Partial Value Number Redundancy Elimination

Rei Odaira[1] and Kei Hiraki[1]

University of Tokyo, Bunkyo-ku, Tokyo, Japan,
{ray, hiraki}@is.s.u-tokyo.ac.jp

Abstract. When exploiting instruction level parallelism in a runtime optimizing compiler, it is indispensable to quickly remove redundant computations and memory accesses to make resources available. 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

Redundancy elimination is an optimizing technique that removes instructions that compute the same value as previously executed instructions, so that it can shorten the critical path and make resources available for instruction level parallelism. When using redundancy elimination in runtime optimizing compilers, which have recently gained widespread use in Java and other execution environments, we need to make it as powerful as possible, and at the same time, keep its analysis time short. In this paper, we aim to manage both optimizing power and analysis speed of redundancy elimination.

So far, the most widely used redundancy elimination algorithms have been Partial Redundancy Elimination (PRE) [1–4] and Global Value Numbering (GVN) [5–7]. PRE can eliminate redundancy on at least one, but not necessarily all execution paths leading to an instruction. 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 [8]. 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 [8]. They first perform GVN, then convert the program back into the non-SSA form, and finally execute PRE.

In reality, we need to perform PRE and copy propagation (CP) iteratively after GVN[10], because otherwise there remain many redundancies in a program as described in Section 2. We propagate copy instructions generated by the previous PRE to get as many instructions lexically identical as possible. We then make the next PRE remove the now-lexically-identical instructions. 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 paper, 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 at join nodes *on demand* 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 paper is organized as follows. Section 2 presents the types of redundancy we deal with in this paper, and explains the inefficiencies of the existing algorithms to remove such redundancies. Section 3 describes the algorithm of PVNRE and Section 4 shows the experimental results. Section 5 reviews related work, and Section 6 sets out the conclusion.

2 Background

We categorize the redundancy we deal with in this paper 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.

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)

2.1 Type I – IV

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

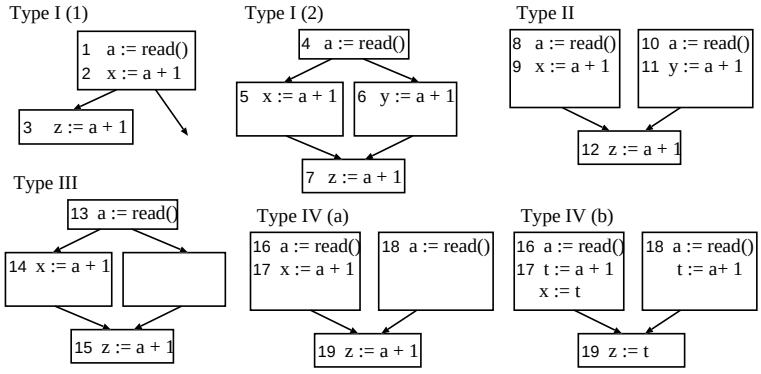

Fig. 1. Examples of optimization by PRE

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, Instruction 12 computes different values, since "a" refers to the different definitions (Instructions 8 and 10). The path-dependency becomes clearer if we convert the program into the SSA form

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, Instructions 8, 10, and the ϕ function are assigned distinct value numbers, and so are Instructions 9, 11, and 12, which makes it impossible to detect the redundancy among them.

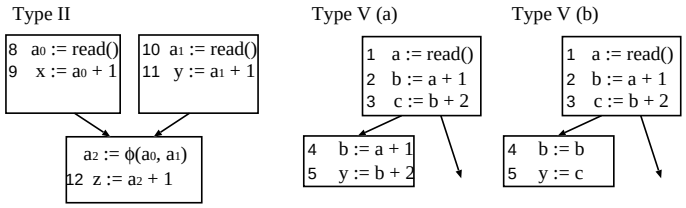

Fig. 2. Examples of optimization by GVN

The reason GVN cannot detect Type III is that Instruction 14 does not dominate 15, nor make the value number of "a + 1" available at 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.

2.2 Type V

Figure 2 Type V illustrates the type of redundancy GVN can remove but PRE cannot. Instructions 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 3 and 5 are lexically identical, their redundancy cannot be removed by PRE because there is a store for the operand between them (Instruction 4), or in other words, because they are data-dependent on different instructions (2 and 4). Therefore, we call this type of redundancy between 3 and 5 a data-dependent redundancy.

2.3 Type VI – VIII

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)

[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.

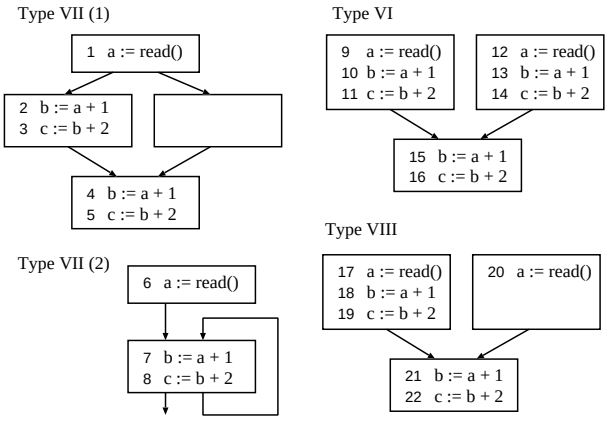

Fig. 3. GVN+PRE cannot optimize these types of redundancy

is the same as Type VII (2) in Fig. 3. The first PRE moves Instruction 7 out of the loop (b). It cannot move Instruction 8 because this is data-dependent on 7. The generated copy instruction is then propagated to 8 (c). Finally, since the data dependency has been removed, the second PRE can move 8 out of the loop (d). The redundancies of Types VI and VIII in Fig. 3 can be eliminated in the same manner.

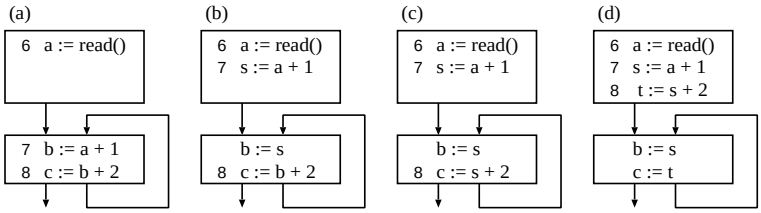

Fig. 4. Sequence of optimization by iteration of PRE and CP

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.
- 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.

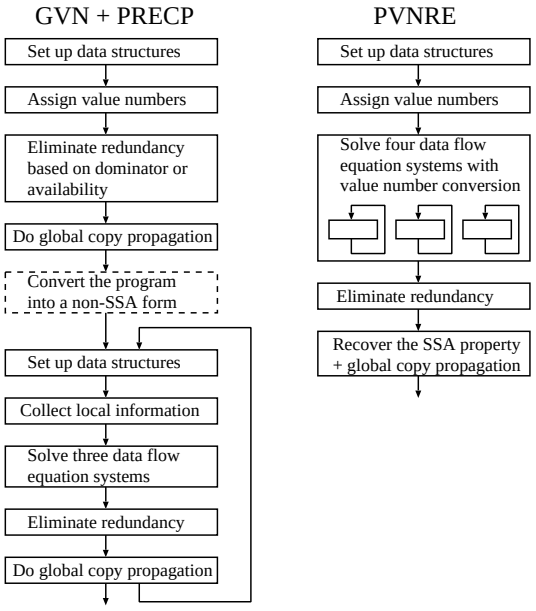

Fig. 5. GVN+PRECP and PVNRE algorithms

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

3 PVNRE Algorithm

PVNRE aims to provide equal or better optimizing power with faster analysis speed than GVN+PRECP. In this section, we give an overview of PVNRE and explain its intuitive behavior, using examples. Due to space limitations, we omit the formalization of its algorithm and the proof of correctness; these can be found in our technical report[11]. 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.
– All instructions including ϕ functions are binary, except for function calls.

The overview of the algorithm is shown on the right-hand side of Fig. 5. In fact, PVNRE can remove redundancies beyond the ones listed in Table 1, but such types of redundancy are rarely found in real programs.

3.1 Value Numbering

The value numbering in PVNRE shown in Fig. 6 is the same as that for GVN, except that (1) it computes backedges (*Backedges*) and transparency (*UnTransp*), and (2) it must assign incremental value numbers (Line 7 and 18). *Operator*, *Left*, and *Right* in the figure represent the operator and the left and right operand instructions, respectively. *GEN* is used as local information in the following data-flow analyses. Backedges and transparency are described in Sect. 3.3.

Incremental value numbers ensure that Instruction x is assigned a greater number than Instruction 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 3 and the pre-order in Line 5), so that we can use the value numbers of the left and right operands as a key to the hash table. PVNRE utilizes this numerical order in the following data-flow analyses to process value numbers in data dependency order.

```
 1 for each edge e do UnTransp(e) := ∅ end
 2 VN := 0
 3 for each basic block N in reverse post order do
 4   GEN(N) = ∅
 5   for each instruction n in N in pre-order do
 6     if Operator(n) is φ or function call then
 7       VN := VN + 1; Num(n) := VN
 8       Backedges(VN) := the set of the enclosing backedges of N
 9     else
10       α := call Hash(Operator(n), Num(Left(n)), Num(Right(n)))
11       Num(n) := α; GEN(N) := GEN(N) ∪ {α}
12     end
13   end
14 end
15
16 Hash(op, l, r) {
17   If ⟨op, l, r⟩ is registered, returns the corresponding VN. If not,
18   VN := VN + 1; Register VN with ⟨op, l, r⟩.
19   Backedges(VN) := Backedges(l) ∪ Backedges(r)
20   for each b in Backedges(VN) do
21     UnTransp(b) := UnTransp(b) ∪ {VN}
22   end
23   return VN
24 }
```

Fig. 6. Algorithm for value numbering

3.2 Data-Flow Analyses

Using value numbers in PRE-like data-flow analyses, PVNRE can detect not only the redundancies of Types I and V like GVN, but also partial redundancies, particularly Types III and VII.

PVNRE solves four equation systems for $AVAIL^{all}$, $AVAIL^{some}$, $ANTIC^{all}$, and $AVAIL^{Msome}$. These are the framework for PRE proposed by Bodik et al. [2]. $AVAIL^{all}$ 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. Because of space limitations, we refer readers to the paper by Bodik et al. [2] for details about these predicates. Here, we focus on the differences between their work and PVNRE, namely, transparency and value number conversion.

3.3 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 Instructions 3 and 4 to flow through the backedge, the information would reach 3 and 4 again; hence, we would consider them to be loop invariants. However, since these instructions are induction variables, this does not happen and they return different values for each iteration of the loop.

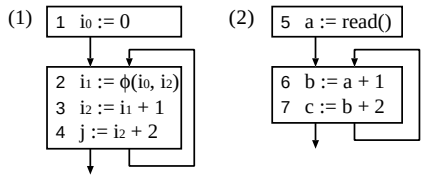

Fig. 7. Examples of backedge and transparency

Thus, we define transparency for value numbers and back-edges: 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. Therefore, during value numbering, PVNRE computes the enclosing backedges for ϕ functions and function calls

(Fig. 6 Line 8). Other instructions inherit backedges from the data-dependent instructions (Line 19), and register themselves to *UnTransp* (the complementary set of the transparency) for the backedges (Line 20 – 22). For example, in Fig. 7(2), even if Instructions 6 and 7 are inside the loop their value numbers are transparent through the backedge, so that PVNRE can detect their partial redundancy (Types III and VII, respectively).

3.4 Conversion of Value Numbers

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. It is worth noting that the reason GVN cannot detect the path-dependent redundancy is the existence of ϕ functions in the SSA form.

PVNRE uses two sets, $JT(N)$ and $JT'(N)$, which are defined for each join node N. Their elements are of the form $\langle t, l, r\rangle$, 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.

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 $\{\langle 7, 1, 4\rangle\}$ by the ϕ function (Instruction 7).

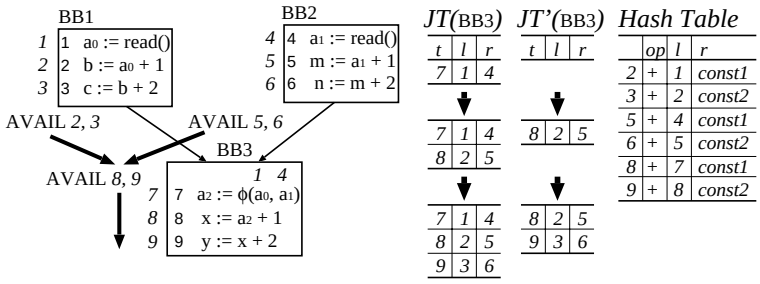

Fig. 8. Example of value number conversion for Type II and VI

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. 3.1.

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 $\langle 8, 2, 5\rangle$ to JT and JT'. In the same manner, we process *3* and add another element $\langle 9, 3, 6\rangle$. 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 Instructions 8 (Type II) and 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 $ANTIC^{all}$.

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

In summary, 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 Types VI and VIII. In addition, by using the on-demand conversion, it need not construct any special representation of redundancy such as a *Value Flow Graph* [12] in advance of the analyses.

Data-Flow Equation System. The resulting data-flow equation system for $AVAIL^{all}$ is as follows.

$$AVAIL^{all}_{in}(\alpha, N) = \bigwedge_{\forall M \in Pred(N)} \Big((AVAIL^{all}_{out}(\alpha, M) \wedge \alpha \notin UnTransp(M \rightarrow N)) \vee (AVAIL^{all}_{out}(\beta, M) \text{ s.t. } \beta \text{ is converted to } \alpha \text{ along } M \rightarrow N \text{ by } JT'(N)) \Big)$$

$$AVAIL^{all}_{out}(\alpha, N) = AVAIL^{all}_{in}(\alpha, N) \vee \alpha \in GEN(N)$$

[2] If we could not find 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.

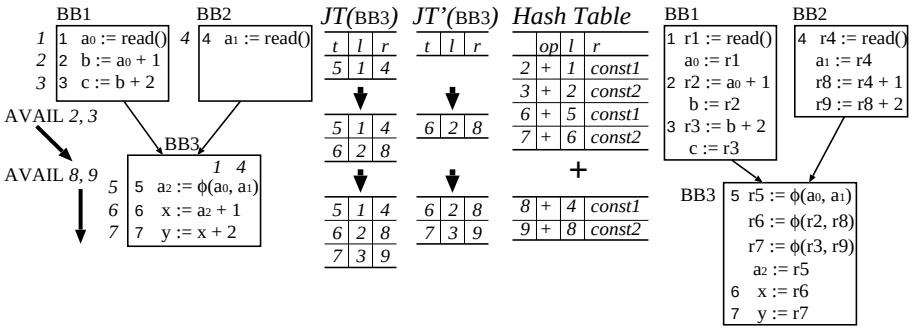

Fig. 9. Example of value number conversion for Type IV and VIII, and the result of redundancy elimination

The equation systems for $AVAIL^{some}$, $ANTIC^{all}$, and $AVAIL^{Msome}$ are similar to the one above [11].

3.5 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 [2]. 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 [2]. 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.

3.6 Complexity

Let p and n be the number of ϕ functions and the other instructions, respectively, in the original program. The complexity of PVNRE is $O(n^2 + np^{2^n})$ in the worst case. However, in practice we can expect the complexity to be $O((1 + p)n^2)$. Details can be found in our technical report[11].

4 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 [13], a free implementa-

tion of a Java virtual machine. This time, we used 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 [14]. All measurements were collected on Linux 2.4.18 with a 2.20 GHz Xeon and 512 MB main memory.

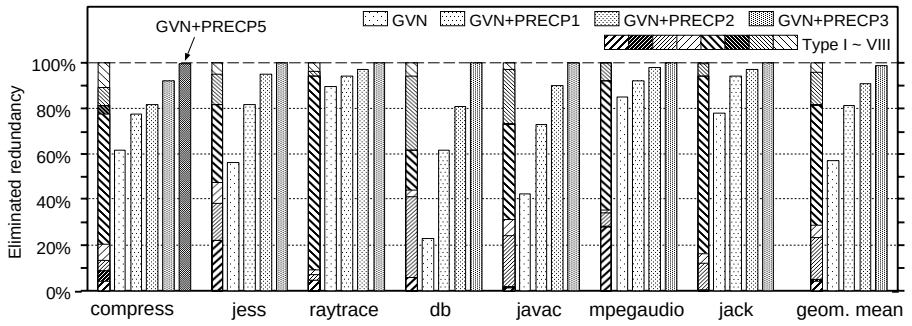

Fig. 10. Dynamic counts of eliminated redundancies (PVNRE = 100%)

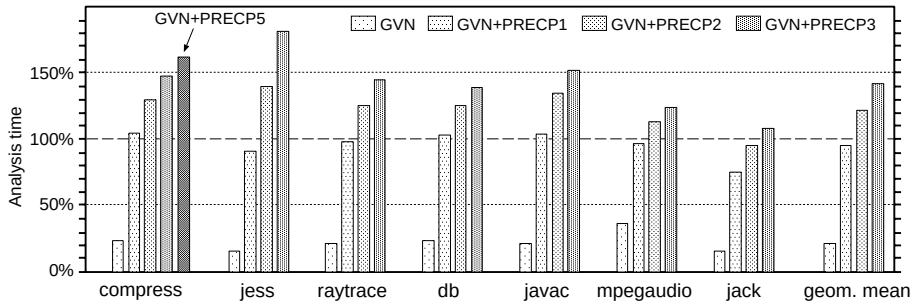

Fig. 11. Analysis time (PVNRE = 100%)

4.1 Effectiveness of Redundancy Elimination

We calculated the *dynamic* counts of reduced instructions for all the executed methods. Figure 10 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 Type V accounts for over 50% on an average.

In these programs, we need to iterate PRE and CP at least three times 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.

4.2 Analysis Time

Figure 11 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.

5 Related Work

Several approaches [10, 12] 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.[10] 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.[12] 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.[4] 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.[2] 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.[15] 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.[5] proposed GVN using a partitioning algorithm, and Rüthing et al.[16] 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 [6] 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.[7] 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. VanDunen et al.[17] proposed "Value-based Partial Redundancy Elimination", but it cannot eliminate part of Type I, II, V, and VI. In addition, their approach cannot use efficient bit-vector implementation in data-flow analyses.

6 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 to

exploit instruction level parallelism in 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. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. ACM SIGPLAN Notices **27** (1992) 224–234
2. Bodik, R., Gupta, R., Soffa, M.L.: Complete removal of redundant expressions. In: SIGPLAN Conference on Programming Language Design and Implementation. (1998) 1–14
3. Briggs, P., Cooper, K.D.: Effective partial redundancy elimination. ACM SIGPLAN Notices **29** (1994) 159–170
4. Bodik, R., Anik, S.: Path-sensitive value-flow analysis. In: Symposium on Principles of Programming Languages. (1998) 237–251
5. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California (1988) 1–11
6. Click, C.: Global code motion: global value numbering. ACM SIGPLAN Notices **30** (1995) 246–257
7. Cooper, K., Simpson, T.: Value-driven code motion. Technical report, CRPC-TR95637-S, Rice University (1995)
8. Muchnick, S.S.: Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers (1997)
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13** (1991) 451–490
10. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (1988) 12–27
11. Odaira, R., Kei, H.: Partial value number redundancy elimination. Technical report, University of Tokyo (2004) http://www-hiraki.is.s.u-tokyo.ac.jp/members/ray/pvnre_tr.ps.gz.
12. Steffen, B., Knoop, J., Rüthing, O.: The value flow graph: A program representation for optimal program transformations. In: European Symposium on Programming. (1990) 389–405
13. Kaffe.org: (Kaffe Open VM) http://www.kaffe.org/.
14. Standard Performance Evaluation Corporation: (SPEC JVM98 Benchmarks) http://www.spec.org/osg/jvm98/.
15. Chow, F., Chan, S., Kennedy, R., Liu, S.M., Lo, R., Tu, P.: 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, ACM Press (1997) 273–286
16. Rüthing, O., Knoop, J., Steffen, B.: Detecting equalities of variables: Combining efficiency with precision. In: Static Analysis Symposium. (1999) 232–247
17. VanDrunen, T., Hosking, A.L.: Value-based partial redundancy elimination. In: Compiler Construction (CC). (2004) 167–184