# Value-Based Partial Redundancy Elimination

Thomas VanDrunen and Antony L. Hosking

Purdue University

**Abstract.** Partial redundancy elimination (PRE) is a program transformation that identifies and eliminates expressions that are redundant on at least one (but not necessarily all) execution paths. Global value numbering (GVN) is a program analysis and transformation that identifies operations that compute the same value and eliminates operations that are redundant. A weakness of PRE is that it traditionally considers only expressions that are lexically equivalent. A weakness of GVN is that it traditionally considers only operations that are *fully* redundant. In this paper, we examine the work that has been done on PRE and GVN and present a hybrid algorithm that combines the strengths of each. The contributions of this work are a framework for thinking about expressions and values without source-level lexical constraints, a system of data-flow equations for determining insertion points, and a practical algorithm for extending a simple hash-based GVN for PRE. Our implementation subsumes GVN statically and, on most benchmarks, in terms of performance.

## 1 Introduction

The goal of a compiler's optimization phases is transforming the program to make it more efficient. Accordingly, researchers have given much attention to methods for eliminating redundant computations; both approaches and terminology to describe them have proliferated. In the progeny of this research, two primary genera can be identified (some cross-breeding not withstanding): Partial Redundancy Elimination (PRE) and Global Value Numbering (GVN). The goal of the present work is to present an algorithm for a hybrid subsuming these two approaches synthesized from known techniques and tools.

### 1.1 Motivation

PRE considers the control flow of the program and identifies operations that are redundant on *some but not necessarily all* traces of the program that include that operation, and hence are *partially redundant*. It hoists operations to earlier program points where originally the operation was unknown and removes operations that the hoisting has rendered fully redundant. In Figure 1(a), $e \leftarrow c + b$ is partially redundant because of $d \leftarrow c + b$. By hoisting and preserving the result of the operation in temporary variable $t$, PRE produces the program in Figure 1(b). Because of this hoisting, PRE is sometimes called (an instance of) *code*

E. Duesterwald (Ed.): CC 2004, LNCS 2985, pp. 167–184, 2004.

*motion.* GVN on the other hand considers the value produced by an operation (in this paper, we always assume *value* to mean static value; operations in loop bodies or otherwise reentrant code may result in different dynamic values at different points in program execution). If a value has already been computed, subsequent operations can be eliminated even though they may differ lexically. Because of the move $c \leftarrow a$ in Figure 1(a), the operations $c \leftarrow b+a$ and $d \leftarrow c+b$ compute the same value. Accordingly GVN preserves this value in a temporary and eliminates the re-computation, as in Figure 1(c). As this example shows, in their basic form, neither PRE nor GVN is strictly more powerful than the other (see Muchnick p. 343 [25]).

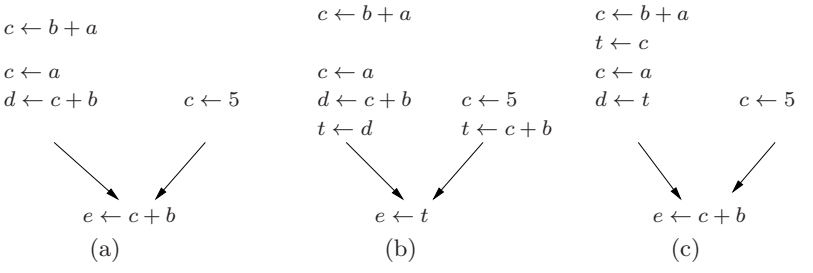



**Fig. 1.** Basic example

### 1.2 Our Result

Answering this challenge, we have found a technique, GVN-PRE, that subsumes both and eliminates redundancies that would be undetected by either working in isolation. We create a system of data flow equations that, using the infrastructure of a simple hash-based value numbering system, calculate insertion points for partially redundant values rather than partially redundant expressions. We present this in a framework that allows clear reasoning about expressions and values and implement it with an algorithm that can easily be reproduced. After exploring the background to this problem and related work, we explain the framework, give the data flow equations and related algorithm, and explore an implementation and its performance. Our implementation subsumes GVN.

### 1.3 Related Work

***PRE.*** Partial Redundancy Elimination in general poses several challenges. First, lexically identical expressions at different program points do not necessarily compute the same result, since operands may be reassigned, so any PRE algorithm must take into account assignments that kill an expression's availability. Furthermore, care must be taken that the hoisting would do no harm: if an

operation is added at a program point, it must occur without reassignment to its operands on all paths from that point to program exit (in the literature, they say the operation must be *downsafe* or *anticipated* at that point). Otherwise the hoisting will lengthen at least one trace of the program, defying optimality; even worse, if the hoisted instruction throws an exception, the program's semantics change. Finally, PRE algorithms should resist the temptation to hoist operations earlier in the program than necessary; while this may not do any harm in terms of lengthening a trace of the program, it will make no improvement either, but may increase register pressure by lengthening the live range of an assignment.

PRE was invented by Morel and Renoise [24]. The original formulation used bit vectors to detect candidates for motion and elimination. Knoop, Rüthing, and Steffen gave a complete, formal, and provably optimal version of PRE which they called Lazy Code Motion (LCM) [19,20]; it was improved by Drechsler and Stadel [13]. LCM used a myriad of predicates and equation systems to determine the earliest and latest placement points for operations that should be hoisted. Chow, Kennedy et al. produced a PRE algorithm for programs in static single assignment form (SSA) [11], called SSAPRE [8,17]. SSA is an intermediate representation property such that program variables and temporaries are divided into different versions for each assignment so that each version of a variable is assigned to exactly once in a static view of the program. If a variable exists in different versions on incoming paths at a join point, they are merged into a new version at the join. SSA makes it easy to identify the live ranges of variable assignments and hence which lexically equivalent expressions are also semantically equivalent. On the other hand, it complicates hoisting since any operand defined by a merge of variable versions must have the earlier version back-substituted. More recently, Dhamdhere has presented a simpler PRE that examines specific paths from earlier to later occurrences of an operation [12].

A major handicap for these versions of PRE is that they are based on lexical (or syntactic) equivalences. Although SSA provides the freedom to think outside the lexical box because each assignment essentially becomes its own variable, even SSAPRE does not take advantage of this but regards expressions in terms of the source variables from which their operands come; in fact, it makes stronger assumptions about the namespace than basic SSA [34]. The fact that many redundant results in a program do not come from lexically identical expressions (nor do all lexically identical expressions produce the same results) has motivated research to make PRE more effective [5] and led Click to assert that "in practice, GVN finds a larger set of congruences than PRE" [9].

***GVN.*** Global Value Numbering partitions expressions and variables into classes (or assigns them a unique *value number*) all of which have the same static value (or are *congruent*). A well-known form having its origin in two papers simultaneously presented by Alpern, Rosen, Wegman, and Zadek [3,28], it uses algebraic properties to associate expressions, such as assignments ($a \leftarrow b$ implies $a$ and $b$ are in the same class), commutativity of certain operations ($a + b$ is in the same class as $b + a$), and facts about certain constants ($a + 0$ is in the same class as $a$). SSA is a major boon for GVN, since (SSA versions of) variables are always in the

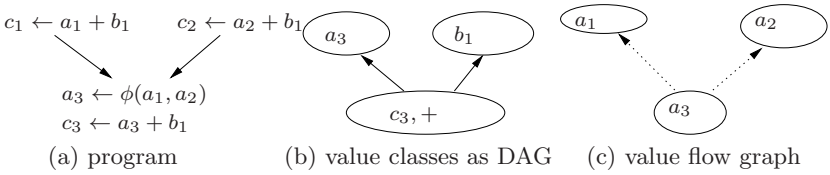



**Fig. 2.** Example showing weakness of traditional GVN and the use of a VFG

same class, and therefore so are expressions on those versions in the same class, statically and globally. Congruences can be found by hashing [9,6] or by first assuming all expressions to be equal and splitting classes when incongruences are found [3,16,30]. Since identifying an operation as redundant depends on the operation having the same *global* number as an earlier operation, in traditional GVN such an earlier operation must dominate (i.e., occur on all paths to) the later operation. However, if any given path is considered in isolation, some value classes can be merged. In Figure 2(a), $\phi(a_1, a_2)$ merges the variables $a_1$ and $a_2$, so if we consider the left path, $a_3$ should be in the same class as $a_1$, whereas on the right path, $a_2$ should be in the same class as $a_1$. Yet the operation $a_3 + b_1$ will not be eliminated because nothing in its global class has been computed earlier. Since traditional GVN does no motion, it does not eliminate some operations that on any given trace of the program are fully redundant—let alone operations that are partially redundant. (Another tradition of global value numbering has been worked on by Steffen et al [31,32,33,29,22]. Its various versions have tended to be more general and theoretically complete but less amenable to implementation than that of Rosen, Alpern, et al.)

***Hybrids.*** One of the original GVN papers [28] is remarkable because it not only introduces GVN, SSA, and critical edge removal; it also does PRE on values. However, the range of congruences it can identify is small, and subsequent research seems to have ignored this as it wallowed in its lexical mire. The first explicit study of the potential for PRE to work on values (or for GVN to consider partial redundancies) is that of Knoop, Rüthing, and Steffen's comparison of code motion and code placement [21]. They distinguish *syntactic* (lexical) code motion from *semantic*, which considers values. They note that the collection of value classes can be modeled by a dag, since a value is represented by an expression (a node) which has other values as operands (edges pointing to nodes). The path-based connections among values is captured by what they call the *value flow graph* (VFG). See Figure 2(b) and (c). Knoop et al. use these constructs to sketch an algorithm for determining safe insertion points for code motion (PRE), but distinguish it from a more powerful optimization which they call code placement and for which determining insertion points traditionally considered safe is not adequate for finding all insertion points necessary for optimality.

Bodík and Anik, independently of Knoop et al., developed an algorithm that also addressed PRE on values [4]. Their work uses value numbering with back

substitution to construct a *value name graph* (VNG) similar to Knoop et al's VFG. The VNG considers expressions to be names of values. If the program is not in SSA, expressions represent different values at different program points, and the nodes of the graph are expression / program-point pairs. The edges capture the flow of values from one name to another according to program control flow. Bodík et al. then use data flow analysis to identify optimal insertion points for PRE, an algorithm they call VNGPRE.

### 1.4 Overview

In this paper, we present a new algorithm for PRE on values which subsumes both traditional PRE and GVN. The contributions of our approach are as follows. First, our analysis considers larger program chunks than Bodík (basic blocks instead of single instructions). Second, we present a framework for expressions and values that takes full advantage of SSA, completely ignoring source-level lexical constraints and thinking solely in terms of values and expressions that represent them. Third, no graph is explicitly constructed nor is any novel structure introduced. Instead, we achieve the same results by synthesizing well-known tools (SSA, control flow graphs, value numbers) and techniques (flow equations, liveness analysis, fixed-point iteration). In fact, our algorithm can be viewed as an extension of a simple hash-based GVN. Finally, we report on an efficient and easily-reproduced implementation for this algorithm. Our intention here is not to make a head-on comparison with the approaches of Bodík et al. or Knoop et al., but rather to present an approach that covers both traditional PRE and GVN that is easily realized in implementation.

The rest of the paper is organized as follows: Section 2 gives preliminaries, such as assumptions we make about the input program, precise definitions of concepts we use for our analysis, and descriptions of global value numbering infrastructure we assume already available. Section 3 contains the meat of the paper, describing the various phases of the approach both formally and algorithmically. Section 4 gives details on our implementation and its results. We conclude by discussing future work in Section 5.

## 2 Framework

In this section, we present our framework in terms of program structure we assume and the model of expressions and values we use.

### 2.1 Program Structure

We assume the input programs are in an intermediate representation that uses a *control flow graph* (CFG) over *basic blocks* [1]. A basic block is a code segment that has no unconditional jump or conditional branch statements except for possibly the last statement, and none of its statements, except for possibly the first, is a target of any jump or branch statement. A CFG is a graph representation

of a procedure that has basic blocks for nodes and whose edges represent the possible execution paths determined by jump and branch statements. We define $\mathsf{succ}(b)$ to be the set of successors to basic block $b$ in the CFG, and similarly $\mathsf{pred}(b)$ the set of predecessors. (When these sets contain only one element, we use this notation to stand for that element for convenience.) We also define $\mathsf{dom}(b)$ to be the dominator of $b$, the nearest block that dominates $b$. This relationship can be modeled by a *dominator tree*, which we assume to be constructed [1]. We assume that all *critical edges*—edges from blocks with more than one successor to blocks with more than one predecessor [28]—have been removed from the CFG by inserting an empty block between the two blocks connected by the critical edge.

We assume the following language to define the set of possible instructions in the program (this excludes jumps and branches, which for our purposes are modeled by the graph itself):

$$
\begin{array}{ll}
k ::= t \mid t\ \mathsf{op}\ s & \textit{Operations} \\
p ::= \phi(t^*) & \textit{Phis} \\
\gamma ::= k \mid p \mid \bullet & \textit{Right-hand terms} \\
i ::= t \leftarrow \gamma & \textit{Instructions} \\
b ::= i^* & \textit{Basic blocks}
\end{array}
$$

The symbol $\bullet$ stands for any operation which we are not considering for this optimization; it is considered to be a black box which produces a result. The meta-variable $t$ ranges over (SSA) variables. We do not distinguish between source-level variables and compiler-generated temporaries, nor are we concerned with what source-level variable any SSA variable come from. Because of SSA form, no variable is reassigned, and a variable's scope is all program points dominated by its definition. For simplicity, we do not include constants in the grammar, though they appear in some examples; constants may be considered globally-defined temporaries. We let $\mathsf{op}$ range over operators.

A phi contains the same number of operands as there are predecessors to its block, and each operand is associated with a distinct predecessor. Semantically, the phi selects the operand associated with the predecessor that precedes the block on a given trace and returns that operand as its result. Phis precede all non-phis in a block.

### 2.2 Values and Value-Numbering

A *value* is a set of expressions. $v$ ranges over values. An expression is similar to an operation except that if an expression involves an operator, its operands are given in terms of values rather than subexpressions. This way we can think of expressions more generally. If $t_1$ and $t_3$ are members of value $v_1$, and $t_2$ is a member of $v_2$, we need not think of $t_1 + t_2$ and $t_3 + t_2$ as separate expressions; instead, we think only of the expression $v_1 + v_2$.

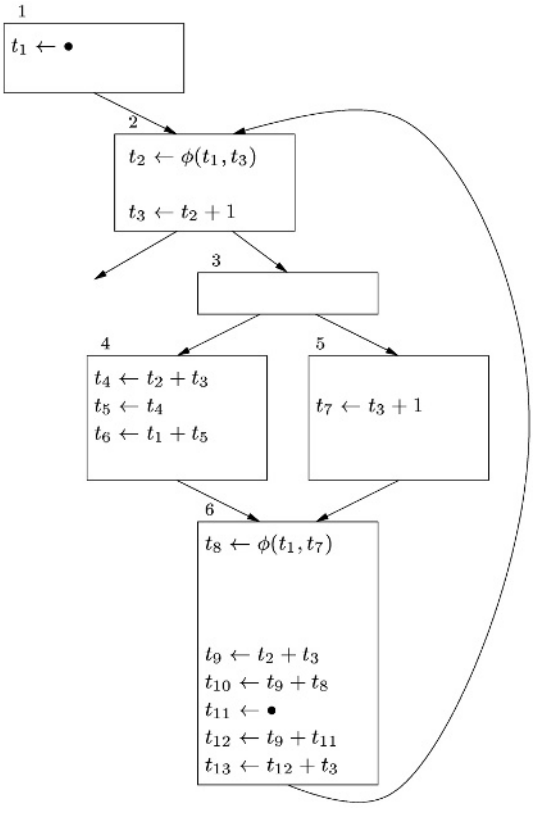
$$e ::= s \mid v \ \mathsf{op} \ v \qquad \textit{Expressions}$$

Our goal in value numbering is to obtain three pieces of information: value numbers, available expressions, and anticipated expressions. Traditional value numbering requires only the first two. First, we partition all expressions in the program into values. We represent this partition with a map (or value table) on which we can add and lookup the values for expressions. We treat the value table as a black box, but assume it to be used for a simple hashing value number scheme, recognizing expressions by their structure and, if smart enough, algebraic properties, associating them with values. It is worth noting that there is no limit on how smart the function lookup is engineered to be. For example, if $v_1$ contains 1, $v_2$ contains 2, $v_4$ contains $v_3 + v_1$ for some $v_3$, and $v_5$ contains $v_4 + v_1$, then $v_5$ also should contain $v_3 + v_2$. To enumerate all such cases would be infinite.

Figure 3(a) displays a running example we will refer to throughout this and the next section. Since it is large, we will often discuss only part of it at a time. For the present, consider block 4. The table in Figure 3(b) displays the values referenced in the block and the expressions they contain. For clarity, we assign a value the same subscript as one of its temporaries wherever possible. Values $v_1$, $v_2$, and $v_3$ are defined in blocks that dominate block 4, so they are in scope here. The instruction $t_4 \leftarrow t_2 + t_3$ leads us to discover expressions $v_2 + v_3$ and $t_4$ as elements of the same value, which we call $v_4$. Because of the move $t_5 \leftarrow t_4$, $t_5$ is also in $v_4$. Finally, $v_1 + v_5$ and $t_6$ are in a value we call $v_6$. Recall that these values are global, so we are showing only part of the value table for the entire program (in fact, values $v_3$ and $v_4$ have more members, discovered in other parts of the program).

The second piece of information is what expressions are available to represent the values; or, put differently, what values are already computed and stored in temporaries at a given program point. If more than one temporary of the same value is live at a point, we want to pick one as the leader, which will be used as the source of the value to replace an operation of that value with a move. To be unambiguous, the leader should be the "earliest" temporary available; that is, the temporary whose defining instruction dominates the defining instructions of all other temporaries of that value live at that point. We define the *leader set* to be the set of leaders representing available values at a program point. Although technically there is a leader set for any program point, for our purposes we will be interested only in the sets at the end of a basic block, AVAIL_OUT. The leaders available out of block 4 are those expressions listed first for each value in Figure 3(b). (This notion of availability is sometimes called *upsafety* [21].)

Finally, we want to know what values are *anticipated* at a program point; that is, it will be computed or used on all paths from that point to program exit. (This notion of anticipation—sometimes called anticability— is essentially the same as that of downsafety.) Just as we want appropriate temporaries to represent values in the leader set, so we want appropriate expressions to represent anticipated values—that is, anticipated values should also have a reverse-flow leader or *antileader*. An antileader can be a live temporary or a non-simple

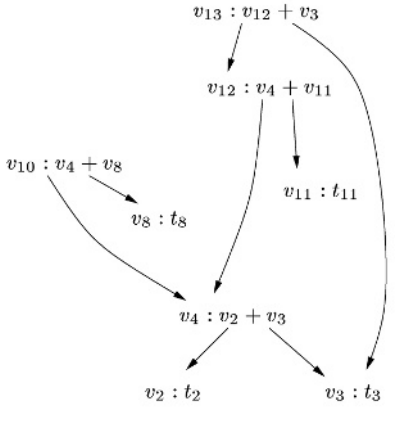



(a) Program CFG

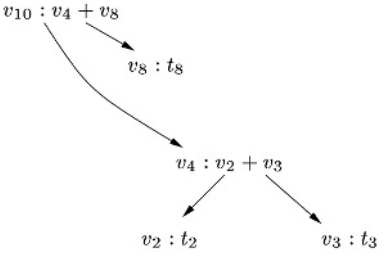
**values**
$v_1$: $t_1$
$v_2$: $t_2$
$v_3$: $t_3$
$v_4$: $t_4, v_2 + v_3, t_5$
$v_6$: $t_6, v_1 + v_4$

**temps generated:**
$t_4, t_5, t_6$
**exprs generated:**
$t_2, t_3, t_4, t_1, t_5, v_2 + v_3, v_1 + v_4$

(b) Values, temporaries, and expressions of block 4




(c) Anticipated expressions at block 6, not considering the kill of $t_{11}$




(d) Actual anticipated expressions at block 6

**Fig. 3.** Running example

expression whose operands are represented in the antileader set. This is so that the expression could become available by an insertion at that point, or several insertions if it is a nested expression. Conversely from the leader set, we are interested only in the antileader sets at the beginning of basic blocks, ANTIC_IN.

For an antileader set, it does not matter which expression represents a value, so long as that value is live. A temporary potentially in ANTIC_IN becomes dead if it is assigned to. If the assignment is from something we can make an expression for (as opposed to •), that expression replaces the temporary as the antileader. If the assignment is from •, then the value is no longer represented at all. Furthermore, any other expression that has that (no longer represented) value as an operand also becomes dead. Therefore antileaders and the values they represent are killed by definitions of temporaries in the block. An antileader set can be pictured as a dag. Consider basic block 6 in Figure 3(a), alternately with and without the instruction $t_4 \leftarrow \bullet$. In the case where we exclude that instruction, assume $t_4$ to be global. Without the definition of $t_4$, ANTIC_IN can be represented by the dag in Figure 3(c). The nodes of the dag are pairs of values and the antileaders representing them; edges are determined by the operands of the antileader. If we suppose the block contains $t_6 \leftarrow \bullet$ as it does in the program, then $v_4 : t_4$ is killed, along with all expressions that depend on $v_4$ also, in cascading effect. See Figure 3(d).

To calculate these sets, we use two other pieces of information: the temporaries generated by a set, meaning those that are defined; and the expressions generated by a set, meaning those that occur as operations, operands to operations, or sources of moves. We say that the set "generates" these because they are used as "gen" sets for the flow equations we define in the next section for the leader and antileader sets. These are given for block 4 in Figure 3(b).

## 3 GVN-PRE

The GVN-PRE algorithm has three steps: BuildSets, Insert, and Eliminate. BuildSets, which we describe formally and algorithmically, populates the value table and the leader and antileader sets. Insert places new instructions in the program to make partially available instructions fully available. Eliminate removes computations whose values are already available in temporaries or as constants.

### 3.1 BuildSets

***Flow equations.*** To compute AVAIL_IN and AVAIL_OUT for a block, we must consider not only the contents of the block itself, but also expressions inherited from predecessors (for AVAIL_OUT) and anti-inherited from successors (for ANTIC_IN). For this we use a system of flow equations. As is common for flow equations, we also define the sets AVAIL_IN and ANTIC_OUT, although only AVAIL_OUT and ANTIC_IN are used later in the optimization. We have three gen sets, EXP_GEN($b$) for expressions (temporaries and non-simple) that appear in the right hand side of an instruction in $b$; PHI_GEN($b$) for temporaries that

are defined by a phi in $b$; and $\mathsf{TMP\_GEN}(b)$ for temporaries that are defined by non-phi instructions in $b$. There are no kill sets for calculating $\mathsf{AVAIL\_OUT}$, since SSA form implies no temporary is ever killed. For $\mathsf{ANTIC\_IN}$, $\mathsf{TMP\_GEN}$ acts as a kill set. Since an available expression must be defined in an instruction that dominates the program point in question, so in calculating $\mathsf{AVAIL\_OUT}$, we consider inherited expressions only from the block's dominator. In terms of flow equations,

$$\mathsf{AVAIL\_IN}[b] = \mathsf{AVAIL\_OUT}[\mathsf{dom}(b)] \tag{1}$$

$$\mathsf{AVAIL\_OUT}[b] = \mathsf{canon}(\mathsf{AVAIL\_IN}[b] \cup \mathsf{PHI\_GEN}(b) \cup \mathsf{TMP\_GEN}(b)) \tag{2}$$

where $\mathsf{canon}$ is a procedure that, given a set of temporaries, partitions that set into subsets which all have the same value and chooses a leader from each. Of course $\mathsf{canon}$ would be inconvenient and inefficient to implement; instead, $\mathsf{AVAIL\_OUT}[b]$ can be calculated easily and efficiently at the same time as the gen sets, as we will show later.

For $\mathsf{ANTIC\_IN}$, handling successors is more complicated. If there is only one successor, we add all its antileaders to $\mathsf{AVAIL\_OUT}$ of the current block; however, we must translate some temporaries based on the phis at the successor. For example, if $t_1$ is anticipated by block $b$, and block $b$ has a phi which defines $t_1$ and has $t_2$ as an operand from block $c$, then $t_2$, rather than $t_1$, is anticipated at the end of block $c$, and it has a different value. For this we assume a function $\mathsf{phi\_translate}$ which takes a successor block, a predecessor block (i.e., there is an edge from the second block to the first), and a temporary; if the temporary is defined by a phi at the successor, it returns the operand to that phi corresponding to the predecessor, otherwise returning the temporary. If there are multiple successors, then there can be no phis (because of critical edge removal), but only values anticipated by all of the successors can be anticipated by the current block. This assures that all anticipated expressions are downsafe—they can be inserted with assurance that they will be used on all paths to program exit.

The flow equations for calculating the antileader sets are

$$\mathsf{ANTIC\_OUT}[b] = \begin{cases} \{e | e \in \mathsf{ANTIC\_IN}[\mathsf{succ}_0(b)] \wedge \\ \quad \forall\, b' \in \mathsf{succ}(b), \exists e' \in \mathsf{ANTIC\_IN}[b'] | \\ \quad \mathsf{lookup}(e) = \mathsf{lookup}(e')\} & \text{if } |\mathsf{succ}(b)| > 1 \\ \\ \mathsf{phi\_translate}(A[\mathsf{succ}(b)], b, \mathsf{succ}(b)) & \text{if } |\mathsf{succ}(b)| = 1 \end{cases} \tag{3}$$

$$\mathsf{ANTIC\_IN}[b] = \mathsf{clean}(\mathsf{canon}_e(\mathsf{ANTIC\_OUT}[b] \cup \mathsf{EXP\_GEN}[b] - \mathsf{TMP\_GEN}(b))) \tag{4}$$

When $\mathsf{phi\_translate}$ translates expressions through phis, it may involve creating expressions that have not been assigned values yet and therefore require new values to be created. Consider calculating $\mathsf{ANTIC\_OUT}[B_5]$ in our example. Value $v_{10}$ is represented by $v_4 + v_8$, which is translated through the phi to $v_4 + v_7$. However, that expression does not exist in the program—it needs a new value.

Thus sometimes the value table will need to be modified while calculating these sets.

$\mathsf{canon}_e$ generalizes canon for expressions. For ANTIC_IN, we do not care what expression represents a value as long as it is live, so $\mathsf{canon}_e$ can make any choice as long as it is consistent. This essentially makes the union in the formula for ANTIC_IN to be "value-wise", meaning that given two sets of expressions representing values, we want every value in each set represented, but by only one expression each. Similarly, the formula for ANTIC_OUT when there are multiple successors performs a "value-wise" intersection—only values that are represented in all successors should be represented here, but which expression represents it does not matter. clean kills expressions that depend on values that have been (or should be) killed. Because information will also flow across back edges in the graph, these sets must be calculated using a fixed-point iteration.

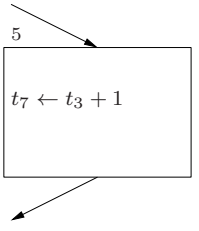
***Algorithm.*** Calculating BuildSets consists of two parts. The first is a top-down traversal of the dominator tree. At each block, we iterate forward over the instructions, making sure each expression has a value assigned to it. We also build EXP_GEN, PHI_GEN, and TMP_GEN for that block.

Computing canon directly is inconvenient and costly. We avoid computing it by maintaining an invariant on the relevant sets, that they never contain more than one expression for any value. Since the value leader set for the dominator will have already been determined, we can conveniently build the leader set for the current block by initializing it to the leader set of the dominator and, for each instruction, adding the target to the leader set *only if its value is not already represented*. This way, we never add a temporary to an AVAIL_OUT set unless its value is not yet represented. Similarly, we need only one representative in EXP_GEN for each value, so we do not add an expression for a value that has already appeared (this way EXP_GEN contains only the first appearance of a value, appropriate for an antileader). We do not calculate AVAIL_IN sets since it is trivial.

The second part calculates flow sets to determine the antileader sets and conducts the fixed-point iteration. Until we conduct a pass on which no ANTIC_IN set changes, we perform top-down traversals of the postdominator tree. This helps fast convergence since information flows backward over the CFG. To keep ANTIC_IN canonical, we remove the killed temporaries from ANTIC_OUT and EXP_GEN separately, and then do what amounts to a value-wise union.

Both phi_translate and clean process elements in such a way that other set elements that they depend on must be processed first. The key to doing this efficiently is to maintain all sets as topological sorts of the dags they model, which can be done by implementing the sets as linked lists. Figure 4 shows a round of this process on block 5, including ANTIC_IN$[B_5]$ before and after the insertions from $S$. Notice that the net effect is to replace $v_7 : t_7$ in ANTIC_OUT with $v_7 : v_3 + v_{100}$, and that although the order has changed, it is still a topological sort. For this to be efficient, the fixed point iteration must converge quickly; convergence depends primarily on CFG structure. In our benchmarks, the maximum number of iterations needed for any method being compiled was 55; that,

however, appears to represent an extreme outlier, as the second highest was 18. When measuring the maximum number of rounds needed for convergence of any method, 15 of our 20 benchmarks had a maximum less than 10, and we found that if we set 10 as the limit, there was no decrease in the number of operations eliminated in the elimination phase, implying that the extra rounds needed in extreme cases produce no useful data.




$$\begin{aligned}
\mathsf{TMP_GEN}[5] &: v_7 : t_7 \\
\mathsf{EXP_GEN}[5] &: v_3 : t_3, v_{100} : 1, v_7 : v_3 + v_{100} \\
\mathsf{ANTIC_OUT} &: v_2 : t_2, v_3 : t_3, v_4 : v_2 + v_3, v_7 : t_7, \\
&\quad\; v_{101} : v_4 + v_7 \\
S &: v_2 : t_2, v_3 : t_3, v_4 : v_2 + v_3, v_{101} : v_4 + v_7 \\
\mathsf{ANTIC_IN}[5]_{\text{orig}} &: v_3 : t_3, v_{100} : 1, v_7 : v_3 + v_{100} \\
\mathsf{ANTIC_IN}[5] &: v_3 : t_3, v_{100} : 1, v_7 : v_3 + v_{100}, v_2 : t_2, \\
&\quad\; v_4 : v_2 + v_3, v_{101} : v_4 + v_7
\end{aligned}$$

**Fig. 4.** Example for the second phase of BuildSets

### 3.2 Insert

Insert is concerned with hoisting expressions to earlier program points. If this phase were skipped, we would be left with a traditional global value numbering scheme. Insertions happen only at merge points. This phase iterates over blocks that have more than one predecessor and inspects all expressions anticipated there. For a non-simple expression, we consider the equivalent expressions in the predecessors. This requires some translation because of the phis at the block, for which we use phi_translate. We look up the value for this equivalent expression and find the leader. If there is a leader, then it is available. If the expression is available in at least one predecessor, then we insert it in predecessors where it is not available. Generating fresh temporaries, we perform the necessary insertions and create a phi to merge the predecessors' leaders.

We can see the results of Insert in Figure 5 at the join point of block 6. The anticipated expressions $v_2 + v_3$ and $v_4 + v_8$ are available from block 4, so $t_{16} \leftarrow t_2 + t_3$ and $t_{18} \leftarrow t_{16} + t_7$ are hoisted to block 5. Here the topological sort of $\mathsf{ANTIC\_IN}[B_6]$ comes in handy again, since these expressions are nested and $v_2 + v_3$ must be inserted first. Note that thus our algorithm handles *second order effects* without assigning ranks to expressions (compare Rosen et al. [28]). Appropriate phis are also inserted.

The hoisted operations and newly created phis imply new leaders for their values in the blocks where they are placed, and these leaders must be propagated to dominated blocks. This could be done by re-running BuildSets, but that is unnecessary and costly. Instead, we assume a map which associates blocks with sets of expressions which have been added to the value leader sets during Insert. Whenever we create a new computation or phi, we possibly make a new

value, and we at least create a new leader for that value in the given block. We update that block's leader set and its new set. Since this information should be propagated to other blocks which the new temporaries reach, for each block we also add all the expressions in its dominator's new set into the block's own leader set and new set. This also requires us to make Insert a top-down traversal of the dominator tree, so that a block's dominator is processed before the block itself.

What we have said so far, however, is not completely safe in the sense that it in some cases it will lengthen a computational path by inserting an instruction without allowing one to be eliminated by the next phase. Since $v_3 + v_{100}$ is also anticipated in at block 6, being anti-inherited from $t_3 \leftarrow t_2 + 1$ in block 2, we insert $t_{14} \leftarrow t_3 + 1$ in block 4. This does not allow any eliminations, but only lengthens any trace through block 4. The value is still not available at the instruction that triggered this in block 1 because block 1 was visited first, at which time it was available on no predecessors and therefore not hoisted. To fix this, we repeat the process until we make a pass where nothing new is added to any new set. On the next pass, $v_7 : t_{15}$ is available in block 6, so we hoist $t_1 + 1$ to block 1. In practice, Insert converges quickly. We have seen only one case where it required 3 rounds. On most benchmarks, the maximum number of required rounds for any method was 2, and on average it took only a single round. Note that we do not need predicates for determining latest and earliest insertion points, since insertions naturally float to the right place. Insertions made too late in the program will themselves become redundant and eliminated in the next phase.

### 3.3 Eliminate

Eliminate is straightforward. For any instruction, find the leader of the target's value. If it differs from that target, then there is a constant or an earlier-defined temporary with the same value. The current instruction can be replaced by a move from the leader to the target. The order in which we process this does not matter. The optimized program is shown in Figure 5.

***Corner cases.*** Certain extensions must be made to this algorithm for theoretical optimality (such as *code placement* in Knoop et al [21]). In practice, we have found them to yield no benefit, but they are described in a technical report [35].

## 4 Implementation and Results

We have proven the concept of this approach by implementing it in an open-source compiler; our implementation is not compared against other hybrid approaches which are difficult to implement, but is shown to subsume GVN. Our experiments use Jikes RVM [2,7,27], a virtual machine that executes Java class-files. We have implemented the algorithm described here as a compiler phase for the optimizing compiler and configured Jikes RVM version 2.3.0 to use the

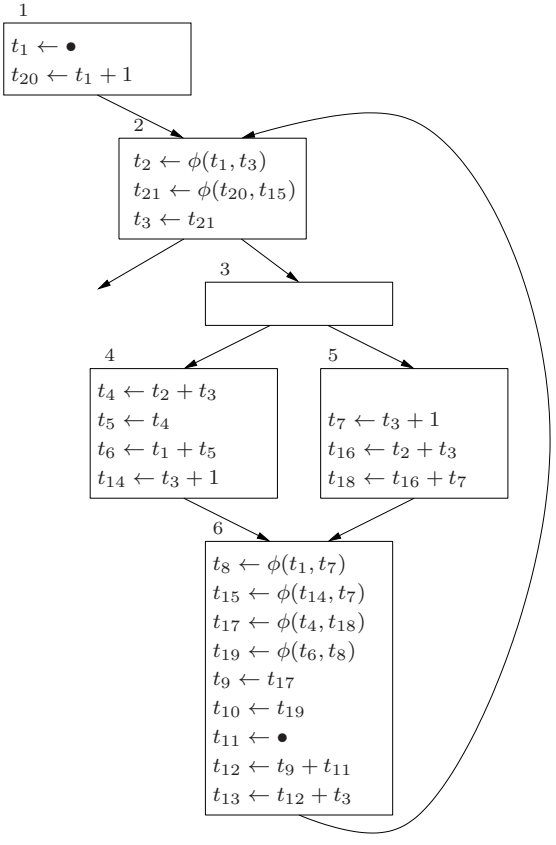



**Fig. 5.** The optimized program

optimizing compiler only and a copy mark-sweep garbage collector. The optimizing compiler performs a series of code transformations on both SSA and non-SSA representations. It already has loop-invariant code motion (LICM) and global common subexpression elimination (GCSE) phases in SSA, which rely on GVN. Placing our phase before these two shows that GVN-PRE completely subsumes GCSE in practice. LICM is not completely subsumed because Jikes RVM's LICM performs unsafe speculative motion which GVN-PRE does not. GCSE is equivalent to GVN as it is presented in this paper.

Figure 6 shows static eliminations performed by each optimization level. The left column in each set represents the number of intermediate representation operations eliminated by GCSE, and the second those eliminated by GVN-PRE. In each case, GVN-PRE eliminates more, sometimes twice as much, although the optimization also inserts operations and in some cases may later eliminate an operation it has inserted. The third bar shows the number GVN-PRE eliminations plus the number of operations GCSE can eliminate after GVN-PRE

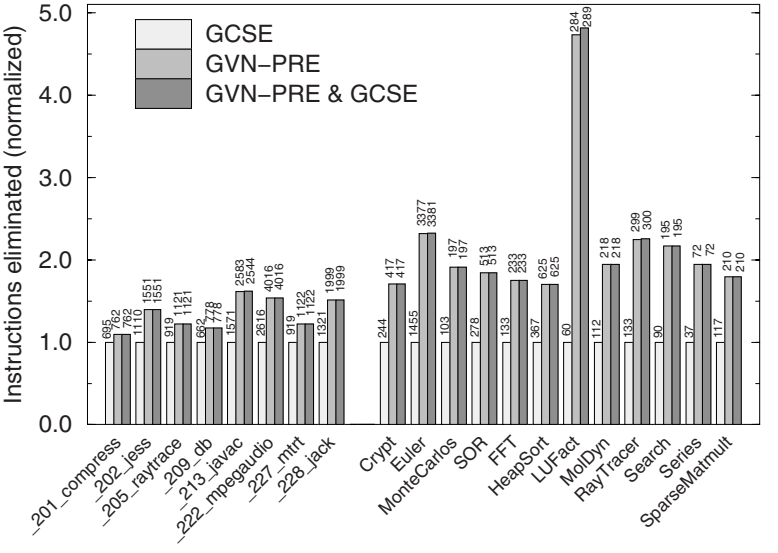



**Fig. 6.** Static results for the benchmarks

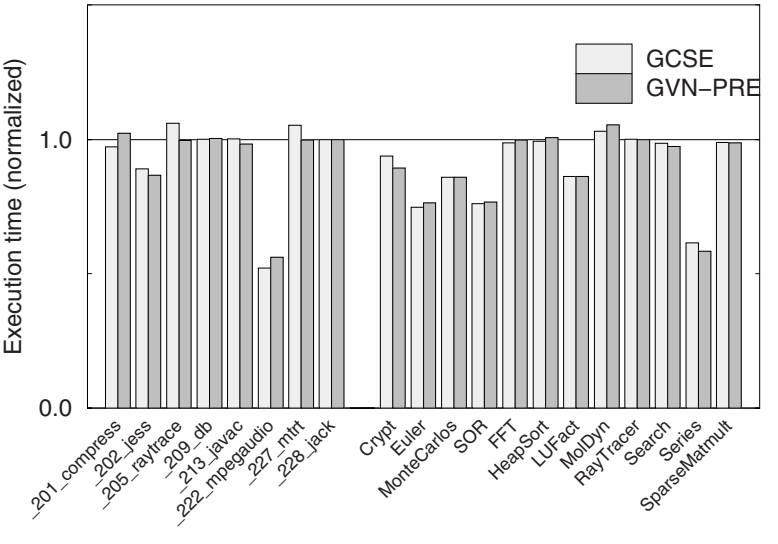



**Fig. 7.** Performance results for the benchmarks

has run. After GVN-PRE, GCSE finds zero in all but four cases, and never more than 6, compared to hundreds or thousands eliminated otherwise. Thus our optimization subsumes GCSE. All these are normalized to the first bar.

Our performance results are in Figure 7. These were obtained on a 1600MHz Intel Pentium 4 with 512 MB of RAM and 256 KB cache, running Red Hat Linux 3.2.2-5. We used three optimization levels: the pre-defined O2 with GCSE and GVN-PRE turned off, O2 with GCSE on but GVN-PRE off, and O2 with GCSE off but GVN-PRE on. In all cases, LICM and load elimination were turned off to isolate the impact of GCSE and GVN-PRE. We used twenty benchmarks in total, eight from the SPECjvm98 suite [10] and twelve from the sequential benchmarks of the Java Grande Forum [14]. For each benchmark at each optimization level we ran the program eleven times in the same invocation of the VM and timed the last nine runs using Java's `System.currentTimeMillis()` method. This way we did not incorporate compilation time in the measurements. We also subtracted garbage collection time. The graph shows the best running time of GCSE and GVN-PRE (out of the nine timed runs) normalized by the best running time of running with neither optimization. In only a few benchmarks is there significant performance gain, and sometimes these optimizations do no pay off. Studies have shown that in an environment like Jikes RVM, in order for an instruction elimination optimization to make a significant impact, it must address the removal of redundant loads and stores for objects and arrays [23], and we plan to extend our algorithm to incorporate those.

## 5 Conclusions and Future Work

We have presented an algorithm in which PRE and GVN are extended into a new approach that subsumes both, describing it formally as a dataflow problem and commenting on a practical implementation. This demonstrates that performing PRE as an extension of GVN can be done simply from a software engineering standpoint, and that it is feasible as a phase of an optimizing compiler. For future work, we plan to produce a version that will deliver real performance gains. Studies have shown that in an environment like Jikes RVM, in order for an instruction elimination optimization to make a significant impact, it must address the removal of redundant loads and stores for objects and arrays [23]. Accordingly, work to extend this algorithm for load and store instructions using Array SSA form [18,15] is underway.

**Acknowledgments.** The authors thank Di Ma for helpful conversation. We also thank the anonymous reviewers for their suggestions. This work is supported by the National Science Foundation under grants Nos. CCR-9711673, IIS-9988637, and CCR-0085792, by the Defense Advanced Research Program Agency, and by gifts from Sun Microsystems and IBM.

## References


1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—principles, techniques, and tools*. Addison Wesley, 1986.
2. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.
3. Bowen Alpern, Mark Wegman, and Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.
4. Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In Proceedings of the Symposium on Principles of Programming Languages [26], pages 237–251.
5. Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the Conference on Programming Language Design and Implemenation*, pages 159–170, Orlando, FL, June 1994. SIGPLAN.
6. Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software—Practice and Experience*, 27(6):701–724, 1997.
7. Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, M Serrano, V Sreedhar, H Srinvasan, and J Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.
8. Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the Conference on Programming Language Design and Implemenation*, pages 273–286, Las Vegas, Nevada, June 1997.
9. Cliff Click. Global code motion, global value numbering. In *Proceedings of the Conference on Programming Language Design and Implemenation*, pages 246–257, La Jolla, CA, June 1995. *SIGPLAN Notices* 30(6), June 1995.
10. Standard Performance Evalution Counsil. SPEC JVM 98 benchmarks, 1998. http://www.spec.org/osg/jvm98/.
11. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
12. Dhananjay M. Dhamdhere. E-path_pre—partial redundancy elimination made easy. *ACM SIGPLAN Notices*, 37(8):53–65, August 2002.
13. Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen's "lazy code motion". *ACM SIGPLAN Notices*, 28(5):29–38, May 1993.
14. EPCC. The java grande forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/index_1.html.
15. Stephen Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *Proceedings of the Static Analysis Symposium*, pages 155–174, Santa Barbara, California, July 2000.
16. Karthik Gargi. A sparse algorithm for predicated global value numbering. In *Proceedings of the Conference on Programming Language Design and Implemenation*, pages 45–56, Berlin, Germany, June 2002.
17. Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, May 1999.

18. Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in Parallelizaiton. In Proceedings of the Symposium on Principles of Programming Languages [26], pages 107–120.
19. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the Conference on Programming Language Design and Implemenation*, pages 224–234, San Francisco, CA, July 1992. *SIGPLAN Notices* 27(7), July 1992.
20. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.
21. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Code motion and code placement: Just synonyms? In *Proceedings of the 7th European Symposium on Programming (ESOP)*, pages 154–169, Lisbon, Portugal, 1998. LNCS 1381.
22. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Expansion-based removal of semantic partial redundancies. In *Proceedings of the 8th International Conference on Compiler Construction (CC)*, pages 91–106, Amsterdam, The Netherlands, 1999. LNCS 1575.
23. Han Lee, Amer Diwan, and J. Eliot B. Moss. Understanding the behavior of compiler optimizations. Submitted.
24. Etienne Morel and Claude Renvoise. Global optimization by supression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.
25. Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
26. *Proceedings of the Symposium on Principles of Programming Languages*, San Diego, California, January 1998.
27. IBM Research. The Jikes Research Virtual Machine. http://www-124.ibm.com/developerworks/oss/jikesrvm/.
28. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM Press, 1988.
29. Oliver Rüthing, Bernhard Steffen, and Jens Knoop. Detecting equalities of variables—combining efficiency with precision. In *Proceedings of the static analysis symposium (SAS)*, pages 232–247, 1999.
30. Loren Taylor Simpson. *Value-driven redundancy elimination*. PhD thesis, Rice University, 1996.
31. Bernhard Steffen. Optimal run-time optimization—proved by a new look at abstract interpretation. In *Proceedings of the Second International Joint conference on theory and practice of software development (TAPSOFT)*, pages 52–68, 1987. LNCS 249.
32. Bernhard Steffen, Jens Knoop, and Oliver Rüthing. The value flow graph—a program representation for optimal program transformations. In *Proceedings of the European Conference on Programming (ESOP)*, pages 389–405, 1990. LNCS 432.
33. Bernhard Steffen, Jens Knoop, and Oliver Rüthing. Efficient code motion and an adaptation to strength reduction. In *Proceedings of the sixth International Joint conference on theory and practice of software development (TAPSOFT)*, 1991. LNCS 494.
34. Thomas VanDrunen and Antony L Hosking. Anticipation-based partial redundancy elimination for static single assignment form. Submitted, 2003.
35. Thomas VanDrunen and Antony L Hosking. Corner cases in value-based partial redundancy elimination. Technical Report CSD-TR#03-032, Purdue University Department of Computer Sciences, 2003.