TECHNICAL REPORT 04-01

Partial Value Number Redundancy Elimination

Rei Odaira and Kei Hiraki

June 9, 2004

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TITLE

Partial Value Number Redundancy Elimination

AUTHORS

Rei Odaira and Kei Hiraki

KEY WORDS AND PHRASES

partial redundancy elimination, global value numbering, optimizing compiler, just-in-time compiler, runtime compiler, Java virtual machine

ABSTRACT

When developing a redundancy elimination algorithm for a runtime optimizing compiler, not only its optimizing power but also its analysis speed must be considered. We propose a fast and efficient algorithm called Partial Value Number Redundancy Elimination (PVNRE), which completely fuses Partial Redundancy Elimination (PRE) and Global Value Numbering (GVN). Using value numbers in the data-flow analyses, PVNRE can deal with data-dependent redundancy, and can quickly remove path-dependent partial redundancy by converting value numbers at join nodes *on demand* during the data-flow analyses. Compared with the naive combination of GVN, PRE, and copy propagation, PVNRE has a maximum 45% faster analyses speed, but the same optimizing power on SPECjvm98.

REPORT DATE June 9, 2004	**WRITTEN LANGUAGE** English
TOTAL NO. OF PAGES 24	**NO. OF REFERENCES** 19

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

DISTRIBUTION STATEMENT

First issue 30 copies.

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Partial Value Number Redundancy Elimination

Rei Odaira and Kei Hiraki,
Department of Computer Science,
Graduate School of Information Science and Technology,
University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo, Japan
{ray, hiraki}@is.s.u-tokyo.ac.jp

Abstract

When developing a redundancy elimination algorithm for a runtime optimizing compiler, not only its optimizing power but also its analysis speed must be considered. We propose a fast and efficient algorithm called Partial Value Number Redundancy Elimination (PVNRE), which completely fuses Partial Redundancy Elimination (PRE) and Global Value Numbering (GVN). Using value numbers in the data-flow analyses, PVNRE can deal with data-dependent redundancy, and can quickly remove path-dependent partial redundancy by converting value numbers at join nodes *on demand* during the data-flow analyses.

Compared with the naive combination of GVN, PRE, and copy propagation, PVNRE has a maximum 45% faster analyses speed, but the same optimizing power on SPECjvm98.

1 Introduction

Runtime optimizing compilers have recently gained widespread use in Java and other execution environments. A runtime optimizing compiler needs to keep analysis time short, but at the same time, it must reduce the execution time of generated code. Thus, the development of fast and efficient advanced optimizations has become increasingly important. In this paper, we aim to manage both optimizing power and analysis speed of redundancy elimination, an optimization that is frequently used in modern optimizing compilers [15].

Redundancy elimination is an optimizing technique that removes instructions that compute the same value as previously executed instructions. Partial Redundancy Elimination (PRE) [2, 3, 4, 12, 13, 14] and Global Value Numbering (GVN) [1, 5, 7, 8] are the most widely used redundancy elimination algorithms.

PRE can eliminate redundancy on at least one (but not necessarily all) execution path leading to an instruction (that is, partial redundancy). It deals with lexically identical instructions between which there is no store for any of their operand variables. To remove the redundancy it has to solve three data-flow equations for availability (*AVAIL*) and anticipatability (*ANTIC*) of instructions.

On the other hand, GVN can remove instructions that compute the same value on all paths even when they are lexically different. GVN uses value numbering, which is an algorithm to detect redundancy by assigning the same *value number* to a group of instructions that can be proved to compute the same value by static analysis. One GVN variants called *the bottom-up method* uses a hash table to assign value numbers to instructions [15]. It first transforms the whole program into the Static Single Assignment (SSA) form [9] and then searches the hash table, using an operator for each instruction and the value numbers of its operands as a key to the table. If the key has already been registered, a redundant instruction is then found. If not, it generates a new value number and registers it to the table together with the key. After value numbering, GVN performs dominator-based or availability-based redundancy elimination, which removes instructions dominated by ones with the same value numbers, or instructions whose value numbers are available.

Because of their complementary power, most modern optimizing compilers perform both GVN and PRE [15]; that is, they first perform GVN, then convert the program back into the non-SSA form, and finally execute PRE.

Table 1: Eight types of redundancy

	data independent		data dependent	
	path independent	path dependent	path independent	path dependent
total	(I): PRE, GVN	(II): PRE	(V): GVN	(VI)
partial	(III): PRE	(IV): PRE	(VII)	(VIII)

There are, however, three types of redundancy in ordinary programs that neither of these methods can eliminate, as described in Section 2. Thus, we need to perform PRE and copy propagation (CP) iteratively after GVN[16]; we make as many instructions lexically identical as possible by propagating copy instructions generated by the previous PRE, and remove the now-lexically-identical instructions by the next PRE. In other words, from the upper stream of data dependency, we must perform both PRE and CP for each depth level of the dependency, because PRE cannot deal with data dependency in one pass. Therefore, this algorithm suffers from an overhead due to the iteration, which a runtime optimizing compiler cannot ignore.

In this report, we propose Partial Value Number Redundancy Elimination (PVNRE), which fuses GVN and PRE, and removes the need for the iteration of PRE and CP. PVNRE performs PRE-like data-flow analyses, in which it uses not the lexical appearances of instructions but rather their value numbers as GVN does. In contrast to PRE, PVNRE can deal with data dependency between value numbers during data-flow analyses, and avoid the iteration of PRE and CP. Thus, it is as powerful as, and faster than the combination of GVN and the iteration of PRE and CP.

The main contributions of our work are as follows.

- PVNRE is the first redundancy elimination algorithm that tackles the iteration of PRE and CP, and succeeds in managing both optimizing power and analysis speed.
- To allow PVNRE to include the powerful features of PRE, we developed a new algorithm to convert value numbers through ϕ functions of the SSA form *on demand* during data-flow analyses. Thus, we need not construct any special representation of redundancy such as a *Value Flow Graph* [19] in advance of the analyses.
- We show that by assigning incremental value numbers to instructions the numbers can themselves represent data dependency. PVNRE is the first method that exploits this property in processing the value numbers in the order of their data dependency during data-flow analyses.
- We present the effectiveness of PVNRE by implementing it in our just-in-time compiler and conducting experiments using real benchmarks.

The rest of the report is organized as follows. Section 2 presents the types of redundancy we deal with in this report, and explains the inefficiencies of the existing algorithms to remove such redundancies. Section 3 and 4 introduce the overview of PVNRE and basic notations. Section 5 describes the algorithm of PVNRE and Section 6 shows the experimental results. Section 7 reviews related work, and Section 8 sets out the conclusion.

2 Background

We categorize the redundancy we deal with in this report into eight types (Type I – VIII) as shown in Table 1. We also show in the table which types of redundancy PRE and GVN can detect and eliminate.

Figure 1 illustrates the four types of redundancy PRE can detect. PRE removes the computations of Instructions (Insns.) 3, 7, 12, 15, and 19 by using a temporary variable "t" as exemplified in Type IV(b) of the figure.

In contrast, GVN can detect Types I and V. Precisely speaking, availability-based GVN can detect both (1) and (2) of Type I in Fig. 1, while dominator-based GVN can detect only (1).

GVN cannot detect Type II because of the path-dependent redundancy; depending on the execution path that leads to it, Insn. 12 computes different values, since "a" refers to the different definitions (Insns. 8 and 10). The path-dependency becomes clearer if we convert the program into the SSA form

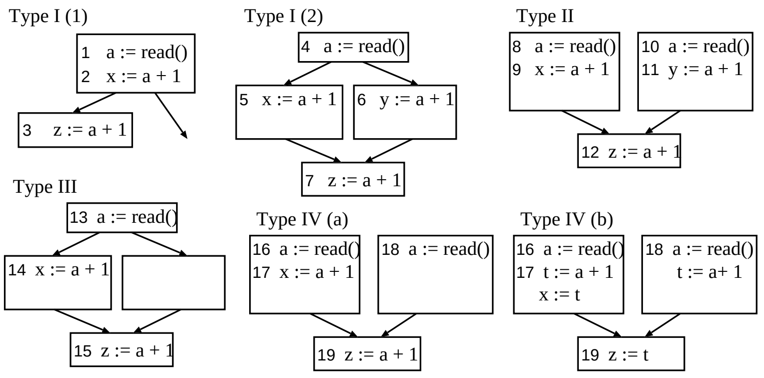


Figure 1: Examples of optimization by PRE

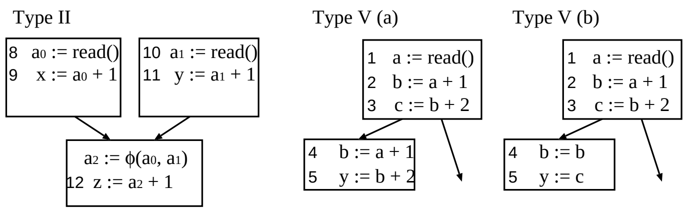


Figure 2: Examples of optimization by GVN

as in Fig. 2 Type II. The newly inserted ϕ function is a pseudo function that merges more than one definitions of a variable at a join node. Thus, Insns. 8, 10, and the ϕ function are assigned distinct value numbers, and so are Insns. 9, 11, and 12, which makes it impossible to detect the redundancy among them.

The reason GVN cannot detect Type III is that Insn. 14 does not dominate Insn. 15, nor make the value number of “a + 1” available at Insn. 15 due to its partial redundancy. Type IV in Fig. 1 is a combination of Types II and III, so that it also cannot be detected by GVN.

Figure 2 Type V illustrates the type of redundancy GVN can remove but PRE cannot. Insns. 2 and 4, and 3 and 5 are assigned the same value numbers; hence, we can eliminate the redundancy by using the transformation shown in (b). In particular, although Insns. 3 and 5 are lexically identical, their redundancy cannot be removed by PRE because there is a store for the operand between them (Insn. 4), or in other words, because they are data-dependent on different instructions (Insns. 2 and 4). Therefore, we call this type of redundancy between Insns. 3 and 5 a data-dependent redundancy.

As described in Sect. 1, most modern optimizing compilers perform both GVN and PRE; but other types of redundancy (Fig. 3), which neither of them can eliminate, can also be encountered in real programs. For example, a data-dependent chain of loop invariants, which can frequently be found in address computations of loop-invariant array loads, is one of the variants of Type VII as shown in Fig. 3 Type VII (2).

Thus, after GVN, we have to perform PRE and copy propagation (CP) iteratively to completely eliminate such redundancies [1] . For example, in Fig. 4, (a) is the same as Type VII (2) in Fig. 3. The first PRE moves Insn. 7 out of the loop (b). It cannot move Insn. 8 because this is data-dependent on Insn. 7. The generated copy instruction is then propagated to Insn. 8 (c). Finally, since the data dependency has been removed, the second PRE can move Insn. 8 out of the loop (d). The redundancies of Types VI and VIII in Fig. 3 can be eliminated in the same manner.

The diagram of the resulting algorithm, which we call GVN+PRECP, is shown in the left-hand side of Fig. 5. However, the inefficiencies in the algorithm are as follows.

- To collect the local information of instructions in PRE, a hash table is used to number each lexical appearance of instructions. This operation is similar to value numbering in GVN.

[1] In fact, if we iterate PRE and CP, we need not perform GVN from the point of view of optimizing ability. However, we should perform GVN first because it can eliminate Type V much faster than the iteration of PRE and CP.

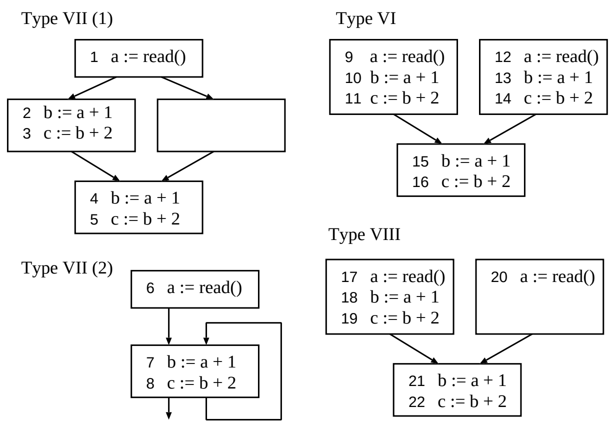


Figure 3: GVN+PRE cannot optimize these types of redundancy

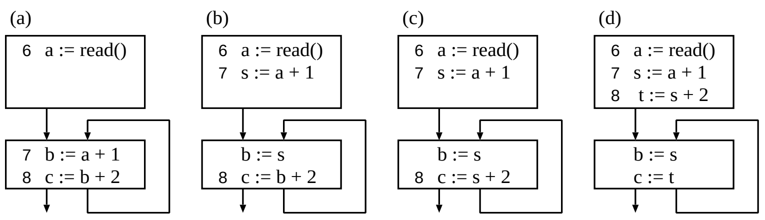


Figure 4: Sequence of optimization by iteration of PRE and CP

- In spite of the fact that copy propagation is trivial in the SSA form, we cannot perform it in that form during PRECP because PRE is based on a non-SSA form.
- For each iteration of the loop, we must set up data structures and collect information for all instructions in the program, although most of them might be unaffected by that iteration.

In the next section, we describe PVNRE, a redundancy elimination algorithm which overcomes these inefficiencies of GVN+PRECP.

3 Overview of PVNRE Algorithm

PVNRE aims to provide equal or better optimizing power with faster analysis speed than GVN+PRECP.

The overview of the algorithm is shown on the right-hand side of Fig. 5. The main features of this algorithm are as follows.

- Using value numbers in the PRE-like data-flow analyses, it can detect not only the redundancies of Types I and V like GVN, but also partial redundancies, particularly Types III and VII. (Sect. 5.1, 5.3, and 5.8)
- Path-dependent redundancy is detected by converting value numbers through ϕ functions on demand during data-flow analyses. It is worth noting that the reason GVN cannot detect the path-dependent redundancy is the existence of ϕ functions in the SSA form. (Sect. 5.4, 5.5, and 5.6)
- PVNRE assigns value numbers so that the numerical order can represent data dependency. This constraint allows it to deal with data-dependent redundancy during data-flow analyses and redundancy elimination. (Sect. 5.1, 5.4, 5.5, and 5.10)
- Instead of iterating PRE and CP for all instructions, PVNRE uses iterations for only the essential subset of the instructions; it iterates the value number conversion at the innermost loop of data-flow analyses to detect the redundancies of Types VI and VIII. Thus, it is faster than GVN+PRECP. (Sect. 5.4 and 5.5)

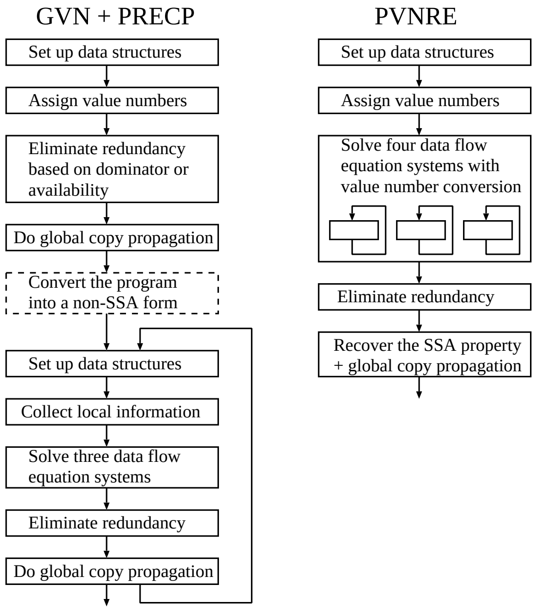

GVN + PRECP
Set up data structures
Assign value numbers
Eliminate redundancy based on dominator or availability
Do global copy propagation
Convert the program into a non-SSA form
Set up data structures
Collect local information
Solve three data flow equation systems
Eliminate redundancy
Do global copy propagation
PVNRE
Set up data structures
Assign value numbers
Solve four data flow equation systems with value number conversion
Eliminate redundancy
Recover the SSA property + global copy propagation

Figure 5: GVN+PRECP and PVNRE algorithms

- PVNRE performs copy propagation on the SSA form. (Sect. 5.4)

In fact, PVNRE can remove redundancies beyond the ones listed in Table 1, but such types of redundancy are rarely found in real programs.

4 Preliminaries

Without loss of generality, we make the following assumptions about a program.

- The program has already been transformed into the SSA form.
- The control flow graph is reducible.
- The nesting relationship of the loops has already been analyzed.
- The critical edges have already been removed.
- Each basic block has at most two preceding blocks, which implies that every ϕ function is binary.

A program is a directed graph $G = \langle \mathit{Nodes}, \mathit{Edges}, \mathit{start}, \mathit{end} \rangle$ with the instruction set *Nodes*, the edge set *Edges*, the unique start instruction *start*, and the unique end instruction *end*. $m \rightarrow n$ denotes an edge from m to n. *BBNodes* represents the basic block set. $\mathit{Pred}(N)$ denotes the preceding blocks of $N \in \mathit{BBNodes}$, $\mathit{Succ}(N)$ the succeeding blocks, $\mathit{Head}(N)$ the first instruction, and $\mathit{BB}(n)$ the block to which $n \in \mathit{Nodes}$ belongs.

$P[m, n]$ represents the set of all the paths from m to n. $\lambda(p)$ is the length of a path p, and p_i is its ith instruction ($1 \leq i \leq \lambda(p)$). In addition, $p[i, j]$ indicates the sub-path from the ith to the jth instruction. $\mathit{Value}(p, i)$ denotes the value computed by the ith instruction on p

The set of all operators *Operators* consists of arithmetic operators *Normal*, a copy operator *Copy*, ϕ functions *Phi*, and the other operators *Fixed*. *Fixed* includes function calls and memory operators. We define *Op* as a function from each instruction to its operator ($\mathit{Nodes} \rightarrow \mathit{Operators}$), writing $n.\mathit{Op}$ for $n \in \mathit{Nodes}$, and ϕ_N as an operator of a ϕ function in a block N.

As with *Phi*, we assume that *Normal* has exactly two operands and *Copy* has only the left operand. We define *Lt* and *Rt* as functions from each instruction to its operands ($\mathit{Nodes} \rightarrow \mathit{Nodes}$), writing $n.\mathit{Lt}$ and $n.\mathit{Rt}$. However, when there is no need to distinguish between them, we write one of them as $n.X$ and the other as $n.Y$.

The value number set *Nums* is a subset of natural numbers $\mathcal{N}$, and a value numbering *Num* can be defined as a function of $\mathit{Nodes} \rightarrow \mathit{Nums}$. We make *Op*, *Lt*, and *Rt* also take the roles of functions from each value number to its operator and operand value numbers, respectively.

5 Algorithm Details

5.1 Value Numbering

The value numbering in PVNRE shown in Fig. 6 is the same as that for GVN, except that (1) it computes backedges (*Bedges*) and transparency (*UnTransp*), and (2) it must assign incremental value numbers (Line 9 and 26). *GEN* is used as local information in the following data-flow analyses. Backedges and transparency is described in Sect. 5.3.

Incremental value numbers ensures the following constraint (Definition 5.1), which means that Insn. x is assigned a greater number than Insn. y if x is data-dependent on y. This is because in value numbering we must number instructions in the data dependency order (the reverse post order in Line 5 and the pre-order in Line 7), so that we can use the value numbers of the left and right operands as a key to the hash table.

Definition 5.1 (Numerical Order Constraint). *For $\alpha \in \mathit{Nums}$,*

$$\alpha.\mathit{Op} \in \mathit{Normal} \Rightarrow \alpha.X < \alpha \ .$$

PVNRE utilizes this numerical order in the following data-flow analyses to process value numbers in data dependency order.

5.2 Data-Flow Analyses

After value numbering, PVNRE solves four data-flow equation systems for AVAIL^{all}, AVAIL^{some}, ANTIC^{all}, and AVAIL^{Msome}. These are the framework for PRE proposed by Bodik et al. [3]. AVAIL^{all} and AVAIL^{some} represent the availability on all paths and some paths respectively, while ANTIC^{all} denotes the anticipatability on all paths. AVAIL^{Msome} is akin to AVAIL^{some}, but a condition is added to prevent speculative insertion of instructions. We refer readers to the paper by Bodik et al. [3] for more details about these predicates. Here, we focus on the differences between their work and PVNRE, namely, transparency and value number conversion.

5.3 Transparency

Transparency is a condition that determines whether or not data-flow information is valid beyond a block or an edge in data-flow analysis. In traditional PRE, the propagated information (*AVAIL* or *ANTIC*) of an instruction is invalidated if a block contains a store for an operand variable of the instruction. PVNRE, on the other hand, does not need such kind of condition because it can deal with data-dependent redundancy as illustrated in Fig. 2 Type V.

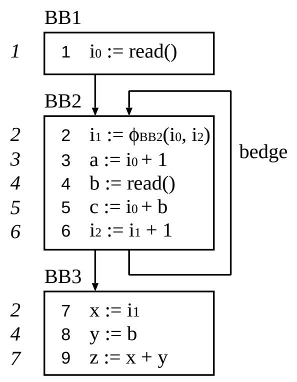
In PVNRE, however, we need special treatment for backedges. For example, in Fig. 7(1), if we allowed the *AVAIL* of the value numbers of Insn. 6 to flow through the backedge, the information would reache Insn. 6 again; hence, we would consider it to be a loop invariant However, since Insn. 6 is an induction variable, this does not happen and it returns different values for each iteration of the loop.

To solve the problem, we define transparency for value numbers and backedges: the value numbers of instructions that may compute different values for each iteration of a loop are invalidated at the backedge of the loop. The reason an instruction computes different values for each iteration is that it is data-dependent on a ϕ function or a function call inside the loop. Thus, we first define *DefBBNodes* for *Fixed* and *Phi* value numbers, which is a set of the basic blocks containing instructions that generate a value number.

```
for each $e \in Edges$ do
  $UnTransp(e) := \emptyset$
end
$VN := 0$
for each $N \in BBNodes$ in reverse post order do
  $GEN(N) = \emptyset$
  for each $n \in N$ in pre-order do
    if $n.Op \in Fixed, Phi$ then
      $VN := VN + 1$
      $VN.Op := n.Op$
      $Bedges(VN) :=$ the set of the enclosing backedges of $N$
      $Num(n) := VN$
    else if $n.Op \in Copy$ then
      $Num(n) := Num(n.Lt)$
    else
      $\alpha :=$ call $Hash(n.Op, Num(n.Lt), Num(n.Rt))$
      $Num(n) := \alpha$
      $GEN(N) := GEN(N) \cup \{\alpha\}$
    end
  end
end

$Hash(op, l, r)$ {
  If $\langle op, l, r\rangle$ is registered, returns the corresponding $VN$.
  If not,
  $VN := VN + 1$
  Register $VN$ with $\langle op, l, r\rangle$.
  $VN.Op := op$; $VN.Lt := l$; $VN.Rt := r$
  $Bedges(VN) := Bedges(l) \cup Bedges(r)$
  for each $b \in Bedges(VN)$ do
    $UnTransp(b) := UnTransp(b) \cup \{VN\}$
  end
  return $VN$
}
```

Figure 6: Algorithm for value numbering



Value No.	*Transp*	Description
1	true	*DefBBNodes*(*1*) is outside the loop.
2	false	*DefBBNodes*(*2*) is inside the loop.
3	true	It is inside the loop, but its operand refers to the value number of the instrucion outside the loop, so it is loop-invariant.
4	false	*DefBBNodes*(*4*) is inside the loop.
5	false	One of its operand refers to the value number of the instruction inside the loop.
6	false	The same as *5*.
7	false	It is outside the loop, but its operands refer to the value numbers of the instructions inside the loop.

Figure 7: Example of backedge and transparency

Definition 5.2 (*DefBBNodes* **for** *Fixed*)**.** *For $\alpha \in Nums$ s.t. $\alpha.Op \in Fixed$,*

$$\begin{aligned} DefNodes(\alpha) &\overset{\text{def}}{=} \{n \in Nodes \mid Num(n) = \alpha \wedge n.Op = \alpha.Op\} \\ DefBBNodes(\alpha) &\overset{\text{def}}{=} \bigcup_{\forall n} BB(n) \text{ s.t. } n \in DefNodes(\alpha) \ . \end{aligned}$$

Definition 5.3 (*DefBBNodes* **for** *Phi*)**.**
For $\alpha \in Nums$ s.t. $\alpha.Op = \phi_N \in Phi$, $\quad DefBBNodes(\alpha) \overset{\text{def}}{=} \{N\}$.

In Fig. 7, *DefBBNodes(2)* = {BB2} and *DefBBNodes(4)* = {BB2}. Since Insn. 7 and 8 are copy instructions, they are not considered to generate their own value numbers.

5.3.1 Backedges

Backedges can be defined as $Bedges \overset{\text{def}}{=} \{m \rightarrow n \mid n \text{ dom m}\}$, where dom denotes domination relationship. Here, we define backedges for each instruction and block as follows.

Definition 5.4 (Backedges for an instruction and a basic block).
For $u \in Nodes$ and $N \in BBNodes$,

$$\begin{aligned} Bedges(u) &\overset{\text{def}}{=} \{e = m \rightarrow n \mid e \in Bedges \wedge \exists p \in P[u, m] \ \forall 1 \leq i \leq \lambda(p) \ . \ p_i \neq n\} \\ Bedges(N) &\overset{\text{def}}{=} Bedges(Head(N)) \ . \end{aligned}$$

That is, they represent all the backedges of the loops enclosing the instruction or the block. In Fig. 7, *Bedges* for Insn. 2–6 and BB2 is {bedge}, and *Bedges* for the other instructions is $\emptyset$.

5.3.2 Transparency

We define the transparency in PVNRE as follows.

Definition 5.5 (*Transp*)**.** *For $\alpha \in Nums$ and $e \in Edges$,*
(1) if $\alpha.Op \in Fixed \vee \alpha.Op \in Phi$, then

$$Transp(\alpha, e) \overset{\text{def}}{\Leftrightarrow} e \notin \bigcup_{\forall N} Bedges(N) \text{ s.t. } N \in DefBBNodes(\alpha),$$

(2) otherwise

$$Transp(\alpha, e) \overset{\text{def}}{\Leftrightarrow} Transp(\alpha.Lt, e) \wedge Transp(\alpha.Rt, e) \ .$$

A *Fixed* value number is not allowed to propagate through the enclosing back-edges because we assume that a memory load or a function call returns different values each time it is executed. Propagation is not allowed for *Phi* in the same way because we cannot determine statically which of the merged definitions is actually referred to for each execution. For *Normal*, its value is considered to be invalid beyond a backedge of a loop iff. one of its operands' values is invalid beyond the backedge. Figure 7 shows *Transp* for bedge for each value number.

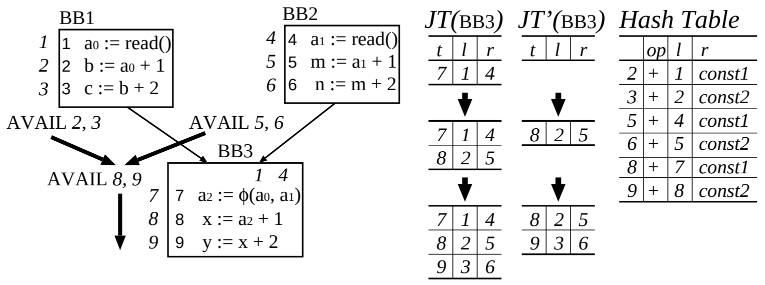
We also naturally extend the definition of *Transp* to any path.

Definition 5.6 ($Transp^{\forall}$)**.**
For $\alpha \in Nums, p \in P[start, end], 1 \leq i < j \leq \lambda(p)$,

$$Transp^{\forall}(\alpha, p[i, j]) \overset{\text{def}}{\Leftrightarrow} \bigwedge_{\forall i \leq k < j} Transp(\alpha, p_k \rightarrow p_{k+1}) \ .$$

During value numbering, PVNRE computes the enclosing backedges for a ϕ function and a function call (Fig. 6 Line 11). Other instructions inherit backedges from the data-dependent instructions (Line 29), and register themselves to *UnTransp* (the complementary set of the transparency) for the backedges (Line 30 – 32).

Alternatively, we could invalidate value numbers not at the backedges but at the ϕ functions or the function calls on which they are data-dependent. However, if we did so we would need to compute *UnTransp* for all the basic blocks, and set up upward and downward *GEN* separately, which would result in longer analysis time. To the best of our knowledge, PVNRE is the first redundancy elimination algorithm to use backedge-based transparency.



t	l	r
7	1	4
8	2	5
9	3	6

	op	l	r
2	+	1	const1
3	+	2	const2
5	+	4	const1
6	+	5	const2
8	+	7	const1
9	+	8	const2

Figure 8: Example of value number conversion for Type II and VI

5.4 Examples of Value Number Conversion

If we only propagate value numbers on PRE-like data-flow analyses, we cannot detect path-dependent redundancies (Types II, IV, VI, and VIII) as we described in Sect. 2. To solve the problem, we convert value numbers at join nodes on demand during data-flow analyses, using ϕ functions.

PVNRE uses two sets, $JT(N)$ and $JT'(N)$, which are defined for each join node N. Their elements are of the form $\langle t, \langle l, r\rangle\rangle$, which means "value number l and r join at N, and are converted into t." $JT(N)$ is initialized by the ϕ functions in N in the original program, and $JT'(N)$ is set to $\emptyset$. PVNRE uses and adds elements to JT and JT' while solving $AVAIL^{all}$ and $AVAIL^{some}$. To solve $ANTIC^{all}$ and $AVAIL^{Msome}$, it just uses JT', adding no more elements.

5.4.1 Types II and VI (Path-Dependent Total Redundancy).

We use the example in Fig. 8. Value numbers (italic numbers *1* − *9*) are already assigned as the hash table shows. Assume that we are now processing basic block 3 (BB3) for the first time while we are computing the maximum fixed points for $AVAIL^{all}$ and $AVAIL^{some}$. JT is initialized as $\{\langle 7, \langle 1, 4\rangle\rangle\}$ by the ϕ function (Insn. 7).

Now we have to compute $AVAIL^{all}_{in}$(BB3) and $AVAIL^{some}_{in}$(BB3). *2* and *3* are available from the left, so that we process *2* first, utilizing the constraints between the numerical order and the data dependency described in Sect. 5.1. We search the "l" column of JT for *1* (the left operand of *2*). We need not search for the right operand, because it is constant 1, and is not subject to conversion. We find an element in JT[2] , which indicates that *1* merges with *4* and is converted into *7*. Now we search the hash table for "*4* + const1" and "*7* + const1," finding *5* and *8*. Thus we add a new element $\langle 8, \langle 2, 5\rangle\rangle$ to JT and JT'. In the same manner, we process *3* and add another element $\langle 9, \langle 3, 6\rangle\rangle$. Note that if we process *3* first, we cannot convert either *2* or *3*. Then we process *5* and *6* from the right, but we need not add any more elements. Consequently, $AVAIL^{all}_{in} = \{8, 9\}$, $AVAIL^{some}_{in} = \{2, 3, 5, 6, 8, 9\}$. We propagate them into BB3, and finally detect the redundancy of Insn. 8 (Type II) and Insn. 9 (Type VI), because their value numbers are available. When we process BB3 again during the computation of the maximum fixed points, we need not repeat the process again for *2*, *3*, *5*, and *6*, but just use JT'. In the same way, we also use JT' to compute $ANTIC^{all}$ and $AVAIL^{Msome}$.

Indeed, it is not mandatory to compute $AVAIL^{some}$, but we do so in order to speed up the analyses that follow; if not, we would have to update JT', even during the computation of $ANTIC^{all}$. This is because the number of propagated value numbers in $AVAIL^{all}$ is so small that the resulting JT' would not have enough elements to convert value numbers backward for $ANTIC^{all}$.

5.4.2 Types IV and VIII (Path-Dependent Partial Redundancy).

For the path-dependent partial redundancy, we must generate new value numbers during the conversion. For example, in Fig. 9, when we process *2*, we cannot find "*4* + const1" in the hash table. Then we generate a new value number *8*, and register it to the hash table[3] . The same goes for *3*, and *9* is

[2] If we could not found any, then *2* would not be converted at BB3.

[3] If we cannot find a conversion target (in this case, "*5* + const1"), we generate another value number, and let it propagate down.

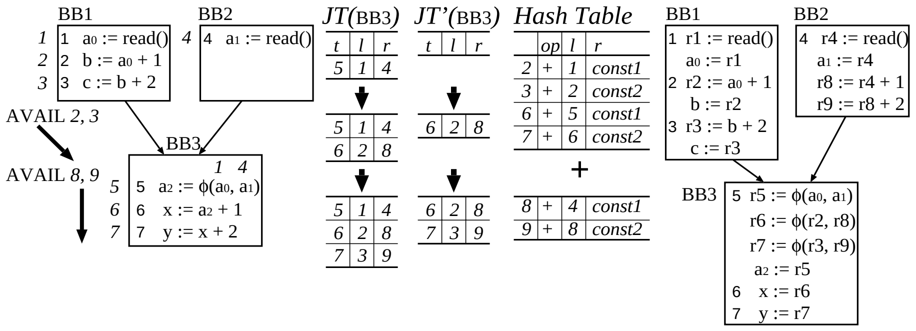


Figure 9: Example of value number conversion for Type IV and VIII, and the result of redundancy elimination

generated. Consequently, $AVAIL_{in}^{all} = \emptyset$, $AVAIL_{in}^{some} = \{2, 3, 6, 7\}$, so that we can detect the partial redundancy of Insn. 6 (Type IV) and Insn. 7 (Type VIII).

5.4.3 Redundancy Elimination

After the computation of $AVAIL^{all}$, $AVAIL^{some}$, $ANTIC^{all}$, and $AVAIL^{Msome}$, we insert new instructions into edges and remove all redundancies as in [3]. We show an example of redundancy elimination in the right-hand side of Fig. 9. We use a new variable "rn" for a value number n. For a non-redundant instruction, the value is stored into "rn," For a redundant instruction, the expression is replaced with "rn." For each element in JT', a new ϕ function is inserted. To make partially redundant instructions totally redundant, new instructions are inserted at certain edges to join nodes [3]. For example, we must insert instructions to compute *8* and *9* at the tail of BB2. Thus, consulting the hash table, "r8 := r4 + 1" and "r9 := r8 + 1" are inserted in the data dependency order, or the numerical order of the value numbers.

After redundancy elimination, we must recover the SSA property because "rn" can be assigned at more than one place. We use the algorithm proposed by Cytron et al. [9], and at the same time perform copy propagation.

5.5 Value Number Conversion Algorithm

The program code to compute $AVAIL^{all}$ and $AVAIL^{some}$ is shown in Fig. 10 and 11. Let *lnk* be $\langle l, r\rangle$ s.t. $\langle trgt, \langle l, r\rangle\rangle \in JT(N)$, and e_l, e_r the incoming edges of N, we denote $lnk(e_l) = l$ and $lnk(e_r) = r$. We define $AVAIL^{some}$ not only for blocks but also for edges to record $AVAIL_{out}^{some}$ of their source blocks (Fig. 11 Line 23).

$JT(N)$ is initialized by the ϕ functions in N in the original program (Fig. 10 Line 3), and $JT'(N)$ is set to $\emptyset$ (Line 5). We compute maximum fixed points for $AVAIL^{all}$ and $AVAIL^{some}$ at Fig. 10 Line 12 – 24. We first update $JT'(N)$ in *ConvThroughPhi* (Line 16), and use it to compute new $AVAIL_{in}^{all}(N)$ and $AVAIL_{in}^{some}(N)$ in *CompAVin* (Line 18).

To update JT' (and JT), we convert the value numbers that have reached N for the first time (Fig. 11 Line 3). The ascending order (Line 4) ensures that we process them from the upper stream of data dependency. We search $JT(N)$ for the left and right operands of α at Line 5 and 6. Line 10 corresponds to the situation where both of the operands are converted, while Line 14 and 19 to the situation where either left or right operand is converted respectively. *DoConv* updates $JT(N)$ and $JT'(N)$ at Line 44 and 45. The newly-added element can be used at Line 5 and 6 of the following iterations of the same loop (Line 4–22) to convert data-dependent value numbers.

The computation of $ANTIC^{all}$ and $AVAIL^{Msome}$ is done in a similar manner except that we need not update JT any more.

```
for each $N \in BBNodes$ do
  for each $n \in N$ . $n.Op \in Phi$ do
    $JT(N) \cup{:=}\{\langle Num(n), \langle Num(n.Lt), Num(n.Rt)\rangle\rangle\}$
  end
  $JT'(N) := \emptyset$
  $AVAIL_{out}^{all}(N) := \top, AVAIL_{out}^{some}(N) := \emptyset$
end
for each $e \in Edges$ do
  $AVAIL^{some}(e) := \emptyset$
end

$AVAIL_{out}^{all}(start) := \emptyset$, $W := Succ(start)$
while $W \neq \emptyset$ do
  $N \in W, W := W \setminus N$
  for each $M \in Pred(N)$ do
    call $ConvThroughPhi(M, N)$
  end
  call $CompAVin(N)$
  $AVAIL_{out}^{all}(N) := AVAIL_{in}^{all}(N) \cup GEN(N)$
  $AVAIL_{out}^{some}(N) := AVAIL_{in}^{some}(N) \cup GEN(N)$
  if $AVAIL_{out}^{all}(N)$ or $AVAIL_{out}^{some}(N)$ changed then
    $W \cup{:=} Succ(N)$
  end
end

$CompAVin(N)$ {
  $AVAIL_{in}^{all}(N) := \top$, $AVAIL_{in}^{some}(N) := \emptyset$
  for each $M \in Pred(N)$ do
    $AVAIL_{in}^{all}(N) \cap{:=}(AVAIL_{out}^{all}(M) - UnTransp(M{\rightarrow}N))$
    $AVAIL_{in}^{some}(N) \cup{:=}(AVAIL_{out}^{some}(M) - UnTransp(M{\rightarrow}N))$
  end
  for each $\langle trgt, lnk\rangle \in JT'(N)$ do
    if $\forall M \in Pred(N)$ . $lnk(M{\rightarrow}N) \in AVAIL_{out}^{all}(M)$ then
      $AVAIL_{in}^{all}(N) \cup{:=}\{trgt\}$
    end
    if $\exists M \in Pred(N)$ . $lnk(M{\rightarrow}N) \in AVAIL_{out}^{some}(M)$ then
      $AVAIL_{in}^{some}(N) \cup{:=}\{trgt\}$
    end
  end
}
```

Figure 10: Algorithm for $AVAIL^{all}$ and $AVAIL^{some}$ (the first half)

```
$ConvThroughPhi(M, N)$ {
  $e := M \rightarrow N$
  $diff := (AVAIL^{some}_{out}(M) - AVAIL^{some}(e))$
  **for each** $\alpha \in diff$ **in ascending order do**
    $ljs := \{\langle trgt, lnk\rangle \in JT(N) \mid lnk(e) = \alpha.Lt\}$
    $rjs := \{\langle trgt, lnk\rangle \in JT(N) \mid lnk(e) = \alpha.Rt\}$
    **if** $ljs \neq \emptyset$ **then**
      **if** $rjs \neq \emptyset$ **then**
        **for each** $\langle l, llnk\rangle \in ljs, \langle r, rlnk\rangle \in rjs$ **do**
          **call** $DoConv(N, \alpha, l, llnk, r, rlnk)$
        **end**
      **else if** $\alpha.Rt \in Transp(e)$ **then**
        **for each** $\langle l, llnk\rangle \in ljs$ **do**
          **call** $DoConv(N, \alpha, l, llnk, \alpha.Rt, nil)$
        **end**
      **end**
    **else if** $rjs \neq \emptyset \wedge \alpha.Lt \in Transp(e)$ **then**
      **for each** $\langle r, rlnk\rangle \in rjs$ **do**
        **call** $DoConv(N, \alpha, \alpha.Lt, nil, r, rlnk)$
      **end**
    **end**
  **end**
  $AVAIL^{some}(e) := AVAIL^{some}_{out}(M)$
}

$DoConv(N, \alpha, l, llnk, r, rlnk)$ {
  $trgt :=$ **call** $Hash(\alpha.Op, l, r)$
  **if** $\exists x \,.\, \langle trgt, x\rangle \in JT(N)$ **then**
    **return end**
  **for each** $M \in Pred(N)$ **do**
    **if** $llnk = nil$ **then**
      $lt := l$
    **else**
      $lt := llnk(M \rightarrow N)$
    **end**
    **if** $rlnk = nil$ **then**
      $rt := r$
    **else**
      $rt := rlnk(M \rightarrow N)$
    **end**
    $\beta :=$ **call** $Hash(\alpha.Op, lt, rt)$
    $lnk(M \rightarrow N) := \beta$
  **end**
  $JT(N) \cup:= \langle trgt, lnk\rangle$
  $JT'(N) \cup:= \langle trgt, lnk\rangle$
}
```

Figure 11: Algorithm for $AVAIL^{all}$ and $AVAIL^{some}$ (the second half)

5.6 Formalization of Value Number Conversion

As shown in the previous section, PVNRE joins two value numbers together which satisfy certain conditions at a join node. It then converts them into a new value number, and transfers it to the following nodes. The *joinability* is defined as follows.

Definition 5.7 (*Jtarget* and *Jtarget′*). *For $M, N \in BBNodes$ s.t. $M \in Pred(N)$ and $\alpha_0, \alpha_1 \in Nums$, $Jtarget(N, \alpha_0, \alpha_1) \overset{\text{def}}{=}$ (1) $\cup$ (2) $\cup$ (3), and $Jtarget'(N, \alpha_0, \alpha_1) \overset{\text{def}}{=}$ (2) $\cup$ (3), where*

(1) $\{\alpha \in Nums \mid \exists n \in Nodes \text{ s.t. } \alpha = Num(n)$
$\quad \wedge\, n.Op = \phi_N \wedge Num(n).Op = \phi_N \wedge \alpha_0 = Num(n.Lt) \wedge \alpha_1 = Num(n.Rt)\}$

(2) $\{\alpha_2 \in Nums \mid \alpha_2.Op = \alpha_0.Op = \alpha_1.Op \in Normal$
$\quad \wedge\, \alpha_2.X \in Jtarget(N, \alpha_0.X, \alpha_1.X)$
$\quad \wedge\, \alpha_2.Y = \alpha_0.Y = \alpha_1.Y \wedge Transp(\alpha_2.Y, M{\rightarrow}N)\}$

(3) $\{\alpha_2 \in Nums \mid \alpha_2.Op = \alpha_0.Op = \alpha_1.Op \in Normal$
$\quad \wedge\, \alpha_2.Lt \in Jtarget(N, \alpha_0.Lt, \alpha_1.Lt) \wedge \alpha_2.Rt \in Jtarget(N, \alpha_0.Rt, \alpha_1.Rt)\}$.

Set (1) denotes that a ϕ function is the basis of joinability; it is *Jtarget′* that represents the actual recursive joinability of value numbers. Set (2) is the case where one of the operands of α_0 joins with the corresponding operand of α_1. Set (3) describes the case where both of the operands join.

In Fig. 8, it can be proved that

$$\begin{aligned}
&Jtarget(\text{BB3}, 1, 4) = \{7\},\\
&Jtarget(\text{BB3}, 2, 5) = Jtarget'(\text{BB3}, 2, 5) = \{8\},\\
&Jtarget(\text{BB3}, 3, 6) = Jtarget'(\text{BB3}, 3, 6) = \{9\}\ .
\end{aligned}$$

In Fig. 9,

$$\begin{aligned}
&Jtarget(\text{BB3}, 1, 4) = \{5\},\\
&Jtarget(\text{BB3}, 2, 8) = Jtarget'(\text{BB3}, 2, 8) = \{6\},\\
&Jtarget(\text{BB3}, 3, 9) = Jtarget'(\text{BB3}, 3, 9) = \{7\}\ .
\end{aligned}$$

We also define *Jlink* (and *Jlink′*) as a set of the value numbers to which a value number is converted.

Definition 5.8 (*Jlink* and *Jlink′*). *For $M, N \in BBNodes$ s.t. $M \in Pred(N)$*
and $\alpha \in Nums$, $Jlink(N, \alpha, M{\rightarrow}N) \overset{\text{def}}{=}$
(1) if M is the left preceding block of N, then $\bigcup_{\forall\beta} Jtarget(N, \alpha, \beta)$
(2) if M is the right preceding block of N, then $\bigcup_{\forall\beta} Jtarget(N, \beta, \alpha)$.
Jlink′ is defined in the same way using Jtarget′.

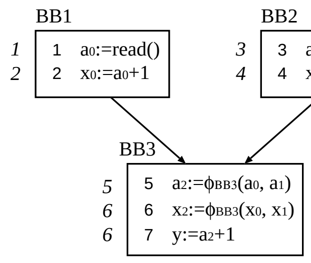
In Fig. 8,

$$\begin{aligned}
&Jlink(\text{BB3}, 1, \text{BB1}{\rightarrow}\text{BB3}) = \{7\}, Jlink(\text{BB3}, 4, \text{BB2}{\rightarrow}\text{BB3}) = \{7\},\\
&Jlink(\text{BB3}, 2, \text{BB1}{\rightarrow}\text{BB3}) = Jlink'(\text{BB3}, 2, \text{BB1}{\rightarrow}\text{BB3}) = \{8\},\\
&Jlink(\text{BB3}, 5, \text{BB2}{\rightarrow}\text{BB3}) = Jlink'(\text{BB3}, 5, \text{BB2}{\rightarrow}\text{BB3}) = \{8\},\\
&Jlink(\text{BB3}, 3, \text{BB1}{\rightarrow}\text{BB3}) = Jlink'(\text{BB3}, 3, \text{BB1}{\rightarrow}\text{BB3}) = \{9\},\\
&Jlink(\text{BB3}, 6, \text{BB2}{\rightarrow}\text{BB3}) = Jlink'(\text{BB3}, 6, \text{BB2}{\rightarrow}\text{BB3}) = \{9\}\ .
\end{aligned}$$

We omit *Jlink* and *Jlink′* for Fig. 9.

5.7 Soundness

In order to have PVNRE work correctly, we must assign value numbers so that *Op*, *Lt*, *Rt*, and *Num* satisfy the soundness defined later in this section. We first define *DefNodes* for *Normal*, *Phi*, and *Copy*. *DefNodes* for *Fixed* has already been defined in Definition 5.2.



	Op	*Lt*	*Rt*	*DefNodes*
1	read()	--	--	1
2	+	*1*	const1	2
3	read()	--	--	3
4	+	*2*	const1	4
5	ϕ_{BB3}	*1*	*3*	5
6	+	*5*	const1	6,7

Figure 12: Example of sound value numbering

Definition 5.9 (*DefNodes*)**.** *For $\alpha \in$ Nums,*
(1) if $\alpha.Op \in$ Normal

$$\begin{aligned}
\mathit{DefNormalNodes}(\alpha) &\overset{\text{def}}{=} \{n \in \mathit{Nodes} \mid \mathit{Num}(n) = \alpha \wedge n.\mathit{Op} = \alpha.\mathit{Op}\} \\
\mathit{DefPhiNodes}(\alpha) &\overset{\text{def}}{=} \{n \in \mathit{Nodes} \mid \mathit{Num}(n) = \alpha \wedge n.\mathit{Op} \in \mathit{Phi} \\
&\quad\ \wedge \mathit{Jtarget}'(\mathit{BB}(n), \mathit{Num}(n.\mathit{Lt}), \mathit{Num}(n.\mathit{Rt})) \neq \emptyset \\
&\quad\ \wedge \mathit{Num}(n.X).\mathit{Op} = \alpha.\mathit{Op}\} \\
\mathit{DefNodes}(\alpha) &\overset{\text{def}}{=} \mathit{DefNormalNodes}(\alpha) \cup \mathit{DefPhiNodes}(\alpha)
\end{aligned}$$

(2) if $\alpha.Op \in$ Phi

$$\begin{aligned}
\mathit{DefNodes}(\alpha) &\overset{\text{def}}{=} \{n \in \mathit{Nodes} \mid \mathit{Num}(n) = \alpha \wedge n.\mathit{Op} = \alpha.\mathit{Op} \\
&\quad\ \wedge \mathit{Num}(n.X) \neq \alpha \\
&\quad\ \wedge \mathit{Jtarget}'(N, \mathit{Num}(n.\mathit{Lt}), \mathit{Num}(n.\mathit{Rt})) = \emptyset\}
\end{aligned}$$

(3) if $\alpha.Op \in$ Copy

$$\begin{aligned}
\mathit{DefNodes}(\alpha) &\overset{\text{def}}{=} \{n \in \mathit{Nodes} \mid \mathit{Num}(n) = \alpha \wedge n.\mathit{Op} = \alpha.\mathit{Op} \\
&\quad\ \wedge \mathit{Num}(n.\mathit{Lt}) \neq \alpha\}\ .
\end{aligned}$$

DefPhiNodes implies that ϕ functions whose operands satisfy joinability can be *DefNodes* for a value number whose operator is *Normal*. For exmaple, in Fig. 12, the operator of Insn. 6 is ϕ, but the operator of its value number *6* is $+ \in$ *Normal* because its operands *2* and *4* merge at BB3.

Using *DefNodes*, we define the soundness for *Op*, *Lt*, and *Rt*.

Definition 5.10 (Soundness of *Op*.) *Op for value numbers satisfies soundness $\overset{\text{def}}{\Leftrightarrow}$ for $\alpha \in$ Nums,*

$$\exists n \in \mathit{Nodes}\ .\ \mathit{Num}(n) = \alpha \Rightarrow \mathit{DefNodes}(\alpha) \neq \emptyset\ .$$

For example, if we define $\alpha.Op =$ read() $\in$ *Fixed* for a value number α and there exists an instruction with value number α in the program, there must be at least one instruction whose operator is read() and whose value number is α.

Definition 5.11 (Soundness of *Lt* and *Rt*.) *Lt for value numbers satisfies soundness $\overset{\text{def}}{\Leftrightarrow}$ for $\alpha, \beta \in$ Nums, $\alpha.Lt = \beta \wedge \exists n \in$ Nodes . $Num(n) = \alpha \Rightarrow$*
(1) if $\alpha.Op \in$ Normal

$$\begin{aligned}
&\exists n \in \mathit{DefNormalNodes}(\alpha)\ .\ \mathit{Num}(n.\mathit{Lt}) = \beta \\
&\vee \exists n \in \mathit{DefPhiNodes}(\alpha)\ . \\
&\quad (\mathit{Jtarget}'(\mathit{BB}(n), \mathit{Num}(n.\mathit{Lt}).\mathit{Lt}, \mathit{Num}(n.\mathit{Rt}).\mathit{Lt}) \ni \beta \\
&\quad \vee \mathit{Num}(n.\mathit{Lt}).\mathit{Lt} = \mathit{Num}(n.\mathit{Rt}).\mathit{Lt} = \beta)
\end{aligned}$$

(2) if $\alpha.Op \in Phi, Copy$

$$\exists n \in DefNodes(\alpha) \ . \ Num(n.Lt) = \beta \ .$$

Soundness of Rt is defined in the same manner.

Definition 5.12 (Soundness of Value Numbering). *A value numbering Num satisfies soundness $\overset{\text{def}}{\Leftrightarrow}$ for $m, n \in Nodes$, if $m \neq n \wedge Num(m) = Num(n)$, then one of (1)–(5) holds, where*

(1) $m.Op \in Copy \wedge Num(m.Lt) = Num(n)$

(2) $m.Op \in Phi \wedge Num(m.Lt) = Num(m.Rt) = Num(n)$

(3) $m.Op = n.Op \in Normal \cup Phi \wedge Num(m.X) = Num(n.X)$

*(4) $m.Op \in Phi \wedge n.Op \in Normal \wedge$
$Num(n) \in Jtarget'(BB(m), Num(m.Lt), Num(m.Rt))$*

(5) (1)–(4) with m and n interchanged .

Following are the explainations for each condition.

(1) A copy instruction is allowed to have the same value number as its operand, because it always returns the same value as its operand.

(2) A ϕ function whose operands have the same value number is allowed to be assigned that value number. This is because if two definitions of a variable are the same value, their merging also generates that value.

(3) For *Normal* and *Phi*, two instructions are allowed to have the same value number if their operator and operands are the same. This condition corresponds to searching the hash table in GVN.

(4) It is Condition (4) that distinguishes PVNRE from GVN, as it can detect redundancy beyond ϕ functions.

Conditions (1) – (5) imply that two different *Fixed* instructions with the same value number do not satisfy the soundness. In addition, it is worth noting that the opposite direction is not required to be satisfied for soundness. In this case, the soundness of the algorithm is satisfied, but its completeness is compromised.

Theorem 5.13 (Redundancy Theorem). *If Op, Lt, Rt, and Num satisfies soundness, then for $p \in P[start, end]$ and $1 \leq i < j \leq \lambda(p)$,*

$$Num(p_i) = Num(p_j) \wedge Transp^{\forall}(Num(p_i), p[i,j]) \Rightarrow Value(p,i) = Value(p,j) \ .$$

Intuitively, if two instructions have the same value number, and the execution between them is along a $Transp^{\forall}$ path, they compute the same value. We can prove the theorem by induction on the depth of data dependency with *Fixed* as a basis as we show in the Appendix A.

Note that the algorithms shown in Fig. 6, 10, and 11 satisfy the soundness of *Op*, *Lt*, *Rt*, and *Num*. Based on the theorem, PVNRE solves data flow equations in order to find instructions with the same value number between which there exists a $Transp^{\forall}$ path.

5.8 Data-Flow Equation System

Equations for $AVAIL^{all}$ and $AVAIL^{some}$ are as follows.

$$\begin{aligned} AVAIL^{all}_{in}(\alpha, n) \quad = \quad & \bigwedge_{\forall m \in Pred(n)} \Big((AVAIL^{all}_{out}(\alpha, m) \wedge Transp(\alpha, m \rightarrow n)) & (1) \\ & \vee (AVAIL^{all}_{out}(\beta, m) \text{ s.t. } \alpha \in Jlink'(BB(n), \beta, m \rightarrow n)) \Big) & (2) \end{aligned}$$

$$
\begin{aligned}
\mathit{AVAIL}^{all}_{out}(\alpha, n) &= \mathit{AVAIL}^{all}_{in}(\alpha, n) \vee (n \in \mathit{DefNodes}(\alpha)) && (3)\\
\mathit{AVAIL}^{some}_{in}(\alpha, n) &= \bigvee_{\forall m \in \mathit{Pred}(n)} \Big((\mathit{AVAIL}^{some}_{out}(\alpha, m) \wedge \mathit{Transp}(\alpha, m \to n)) && (4)\\
&\qquad \vee (\mathit{AVAIL}^{some}_{out}(\beta, m) \text{ s.t. } \alpha \in \mathit{Jlink}'(\mathit{BB}(n), \beta, m \to n))\Big) && (5)\\
\mathit{AVAIL}^{some}_{out}(\alpha, n) &= \mathit{AVAIL}^{some}_{in}(\alpha, n) \vee (n \in \mathit{DefNodes}(\alpha)) \ . && (6)
\end{aligned}
$$

Predicate (1) and (4) are the cases where α propagates unchanged, while (2) and (5) cover the situations where α is converted into β.

Equations for ANTIC^{all} and AVAIL^{Msome} are as follows.

$$
\begin{aligned}
\mathit{ANTIC}^{all}_{out}(\alpha, m) &= \bigwedge_{\forall n \in \mathit{Succ}(m)} \Big((\mathit{ANTIC}^{all}_{in}(\alpha, n) \wedge \mathit{Transp}(\alpha, m \to n)) && (7)\\
&\qquad \vee \bigvee_{\forall \beta} (\mathit{ANTIC}^{all}_{in}(\beta, n) \text{ s.t. } \beta \in \mathit{Jlink}'(\mathit{BB}(n), \alpha, m \to n))\Big) && (8)\\
\mathit{ANTIC}^{all}_{in}(\alpha, n) &= \mathit{ANTIC}^{all}_{out}(\alpha, n) \vee (n \in \mathit{DefNodes}(\alpha)) && (9)\\
\mathit{AVAIL}^{Msome}_{in}(\alpha, n) &= \bigvee_{\forall m \in \mathit{Pred}(n)} \Big((\mathit{AVAIL}^{Msome}_{out}(\alpha, m) \wedge \mathit{Transp}(\alpha, m \to n)) && (10)\\
&\qquad \vee (\mathit{AVAIL}^{Msome}_{out}(\beta, m) \text{ s.t. } \alpha \in \mathit{Jlink}'(\mathit{BB}(n), \beta, m \to n))\Big) && (11)\\
\mathit{KILL}(\alpha, n) &= \mathit{AVAIL}^{all}_{in}(\alpha, n) \vee \mathit{ANTIC}^{all}_{in}(\alpha, n) && (12)\\
\mathit{AVAIL}^{Msome}_{out}(\alpha, n) &= (\mathit{AVAIL}^{Msome}_{in}(\alpha, n) \wedge \mathit{KILL}(\alpha, n))\\
&\qquad \vee (n \in \mathit{DefNodes}(\alpha)) && (13)
\end{aligned}
$$

Predicate (8) denotes the case where β is converted into α when the control flow is traced backward. AVAIL^{Msome} is akin to AVAIL^{some}, but the condition KILL is added to prevent speculative insertion of instructions.

PVNRE solves the above equations by computing maximum fixed points.

Theorem 5.14 *Maximum fixed point (MFP) solutions to AVAIL^{all}, AVAIL^{some}, and AVAIL^{Msome} are equal to their meet-over-all-paths (MOP) solutions. An MFP solution to ANTIC^{all} does not match its MOP solution, but is a correct one.*

The proof of the theorem can be seen in the Appendix B.

The correctness of the MFP solution to ANTIC^{all} ensures soundness of PVNRE, but the incompleteness of the solution implies that there might exist some redundancy that PVNRE cannot eliminate. In fact, however, programs encountered in real life rarely contain such redundancies.

5.9 Formalization of Insertion Points

In order to make partially redundant instructions totally redundant, PVNRE inserts arithmetic instructions and ϕ functions into the original program. In this section, we use the following notations.

$$
\begin{aligned}
\mathit{AVAIL} = \mathsf{Must} &\overset{\text{def}}{\Leftrightarrow} \mathit{AVAIL}^{all} \wedge \mathit{AVAIL}^{Msome}\\
\mathit{AVAIL} = \mathsf{May} &\overset{\text{def}}{\Leftrightarrow} \neg\mathit{AVAIL}^{all} \wedge \mathit{AVAIL}^{Msome}\\
\mathit{AVAIL} = \mathsf{No} &\overset{\text{def}}{\Leftrightarrow} \neg\mathit{AVAIL}^{all} \wedge \neg\mathit{AVAIL}^{Msome} \ .
\end{aligned}
$$

Definition 5.15 (*InsertNormal*)**.** *For $\alpha \in \mathit{Nums}, m, n \in \mathit{Nodes}$,*

$$
\begin{aligned}
&\mathit{InsertNormal}(\alpha, m \to n) \overset{\text{def}}{\Leftrightarrow}\\
&(\mathit{AVAIL}_{in}(\alpha, n) = \mathsf{No} \wedge (\mathsf{n} \in \mathit{DefNormalNodes}(\alpha))) && (14)\\
&\vee(\neg\, \exists \beta \ . \ \alpha \in \mathit{Jlink}'(\mathit{BB}(n), \beta, m \to n)\\
&\ \wedge \mathit{AVAIL}_{out}(\alpha, m) = \mathsf{No} \wedge \mathit{AVAIL}_{in}(\alpha, n) = \mathsf{May} \wedge \mathit{ANTIC}^{all}_{in}(\alpha, n)) && (15)\\
&\vee(\exists \gamma \ . \ \gamma \in \mathit{Jlink}'(\mathit{BB}(n), \alpha, m \to n)\\
&\ \wedge \mathit{AVAIL}_{out}(\alpha, m) = \mathsf{No} \wedge \mathit{AVAIL}_{in}(\gamma, n) = \mathsf{May} \wedge \mathit{ANTIC}^{all}_{in}(\gamma, n)) \ . && (16)
\end{aligned}
$$

Predicate (14) represents the points immediately before the non-redundant instructions. PVNRE (and also PRE) stores the value of a non-redundant instruction to a temporary variable, and modifies the following instructions which compute the same value, so that they refer to this variable instead of computing the value. (15) and (16) are the conditions to insert instructions at the edges where availability turns from No to May. They denote the cases where a value number is converted or not converted, respectively, at a join node.

Definition 5.16 **(*InsertPhi*).** *For* $\alpha_0, \alpha_1, \alpha_2 \in Nums, N \in BBNodes,$

$$
\begin{aligned}
&InsertPhi(\alpha_2, N, \alpha_0, \alpha_1) \overset{\text{def}}{\Leftrightarrow} \\
&\alpha_2 \in Jtarget'(N, \alpha_0, \alpha_1) && (17)\\
&\wedge(AVAIL_{in}(\alpha_2, Head(N)) = \mathsf{Must} \\
&\quad \vee(AVAIL_{in}(\alpha_2, Head(N)) = \mathsf{May} \wedge ANTIC^{all}_{in}(\alpha_2, Head(N)))) && (18)\\
&\wedge\neg\ \exists n \in Nodes\ .\ (n.Op = \phi_N \wedge Num(n) = \alpha_2 \\
&\qquad \wedge Num(n.Lt) = \alpha_0 \wedge Num(n.Rt) = \alpha_1)\ . && (19)
\end{aligned}
$$

ϕ functions are inserted to explicitly represent the conversion of value numbers in a program. Condition (18) is necessary because there must exist instructions referred to by the operands of ϕ functions. In addition, (19) implies that it is not necessary to insert ϕ functions whose equivalents already exist in the original program.

5.10 Instruction Insertion and Redundancy Elimination

PVNRE assigns a unique variable to each value number. These new variables must not be the same as the variables in the original program. $Var(\alpha)$ denotes a variable assigned to value number α.

Let $InsertionNums(m \rightarrow n)$ be a set of value numbers of arithmetic instructions that is to be inserted at $m \rightarrow n$. PVNRE inserts instructions according to the ascending order of value numbers in *InsertionNums*. The inserted instruction for α is $Var(\alpha) := Var(\alpha.Lt)\ \alpha.Op\ Var(\alpha.Rt)$. The reason for the ascending order insertion is that if there is true data dependency between value numbers in *InsertionNums*, we need to insert the dependent one below. Here, we utilize the numerical order constraint (Definition 5.1).

As for ϕ functions, given $InsertPhi(\alpha_2, N, \alpha_0, \alpha_1)$, the instruction to be inserted at the head of block N is $Var(\alpha_2) := \phi_N(Var(\alpha_0), Var(\alpha_1))$.

Furthermore, for $n \in Nodes$ s.t. $n.Op \notin Normal \wedge n \in DefNodes(Num(n))$, x := $Var(Num(n))$, where x is a destination variable of n, is inserted immediately after n. The destination of n is then altered into $Var(Num(n))$.

Since the above insertion and transformation make all the arithmetic instructions in the original program totally redundant, we can replace the right-hand side of all those instructions with $Var(Num(n))$, thus, eliminating all the redundancy.

A brief proof of the soundness of PVNRE and its complexity can be seen in the Appendix C.

5.11 Complexity

Let p and n be the number of ϕ functions and the other instructions, respectively, in the original program. If we consider and analyze each instruction separately, one instruction generates $O(x^2)$ new ϕ functions in the worst case, where x denotes the number of ϕ functions before the data flow analysis of the instruction. Thus the final number of ϕ functions will be $O(p^{2^n})$, which means that the maximum value number can be $O(n + p^{2^n})$. This results in the complexity of $O(n^2 + np^{2^n})$ in the data flow analyses. In practice, however, one instruction generates at most p new ϕ functions, so that we can expect the complexity of $O((1 + p)n^2)$.

6 Experimental Results

We implemented PVNRE in a Java just-in-time (JIT) compiler that we are now developing, called RJJ. RJJ is invoked from kaffe-1.0.7 [10], a free implementation of a Java virtual machine. This time, we used

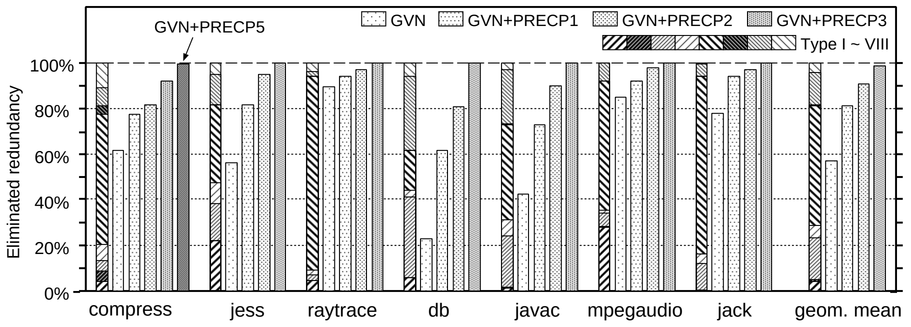


Figure 13: Dynamic counts of eliminated redundancies (PVNRE = 100%)

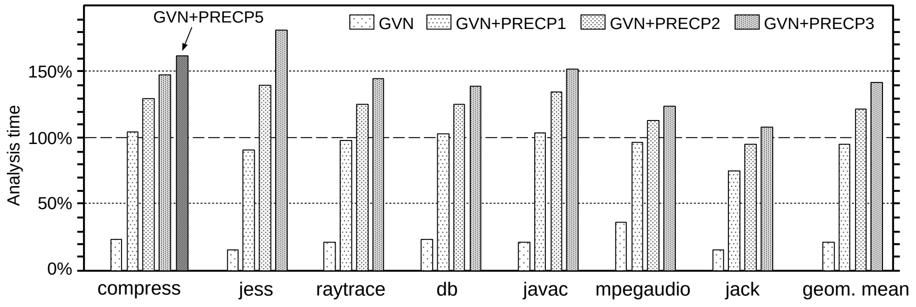


Figure 14: Analysis time (PVNRE = 100%)

RJJ only for counting the *dynamic* number of redundancies eliminated and measuring analysis time; we delegated the generation of execution code to a JIT compiler in kaffe.

We also implemented dominator-based GVN and GVN+PRECP. We refer to algorithms that execute both PRE and CP exactly once, at most twice, and at most three times as GVN+PRECP1, GVN+PRECP2, and GVN+PRECP3, respectively. GVN+PRECP2 and GVN+PRECP3 stop iteration when further improvement cannot be achieved.

We eliminated redundancy not only in arithmetic but also in load instructions. We assumed a new memory model for Java (JSR-000133), so that we aggressively eliminated redundant loads. To measure the intrinsic power of PVNRE, we did not preserve the exception order before and after optimization.

As benchmarks, we used SPECjvm98 [18]. All measurements were collected on Linux 2.4.18 with a 2.20 GHz Xeon and 512 MB main memory.

6.1 Effectiveness of Redundancy Elimination

We calculated the *dynamic* counts of reduced instructions for all the executed methods. Figure 13 shows the results. Each bar corresponds to PVNRE, GVN, GVN+PRECP1, GVN+PRECP2, and GVN+PRECP3 from left to right, except that we also show the result of GVN+PRECP5 for "compress." Further iteration of PRE and CP made no improvement. The bars are normalized to PVNRE = 100%, and the breakdown of PVNRE represents Types I – VIII from bottom to top. Note that GVN = Type I + V, and GVN+PRECP1 = Type I + II + III + IV + V. It is also worth noting that Types I and V in the results represent only the redundancies dominator-based GVN can detect. Thus, for example, Type I(2) in Fig. 1 is included in Type II in the graph. We can observe that Types II and VI do not appear in the graph except in "compress," while Type V accounts for over 50% on an average.

The result indicates that we need to iterate PRE and CP three times (or five times for "compress") to completely remove redundancies of Types VI, VII, and VIII, or in other words, to match the ability of GVN+PRECP to that of PVNRE. Those redundancy types account for 19% in dynamic counts, thus it is not sufficient to perform only GVN, or GVN and PRE with no iteration.

6.2 Analysis Time

Figure 14 shows the total analysis time of GVN and GVN+PRECPs for all the executed methods in comparison with that of PVNRE. We did not include the time to convert the program into a non-SSA form in GVN+PRECPs. Note that the analysis time of GVN+PRECPs does not increase proportionally to the number of iterations because iteration is stopped when further improvement cannot be achieved. GVN+PRECP3, which has almost the same ability to eliminate redundancy as PVNRE, is 82% slower than PVNRE for "jess," and 42% slower on an average. That means PVNRE achieves a 45% maximum speedup over GVN+PRECP3, and a 30% speedup on an average.

7 Related Work

Several approaches [16, 19] that are as powerful as PVNRE have been proposed, but actual data have never been presented concerning their analysis speed compared with traditional algorithms. In contrast, we measured the analysis time of PVNRE on real benchmarks and showed that it is faster than GVN+PRECP. Moreover, we estimate that the existing approaches are slower than PVNRE as follows. Rosen et al.[16] proposed an algorithm that converts the lexical appearance of instructions through ϕ functions. It first assigns an integer called *rank* to each instruction that represents the depth of data dependency. It then performs copy propagation, code motion and redundancy elimination separately for each class of instructions with the same rank. Therefore it is essentially as slow as PRECP. Steffen et al.[19] proposed a two-phased algorithm that first computes *Herbrand equivalences* of instructions represented by a *Value Flow Graph*. It then performs PRE on that graph. In comparison, our value numbering is much faster than its equivalence computation. In addition, our work relies solely on the traditional SSA form and on-demand value number conversion at join nodes, while their algorithm suffers from an overhead due to a necessary conversion of the whole-program graph representation.

Bodik et al.[2] proposed "Path-Sensitive Value-Flow Analysis," which extends the optimizing power of GVN and PRE by using symbolic back-substitution at the cost of analysis time. Actually, PVNRE also includes part of the extended ability, but we did not encounter such a situation in SPECjvm98 where this kind of ability is of use. Rather, the emphasis of our work is on providing the power of GVN+PRECP in a shorter analysis time.

Bodik et al.[3] also proposed the framework for PRE that we utilize in PVNRE. Its optimizing power is the same as that of the existing frameworks for PRE, hence, is weaker than that of PVNRE. Chow et al.[6] proposed PRE on the SSA form. The goal of their work was to speed up analysis by exploiting sparse data structures. Thus, its optimizing ability is just the same as that of the traditional PRE algorithms.

Alpern et al.[1] proposed GVN using a partitioning algorithm, and Rüthing et al.[17] extended it to detect *Herbrand equivalences*. PVNRE cannot use partitioning because it has to assign value numbers even while it is solving data-flow equations. Click [7] proposed an algorithm combining GVN and aggressive code motion. It first performs hash-table-based GVN, and then moves instructions out of loops using dominator relationship. It can eliminate partial redundancy, but cannot detect path-dependent redundancy. Cooper et al.[8] proposed an algorithm that first performs GVN using a hash table and then performs PRE on value numbers. It is similar to PVNRE in that it propagates value numbers in data-flow analyses, but it does not convert them using ϕ functions. Thus, it cannot remove path-dependent redundancy.

8 Conclusion

We proposed PVNRE, a redundancy elimination algorithm that manages both optimizing power and analysis time. Using value numbers in data-flow analyses, PVNRE can deal with data-dependent redundancy. It can also detect path-dependent partial redundancy by converting value numbers through ϕ functions on demand. It represents data dependency by the numerical order between value numbers; therefore it can quickly process data-dependent redundancy during data-flow analyses, and avoid the overhead due to the iteration of PRE and CP.

We implemented PVNRE in a Java JIT compiler and conducted experiments using SPECjvm98. The results showed that PVNRE has the same optimizing power but has a maximum 45% faster analysis speed than algorithms that iterate PRE and CP. These results show that PVNRE is an outstanding algorithm for a runtime optimizing compiler, where both optimizing power and analysis time are important.

We are currently integrating PVNRE with redundant exception elimination in Java. We expect that all redundancy eliminations that preserve exception orders can be performed in one pass.

References

[1] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, 1988.

[2] Rastislav Bodik and Sadun Anik. Path-sensitive value-flow analysis. In *Symposium on Principles of Programming Languages*, pages 237–251, 1998.

[3] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant computations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 1998.

[4] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *ACM SIGPLAN Notices*, 29(6):159–170, 1994.

[5] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software Practice and Experience*, 27(6):701–724, 1997.

[6] 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 1997 ACM SIGPLAN conference on Programming language design and implementation*, pages 273–286. ACM Press, 1997.

[7] Cliff Click. Global code motion: global value numbering. *ACM SIGPLAN Notices*, 30(6):246–257, 1995.

[8] Keith Cooper and Taylor Simpson. Value-driven code motion. Technical report, CRPC-TR95637-S, Rice University, 1995.

[9] 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.

[10] Kaffe.org. Kaffe Open VM. http://www.kaffe.org/.

[11] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM Press, 1973.

[12] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *ACM SIGPLAN Notices*, 27(7):224–234, 1992.

[13] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: theory and practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1117–1155, 1994.

[14] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[15] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.

[16] 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.

[17] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. Detecting equalities of variables: Combining efficiency with precision. In *Static Analysis Symposium*, pages 232–247, 1999.

[18] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/.

[19] Bernhard Steffen, Jens Knoop, and Oliver Rüthing. The value flow graph: A program representation for optimal program transformations. In *European Symposium on Programming*, pages 389–405, 1990.

Appendix

A Theorem 5.13 (Redundancy Theorem)

Definition A.1 (*DomBedges*.) *For $u \in Nodes$,*

$$DomBedges(u) \overset{\text{def}}{=} \{e = m \rightarrow n \mid e \in Bedges \wedge n \text{ dom u}\} .$$

Using the reducibility of the control-flow graph, we can prove the following two lemmas.

Lemma A.2 *For $p \in P[start, end]$ and $1 \leq j < i \leq \lambda(p)$,*

$$\begin{aligned}&p_i.Op \in Normal \cup Copy \wedge p_j = p_i.X\\ &\Rightarrow \forall j \leq k < i \;.\; p_k{\rightarrow}p_{k+1} \notin DomBedges(Node(p_i.X)) .\end{aligned}$$

Lemma A.3 *For $e \in Edges$ and $n \in Nodes$,*

$$e \notin DomBedges(n) \Rightarrow Transp(Num(n), e) .$$

The following Theorem A.4 indicates that the value numbers of the operands of an instruction is transparent along the paths between the operand instructions and the instruction.

Theorem A.4 (Transparency of Operands.)
For $p \in P[start, end]$ and $1 \leq j < i \leq \lambda(p)$,

$$p_i.Op \in Normal \cup Copy \wedge p_j = p_i.X \Rightarrow Transp^{\forall}(Num(p_j), p[i, j])$$

Proof: *This theorem follows from Lemma A.2, A.3, and Definition 5.5(2).* □

We cannot propagate *Jtarget* to the original point along the backedges of the enclosing loops as follows.

Lemma A.5 *For $\alpha_0, \alpha_1, \alpha_2 \in Nums$, $N \in BBNodes$, $p \in P[start, end]$, and $1 \leq i < j \leq \lambda(p)$,*

$$\begin{aligned}&\alpha_2 \in Jtarget(N, \alpha_0, \alpha_1) \wedge Node(p_i) = Node(p_j) = Head(N)\\ &\Rightarrow \neg Transp^{\forall}(\alpha_2, p[i, j])\end{aligned}$$

From Theorem A.4 and Lemma A.5, we can prove Theorem 5.13.

Proof: *Our proof will proceed by induction on the depth of data dependency with Fixed as a basis. From the fact that $Num(p_i) = Num(p_j)$, we can rely on the soundness of value numbering (Definition 5.12). Induction hypothesis is satisfied by Theorem A.4. For the case of Definition 5.12(4), we appeal to Lemma A.5.* □

B Theorem 5.14 (MFP and MOP solution)

The MFP solution to a problem matches its MOP solution if all of its flow functions are monotone and distributive [11]. *Transp*, ($n \in \mathit{DefNodes}(\alpha)$), and *KILL*, which appear in the data-flow equations for AVAIL^{all}, AVAIL^{some}, ANTIC^{all}, and AVAIL^{Msome}, are monotone and distributive. Jlink' is monotone, but is not distributive for $\wedge$. The flow function of Jlink' can be generalized into the form of $f(\langle x_\alpha \rangle) = \langle x_\alpha \vee x_\beta \rangle$, where $1 \leq \alpha, \beta \leq \mathit{max}(\mathit{Nums})$.

Lemma B.1 *Jlink' is monotone.*

Proof: $\langle x_\alpha \rangle \leq \langle y_\alpha \rangle \Rightarrow x_\alpha \leq y_\alpha \Rightarrow x_\alpha \vee x_\beta \leq y_\alpha \vee y_\beta \Rightarrow \langle x_\alpha \vee x_\beta \rangle \leq \langle y_\alpha \vee y_\beta \rangle \Rightarrow f(\langle x_\alpha \rangle) \leq f(\langle y_\alpha \rangle)$ □

Lemma B.2 *Jlink' is distributive for $\vee$, but not for $\wedge$.*

Proof: *For $\wedge$, $f(\langle x_\alpha \rangle \wedge \langle y_\alpha \rangle) = \langle (x_\alpha \wedge y_\alpha) \vee (x_\beta \wedge y_\beta) \rangle \neq \langle (x_\alpha \vee x_\beta) \wedge (y_\alpha \vee y_\beta) \rangle = f(\langle x_\alpha \rangle) \wedge f(\langle y_\alpha \rangle)$. We can prove the distributivity for $\vee$ in the same manner.* □

We can prove Theorem 5.14 as follows.

Proof: *From Lemma B.1 and B.2, it is clear that MFP solutions to AVAIL^{some} and AVAIL^{Msome} are equal to their MOP solutions. The proof by Kildall [11] implies that an MFP solution to ANTIC^{all} is a correct solution, but does not match its MOP solution.*

Jlink' for AVAIL^{all} is distributive because Lemma A.5 means that

$$\forall \alpha \exists \beta \;.\; \alpha \in \mathit{Jlink}'(BB(n), \beta, m \rightarrow n) \Rightarrow \neg \mathit{AVAIL}^{all}_{out}(\alpha, m),$$

and the flow function is essentially $f(\langle x_\alpha \rangle) = \langle x_\beta \rangle$. Consequently, an MFP solution to AVAIL^{all} matches its MOP solution. □

C Soundness of PVNRE

Theorem C.1 (Soundness of PVNRE). *The value of any instruction on any execution path after optimization by PVNRE is the same as that of the corresponding instruction (if any) before the optimization.*

Proof: *We can prove the syntactic soundness; for example, the operands of a newly-inserted Normal instruction whose value number is α refer to instructions whose value numbers are $\alpha.X$. Thus the value numbers of the newly-inserted instructions satisfy Definition 5.12. It follows that Theorem 5.13 holds true even after the optimization. From Theorem 5.14, Definition 5.15, and 5.16, all the Normal instructions become totally redundant in terms of Theorem 5.13. Therefore we can replace the right-hand side of all those instructions n with $\mathit{Var}(\mathit{Num}(n))$.* □