Detection of Redundant Expressions: A Complete and Polynomial-Time Algorithm in SSA

Rekha R. Pai(✉)

National Institute of Technology Calicut, Calicut, Kerala, India
rekharamapai@nitc.ac.in

Abstract. Detection of redundant expressions in a program based on values is a well researched problem done with a view to eliminate the redundancies so as to improve the run-time efficiency of the program. The problem entails the detection of equivalent expressions in a program. Here we present an iterative data-flow analysis algorithm to detect equivalent expressions in SSA for the purpose of detection of redundancies. The central challenge in this static analysis is to define a "join" operation to detect all equivalences at a join point such that any later occurrences of redundant expressions are detected in polynomial time. We achieve this by introducing the notion of *value ϕ-function*. We claim the algorithm is complete and takes only polynomial time. We implemented the algorithm in LLVM and demonstrated its performance.

Keywords: Equivalence detection · Global value numbering · Redundancy detection · Value ϕ-function

1 Introduction

Elimination of redundant expressions in a program, based on values, is an important code optimization done with a view to improve run-time efficiency of a program. The fundamental problem here is the detection of equivalent expressions in the program. The detection of all equivalences in a program is undecidable and hence we focus only on the detection of Herbrand equivalences [8], as is done traditionally. Two expressions are *Herbrand equivalent* if they have the same operator and corresponding operands are Herbrand equivalent.

Equivalences are detected by assigning *value numbers* to each expression. The value number v_i is assigned to two expressions if they are detected to be equivalent [3]. Global Value Numbering (GVN) is the problem of assigning value numbers to expressions to detect equivalences in whole programs. Efforts in the literature have been to propose a GVN algorithm which is both complete and efficient. A GVN algorithm is "complete" if it detects all Herbrand equivalences such that all associated total redundancies are detected.

Current GVN algorithms are either complete [5] or take only polynomial time [2–4,6,8–10], but not both, in the context of detection of redundancies. As in a

X. Feng and S. Park (Eds.): APLAS 2015, LNCS 9458, pp. 49–65, 2015.
DOI: 10.1007/978-3-319-26529-2_4

data-flow analysis, the central challenge in GVN is to define a "join" operation to detect equivalences at a *join* point. Though the detection of all equivalences at a join point makes a GVN algorithm complete, it blows up the size of partition of equivalent expressions thus making the algorithm inefficient [5].

In order to make a GVN algorithm polynomial, a solution is to detect only those equivalences at a point p' that may be used later at a point p where an expression, say e, appears. Here, we view the solution from a different perspective. Instead of detecting equivalences that may be used later, we propose that given an expression e at a point p in the program, detect whether e is equivalent to some expression(s) e' that appear in paths to e. For this we use semantics of ϕ-function in Static Single Assignment (SSA) form and introduce the new concept of *value ϕ-function* which is a set of equivalent ϕ-functions. We then propose an iterative data-flow analysis algorithm to detect equivalences in SSA form of programs which is complete and takes only polynomial time. We later prove the soundness and completeness of the algorithm.

We implemented the proposed algorithm and the algorithms by Kildall [5] and Gulwani [4] in LLVM to substantiate our claims on completeness and efficiency. The SPEC2006 programs were analyzed using the three algorithms and experimental results demonstrate that the proposed algorithm is complete as it detects same number of redundancies as the complete algorithm by Kildall. The proposed algorithm is also efficient compared to the widely accepted Gulwani's polynomial time algorithm since it takes less time to analyze the SPEC2006 programs.

The rest of the paper is organized as follows: in Sect. 2 we analyze two classic GVN algorithms to get a clarity on the problems in global value numbering. The terms used in this paper are given in Sect. 3. *Value ϕ-function* and the new algorithm are described in Sect. 4. The algorithm is formally defined in Sect. 5 and an experimental comparison of our algorithm with Kildall's [5] and Gulwani's [4] is made in Sect. 6. In Sect. 7 we review some of the algorithms in the literature. Section 8 concludes the work.

2 Motivation

In this section we analyze the classic works by Kildall [5] and Gulwani [4] to understand the problems in the detection of equivalent expressions. The algorithm by Kildall is complete and the one by Gulwani takes only polynomial time.

2.1 Kildall's Algorithm

The iterative data-flow analysis algorithm by Kildall detects equivalences at each point in the program. The equivalences are represented as a partition of expressions into equivalence classes, known as *expression pool*. The algorithm uses a powerful concept known as "structuring" in its transfer function. When a new equivalence class is created in an expression pool corresponding to an expression e in the program, the algorithm structures the partition by the construction

and addition of all expressions (Herbrand) equivalent to e in the new class. This ensures detection of all redundant expressions which means that the algorithm is complete. But this leads to an exponential growth in the size of an equivalence class. The use of value numbers, as given in 'Implementation Notes' section in [5], avoids this problem. Kildall uses *value expression*, a compact representation for a set of equivalent expressions [5,9] to make the size of an equivalence class linear. But the problem of exponential growth in the size of expression pools (expressed in terms of number of equivalence classes) persists due to the definition of join operation as shown by [4]. This is because the join operation applied on n_j input pools may result in an expression pool whose size is exponential in the size of the input pools [4].

2.2 Gulwani's Algorithm

This algorithm works similar to that of Kildall's with equivalence information represented as a directed graph known as *Strong Equivalence DAG* (SED). The SED provides a compact representation of equivalence classes in a partition. The algorithm detects equivalences among all expressions of size at most s, where s is the size of program expression. This reduces the number of equivalence classes in a partition computed by join operation (compared with Kildall's) which makes it take only polynomial time. The join operation as defined in Gulwani (see Sect. 3.5, JOIN algorithm, lines 3–5 in [4]) intersects classes only if they have at least one variable in common. This leads to missing in the detection of some equivalences that will be useful in detecting redundancies [9].

2.3 Our View

The central problem in GVN is to define a *join* operation to detect equivalences at a join point. Detecting all equivalent expressions at a point makes the algorithm exponential. A solution, to overcome this problem, is to detect only those equivalent expressions at a point that are used to detect later occurrences of a redundant expression. To the best of our knowledge, currently there are no methods to precisely predict whether such redundant expressions might appear or not. Here we propose to view the problem from a completely different perspective. Instead of detecting equivalent expressions at a join point j that is used later, we postpone detection of such equivalences till a point where an expression actually occurs.

3 Terminology

Program Representation. The program in SSA is represented as a Control Flow Graph (CFG) [1] that has an empty *entry* and *exit* block. Other blocks contain assignment statements of the form $x = e$, where e is an expression. We assume a block can have at most two predecessors and a block with exactly two predecessors is called *join* block. The input and output points of a block are called *in* and *out* points, respectively, of the block.

Expression. An *expression* can be either a constant, a variable, or of the form $x \oplus y$ where x and y are constants or variables and $\oplus$ is a generic binary operator. An expression can also be of the form $\phi_k(x, y)$ where x and y are variables and k is the join block in which it appears. Such expressions are *ϕ-functions*. We may omit the subscript k when the join block is clear from the context. In the CFGs we draw, ϕ-functions appear in join blocks. But for the sake of clarity, we assume ϕ-functions are transformed to *copy* statements[1] and appended to appropriate predecessors of the join block.

Equivalence. Two expressions e_1 and e_2 are *Herbrand equivalent*, denoted $e_1 \equiv e_2$, if they have the same operator and corresponding operands are Herbrand equivalent. Two expressions e_1 and e_2 in a path $\mathcal{P}$ are said to be *equivalent in the path*, denoted $e_1 \equiv_{\mathcal{P}} e_2$, if they are Herbrand equivalent in that path.

Value Expression. A *value expression* $v_i \oplus v_j$ represents an operation between two equivalent classes where v_i and v_j are the value numbers of the two equivalent classes. $v_i \oplus v_j = \{x \oplus y; x \in C_i$, equivalent class with value number v_i and $y \in C_j$, equivalent class with value number $v_j\}$. A value expression is a representative expression of the set of equivalent expressions. Value expression of an expression $x \oplus y$ is constructed by replacing the operands with their value numbers.

4 Basic Concept

Our goal is to develop a complete and polynomial time algorithm for redundancy detection. The cause of redundancy is the equivalence of expressions in a program and hence detection of redundancies can be stated as a problem of computation of equivalence classes of expressions at each point in the CFG. The problem can be formally stated as: *given an expression e at a point p detect whether there are expressions e' in each path to p such that e' and e are equivalent in that path.* Here the concept of *value ϕ-function* is introduced for the purpose. In this section we first explain value ϕ-function and then propose our method to detect redundancies.

4.1 Value ϕ-function

Consider the code segment in Fig. 1. Depending on the path taken expression $x_3 + 1$ is equivalent to either $x_1 + 1$ or $x_2 + 1$. In other words, depending on the path taken, variable w_3 is equivalent to one of variables y_1 and z_2. That is, w_3 can be viewed as equivalent to the "merge of different variables" – y_1 and z_2 – at the join point, denoted $\phi(y_1, z_2)$. This kind of a "merge of different variables" can be seen as an extended form of the ϕ-function in the literature[2]. We use this extended notion of ϕ-function or "merge of different variables" to

[1] A *copy* statement is an assignment statement of the form $x = y$, where y is a variable.

[2] In the literature, a ϕ-function restricts its operands to different subscripted versions of the same non-SSA variable, say $\phi(x_1, x_2)$.

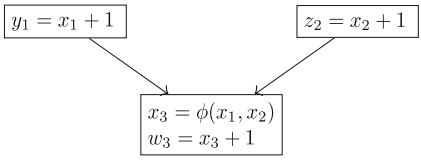

Fig. 1. Program with branches

express equivalences in such cases. Similar to the concept of value expression, we define the concept of *value ϕ-function* as an abstraction of a set of equivalent ϕ-functions.

Partition. A *partition* at a point represents equivalences that hold in the paths to the point. An equivalence class in the partition has a value number and elements like variables, constant, and value expression. It is also annotated with a value ϕ-function when necessary. For example, in a partition $\{\cdots | v_r, x_1, y_1 | v_s, z_1, v_r + 1 | v_m, x_n : \phi_k(v_i, v_j) | \cdots\}$ at a point p the class with value number v_r represents equivalence among variables x_1 and y_1. The class with value number v_s represents equivalence among expressions represented by value expression $v_r + 1$, that is $x_1 + 1$ and $y_1 + 1$. The expressions are also equivalent to variable z_1. From the class with value number v_m we can infer that variable x_n is equivalent to expressions with value number v_i in the path to point p through left edge to join block k. Also, x_n is equivalent to expressions with value number v_j in the path to p through right edge to the join block. Note that the value ϕ-function $\phi_k(v_i, v_j)$ appears as the last element in the class and is separated from the rest of the elements by ":" symbol to indicate that the value ϕ-function is an annotation of the class.

4.2 Proposed Method

Using the concept of value ϕ-function we propose an iterative data-flow analysis algorithm to compute equivalences at each point in the program. The two main components of this algorithm are *join* operation and *transfer function.*

Join Operation. A *join* operation detects equivalences that are common in all paths to the join point. Since in SSA there is only one definition for a variable, equivalences that hold at a point p, which dominates[3] join point j, hold at the join point. These equivalences are detected at the join point by doing a simple class-wise intersection of partitions. However, the detection of some common equivalences that are generated in branches require extra processing and we illustrate the latter. For clarity we separate the cases of detection of equivalences among variables from those among expressions-with-operators.

[3] Point p in a CFG *dominates* point p' if all paths from *entry* point to p' go through p.

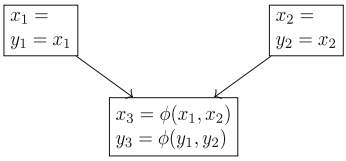

Fig. 2. Detecting equivalence of variables

Equivalence of Variables. Consider the code segment in Fig. 2. In the left path to the join point, variables x_1 and y_1, defined in the left branch, are equivalent. Similarly, x_2 and y_2, defined in the right branch, are equivalent in right path. By the use of ϕ-functions that appears in the join block we can detect that x_1 is equivalent to x_3 in the left path and x_2 is equivalent to x_3 in the right path. Similarly, y_1 is equivalent to y_3 in the left path and y_2 is equivalent to y_3 in the right path. By transitivity of equivalence relation, we conclude that x_3 and y_3 are equivalent at the join point.

In other words, our join operation merge different variables (corresponding to the same non-SSA variable) defined in different branches - x_1 with x_2 and y_1 with y_2 - to obtain x_3 and y_3, respectively, at the join point. We then conclude variables x_3 and y_3 are equivalent at the join point; in addition, they are equivalent to ϕ-functions $\phi(x_1, x_2)$ and $\phi(y_1, y_2)$. The detection of such equivalences are done by the transformation of ϕ-functions to copy statements which are then appended to appropriate predecessors of the join block.

Equivalence of Expressions-with-Operators. Now consider the code segment in Fig. 3. The expressions $x_1 + 1$ and $x_2 + 1$, that appear in different branches are merged[4] to obtain $x_3 + 1$ at the join point. By doing this "merge" we detect the equivalence of $x_1 + 1$ (and $x_2 + 1$) with $x_3 + 1$ if it appears at a point in some path from join point. However, this merge can be avoided if $x_3 + 1$ (or some expression equivalent to it) does not appear.

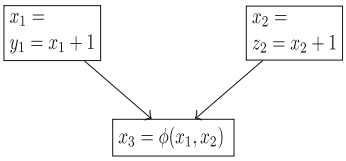

Fig. 3. Detecting equivalence of expressions-with-operators

[4] Merge of expressions can be viewed as an extended notion of merge of variables. "Merge of expressions" $e_{i1} + e_{i2}$ and $e_{j1} + e_{j2}$ is the expression $e_i + e_j$ such that e_i is the merge of e_{i1} and e_{j1}. Similarly, e_j is the merge of e_{i2} and e_{j2}.

As discussed in Sect. 2 the question is whether to merge different expressions at join points to detect equivalences. Here we take a completely different approach and the join operation does not merge the expressions at the join point. Instead merge operation is deferred till the occurrence of $x_3 + 1$ (or some expression equivalent to it). This is discussed when the concept of transfer function is explained.

Example. Consider the case of application of join on partitions $P_1 = \{v_1, x_1, x_3 | v_2, y_1, y_3, v_1 + 1 | v_3, z_1, z_3\}$ and $P_2 = \{v_4, x_2, x_3 | v_5, y_2, y_3 | v_6, z_2, z_3, v_4 + 1\}$. In the classes with value numbers v_1 in P_1 and v_4 in P_2 there is only one common variable x_3 and it appears in a class in the resulting partition P_3. Since the two classes in P_1 and P_2 have different value numbers v_1 and v_4, respectively, we can infer that x_3 is actually a merge of variables. Hence the resulting class is annotated with value ϕ-function $\phi(v_1, v_4)$. The class is assigned a new value number, say v_7. The resulting class is $|v_7, x_3 : \phi(v_1, v_4)|$.

Now consider the classes with value numbers v_2 in P_1 and v_6 in P_2. There are no obvious common equivalences in the classes; however we can infer from the partitions that the different value expressions $v_1 + 1$ in P_1 and $v_4 + 1$ in P_2 actually represent common equivalences of $x_3 + 1$, which is a merge of different expressions[5] $x_1 + 1$ and $x_2 + 1$. But as stated above the different expressions (or different value expressions to be precise) are not merged now and hence no new class is created in the resulting partition P_3.

Similar strategies are adopted to detect common equivalences in other pairs of classes one each from P_1 and P_2. The resulting partition P_3 is $\{v_7, x_3 : \phi(v_1, v_4) | v_8, y_3 : \phi(v_2, v_5) | v_9, z_3 : \phi(v_3, v_6)\}$.

Transfer Function. Based on the equivalences that hold at *in* point of a statement $s : x = e$, transfer function for the statement defines the equivalences that hold at its *out* point. This involves detection of whether the expression e is equivalent to expression e' in each path to it. Accordingly, the transfer function computes partition at *out* point, denoted $POUT_s$ (from that at *in* point, denoted PIN_s) by updating an existing class or creating a new class. The transfer function we define here is similar to the ones in the literature except that it uses value ϕ-function to detect equivalences in each path to e. The first step is to check partition PIN_s for existence of value expression of e. If not found, the transfer function proceeds to check whether expression e could be expressed as a "merge of different expressions". This is illustrated below using the code segment in Fig. 4.

Consider processing the last statement $w_3 = x_3 + 1$. Since value expression $v_7 + 1$ of expression $x_3 + 1$ does not appear in PIN_3, the transfer function proceeds to check whether $x_3 + 1$ could be expressed as a merge of expressions as follows:

[5] Since x_3 is a merge of variables x_1 and x_2, expression $x_3 + 1$ is a merge of $x_1 + 1$ and $x_2 + 1$.

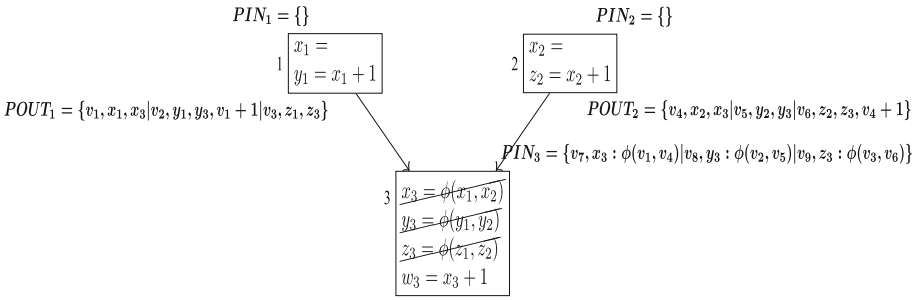

Fig. 4. Concept of transfer function

$$
\begin{aligned}
x_3 + 1 &\equiv v_7 + 1 && \text{// value expression of } x_3 + 1 \text{ computed using } PIN_3 \\
&\equiv \phi(v_1, v_4) + 1 && \text{// class with } v_7 \text{ in } PIN_3 \text{ is annotated with } \phi(v_1, v_4) \\
&\equiv \phi(v_1 + 1, v_4 + 1) && \text{// using semantics of value } \phi\text{-function} \\
&\equiv \phi(v_2, v_6) && \text{// } v_1 + 1 \in POUT_1 \text{ having } v_2, \text{ where block 1 is left} \\
& && \text{predecessor of join block and } v_4 + 1 \in POUT_2 \\
& && \text{having } v_6, \text{ where block 2 is the right predecessor.}
\end{aligned}
$$

Thus using value ϕ-function we detect that $x_3 + 1$ is actually equivalent to expression(s) with value number v_2, when the left path is considered in isolation. Similarly, $x_3 + 1$ is equivalent to expression(s) with value number v_6, when the right path is considered in isolation. That is, $x_3 + 1$ is a "merge of expressions", here $x_1 + 1$ and $x_2 + 1$. In terms of variables, w_3 is merge of different variables, here y_1 and z_2.

Since neither value expression $v_7 + 1$ nor value ϕ-function $\phi(v_2, v_6)$ is present in PIN_3, the transfer function creates a new class in $POUT_3$ with new value number, say v_{10}, and add w_3 and $v_7 + 1$ to it. The class is also annotated with value ϕ-function $\phi(v_2, v_6)$. The classes in PIN_3 are added as such to $POUT_3$. The resulting partition $POUT_3$ is $\{v_7, x_3 : \phi(v_1, v_4) | v_8, y_3 : \phi(v_2, v_5) | v_9, z_3 : \phi(v_3, v_6) | v_{10}, w_3, v_7 + 1 : \phi(v_2, v_6)\}$.

Detection of Redundancies. Expression e in statement $x = e$ is redundant if there exists expression e' equivalent to e in each path to the statement. In terms of variables, this implies x is equivalent to some variable y irrespective of path taken or to the merge of different variables. Given partition $POUT$ at *out* of statement $x = e$, expression e is redundant if there exists a variable in the class of x in $POUT$, other than x, or the class of x in $POUT$ is annotated with value ϕ-function.

In the example code in Fig. 4, redundancy of expression $x_3 + 1$ in the last statement $w_3 = x_3 + 1$ is detected since the class of w_3 in $POUT_3$ (computed in the previous subsection) is annotated with a value ϕ-function.

5 Algorithm

Here we formally present the iterative data-flow analysis algorithm to detect equivalences at each point in the program. The two main components of this algorithm are join operation and transfer function which are defined below. The algorithm to detect redundancies is trivial and is not written here.

5.1 Join

The algorithm JOIN given below defines the join operation. Before the join operation is performed, the ϕ-functions in a join block are transformed to copy statements and appended to appropriate predecessors of the join block. Transfer function is then applied on these copies.

JOIN(P_1, P_2)
  $P = \{\}$
  for each pair of classes $C_i \in P_1$ and $C_j \in P_2$
    $C_k =$ INTERSECT(C_i, C_j)
    $P = P \cup C_k$ // Ignore when C_k is empty
  return P

INTERSECT(C_i, C_j)
  $C_k = C_i \cap C_j$ // set intersection
  if $C_k \neq \{\}$ and C_k does not have value number
    then $C_k = C_k \cup \{v_k\}$ // v_k is new value number
      $C_k = (C_k - \{vpf\}) \cup \{\phi_b(v_i, v_j)\}$
      // *vpf* is value ϕ-function in C_k, $v_i \in C_i$, $v_j \in C_j$, b is join block
  return C_k

Lemma 1. *If $e_1 \equiv e_2$ at a point p and the point p dominates join point j then $e_1 \equiv e_2$ at j iff the* JOIN *algorithm detects their equivalence.*

Lemma 2. *If variable $x \equiv y$ in each path to join point j then $x \equiv y$ at j iff the* JOIN *algorithm detects their equivalence.*

5.2 Transfer Function

Given a partition PIN_s at *in* of a statement s, the transfer function for the statement[6] computes the partition $POUT_s$ at its *out* point and is defined below. The transfer function uses the function VALUEEXPR which accepts an expression e and returns value expression of e, if e is of the form $x \oplus y$, otherwise returns e itself. The function VALUEPHIFUNC accepts value expression and a partition and returns value ϕ-function if the expressions represented by the value expression

[6] Transfer function for a block is the composition of transfer function of each statement in the block [1].

are a merge of expressions. Otherwise it returns NULL. This function assumes partitions at *out* of each block are accessible to it. The concept of this function is given below and the detailed algorithm is in the appendix.

```
transferFunction(x = e, PIN_s)
    POUT_s = PIN_s
    if x is in a class C_i in POUT_s
        then C_i = C_i − {x}
    ve = valueExpr(e)
    vpf = valuePhiFunc(ve, PIN_s)                    // can be NULL
    if ve or vpf is in a class C_i in POUT_s         // ignore vpf when NULL
        then C_i = C_i ∪ {x, ve}                     // set union
        else POUT_s = POUT_s ∪ {v_n, x, ve : vpf}    // v_n is new value number
    return POUT_s

valuePhiFunc(ve, P)
    if ve is of the form φ_k(v_i1, v_j1) ⊕ φ_k(v_i2, v_j2)
        then v_i = getVN(POUT_k_l, v_i1 ⊕ v_i2)
             if (v_i == NULL)
                 then v_i = valuePhiFunc(v_i1 ⊕ v_i2, POUT_k_l)
             v_j = getVN(POUT_k_r, v_j1 ⊕ v_j2)
             if (v_j == NULL)
                 then v_j = valuePhiFunc(v_j1 ⊕ v_j2, POUT_k_r)
    return φ_k(v_i, v_j)      // v_i, v_j are non-NULL
```

Lemma 3. *Let $x = e$ be a statement at a point p in the program and there exist expressions e_i at points p_i in each path to p such that at least one of the p_i's does not dominate p. Then expression e has a value ϕ-function, as computed by* valuePhiFunc *algorithm, iff expressions e_i and e are equivalent in respective paths.*

Lemma 4. *Let $x = e$ be a statement at a point p in the program and there exist expressions e_i in each path to p. Expressions e_i and e are equivalent in their respective paths iff the* transferFunction *algorithm detects the equivalences.*

5.3 The Iterative Algorithm

The algorithm detectEquivalences given below analyzes the program (represented as a CFG G) and computes partitions of equivalent expressions at each point in the program. The iterative analysis method is adapted from [1]. The algorithm initializes *out* point of each statement (except first statement) with partition $\top$ (*top*). $\top$ is a special partition with the property $\text{Join}(P, \top) = P = \text{Join}(\top, P)$. The algorithm iteratively computes partitions at each point till there are no changes in the equivalences detected (from the previous iteration).

```
detectEquivalences(G)
    PIN_1 = {}                          // "1" is the first statement in the program
    POUT_1 = transferFunction(PIN_1)
    for each statement s other than the first statement in the program
        POUT_s = ⊤
    while changes to any POUT occur     // i.e. changes in equivalences
        for each statement s other than the first statement in the program
            if s appears in block b that has two predecessors
                then PIN_s = Join(POUT_s', POUT_s'')[a]
                else PIN_s = POUT_s'
            POUT_s = transferFunction(PIN_s)[b]
```

[a] s' and s'' are last statements in respective predecessors.

[b] s' is the statement just before s.

Theorem 1 (Soundness and Completeness). *Let P be a partition at a point p computed by the iterative data-flow analysis algorithm. Two expressions are equivalent at p iff the algorithm detects their equivalence.*

An outline of the correctness proofs of the algorithms are given in appendix.

5.4 Complexity Analysis

Let there be n number of expressions in a program. By definitions of Join and transferFunction a partition can have $O(n)$ classes with each class of $O(v)$ size, where v is the number of variables and constants in the program. The join operation is class-wise intersection of partitions. With efficient data structure that supports lookup, intersection of each class takes $O(v)$ time. With a total of n^2 such intersections, a join takes $O(n^2.v)$ time. If there are j join points, the total time taken by all the join operations in an iteration is $O(n^2.v.j)$. The transfer function involves construction and lookup of value expression or value ϕ-function in the input partition. A value expression is computed and searched for in $O(n)$ time. Computation of value ϕ-function for an expression $x+y$ essentially involves lookup of value expressions, recursively, in partitions at left and right predecessors of a join block. If a lookup table is maintained to map value expressions to value ϕ-functions (or NULL when a value expression does not have a value ϕ-function), then computation of a value ϕ-function can be done in $O(n.j)$ time. Thus transfer function of a statement $x = e$ takes $O(n.j)$ time. In a program with n expressions total time taken by all the transfer functions in an iteration is $O(n^2.j)$. Thus the time taken by all the joins and transfer functions in an iteration is $O(n^2.v.j)$. As shown in [4], in the worst case the iterative analysis takes n iterations and hence the total time taken by the analysis is $O(n^3.v.j)$.

6 Implementation and Results

In this section we compare the new algorithm with the algorithms by Kildall [5] and Gulwani [4]. We chose Kildall's algorithm since it is complete and the widely

accepted Gulwani's algorithm was chosen since it takes only polynomial time. The three iterative data-flow analysis algorithms compute equivalence information at each point in the program. We implemented the algorithms in LLVM*v*3.4 compiler with *clang* as front end. The implementations consider all arithmetic operations, conversion operations, vector operations and aggregate operations, while to simplify the implementations, memory and branch operations were ignored. The implementations uses the *llvm::DenseMap*, *llvm::SmallPtrSet* and *llvm::SmallVector* classes to define the partitions, equivalence classes and value expressions. Instances of partitions are associated with *in* and *out* points of each instruction. The input to the implementations are in SSA form of LLVM-IR. Since Kildall's and Gulwani's algorithm work on non-SSA form of programs, we modified the algorithms to process ϕ-functions. ϕ-functions are transformed to copy statements and appended to predecessors of the join blocks. The implementations were compared using SPEC2006 programs and the results were obtained on 2 GHz Intel Xeon processor with 8 GB RAM running Ubuntu 12.04.

Table 1. Number of redundancies detected by Gulwani, Kildall, and our algorithm

CINT2006	Gulwani	Kildall	Proposed	Improvement(%)
mcf	32	36	36	12.5
astar	130	153	153	17.7
libquantum	210	259	259	23.3
bzip2	580	691	691	19.1
sjeng	1141	1265	1265	10.9
hmmer	3810	4204	4204	10.3
gobmk	8907	10005	10005	12.3
h264ref	8982	10216	10216	13.7
gcc	19837	23300	23300	17.5
CFP2006	Gulwani	Kildall	Proposed	Improvement(%)
milc	775	867	867	11.9
sphinx3	827	919	919	11.1
lbm	1085	1169	1169	07.7
soplex	2685	3022	3022	12.6
povray	3319	3623	3623	09.2

Table 1 shows the number of redundancies detected in SPEC2006 CINT and CFP C/C++ programs using Gulwani, Kildall, and the new algorithm. The table also gives the percentage improvement made by the new algorithm in detecting redundancies over Gulwani. The results show that the proposed algorithm detects same number of redundancies as the complete algorithm by Kildall thus demonstrating completeness. Also both these algorithms detect more redundancies when compared to Gulwani's with an average improvement of 14.2 %.

The figures indicate that there can be statements, say of the form $z = x \oplus y$, in real programs such that variable z is equivalent to different variables in different paths to the statement. Detection of redundancy of $x \oplus y$ is missed by Gulwani while both Kildall and the new algorithm could capture it.

Table 2. Time taken (in seconds) to analyze the input SPEC2006 programs along with their size (when converted to LLVM-IR SSA form)

CINT2006	Size of program		Time for analysis		
	#joins	#instructions	Kildall	Gulwani	Proposed
mcf	171	1815	2.5961	0.8520	0.4917
libquantum	277	5045	7.5244	1.8921	1.1035
astar	450	6586	13.7687	4.1121	2.0936
bzip2	814	13346	66.4680	9.3012	6.3841
sjeng	1874	18658	119.0993	15.7408	9.5835
hmmer	3279	48387	203.3485	34.6138	30.2571
gobmk	9754	105994	361.5141	49.1068	45.5976
h264ref	6804	116253	358.3743	70.6684	67.6074
gcc	45861	605303	750.6864	110.2226	98.3966
CFP2006	#joins	#instructions	Kildall	Gulwani	Proposed
lbm	55	3773	7.6245	3.9202	1.4973
milc	1103	18867	30.4560	8.7964	5.5625
sphinx3	1836	22929	62.6717	22.4852	18.4850
soplex	3206	48513	136.0443	25.3210	21.4148
povray	8349	128305	320.8518	79.0802	74.7350

To show the efficiency of our algorithm, we measured the CPU time taken to perform the analyses during compilation of the SPEC2006 programs. The times were measured using *-ftime-report* option of *clang*. Table 2 gives the time taken (in seconds) by the implementations to analyze the SPEC programs. The table also gives the number of join blocks and instructions considered to indicate the number of join operation and transfer functions being applied on the partitions.

Table 2 shows that the new algorithm takes less time to analyze the SPEC programs than Gulwani's polynomial time algorithm. This we believe is because Gulwani recursively intersects equivalence classes (that has at least one variable in common) to detect equivalent expressions at a join point (see Sect. 3.5, Join algorithm, Lines 3–5 in [4]). However, the proposed algorithm does only a simple intersection of equivalence classes. Equivalences in paths to an expression is detected only when needed by computing value ϕ-function. Both these algorithms take considerably less time than Kildall's exponential time algorithm. The join operation in Kildall is similar to that in Gulwani except that, in Kildall's, the join operation recursively intersects equivalence classes even if there

are no common variables in the classes which makes it least efficient among the three algorithms.

The results in the tables clearly demonstrate that the proposed algorithm is complete as it detects the same number of redundancies as the complete algorithm by Kildall. Also the algorithm is efficient when compared to the polynomial time algorithm by Gulwani as it takes less time. Since more redundancies can be detected by the proposed algorithm in comparatively less time, the algorithm may be used in redundancy elimination algorithms that aid in the generation of faster code.

7 Related Work

The seminal work on GVN by Kildall [5] detects equivalences at each point in the program using an iterative data-flow analysis algorithm. This algorithm uses "structuring" of partitions of equivalent expressions, which makes it complete. However, structuring of partitions blows up its size and hence affects efficiency of the algorithm. The strive to improve efficiency in the detection of equivalences motivated the algorithm by Alpern and others (referred to as AWZ algorithm) [2] which works on Static Single Assignment (SSA) form of programs and uses the concept of *congruence*. The algorithm though efficient is less precise than Kildall's, one of the reasons being that it does not interpret ϕ*-functions*. Rüthing, Knoop, and Steffen [8] improves on AWZ in terms of the number of equivalences detected by using *normalization rules*. These normalization rules essentially interpret the ϕ-functions. The algorithm is efficient but not complete, as proved by Gulwani [4]. The *Dominator-based value numbering* algorithm by Briggs and others [3] works on SSA form. The algorithm is not complete as it makes pessimistic assumptions about loops in programs. The *SCC-based Value Numbering* algorithm by Simpson [10] considers semantics of operators to improve on AWZ. However the algorithm has similar issues as AWZ since it does not interpret ϕ-functions. The GVN algorithms in SSA by VanDrunen [11] and Odaira [7] detect and eliminate a broader class of partial redundancies and not just total redundancies. The polynomial time algorithm by Gulwani and Necula [4] is claimed to detect all equivalences among expressions of a particular size. However, some of the redundancies could not be detected using this GVN algorithm [9]. Nie proposed an SSA version of Gulwani's algorithm [6]. In general, the algorithms are either complete or take only polynomial time but not both.

8 Conclusion

Detection of equivalent expressions in a program is a static analysis aimed at elimination of redundant expressions. The fundamental problem here is the detection of equivalences at each point in the program such that all redundancies are detected in polynomial time. For this we introduced the novel concept of value ϕ-function. We then presented an iterative data-flow analysis algorithm which uses value ϕ-function to detect equivalences. We showed that the algorithm is complete and

takes only polynomial time. Moreover, we implemented our algorithm and compared it with two widely accepted GVN algorithms in the literature. The experimental results demonstrate that the proposed algorithm is complete and efficient.

Acknowledgements. We thank Vineeth Paleri, Muralikrishnan K, Vinith R, and the anonymous reviewers for their insightful comments.

A Appendix

A.1 valuePhiFunc

This recursive function computes value ϕ-function of a given value expression. The function assumes partitions at *out* of each block is available to it. The function uses equiVE to replace operands of a given value expression with equivalent value ϕ-functions, whenever possible. Else it returns the value expression as such. The getVN function used here takes a partition at *out* of either the left or right predecessor of a join block k. It searches for the input value expression in the partition and returns its value number, if present. If the partition was searched for previously then the function returns a new value number. This case can arise with loops in the program.

valuePhiFunc(ve, P)
  $v_i = v_j = vpf$ = NULL
  ve' = equiVE(ve, P)
  if ve' is of the form $\phi_k(v_{i1}, v_{j1}) + \phi_k(v_{i2}, v_{j2})$
    then v_i = getVN($POUT_{\mathrm{k_l}}, v_{i1} + v_{i2}$)
      v_j = getVN($POUT_{\mathrm{k_r}}, v_{j1} + v_{j2}$)
      if v_i == NULL
        then v_i = valuePhiFunc($v_{i1} + v_{i2}, POUT_{\mathrm{k_l}}$)
      if v_j == NULL
        then v_j = valuePhiFunc($v_{j1} + v_{j2}, POUT_{\mathrm{k_r}}$)
    elseif ve' is of the form $\phi_k(v_{i1}, v_{j1}) + v_m$ [a]
      then v_i = getVN($POUT_{\mathrm{k_l}}, v_{i1} + v_m$)
        v_j = getVN($POUT_{\mathrm{k_r}}, v_{j1} + v_m$)
        if v_i == NULL
          then v_i = valuePhiFunc($v_{i1} + v_m, POUT_{\mathrm{k_l}}$)
        if v_j == NULL
          then v_j = valuePhiFunc($v_{j1} + v_m, POUT_{\mathrm{k_r}}$)
      elseif ve' is of the form $v_m + \phi_k(v_{i2}, v_{j2})$
        then v_i = getVN($POUT_{\mathrm{k_l}}, v_m + v_{i2}$)
          v_j = getVN($POUT_{\mathrm{k_r}}, v_m + v_{j2}$)
          if v_i == NULL
            then v_i = valuePhiFunc($v_m + v_{i2}, POUT_{\mathrm{k_l}}$)
          if v_j == NULL
            then v_j = valuePhiFunc($v_m + v_{j2}, POUT_{\mathrm{k_r}}$)
  if $v_i \wedge v_j$ // both are non-NULL
    then $vpf = \phi_k(v_i, v_j)$
  return vpf

[a] class with value number v_m does not have value ϕ-function or has $\phi_r(v_s, v_t)$ such that block r *dominates* k.

A.2 Proof

Correctness of JOIN Algorithm

Lemma 1. *If $e_1 \equiv e_2$ at a point p and the point p dominates join point j then $e_1 \equiv e_2$ at j iff the algorithm detects their equivalence.*

Proof. Let expressions e_1 and e_2 be equivalent at a point p such that p dominates join point j. Since a variable is defined only once in SSA the expressions are equivalent in each path to j. Line 1 in the algorithm INTERSECT ensures such common equivalences are detected at the join point. □

Lemma 2. *If variable $x \equiv y$ in each path to join point j then $x \equiv y$ at j iff the algorithm detects their equivalence.*

Proof. Let two variables x and y be equivalent in each path to join point j. Then by suitably transforming the ϕ-functions in the join block j and by line 1 of the INTERSECT algorithm such equivalences could also be detected. □

Let there be expressions e_i in each path to an expression e and $e_i \equiv e$ in respective paths. The equivalences are detected by the TRANSFERFUNCTION algorithm which is proved below.

Correctness of TRANSFERFUNCTION Algorithm

Lemma 3. *Let $x = e$ be a statement at a point p in the program and there exist expressions e_i at points p_i in each path to p such that at least one of the p_i's does not dominate p. Then expression e has a value ϕ-function, as computed by* VALUEPHIFUNC *algorithm, iff expressions e_i and e are equivalent in respective paths.*

Proof. This can be proved by induction on the number of join points in paths with the base case similar to that in Fig. 4. □

Lemma 4. *Let $x = e$ be a statement at a point p in the program and there exists expressions e_i in each path to p. Expressions e_i and e are equivalent in respective paths iff the* TRANSFERFUNCTION *algorithm detects the equivalences.*

Proof. Let the expression(s) e_i appear at point p' such that p' dominate p. Then an equivalence class for e_i with its value expression will appear in the partition at p' (ensured by lines 7 and 8 in the algorithm). Since a variable is defined only once in SSA the partition at *in* point of the statement $x = e$ will have a class with the value expression of e_i. Then line 6 in the algorithm ensures equivalence of e_i and e is detected.

Now consider the case where an expression e_i appear at a point p' such that p' does not dominate p. In this case computation of value ϕ-function in line 5 (Lemma 3) and subsequent check for its existence in line 6 ensures detection of equivalences of e_i and e in respective paths. □

Correctness of Iterative Data-Flow Analysis Algorithm

Theorem 1 (Soundness and Completeness). *Let P be a partition at a point p computed by the iterative data-flow analysis algorithm. Two expressions are equivalent at p iff the algorithm detects their equivalence.*

Proof This follows from Lemmas 1, 2, and 4. □

Correctness of Algorithm for Detection of Redundancies

Theorem 2 (Soundness and Completeness). *Let $s : z = x + y$ be a statement at a point p. The expression $x + y$ is redundant iff the algorithm detects its redundancy.*

Proof This follows from Theorem 1. □

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison Wesley, Boston (2006)
2. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 1–11. ACM, New York (1988)
3. Briggs, P., Cooper, K., Simpson, L.: Value numbering. Software: Practice and Experience 27(6), 701–724 (1997)
4. 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)
5. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1973, pp. 194–206. ACM, New York (1973)
6. Nie, J.-T., Cheng, X.: An efficient SSA-based algorithm for complete global value numbering. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 319–334. Springer, Heidelberg (2007)
7. 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)
8. 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)
9. Saleena, N., Paleri, V.: Global value numbering for redundancy detection: a simple and efficient algorithm. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC 2014, pp. 1609–1611. ACM, New York (2014)
10. Simpson, L.T.: Value-driven redundancy elimination. Ph.D. thesis, Rice University, Houston, TX, USA (1996)
11. VanDrunen, T., Hosking, A.L.: Value-based partial redundancy elimination. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 167–184. Springer, Heidelberg (2004)