficient GVN algorithms used in practice are all incomplete. The simple hash-based GVN algorithm fails to detect many kinds of equivalences in the presence of loops and joins. Alpern, Wegman and Zadeck's (AWZ) partition refinement algorithm [1] is based on the static single assignment (SSA) form [4,5]. It treats phi nodes as uninterpreted operators, so equivalences

Z. Shao (Ed.): APLAS 2007, LNCS 4807, pp. 319–334, 2007.

among phi nodes and ordinary expressions can't be discovered. Rüthing, Knoop and Steffen (RKS) improved on AWZ algorithm by incorporating several rewriting rules to remedy this problem [13]. However, their algorithm remains incomplete both for acyclic and cyclic programs [8]. Gargi proposed a set of balanced algorithms that are efficient, but also incomplete [6].

Recently, Gulwani and Necula proposed a randomized polynomial GVN algorithm [7] based on the idea of random interpretation, which involves performing abstract interpretation using randomized data structures and algorithms. This algorithm is complete and efficient. However, unlike other GVN algorithms, there is a small probability that this algorithm deduces false equivalences, i.e. it's not sound. False equivalences are acceptable for program verification tools as long as their appearance probability can be made small enough. However, for compilers this is strictly disallowed.

The obstacle of applying powerful sound and complete GVN algorithms [9,15,8] in practice is their unacceptable low efficiency. Theoretically, all complete GVN algorithms have exponential complexity in the size of the program [8]. To this problem, Gulwani and Necula have proposed a polynomial algorithm that computes all Herbrand equivalences among terms with limited sizes [8]. In practice, choosing the size limitation to the program size is sufficient. Moreover, the exponential case is quite rare in practical programs. Thus, the exponential complexity problem is not really crucial now. The more important reason of the low efficiency of complete GVN algorithms is the huge data structures annotated to every program point and slow abstract evaluations on them site by site [13].

To this problem, we propose a new SSA-based complete GVN algorithm, which uses just one global graph to represent all equivalences at different program points and performs fast abstract evaluations on it. Previous complete GVN algorithms all perform abstract interpretation [3] on ordinary programs. Transforming these algorithms to those working on programs in SSA form is trivial (only need to add abstract interpretation function for phi nodes that can be regarded as copy statements copying values from their operands corresponding to incoming edges to their target). However, we observe that in SSA form, since each variable has only one definition site and its available scope is program points dominated by its definition site, interferences among equivalence relations at different program points can be entirely resolved with dominance information. Therefore, performing abstract evaluations on just one global value number graph is possible. Moreover, using global equivalence representation also greatly speeds up abstract evaluations, since equivalence relation changes caused by each statement only needs to be applied to the global graph once rather than being transferred to all affected local graphs one by one. The difficulty of this achievement is choosing the abstract evaluation order. Naively performing abstract evaluations in an arbitrary order with the global value number graph may cause information inconsistency problem and loss of precision. However, we find that performing abstract evaluations through all edges of a spanning tree of the control flow graph and all other edges separately can conquer this problem.

In the rest of this paper, Section 2 defines the program representation and some relevant notations used in this paper. Section 3 reviews the traditional complete GVN algorithm working on programs not in SSA form. Section 4 extends the basic algorithm to the SSA-based version, and shows how a global value number graph is used to represent all local equivalences. Then, the abstract evaluation order problem is discussed, and the order used by our algorithm is shown to be correct. Section 5 gives implementation details of our algorithm and shows how to restrict it to be polynomial based on the approach proposed in [8]. Section 6 gives experimental results, and Section 7 concludes the paper.

2 Program Representation

We use notations V, O and F to denote program variable set, operator set, and nonfunctional operation set respectively. For notation simplicity, we also regards constants as variables belonging in V. For example,

$$V = \{x_1, x_2, 21, -9, \ldots\} \;\; O = \{+, -, \ldots\} \;\; F = \{load, store, call, branch, \ldots\}$$

The function $arity : (V \cup O \cup F) \rightarrow \omega$ is defined as follows:

$$arity(x) = \begin{cases} 0 & x \in V \\ \text{operands number of } x & x \in O \cup F \end{cases}$$

We assume statements have been decomposed into such a simple form:

$$x_0 = f(x_1, x_2, \ldots, x_{arity(f)}), \text{ where } f \in O \cup F,\ x_i \in V.$$

If $f \in F$, we say that statement/expression is a relevant statement/expression, whose result relies not only on its operands but its position relative to other relevant statements, and it may also cause side effects. For concept unification, we regard parameters and constants as results of hidden relevant statements just after program entry. For a variable x, $def(x)$ denotes x's definition statement. For a statement s, $lhs(s)$ and $rhs(s)$ denote the left and right hand side expressions of s respectively. For an expression e, $Vars(e)$ denotes the set of variables appearing in e.

A program P is represented by a directed flow graph $P = (N_P, E_P, entry, exit)$. The node set N_P consists of statements, join nodes that merge more than one control flow, and $entry$ and $exit$ nodes of P. The edge set E_P represents nondeterministic control flows. For a node n of a directed multigraph, we use $succ(n)$ to denote the successor sequence of n and $succ(n)[i]$ the i-th successor. Correspondingly, $pred(n)$ and $pred(n)[i]$ are for the predecessor sequence and the i-th predecessor of n.

3 Traditional Complete Global Value Numbering

In this section, we review the three components of the traditional complete GVN algorithm as an abstract interpretation problem.

3.1 Abstract Semantic Domain

The abstract semantic domain of the GVN problem is the lattice of equivalence relations of expressions. An equivalence relation of expressions can be compactly represented by an annotated directed acyclic graph (DAG), which is called *Structured Partition DAG* (SPDAG) in [15] and *Strong Equivalence DAG* (SED) in [8]. In this paper, we use a similar data structure called *Value Number DAG* (VNDAG) to represent equivalences. A VNDAG is a labeled directed acyclic graph $D = (N_D, E_D, L_D, M_D)$ satisfying:

1. (N_D, E_D) is a directed acyclic graph with node set N_D and edge set E_D.
2. $L_D : N_D \to V \cup O \cup \{\bot, \top\}$ is a labeling function satisfying $\forall \nu \in N_D.\ arity(L_D(\nu)) = |succ(\nu)|$.
3. $\forall \nu_1, \nu_2 \in N_D.\ (L_D(\nu_1) = L_D(\nu_2) = l \wedge \forall\ i \in [1, arity(l)].\ succ(\nu_1)[i] = succ(\nu_2)[i]) \to \nu_1 = \nu_2$.
4. $M_D : V \to N_D$ is a function mapping each variable to a node of the DAG.

Every node of a VNDAG represents a value number. A node is labeled by either a variable $x \in V$, indicating that it's a leaf node, or an operator $o \in O$, indicating that it has $arity(o)$ successors, or the special symbols $\bot$ or $\top$. In a VNDAG, there is at most one node with a given label and a given sequence of successors.

For any VNDAG D, every value number $\nu \in N_D$ represents a variable set

$$A_D(\nu) = \{x \in V \mid M_D(x) = \nu\}$$

and an expression set

$$T_D(\nu) = \begin{cases} A_D(\nu) & L_D(\nu) \in V \\ A_D(\nu) \cup \{o(t_1, \ldots, t_n) \mid t_i \in T_D(succ(\nu)[i])\} & L_D(\nu) = o \in O \\ \emptyset & L_D(\nu) = \bot \\ \{t \mid Vars(t) \subseteq A_D(\nu)\} & L_D(\nu) = \top \end{cases}$$

Note that $T_D(\nu_\bot)$ is $\emptyset$ and $T_D(\nu_\top)$ contains all expressions whose variables have the value number $\nu_\top$.

We use the notation $D \models t_1 = t_2$ to denote that the VNDAG D implies that expressions t_1 and t_2 are equivalent. Then, the equivalence of any two expressions represented by VNDAG D is deduced as follows (here $x \in V$, $o \in O$, and t, t_i and t'_i denote any expressions):

$$D \models x = t \ \text{ iff } \ \{x, t\} \subseteq T_D(M_D(x))$$
$$D \models o(t_1, \ldots, t_n) = o(t'_1, \ldots, t'_n) \ \text{ iff } \ D \models t_1 = t'_1 \wedge \ldots \wedge D \models t_n = t'_n$$

The abstract semantic domain is in fact the lattice $\mathbb{D}$ of all VNDAGs including two special VNDAGs $D_\bot$ denoting the empty relation, and $D_\top$ denoting the universal relation.

For a VNDAG D and a variable x, the function $newvn(D, x)$ adds a new node labeled by x to D and returns it. For a non-relevant expression t, the following function returns a value number ν such that $t \in T_D(\nu)$.

$$vn(D, t) = \begin{cases} M_D(t) & t \equiv x \in V \\ find(D, o, vn(D, t_1), \ldots, vn(D, t_n)) & t \equiv o(t_1, \ldots, t_n) \end{cases}$$

where, $find$ returns an existing node with corresponding label and successors, or a newly created one (which is also added to D) if such a node is not found. In practice, the VNDAG and $find$ can be implemented with a hash table. Simplifications and normalizations, such as constant folding and expression reassociation can be integrated into $find$ so that more equivalences can be detected.

3.2 Abstract Interpretation Function

Each node in N_P of a program P is interpreted by the abstract interpretation function $\theta : N_P \times \mathbb{D} \to \mathbb{D}$ defined as follows:

$$\theta(n, D) = \begin{cases} \text{let } D' = D \text{ in } D'[M_{D'}[newvn(D', x)/x]/M_{D'}] & n \equiv x = f(\ldots) \\ \text{let } D' = D \text{ in } D'[M_{D'}[vn(D', t)/x]/M_{D'}] & n \equiv x = t \\ \text{let } D' = D \text{ in } D' & otherwise \end{cases}$$

The informal meaning of θ is that: copy D to D' first; then 1) for a relevant statement, add a new node labeled by its left hand side variable to D' and set its value number to the new node, and return the updated D'; 2) for a non-relevant statement, get the node of its right hand side expression and set its left hand side variable's value number to that node, and return the updated D'; 3) for other nodes (join nodes and *entry* and *exit*), return the unchanged D'.

3.3 Abstract Evaluation

For a program P, each edge $e \in E_P$ is associated with a VNDAG denoted as $vndag(e)$. We use $dest(e)$ to denote the destination node of e. For a program node $n \in N_P$, we use $succ_e(n)$ to denote the set of edges whose source node is n, and $pred_e(n)$ the set of edges whose destination node is n. $D_1 \sqcap D_2$ returns a VNDAG that represents equivalences represented by both D_1 and D_2 (refer to [8] for the implementation of $\sqcap$). $D_1 \sqsubset D_2$ iff the equivalence relation represented by D_1 is a strict subset of that represented by D_2. Then, the abstract evaluation algorithm for the complete GVN is given in Figure 1.

```
1 foreach e ∈ E_P do vndag(e) := D_⊤
2 worklist := succ_e(entry)
3 while worklist ≠ ∅ do
4     Take e from worklist
5     foreach e' ∈ succ_e(dest(e)) do
6         D := vndag(e') ⊓ θ(dest(e), vndag(e))
7         if D ⊏ vndag(e') then
8             vndag(e') := D
9             worklist := worklist ∪ {e'}
```

Fig. 1. Traditional complete GVN algorithm

4 SSA-Based Complete Global Value Numbering

4.1 The Trivial SSA-Based Algorithm

In SSA form, every variable has exactly one definition site, and all statements using it as an operand must be dominated by its definition site. This property is achieved by inserting phi nodes at appropriate join nodes and renaming variables for operands and targets of statements and phi nodes [4,5]. The only new notion of the SSA form is the phi node. Thus, to transform the traditional complete GVN to the SSA-based version, we only need to replace join nodes with phi nodes in the program's representation, and define the abstract interpretation function for phi nodes as follows:

$$\theta(x_0 = \phi(x_1, \ldots, x_n), D, i) = \text{let } D' = D \text{ in } D'[M_{D'}[M_{D'}(x_i)/x_0]/M_{D'}]$$

where, the new parameter i is the number of the incoming edge through which the abstract evaluation reaches that phi node. For other nodes, parameter i is ignored. During the abstract evaluation, phi nodes at the same join node are evaluated in parallel, since just evaluating parts of them doesn't make sense.

4.2 Use One VNDAG to Represent All Equivalences

With the property of the SSA form, we extend the expression set represented by each VNDAG node by adding a program edge parameter and further extend the equivalence deducing rules so that just one VNDAG can represent all equivalence relations at different program edges. We use the notation *dom* to denote the dominance relation between program nodes or edges. For a program P, each edge $e \in E_P$, the expression set

$$T_D(e, \nu) = \{t \in T_D(\nu) \,|\, \forall x \in Vars(t).def(x)\ dom\ e\}$$

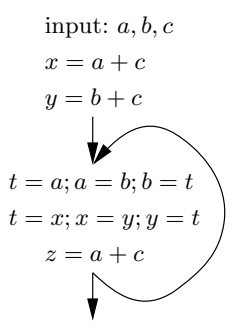
contains all expressions represented by ν and available at e. We use the notation $D \models_e t_1 = t_2$ to denote that the VNDAG D implies that expressions t_1 and t_2 are equivalent at e. Then, the equivalence of any two expressions at e represented by D is deduced as follows (here $x \in V$, $o \in O$, and t, t_i and t'_i denote any expressions):

$$D \models_e x = t \ \text{ iff } \ \{x, t\} \subseteq T_D(e, M_D(x))$$
$$D \models_e o(t_1, \ldots, t_n) = o(t'_1, \ldots, t'_n) \ \text{ iff } \ D \models_e t_1 = t'_1 \wedge \ldots \wedge D \models_e t_n = t'_n$$

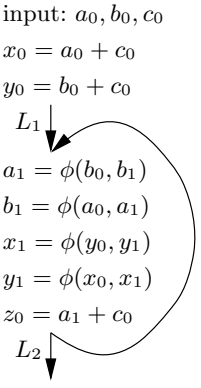
Figure 2 shows a program and the VNDAG representing all Herbrand equivalences in it. For example, we have the following term sets:

$$\begin{aligned} T_D(5) &= \{a_1\} \\ T_D(2) &= \{c_0\} \\ T_D(7) &= \{x_1, z_0\} \cup \{a_1 + c_0\} \\ T_D(L_2, 7) &= \{x_1, z_0, a_1 + c_0\} \end{aligned}$$

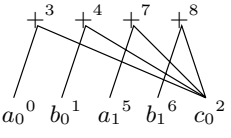
Then, we can deduce that $D \models_{L_2} x_1 = z_0$, $D \models_{L_2} x_1 + c_0 = (a_1 + c_0) + c_0$, etc. (RKS-algorithm fails to detect the equivalence of x_1 and z_0 at L_2.) Note that, the only VNDAG represents all Herbrand equivalences at all program edges.



(a) Original program



(b) SSA form of the program



(c) VNDAG of the program

$a_0 : 0$	$y_0 : 4$	$y_1 : 8$
$b_0 : 1$	$a_1 : 5$	$z_0 : 7$
$c_0 : 2$	$b_1 : 6$	
$x_0 : 3$	$x_1 : 7$	

(d) Value number of each variable (M_D)

Fig. 2. A program and the VNDAG representing all Herbrand equivalences in it. In (c), the superscript of variable and operator is a unique number representing that node in this figure.

4.3 Abstract Evaluation with One Global VNDAG

The hard part is performing abstract evaluations with just one global VNDAG. The naive idea is using the global VNDAG to replace all local VNDAGs appearing in the original abstract evaluation algorithm. Unfortunately, it doesn't always work correctly when joins and loops exist. This is because performing abstract evaluations in an arbitrary order can't ensure that equivalence relations represented by the global VNDAG involved in an operation are always consistent. For example, if the abstract evaluation process reaches a phi node $x_0 = \phi(x_1, x_2)$ through its left incoming edge without touching its right incoming edge, then x_1 may have been set to a new equivalence class but x_2 is still in the old one. Thus, the meet operation after this node doesn't make sense and generates wrong results.

To solve this problem, we adopt a particular evaluation order since the evaluation order can be arbitrary [9]. The edges of a program P are divided into two subsets: the spanning tree (whose root node is *entry*) edges E_{tP} and others $E_P - E_{tP}$. Now, we use two global VNDAGs D and D_1. D is initialized to $D_\top$ at the beginning and D_1 backups the original D before each evaluation iteration. In the i-th evaluation iteration, the $(i-1)$-th (by referring to D_1) equivalence relations of edges in $E_P - E_{tP}$ and the i-th (by referring to D) equivalence relations of edges in E_{tP} are processed in a top-down order of the spanning tree, and the

results are saved in D. The iteration continues until D reaches the fixed point. Figure 3 gives the formal algorithm. Though two global VNDAGs are used here, one is enough if the algorithm is subtly designed. The next section shows how we achieve this.

```
1 D := D_⊤
2 repeat
3     D_1 := D
4     foreach e ∈ E_tP in a top-down order do
5         n := dest(e)
6         let i satisfy that pred_e(n)[i] = e
7         D := θ(n, D_1, 1) ⊓ ... ⊓ θ(n, D_1, |pred_e(n)|) ⊓ θ(n, D, i)
8 until D = D_1
```

Fig. 3. SSA-based complete GVN algorithm with global VNDAG

We show the correctness of this algorithm by building the correspondence between it and the algorithm shown in Figure 4, which is obviously a sound and complete GVN algorithm since compared with the trivial SSA-based algorithm, only the abstract evaluation order is changed (first for non-tree edges and then for tree edges in a top-down order).

```
1 foreach e ∈ E_P do vndag(e) := D_⊤
2 repeat
3     foreach e ∈ E_P − E_tP do
4         foreach e' = succ_c(dest(e))[i] do
5             vndag(e') := vndag(e') ⊓ θ(n, vndag(e), i)
6     foreach e ∈ E_tP in a top-down order do
7         foreach e' = succ_c(dest(e))[i] do
8             vndag(e') := vndag(e') ⊓ θ(n, vndag(e), i)
9 until no vndag(e) changed in this iteration
```

Fig. 4. SSA-based complete GVN algorithm with local VNDAGs

To connect a global VNDAG with a set of local VNDAGs, we introduce the function $\eta : \mathbb{D} \times E_P \to \mathbb{D}$. For any global VNDAG $D \in \mathbb{D}$, program edge $e \in E_P$ and any expressions t_1 and t_2, $\eta(D, e)$ is defined to be a local VNDAG $D' \in \mathbb{D}$ satisfying that $D' \models t_1 = t_2$ iff $D \models_e t_1 = t_2$, i.e. D' is the local VNDAG at e representing the same equivalence relation as D at e. Let $R(D)$ denote the equivalence relation represented by D, $Na(e)$ denote the set of variables not available at e, and for a variable set X, $Pairs(X) = \{\langle t_1, t_2\rangle \mid Vars(t_1) \cap X \neq \emptyset \vee Vars(t_2) \cap X \neq \emptyset\}$. Then, $R(\eta(D, e)) = R(D) - Pairs(Na(e))$. About η, θ and $\sqcap$, the following two lemmas hold:

Lemma 1. *For any $D_1, D_2 \in \mathbb{D}$, $e \in E_P$, $\eta(D_1 \sqcap D_2, e) = \eta(D_1, e) \sqcap \eta(D_2, e)$.*

Proof. We only need to show that the equivalence relations represented by VNDAGs of the two sides are equivalent. $R(\eta(D_1 \sqcap D_2, e)) = R(D_1 \sqcap D_2) - Pairs(Na(e)) = R(D_1) \cap R(D_2) - Pairs(Na(e)) = (R(D_1) - Pairs(Na(e))) \cap (R(D_2) - Pairs(Na(e))) = R(\eta(D_1, e)) \cap R(\eta(D_2, e)) = R(\eta(D_1, e) \sqcap \eta(D_2, e))$. Thus, the proposition holds. □

Lemma 2. *For any $e \in E_P$, let $n = dest(e)$, $e' \in succ_e(n)$, then $\eta(\theta(n, D, i), e') = \eta(\theta(n, \eta(D, e), i), e')$.*

Proof. Let θ_R denote the abstract interpretation function on equivalence relations. We omit the first parameter n of θ and θ_R for simplicity in this proof since they are all the same. $R(\eta(\theta(\eta(D, e), i), e')) = R(\theta(\eta(D, e), i)) - Pairs(Na(e')) = \theta_R(n, R(D) - Pairs(Na(e)), i) - Pairs(Na(e')) =^a \theta_R(n, R(D), i) - Pairs(Na(e) - \{lhs(n)\}) - Pairs(Na(e')) =^b \theta_R(n, R(D), i) - Pairs(Na(e')) = R(\eta(\theta(D, i), e'))$. In the equation, $=^a$ is because θ_R only removes and adds equivalence pairs belonging in $Pairs(\{lhs(n)\})$ from and to the input relation. $=^b$ is because $Pairs(Na(e) - \{lhs(n)\}) \subseteq Pairs(Na(e'))$. □

In the new algorithm, each step of interpretation and meet operations on the global VNDAG corresponds to a set of operations on a set of local VNDAGs. The following lemma builds the connection between the new algorithm working on the global VNDAG and that working on local VNDAGs.

Lemma 3. *At line 7 of Figure 3, for any $e' \in succ_e(n)$ and $e_j = pred_e(n)[j]$, let $k = |pred_e(n)|$, then $\eta(D, e') = \theta(n, \eta(D_1, e_1), 1) \sqcap \ldots \sqcap \theta(n, \eta(D_1, e_k), k) \sqcap \theta(n, \eta(D, e), i)$*

Proof. We omit the first parameter n of θ for simplicity in this proof since they are all the same.

$$
\begin{aligned}
&R(\eta(D, e')) = R(\eta(\theta(D_1, 1) \sqcap \ldots \sqcap \theta(D_1, k) \sqcap \theta(D, i), e')) \\
&= R(\eta(\theta(\eta(D_1, e_1), 1), e') \sqcap \ldots \sqcap \eta(\theta(\eta(D_1, e_k), k), e') \sqcap \eta(\theta(\eta(D, e), i), e')) \\
&= R(\theta(\eta(D_1, e_1), 1)) \cap \ldots \cap R(\theta(\eta(D_1, e_k), k)) \cap R(\theta(\eta(D, e), i)) - Pairs(Na(e')) \\
&=^a R(\theta(\eta(D_1, e_1), 1)) \cap \ldots \cap R(\theta(\eta(D_1, e_k), k)) \cap R(\theta(\eta(D, e), i)) \\
&= R(\theta(\eta(D_1, e_1), 1) \sqcap \ldots \sqcap \theta(\eta(D_1, e_k), k) \sqcap \theta(\eta(D, e), i))
\end{aligned}
$$

$=^a$ is because that if a variable is available at all predecessors of e', then it must also be available at e'. □

The correctness of the new algorithm follows from the following theorem.

Theorem 1. *Let $vndag_j(e)$ denote the local VNDAG of the program edge e before the j-th iteration of the algorithm in Figure 4. At the beginning of each j-th iteration of algorithms in Figure 4 and Figure 3, for any program edge e, $\eta(D, e) = vndag_j(e)$.*

Proof. When $j = 1$, the proposition holds obviously. Assume that the proposition holds for $j \leq m$ ($m \geq 1$). When $j = m + 1$, in the m-th iteration, after each meet operation in line 7 of Figure 3, due to Lemma 3 and that any changes

on D by this meet operation don't affect the equivalence relations represented by D at successor edges of processed tree edges (since removed and added pairs are not available there due to the top-down spanning tree order), by induction on the spanning tree, we can prove that for any $e' \in succ_e(n)$, $\eta(D, e') = \theta(n, \eta(D_1, e_1), 1) \sqcap \ldots \sqcap \theta(n, \eta(D_1, e_k), k) \sqcap \theta(n, \eta(D, e), i) = \theta(n, vndag_m(e_1), 1) \sqcap \ldots \sqcap \theta(n, vndag_m(e_k), k) \sqcap \theta(n, vndag_{m+1}(e), i) = vndag_{m+1}(e')$. After all tree edges are processed, for any $e \in E_P$, $\eta(D, e) = vndag_{m+1}(e)$. □

5 Implementation of the New Algorithm

Using the global VNDAG to represent all equivalences not only greatly reduces the data structure size used by the algorithm, but also can greatly speed up the abstract evaluation process. Since all evaluation results are stored in the same VNDAG, only those (rather than all) program nodes that will affect the VNDAG need to be evaluated. Thus, in practice, evaluations are performed along those define-use (DU) chains through spanning tree edges starting from initial program nodes affecting the global VNDAG. The algorithm shown in Figure 5 builds such a kind of DU chains. The variable set $uses(x)$ stores all variables whose definition site is a non-relevant statement referring to x or a phi node referring to x through a spanning tree edge. At the same time, the algorithm also computes the affected basic block set $affected(x)$ for each variable x, which stores blocks containing at least one phi node referring to x. If the value number of x is changed, phi nodes of blocks in $affected(x)$ must be evaluated.

```
1  BuildDUChains()
2  begin
3      foreach non-relevant statement x = t do
4          foreach y ∈ Vars(t) do
5              uses(y) := uses(y) ∪ {x}
6      foreach phi node x = φ(x_1, ..., x_n) do
7          let b be the containing block of the phi node
8          foreach i ∈ [1, n] do
9              affected(x_i) := affected(x_i) ∪ {b}
10             if (pred(b)[i], b) is a spanning tree edge then
11                 uses(x_i) := uses(x_i) ∪ {x}
12 end
```

Fig. 5. The algorithm for building define-use chains ($uses$)

Notice that in the algorithm in Figure 3, the only use of VNDAG D_1 is computing D at line 7 and testing if D is changed. These two tasks can be achieved without explicitly copying D to D_1. We only need to save changes into a variable-value-number pair set $newupdates$. The sub-expression $\theta(n, D_1, 1) \sqcap \ldots \sqcap \theta(n, D_1, |pred_e(n)|)$ at line 7 for all $n = dest(e), e \in E_{tP} \wedge bb(n) \in changed$ are computed by the algorithm shown in Figure 6, and the results are saved in

```
1  DetectNewUpdates()
2  begin
3      clear newupdates
4      foreach b ∈ changed do
5          if sorted(b) = false then
6              sort phi nodes in b in a topological order of their value numbers in
               the VNDAG
7              sorted(b) := true
8          foreach phi node x := φ(x_1, ..., x_n) in b do
9              ν := Intersect(x, M_D(x_1), ..., M_D(x_n))
10             if ν ≠ M_D(x) then
11                 newupdates := newupdates ∪ {⟨x, ν⟩}
12         clear memorize
13 end
```

Fig. 6. The algorithm for detecting new updates

newupdates as update pairs. It calls `Intersect` [1] shown in Figure 7 to compute value numbers of phi nodes of each affected blocks (in *changed*), and uses the map $memorize : N_D^n \to N_D$ to build the equivalence relation among them. If and only if the returned value number differs from the original one, the result of the intersection is less than the original equivalence relation, and the new update pair is added to *newupdates*. Phi nodes with leaf value numbers must be processed before others so that these value numbers can be correctly labeled with a unique variable rather than returned as the $\perp$ that denotes the equivalence class containing no available variables. This is achieved by sorting phi nodes in a topological order[2] of their value number nodes in the VNDAG, when their containing blocks are first touched.

The rest part of the expression at line 7 of Figure 3 ($\sqcap\theta(n, D, i)$) is implemented by the algorithm shown in Figure 8. The procedure `SetCounter` shown in Figure 9 counts the number of dependent variables $cnt(x)$ along spanning tree edges for each affected variable x. The meet operation is reflected by the condition at line 6, i.e. only if the previous VNDAG deduces that the left hand side variable is equivalent to the right hand side expression, the update should be done for that variable. Then, `UpdateVN` updates value numbers for variables whose *cnt* is set in a topological order of their dependence DAG, and adds affected blocks into *changed*, where new update detection should be performed. To strictly correspond to the algorithm in Figure 3, an additional meet and update process along spanning tree edges should be performed, but it can be omitted since the subsequent process of the main procedure subsumes it.

The main procedure of the new algorithm is shown in Figure 10. It initializes *newupdates* with pairs of targets of relevant statements and unique value

[1] We omit the first parameter of *newvn*, *vn* and *find* since all of them work on the only global VNDAG D.

[2] If we attach an increasing sequence number to each value number when they are created, we can use the ascending order of their sequence numbers.

```
1  Intersect(x, ν1, ..., νn)
2  begin
3      if ν1 = ... = νn then return ν1
4      ν := memorize(ν1, ..., νn)
5      if ν ≠ nil then return ν
6      if ∀i ∈ [1, n].νi = o(μi1, ..., μim) then
7          foreach j ∈ [1, m] do
8              ξj := Intersect(nil, μ1j, ..., μnj)
9          if ∃j ∈ [1, m].ξj = ⊥ then ν := ⊥
10         else ν := find(o, ξ1, ..., ξm)
11     else ν := ⊥
12     if ν = ⊥ and x ≠ nil then ν := newvn(x)
13     memorize(ν1, ..., νn) := ν
14     return ν
15 end
```

Fig. 7. The algorithm for intersecting value numbers

```
1  UpdateVN()
2  begin
       // Count dependence numbers of affected variables
3      foreach ⟨x, ν⟩ ∈ newupdates do SetCounter(x)
       // Update MD for variables in newupdates
4      foreach ⟨x, ν⟩ ∈ newupdates do
5          cnt(x) := 0
6          MD(x) := ν
7          wl := wl ∪ {x}
       // Update MD for other affected variables
8      while wl ≠ ∅ do
9          take v from wl
10         changed := changed ∪ affected(v)
11         foreach x ∈ uses(v) do
12             if x is a result of a phi node then
13                 if cnt(x) = 1 then
14                     cnt(x) := 0
15                     MD(x) := MD(v)
16                     wl := wl ∪ {x}
17             else
18                 cnt(x) := cnt(x) − 1
19                 if cnt(x) = 0 then
20                     MD(x) := vn(rhs(def(x)))
21                     wl := wl ∪ {x}
22 end
```

Fig. 8. The algorithm for updating value numbers

```
1 SetCounter(v)
2 begin
3     cnt(v) := cnt(v) + 1
      // Visit v's successors at the first time
4     if cnt(v) = 1 then
5         foreach x ∈ uses(v) do
6             if x is not a result of phi node or M_D(x) = M_D(v) then
7                 SetCounter(x)
8 end
```

Fig. 9. The algorithm for setting counters of variables

```
1 GVN()
2 begin
3     BuildDUChains()
4     foreach x ∈ V do M_D(x) := ⊤
5     foreach x ∈ V and x is a result of f ∈ F do
6         newupdates := newupdates ∪ {⟨x, newvn(x)⟩}
7     while newupdates ≠ ∅ do
8         UpdateVN()
9         DetectNewUpdates()
10 end
```

Fig. 10. The efficient complete GVN algorithm

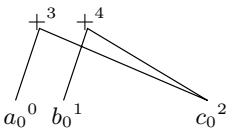
numbers of them. Then, it updates value numbers of affected variables and detects new updates iteratively until a fixed point is reached.

Figure 11 shows the process of applying our algorithm on the program in Figure 2. The initial *newupdates* comprises pairs of three parameters and leaf value numbers of them. After updating value numbers along DU chains, two new value numbers, $+^3$ and $+^4$ are created and M_D is updated. By applying `Intersect` on phi nodes in the loop body block, all of their value numbers are changed, so four new update pairs are added to *newupdates*. Since *newupdates* is not empty, another iteration runs and the result of the second value number updating is shown in (d). Now, applying `Intersect` on these phi nodes again, we get the same value numbers of them as the last time. Thus, *newupdates* is empty and the algorithm terminates. The resulting VNDAG of the algorithm deduces complete Herbrand equivalences of that program.

The approach used in [8] that restricts the exponential complete GVN to a polynomial GVN that detects equivalences among expressions with limited sizes can be directly adopted in our algorithm. We can set a counter to limit the VNDAG size before calling `Intersect` in `DetectNewUpdates`. In `Intersect`, the counter is decreased by one whenever expressions are decomposed into subexpressions (before line 7 of Figure 7). When the counter is decreased to zero, it simply set the value number variable ν to $\bot$. Refer to [8] for more details about this approach and its complexity analysis.

$$\{\langle a_0, {a_0}^0\rangle, \langle b_0, {b_0}^1\rangle, \langle c_0, {c_0}^2\rangle\}$$

(a) The *newupdates* before the first iteration

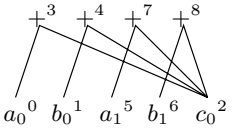


$a_0 : 0$ $y_0 : 4$ $y_1 : 3$
$b_0 : 1$ $a_1 : 1$ $z_0 : 4$
$c_0 : 2$ $b_1 : 0$
$x_0 : 3$ $x_1 : 4$

(b) The VNDAG and M_D after the first iteration

$$\{\langle a_1, {a_1}^5\rangle, \langle b_1, {b_1}^6\rangle, \langle x_1, {a_1}^2 +^7 {c_0}^2\rangle, \langle y_1, {b_1}^6 +^8 {c_0}^2\rangle\}$$

(c) The *newupdates* before the second iteration



$a_0 : 0$ $y_0 : 4$ $y_1 : 8$
$b_0 : 1$ $a_1 : 5$ $z_0 : 7$
$c_0 : 2$ $b_1 : 6$
$x_0 : 3$ $x_1 : 7$

(d) The VNDAG and M_D after the second iteration

Fig. 11. Applying our algorithm on the program in Figure 2

6 Experimental Results

We have implemented the new algorithm without VNDAG size restrictions and a semantic code motion algorithm using its result in GCC-4.2.0 as a standard pass. To show the efficiency of the new algorithm, we measured elapsed user times of it and the total compilation, as well as ratios between them, in compiling SPEC CPU2000 on a machine with a 800MHz Pentium-M and 512MB memory. Times are measured by GCC's own timing facility exported in `timevar.h`. The results are shown in Table 1, which indicate that the new GVN algorithm is extremely

Table 1. Times (in sec.) spent in GVN and total compilation, as well as ratios between them in compiling SPEC CPU2000

CINT2000	gzip	vpr	gcc	mcf	crafty	parser	gap	vortex	bzip2	twolf
CGVN	0.03	0.07	0.61	0.00	0.09	0.14	0.22	0.17	0.01	0.16
Total	6.88	19.23	213.60	2.94	30.83	24.01	75.83	61.79	5.23	35.93
Ratio (%)	0.44	0.36	0.29	0.00	0.29	0.58	0.29	0.28	0.19	0.45

CFP2000	wupwise	swim	mgrid	applu	mesa	art	equake	facerec	ammp	lucas	fma3d	sixtrack	apsi
CGVN	0.00	0.00	0.03	0.04	0.30	0.02	0.01	0.04	0.07	0.04	0.70	0.54	0.12
Total	3.09	0.71	2.08	7.58	70.58	2.13	3.08	8.78	17.68	6.00	223.23	111.15	18.50
Ratio (%)	0.00	0.00	1.44	0.53	0.43	0.94	0.32	0.46	0.40	0.67	0.31	0.49	0.65

Overall	GVN: 3.41 Total: 950.86 Ratio: 0.36%

fast, and efficient enough to be applied in practice. Notice that this is achieved without VNDAG size restriction, which suggests that the exponential complexity problem almost does not exist in practice.

7 Conclusions and Future Works

There are two reasons causing existing complete GVN algorithms inefficient. The first one, the intrinsic exponential complexity problem has been discussed in previous works, but in practice, it is not the important reason (attested by our experiments). This paper conquers the more important and practical reason, the huge data structures and slow abstract evaluations of previous complete algorithms. Based on the SSA form, the new algorithm uses just one global VNDAG to represent all equivalences at different program points and performs fast abstract evaluations on it. The experimental results show that the new algorithm is efficient enough to be applied in practice. To the best of our knowledge, this is the first practical complete GVN algorithm.

In the future, one interesting direction is to generalize this SSA-based approach to apply on other program analysis problems. Another direction is to improve the precision of the basic complete GVN algorithm by considering operator properties, conditionals, memory access instructions and inter-procedural problems.

References

1. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: POPL, pp. 1–11 (1988)
2. Briggs, P., Cooper, K.D.: Effective partial redundancy elimination. In: PLDI, pp. 159–170 (1994)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
4. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: POPL, pp. 25–35 (1989)
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
6. Gargi, K.: A sparse algorithm for predicated global value numbering. In: PLDI, pp. 45–56 (2002)
7. Gulwani, S., Necula, G.C.: Global value numbering using random interpretation. In: Jones, N.D., Leroy, X. (eds.) POPL, pp. 342–352. ACM, New York (2004)
8. Gulwani, S., Necula, G.C.: A polynomial-time algorithm for global value numbering. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 212–227. Springer, Heidelberg (2004)
9. Kildall, G.A.: A unified approach to global program optimization. In: POPL, pp. 194–206 (1973)
10. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI, pp. 83–94 (2000)

11. Odaira, R., Hiraki, K.: Partial value number redundancy elimination. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 409–423. Springer, Heidelberg (2005)
12. Reif, J.H., Lewis, H.R.: Symbolic evaluation and the global value graph. In: POPL, pp. 104–118 (1977)
13. Rüthing, O., Knoop, J., Steffen, B.: Detecting equalities of variables: Combining efficiency with precision. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 232–247. Springer, Heidelberg (1999)
14. Simpson, L.T.: Value-Driven Redundancy Elimination. PhD thesis, Rice University (May 1996)
15. Steffen, B., Knoop, J., Rüthing, O.: The value flow graph: A program representation for optimal program transformations. In: Jones, N.D. (ed.) ESOP 1990. LNCS, vol. 432, pp. 389–405. Springer, Heidelberg (1990)
16. van Drunen, T.J.: Partial redundancy elimination for global value numbering. PhD thesis, Purdue University, West Lafayette, IN, USA, Major Professor-Antony L. Hosking (2004)