Effective Partial Redundancy Elimination

Preston Briggs
Keith D. Cooper

Department of Computer Science
Rice University
Houston, Texas 77251–1892

Abstract

Partial redundancy elimination is a code optimization with a long history of literature and implementation. In practice, its effectiveness depends on issues of naming and code shape. This paper shows that a combination of *global reassociation* and *global value numbering* can increase the effectiveness of partial redundancy elimination. By imposing a discipline on the choice of names and the shape of expressions, we are able to expose more redundancies.

As part of the work, we introduce a new algorithm for global reassociation of expressions. It uses global information to reorder expressions, creating opportunities for other optimizations. The new algorithm generalizes earlier work that ordered FORTRAN array address expressions to improve optimization [25].

1 Introduction

Partial redundancy elimination is a powerful optimization that has been discussed in the literature for many years (*e.g.*, [21, 8, 14, 12, 18]). Unfortunately, partial redundancy elimination has two serious limitations. It can only recognize lexically-identical expressions; this makes effectiveness a function of the choice of names in the front end. It cannot rearrange subexpressions; this makes effectiveness a function of the shape of the code generated by the front end. The net result is that decisions made in the design of the front end dictate the effectiveness of partial redundancy elimination.

This paper shows how an optimizer can use *global reassociation* (see Section 3.1) and a form of *partition-based global value numbering* [2] to improve the effectiveness of partial redundancy elimination. We consider these to be *enabling* transformations. They do not improve the code directly; instead, they rearrange the code to make other transformations more effective. The combination of these transformations with partial redundancy elimination results in removing more redundant expressions, hoisting more loop-invariant expressions (and sometimes hoisting them farther), and removing some *partially-dead expressions*. By using global reassociation and partition-based global value numbering to generate the code shape and name space automatically, the optimizer can isolate partial redundancy elimination from the vagaries of the front end. This lets the optimizer obtain good results on code generated by sources other than a careful front end – for example, on code resulting from other optimization passes or from restructuring front ends.

The primary contributions of this paper are: (1) the use of reassociation to achieve a canonical *code shape* for expressions, (2) the use of partition-based global value numbering to achieve a canonical *naming*, and (3) a new technique for global reassociation of expressions. Additionally, we present experimental evidence that demonstrates the effectiveness of partial redundancy elimination, with and without our transformations.

2 Partial Redundancy Elimination

Partial redundancy elimination (PRE) is a global optimization introduced by Morel and Renvoise [21]. It combines and extends two other techniques.

common subexpression elimination An expression is *redundant* at some point p if and only if it is computed along every path leading to p and none of its constituent subexpressions has been redefined. If e is redundant at p, the evaluation of e at p can be replaced with a reference.

loop-invariant code motion An expression is *loop invariant* if it is computed inside a loop and its value is identical in all the iterations of the loop. If e is invariant in a loop, it can be computed before the loop and referenced, rather than evaluated, inside the loop.

PRE combines and extends these two techniques.

This work has been supported by ARPA through ONR grant N00014-91-J-1989.

To appear in Sigplan PLDI, June 1994.

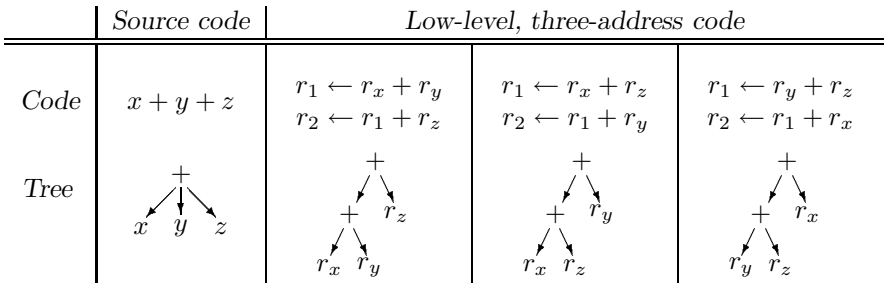
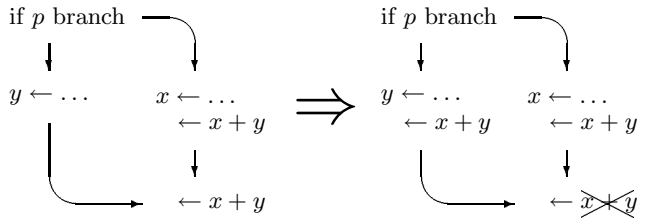


Figure 1: Alternate Code Shapes

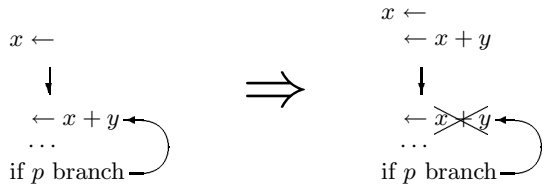
An expression is *partially redundant* at point p if it is redundant along some, but not all, paths that reach p. PRE converts partially-redundant expressions into redundant expressions. The basic idea is simple. First, it uses data-flow analysis to discover where expressions are partially redundant. Next, it solves a data-flow problem that shows where inserting copies of a computation would convert a partial redundancy into a full redundancy. Finally, it inserts the appropriate code and deletes the redundant copy of the expression.

A key feature of PRE is that it never lengthens an execution path. To see this more clearly, consider the example below. In the fragment on the left, the second computation of $x + y$ is partially redundant; it is only available along one path from the `if`. Inserting an evaluation of $x+y$ on the other path makes the computation redundant and allows it to be eliminated, as shown in the right-hand fragment. Note that the left path stays the same length while the right path has been shortened.



partially redundant *redundant*

Loop-invariant expressions are also partially redundant, as shown in the example below. On the left, $x + y$ is partially redundant since it is available from one predecessor (along the back edge of the loop), but not the other. Inserting an evaluation of $x + y$ before the loop allows it to be eliminated from the loop body.



partially redundant *redundant*

2.1 Code Shape

The optimizer in our compiler uses a low-level intermediate language. Most operations have three addresses: two source operands and a target. Translating a source expression to three-address code can introduce artificial ordering constraints. Figure 1 shows the different possibilities for the source expression $x + y + z$.

Consider the case where $r_x = 3$, $r_z = 2$, and r_y is a variable. Only the middle shape will allow constant propagation to transform the expression into $y + 5$. Alternatively, if r_y and r_z are both loop invariant, only the rightmost shape will allow PRE to hoist the loop-invariant subexpression. This case is quite important, since it arises routinely in multi-dimensional array addressing computations.

The choice of expression ordering occurs with associative operations such as *add*, *multiply*, *and*, *or*, *min*, and *max*. In general, there are a combinatorial number of orderings for an associative expression having n operands. Source language specifications sometimes restrict possible reorderings, especially in the case of floating-point arithmetic where numerical precision may be affected. The large number of possible orderings makes an exhaustive search for optimal solutions impractical.

2.2 Naming

Another important issue is the selection of *names*. Our implementation of PRE distinguishes between variable names and expression names. This distinction was introduced by Morel and Renvoise [21, page 97]. A *variable name* is the target of a copy instruction; conceptually, these correspond to source-level assignments. An *expression name* is the target of a computation – in practice, an instruction other than a branch or copy. This gives every expression (and subexpression) a name. Thus, the statement `i = i + 1` might be represented as:

$$r_1 \leftarrow 1$$
$$r_2 \leftarrow r_i + r_1$$
$$r_i \leftarrow r_2$$

where r_i is the name of the variable `i`, r_1 is the name of the expression "1", and r_2 is the name of expression

"$r_i + r_1$". Within a single routine, lexically-identical expressions always receive the same name. Therefore, whenever we see the expression $r_i + r_1$, we would expect to see it named r_2.

This naming discipline can be implemented in the compiler's front end by maintaining a hash table of expressions and creating a new name whenever a new expression is discovered [3]. Unfortunately, relying on the front end limits the applicability of PRE. It is difficult to maintain the naming rules across other optimizations; thus, PRE must be run first and only once. Furthermore, the ability of PRE to recognize identities is limited by the programmer's choice of variable names. Consider the following source sequence and its corresponding intermediate representation:

```
x = y + z          r1 ← ry + rz
                   rx ← r1
a = y              ra ← ry
b = a + z          r2 ← ra + rz
                   rb ← r2
```

Obviously, r_1 and r_2 receive the same value (that is, the expression named by r_2 is redundant). PRE cannot discover this fact even though value numbering can eliminate this redundancy [10]. Of course, this is a simple example, but its very simplicity should suggest the large number of opportunities missed by PRE when considering an entire routine.

3 Effective PRE

To address the limitations of PRE, we propose a set of techniques that reorder and rename expressions. *Global reassociation* uses information about global code shape to rearrange individual expressions. *Global value numbering* uses knowledge about run-time equivalence of values to rename expressions. In combination, they transform the program in a way that exposes more opportunities to PRE.

3.1 Global Reassociation

To address the code shape problems, we use a technique called global reassociation. It uses algebraic properties of arithmetic to rearrange the code. In broad terms, it uses commutativity, associativity, and distributivity to expose common subexpressions and loop-invariant expressions. The effects can be substantial; Cocke and Markstein note that as much as 50% of the code in some inner loops can be eliminated as a result of reassociation [9, page 225]. Our approach has three steps:

1. Compute a *rank* for every expression.
2. Propagate expressions forward to their uses.
3. Reassociate expressions, sorting their operands by ranks.

```
FUNCTION foo(y, z)
  s = 0
  x = y + z
  DO i = x, 100
    s = 1 + s + x
  ENDDO
  RETURN s
END foo
```

Figure 2: Source Code

The next three sections discuss these steps and introduce several important refinements. To help clarify the process, we provide a running example. Figure 2 shows the source code and Figure 3 shows a translation into a simple intermediate form. This translation does *not* conform to the naming discipline discussed in Section 2.2.

Computing Ranks To guide reassociation, the optimizer assigns a *rank* to each expression and subexpression. Intuitively, we want loop-invariant expressions to have lower ranks than loop-variant expressions. In a deeply nested loop, we would like the rank of an expression that is invariant in the inner two loops to be lower than the rank of an expression that is invariant only in the innermost loop. In practice, we compute ranks on the SSA form of the routine during a reverse-postorder traversal of the control-flow graph; therefore, our first step is to build the *pruned* SSA form of the routine [11, 7]. During the renaming step [11, Figure 12], we remove all copies, effectively folding them into ϕ-nodes. This approach simplifies the intermediate code by removing our dependence on the programmer's choice of variable names (recall Section 2.2).

Given the SSA form, we traverse the control-flow graph in reverse postorder, assigning ranks. Each block is given a rank as it is visited, where the first block visited is given rank 1, the second block is given rank 2,

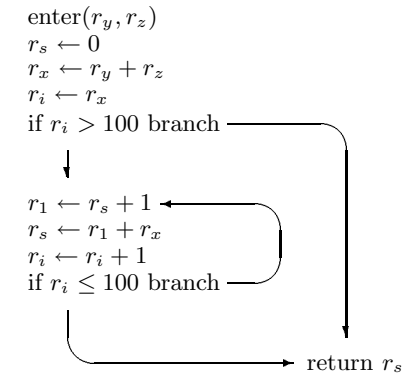


Figure 3: Intermediate Form

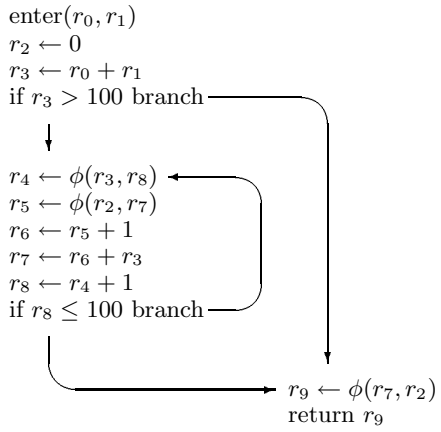


Figure 4: Pruned SSA Form

and so forth. Each expression in a block is ranked using three rules:

1. A constant receives rank zero.
2. The result of a ϕ-node receives the rank of the block, as do any variables modified by procedure calls. This includes the result of a load instruction.
3. An expression receives a rank equal to its highest-ranked operand. Since the code is in SSA form, each operand will have one definition point and will have been ranked before it is referenced.

Figure 4 shows the result of rewriting into pruned SSA form (*minimal* SSA would have required many more ϕ-nodes). Notice that the copy $r_i \leftarrow r_x$ has been folded into the first ϕ-node. The rank of r_2 is 0, the rank of r_0, r_1, and r_3 is 1, the rank of $r_4, r_5, \ldots, r_8$ is 2, and the rank of r_9 is 3. These ranks have the intuitive properties described above – loop-invariant expressions are of lower rank than loop-variant expressions and the rank of a loop-variant expression corresponds to the nesting depth of the loop in which it changes.

Forward Propagation After ranks have been computed, we copy expressions forward to their uses. Forward propagation is important for several reasons. It builds large expressions from small expressions, allowing more scope for reassociation. Additionally, without forward propagation into loops, the compiler would have to cycle between reassociation and PRE to ensure best results with deeply-nested loops. Finally, forward propagation avoids a subtle problem in PRE that arises from the distinction between variable names and expression names (see Section 5.1). As a matter of correctness, the last reason seems to require forward propagation.

We propagate each expression and subexpression as far forward as possible, effectively building expression trees for ϕ-node inputs, values used to control program flow, parameters passed to other routines, and values returned from the current routine. In practice, we first remove each ϕ-node $x \leftarrow \phi(y, z)$ by inserting the copies $x \leftarrow y$ and $x \leftarrow z$ at the end of the appropriate predecessor blocks, then trace from each copy back along the SSA graph to construct the expression trees. (If necessary, the entering edges are split and appropriate predecessor blocks are created.)

Continuing our example, Figure 5 shows how ϕ-nodes are eliminated by inserting copies. New blocks were required to hold the copies. Figure 6 shows the effect of forward propagation.

It is interesting to note that forward propagation eliminates *partially-dead* expressions [15, 19]. An expression is live at its definition point if its result is used on some path to an exit. Alternatively, an expression is dead if its result will never be used on any path. By copying expressions to their use points, forward propagation trivially ensures that every expression is used on every path to an exit. Subsequent application of PRE will preserve this invariant, since PRE will never place an expression on a path where it is partially dead.

On the other hand, forward propagation is not really an optimization. Since it duplicates code, it can expand the size of the routine (see Section 4.3). Furthermore, it can move code *into* loops, substantially increasing path lengths. However, recall that our plan is to transform the code so that later application of PRE will achieve greater optimization. We expect that PRE will be able to reverse the negative effects of forward propagation and achieve significantly improved code as a result of the opportunities afforded by forward propagation.

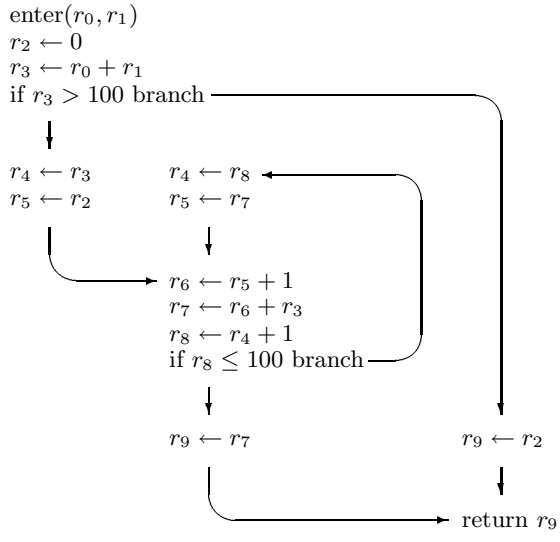


Figure 5: After Inserting Copies

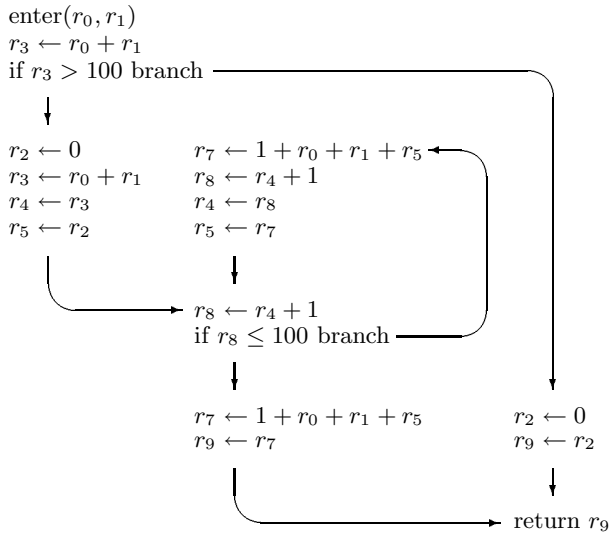


Figure 6: After Forward Propagation

Sorting Expressions Given ranks and expression trees, we are almost ready to reassociate expressions. First, though, we rewrite certain operations to expose more opportunities for reassociation. As suggested by Frailey [17], we rewrite expressions of the form $x-y+z$ as $x + (-y) + z$, since addition is associative and subtraction is not. We also perform similar transformations for Boolean operations. On the other hand, we avoid rewriting x/y as $x \times 1/y$ to avoid introducing precision problems. We rely on a later pass, a form of global peephole optimization, to reconstruct the original operations when profitable.

To reassociate, we traverse each expression, sorting the operands of each associative operation by rank so that the low-ranked operands are placed together. This allows PRE to hoist the maximum number of subexpressions the maximum distance. Furthermore, since constants are given rank 0, all the constant operands in a sum will be sorted together. For example, the expression $1+r_x+2$ becomes $1+2+r_x$. Constant propagation cannot improve the original form; it can easily turn the reordered expression into $3 + r_x$.

Figure 7 shows the result of reassociation. Notice that the low-ranked expressions, $1, r_0$, and r_1, have been sorted to the beginning of the sums.

After sorting expressions, we look for opportunities to distribute multiplication over addition; that is, we rewrite expressions of the form $w \times (x + y + z)$ as $w \times x + w \times y + w \times z$. This distribution is not always profitable, so we again use ranks as a guide. In our current implementation, we distribute a low-ranked multiplier over a higher-ranked sum. For example, if we have an expression $a + b \times ((c + d) + e))$, where *a, b, c,* and d have rank 1 and e has rank 2, we would distribute partially, giving $a+b\times(c+d)+b\times e$. This allows PRE to hoist $a+b\times(c+d)$ even if $b\times e$ cannot be hoisted. Note that a complete distribution would result in extra multiplications without allowing any additional code motion. It is important to re-sort sums after distribution.

3.2 Global Renaming

To address the naming problems, we use a global renaming scheme based on Alpern, Wegman, and Zadeck's algorithm for determining when two variables have the same value [2]. We refer to their technique as "partition-based global value numbering". Instead of building up complex equality relationships from simpler ones, as in traditional value numbering, their technique works from the "optimistic" assumption that all variables are equivalent and uses the individual statements in the code to disprove equivalences.

We use a straightforward version of their algorithm to discover when two names have the same value and then rename all values to reflect these equivalences. Renaming encodes the value equivalences into the name space; this exposes new opportunities to PRE. It also constructs the name space required by PRE (recall Section 2.2). Each lexically-identical expression will have the same name; copies inserted during reassociation will only target variable names. Of course, the "variables" named at this point do not necessarily correspond to source variables; instead, they correspond to the ϕ-nodes introduced during conversion to SSA form. The names are the only things changed during this phase; no instructions are added, deleted, or moved.

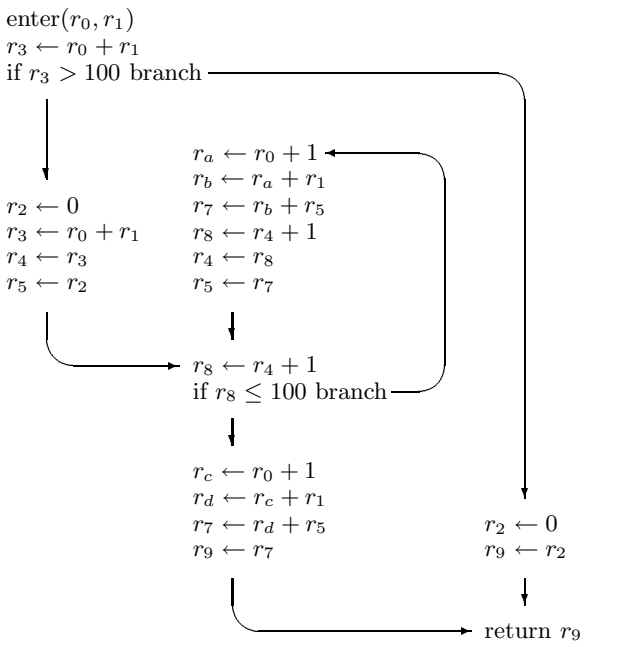


Figure 7: After Reassociation

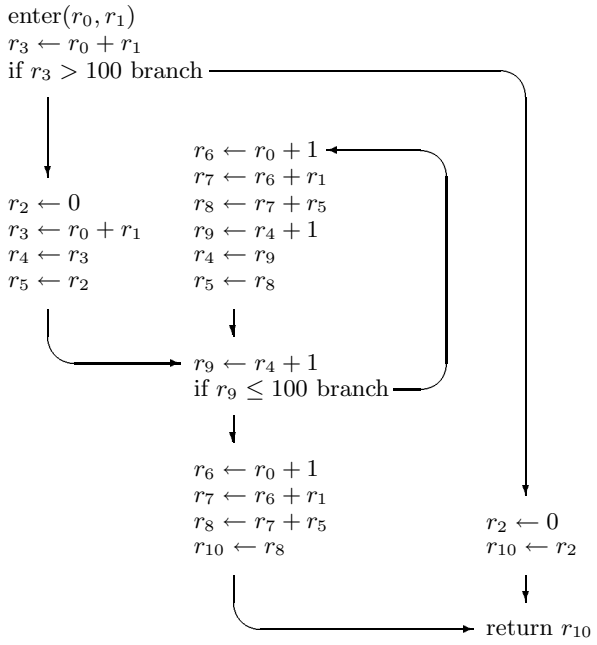


Figure 8: After Value Numbering

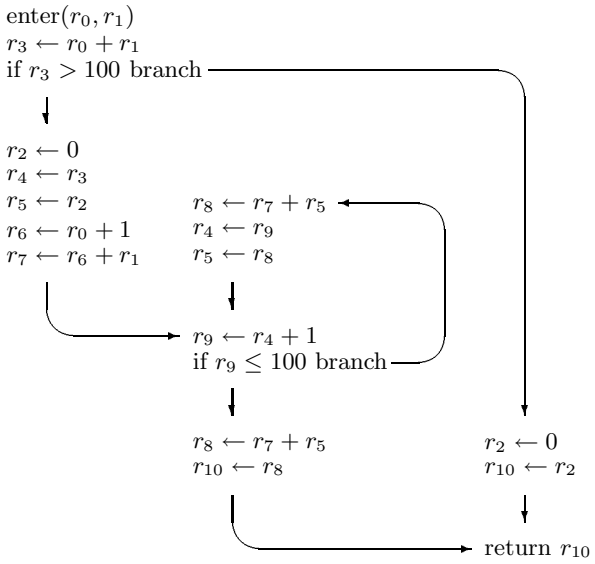


Figure 9: After Partial Redundancy Elimination

Figure 8 shows a naming that might be discovered by global value numbering. In this case, none of the exposed redundancies are particularly surprising, since we created them during forward propagation. However, it is important to note that the code now conforms with the naming requirements stated in Section 2.2. Expressions are named uniquely by $r_0, r_1, r_2, r_3, r_6, r_7, r_8$, and r_9. The remaining names, r_4, r_5, and r_{10}, are defined exclusively by copies and serve as variable names.

Finishing the Example Applying partial redundancy elimination to the code in Figure 8 produces the code in Figure 9. The invariant expressions r_6 and r_7 have been hoisted from the loop and the redundant computations of r_3, r_6, and r_7 have been removed. Finally, the coalescing phase of a Chaitin-style global register allocator will remove unnecessary copy instructions [6]. In this example, coalescing is able to remove all the copies (as shown in Figure 10), though this will not always be possible.

Taken together, the sequence of transformations reduced the length of the loop by 1 operation without increasing the length of any path through the routine. However, it is worth noting that the final code is not optimal. If the expressions r_6 and r_7 had been arranged differently, we would have been able to take advantage of the fact that $r_0 + r_1$ had already been computed. As noted in Section 2.2, finding the optimal solution would require examination of a combinatorial number of cases. We use a fast heuristic that produces good, though not optimal, results.

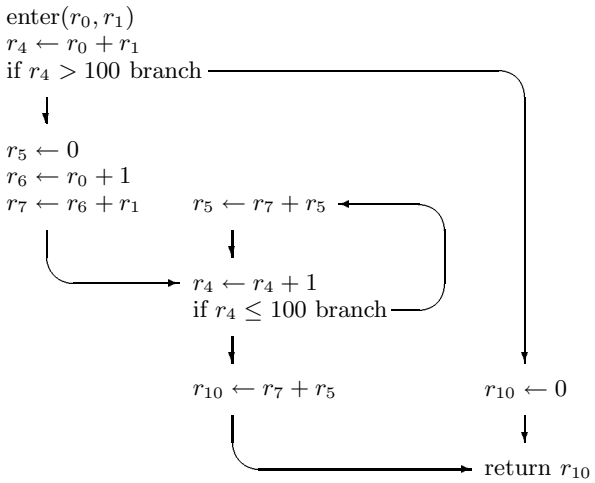


Figure 10: After Coalescing

4 Experimental Study

To test the effectiveness of our techniques, we have implemented versions of global reassociation, global value numbering, and partial redundancy elimination in the context of an experimental FORTRAN compiler. The compiler is structured as a front end that consumes FORTRAN and produces ILOC (our intermediate language), an optimizer that consumes and produces ILOC, and a back end that consumes ILOC and produces C. The generated C code is instrumented to accumulate dynamic counts of ILOC operations. Thus, we are able to compile individual FORTRAN routines, perhaps selected from a large program, and test the effectiveness of different optimizations on the routine in the context of its complete program.

The optimizer is structured as a sequence of passes, where each pass is a Unix filter that consumes and produces ILOC. Each pass performs a single optimization, including all the required control-flow and data-flow analyses. While this approach is not suitable for production compilers, its flexibility makes it ideal for experimentation.

Our implementation of PRE uses a variation described by Drechsler and Stadel [14]. Their formulation supports edge placement for enhanced optimization and simplifies the data-flow equations that must be solved, avoiding the bidirectional equations typical of some other approaches [13]. Our implementation of global value numbering uses the simplest variation described by Alpern, Wegman, and Zadeck, possibly missing some opportunities discovered by their more powerful approaches [2, Sections 3 and 4].

4.1 Results

We ran several versions of the optimizer on a suite of test routines. Each version adds new passes to the previous one. Our test suite consists of 50 routines, drawn from the Spec benchmark suite and from Forsythe, Malcolm, and Moler's book on numerical methods [16]. The results are given in Table 1. We report results for four different levels of optimization:

baseline This column provides the dynamic operation count, including branches, for each routine when optimized using a sequence of global constant propagation [26], global peephole optimization, global dead code elimination [11, Section 7.1], coalescing, and a final pass to eliminate empty basic blocks.[1]

partial The left column gives the operation counts for routines optimized with PRE, followed by the sequence of optimizations used to establish the baseline. The right column gives the percentage improvement over the baseline.

reassociation The left column provides the operation counts for routines optimized using global reassociation (*without* distribution of multiplication over addition) and global value numbering before PRE and the other optimizations. The right column gives the percentage improvements over *partial*.

distribution The left column gives the operation counts for routines optimized using global reassociation (including distribution of multiplication over addition) and global value numbering before PRE and the other optimizations. The right column gives the percentage improvements over *reassociation*.

The *total* column gives the percentage improvements achieved over the baseline by the entire set of optimizations, while the *new* column gives the improvement over *partial* contributed by the combination of reassociation and distribution with global value numbering.

Empty entries indicate no improvement, whereas entries of 0% and −0% indicate very small improvements and degradations.

Limitations of the Optimizer Our optimizer is not complete. In particular, we are currently missing passes for strength reduction and hash-based value numbering. Nevertheless, we believe our results are still valid indications of the importance of reassociation. Indeed, it may be that our results understate the eventual benefits – strength reduction should reduce non-essential overhead and hash-based value numbering should also benefit from reassociation.

4.2 Code Degradation

The results in Table 1 reveal several cases where our "improvements" slowed down the code. Since we are using heuristic approaches to difficult problems, we should not be surprised by occasional losses, annoying as they are. Examination of the code revealed three sources of difficulty; each is discussed in the sections below.

Reassociation Sometimes reassociation can disguise common subexpressions. Recall our example from Figures 2 though 10. The final arrangement of the code,

$$r_4 \leftarrow r_0 + r_1$$

and

$$r_6 \leftarrow r_0 + 1$$
$$r_7 \leftarrow r_6 + r_1$$

hid the fact that $r_0 + r_1$ was being recomputed. We found that this sort of problem occurred quite often in the routines of our test suite. Fortunately, the effect is usually dominated by the improved motion of loop invariants.

[1] The sizes of the test cases for `matrix300` and `tomcatv` have been reduced to ease testing.

routine	*baseline*	*partial*		*reassociation*		*distribution*		*new*	*total*
fmin	4,817	3,807	20%	1,908	49%	1,908		49%	60%
gamgen	462,285	180,260	61%	143,065	20%	107,031	25%	40%	76%
fmtset	705	538	23%	460	14%	397	13%	26%	43%
rkf45	62	62		58	6%	46	20%	25%	25%
sgemv	1,496	1,341	10%	1,241	7%	1,003	19%	25%	32%
saxpy	867	667	23%	667		525	21%	21%	39%
iniset	75,289	56,912	24%	56,766	0%	47,426	16%	16%	37%
spline	1,659	961	42%	885	7%	802	9%	16%	51%
tomcatv	858,364,988	250,343,458	70%	251,509,201	−0%	213,985,244	14%	14%	75%
debico	6,645	3,234	51%	2,946	8%	2,802	4%	13%	57%
seval	105	98	6%	87	11%	86	1%	12%	18%
sgemm	1,393	1,095	21%	1,096	−0%	954	12%	12%	31%
cardeb	1,716	989	42%	999	−1%	889	11%	10%	48%
hmoy	47	28	40%	27	3%	25	7%	10%	46%
orgpar	188	135	28%	135		121	10%	10%	35%
repvid	4,270	3,042	28%	3,038	0%	2,762	9%	9%	35%
drepvi	409	321	21%	303	5%	294	2%	8%	28%
heat	229	201	12%	190	5%	184	3%	8%	19%
svd	6,834	4,555	33%	4,523	0%	4,234	6%	7%	38%
x21y21	403	258	35%	258		239	7%	7%	40%
inideb	1,733	888	48%	954	−7%	829	13%	6%	52%
pastem	6,353	4,070	35%	3,941	3%	3,821	3%	6%	39%
si	206	176	14%	177	−0%	165	6%	6%	19%
deseco	33,873	14,430	57%	13,864	3%	13,707	1%	5%	59%
fmtgen	236	207	12%	202	2%	195	3%	5%	17%
fpppp	7,767	5,838	24%	5,514	5%	5,514		5%	29%
yeh	160	139	13%	132	5%	132		5%	17%
paroi	7,489	3,724	50%	3,677	1%	3,571	2%	4%	52%
twldrv	122,220,766	90,895,146	25%	86,945,328	4%	87,122,050	−0%	4%	28%
debflu	8,066	5,170	35%	5,156	0%	4,965	3%	3%	38%
colbur	152	126	17%	121	3%	123	−1%	2%	19%
decomp	876	635	27%	641	−0%	617	3%	2%	29%
inithx	5,918	3,086	47%	3,067	0%	3,018	1%	2%	49%
coeray	117	105	10%	104	0%	104		0%	11%
rkfs	456	298	34%	297	0%	297		0%	34%
integr	5,803	2,424	58%	2,436	−0%	2,447	−0%	−0%	57%
subb	704	632	10%	636	−0%	636		−0%	9%
supp	906	813	10%	814	−0%	814		−0%	10%
urand	221	220	0%	221	−0%	222	−0%	−0%	−0%
zeroin	1,020	739	27%	743	−0%	743		−0%	27%
fehl	785	510	35%	510		517	−1%	−1%	34%
ihbtr	513	453	11%	452	0%	458	−1%	−1%	10%
saturr	322	318	1%	323	−1%	323		−1%	−0%
solve	223	169	24%	168	0%	172	−2%	−1%	22%
ddeflu	1,127	827	26%	854	−3%	845	1%	−2%	25%
dcoera	182	160	12%	165	−3%	165		−3%	9%
bilan	10,188	3,355	67%	3,447	−2%	3,509	−1%	−4%	65%
drigl	161	113	29%	126	−11%	125	0%	−10%	22%
prophy	15,541	3,904	74%	4,016	−2%	4,351	−8%	−11%	72%
efill	226	205	9%	230	−12%	230		−12%	−1%
routine	*baseline*	*partial*		*reassociation*		*distribution*		*new*	*total*

Table 1: Experimental Results

Distribution Similarly, distribution of multiplication over addition can cause problems in some cases. Consider the following pair of expressions arising from a pair of array accesses, one to a single-precision array and the other to a double-precision array:

$4 \times (r_i - 1)$
$8 \times (r_i - 1)$

Distribution of the multiplies would yield:

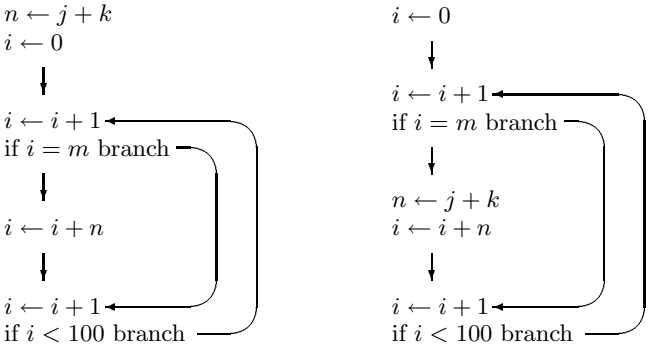
$4 \times r_i - 4 \times 1$
$8 \times r_i - 8 \times 1$

and eventually, via constant folding:

$4 \times r_i - 4$
$8 \times r_i - 8$

Unfortunately, this version is slightly worse than the original code since the original allowed commoning of the subexpression $r_i - 1$. Despite disappointments of this sort, it is clear from the results in Table 1 that distribution is quite important. We believe that some of the problems of distribution can be avoided by employing a slightly more sophisticated approach, though this is a topic for further study.

Forward Propagation Earlier, we mentioned that forward propagation eliminates partially-dead expressions. However, forward propagation can also result in code degradation if expressions are moved into loops where they will be invariant but PRE will be unable to hoist them. For an example, consider the (simplified) code below, where the left and right fragments show the same code before and after forward propagation:



In this case, the computation of $n \leftarrow j + k$ has been pushed into the loop, potentially shortening some paths through the program. However, since we expect the loop to execute many times, the code on the right is potentially much slower (of course, the actual tradeoff is undecidable, as it depends on the values of *j, k,* and m). Recalling from Section 2 that PRE will never lengthen a path through the code, we realize that PRE will not be able to hoist the evaluation of $j + k$ out of the loop without lengthing the path *around* the use of n.

4.3 Code Expansion

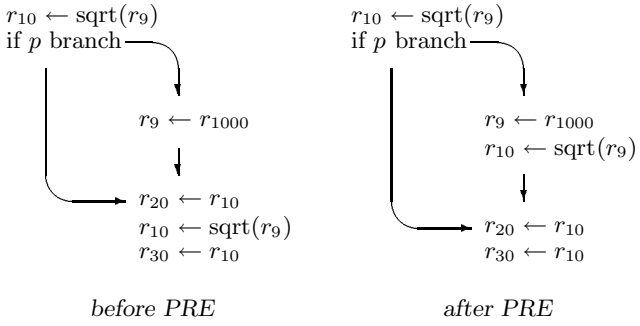
The speed and space requirements of our approach are primarily dependent on the amount of code expansion introduced by forward propagation. In the worst case, this expansion can be exponential in the size of the routine. To see how bad the expansion is likely to be in practice, we measured the the effect of forward propagation on the routines in our test suite. Table 2 shows the results of these tests. The entries in the *before* and *after* columns represent static counts of the number of ILOC operations in each routine. The column labeled *expansion* indicates the code growth factor due to forward propagation.

5 Discussion

In implementing these techniques, we encountered several issues that merit further attention.

5.1 Forward Propagation and Correctness

If an expression name is live across a basic block boundary, PRE will sometimes hoist an expression past a use of its name. Consider the example below:



before PRE *after PRE*

The problem is that the fragment on the left violates a requirement for correct behavior of PRE; namely, an expression defined in one basic block may not be referenced in another basic block.[2] Forward propagation satisfies this rule by moving the computation of $r_{10} \leftarrow \text{sqrt}(r_9)$ directly before its use, relying on the renaming introduced by SSA to preserve the correct version of r_9. We note that Chow also mentions using forward propagation [8]; we conjecture that it helped him avoid the same difficulty with PRE.

An alternative approach to ensuring that no expression name is live across a basic block boundary is to insert copies to newly created variable names and rewrite later references so that they refer to the variable name rather than the expression name. While it is possible that this approach could be used to avoid some of the negative effects of forward propagation, it may detract from the effectiveness of reassociation. This remains a topic for future research.

[2]We have never seen this requirement stated in the literature and believe it to be a source of confusion in the community.

routine	*before*	*after*	*expansion*
bilan	2,000	2,357	1.179
cardeb	916	1,024	1.118
coeray	280	397	1.418
colbur	659	1,155	1.753
dcoera	422	1,050	2.488
ddeflu	4,040	7,089	1.755
debflu	3,767	4,822	1.280
debico	2,728	2,984	1.094
decomp	941	1,144	1.216
deseco	11,545	13,537	1.173
drepvi	1,750	2,262	1.293
drigl	565	667	1.181
efill	1,257	1,996	1.588
fehl	552	581	1.053
fmin	372	661	1.777
fmtgen	588	696	1.184
fmtset	551	600	1.089
fpppp	20,147	27,358	1.358
gamgen	842	1,070	1.271
heat	925	1,817	1.964
hmoy	153	168	1.098
ihbtr	772	790	1.023
inideb	1,064	1,200	1.128
iniset	6,566	6,747	1.028
inithx	2,378	2,539	1.068
integr	812	967	1.191
orgpar	1,352	1,641	1.214
paroi	4,300	4,921	1.144
pastem	2,567	2,794	1.088
prophy	2,695	3,473	1.289
repvid	1,584	1,922	1.213
rkf45	164	228	1.390
rkfs	881	1,180	1.339
saturr	1,524	2,131	1.398
saxpy	95	102	1.074
seval	167	190	1.138
sgemm	677	976	1.442
sgemv	293	340	1.160
si	178	202	1.135
solve	319	375	1.176
spline	1,173	1,220	1.040
subb	1,199	1,199	1.000
supp	1,589	2,075	1.306
svd	2,563	3,984	1.554
tomcatv	2,645	3,610	1.365
twldrv	13,405	15,870	1.184
urand	189	212	1.122
x21y21	70	75	1.071
yeh	929	1,628	1.752
zeroin	276	400	1.449
totals	107,475	136,377	1.269

Table 2: Code Expansion from Forward Propagation

5.2 Interaction with Other Optimizations

Some optimizations interact poorly with our technique. For example, many compilers replace an integer multiply with one constant argument by a series of shifts, adds, and subtracts [4]. Since shifts are not associative, this optimization should not be performed until after global reassociation. For example, if $((x \times y) \times 2) \times z$ is prematurely converted into $((x \times y) \ll 1) \times z$, we lose the opportunity to group z with either x or y. This effect is measurable; indeed, we have accidentally measured it more than once.

We expect that strength reduction will improve the code beyond the results shown in this paper. Reassociation should let strength reduction introduce fewer distinct induction variables, particularly in code with complex subscripts like that produced by cache and register blocking [5, 27]. Of course, some particularly sophisticated approaches to strength reduction include a form of reassociation [20]; we believe that a separate pass of reassociation will significantly simplify the implementation of strength reduction. Additionally, implementing global reassociation as a separate pass provides benefits to other optimizations, even in loop-free code.

5.3 Common Subexpression Elimination

The experiments described in Section 4 show that PRE is a powerful component of an optimizing compiler. A natural question is: "How does it compare to other approaches?" To answer this, we will consider three different approaches. Assume for each that we have used the techniques described in Sections 3.1 and 3.2 to encode value equivalence into the name space.

1. Alpern, Wegman, and Zadeck suggest the following scheme: If a value x is computed at two points, p and q, and p dominates q, then the computation at q is redundant and may be deleted [2, page 2].

2. The classic approach to global common subexpression elimination is to calculate the set of expressions available at each point in a routine. If x is available on every path reaching p, then any computation of x at p is redundant and may be deleted.

3. Partial redundancy elimination, as described in Section 2.

These methods form a hierarchy. The first method removes only a subset of the redundancies in the code. For instance, it cannot remove the redundancy shown in the first example of Section 2 where $x + y$ occurs in each clause of an if-then-else and again in the block that follows. The second method, based on available expressions, will handle this case; it removes all redundancies. PRE is stronger yet – it removes all redundancies and many partial redundancies as well.

6 Related Work

While there have been many papers discussing partial redundancy elimination (*e.g.,* [21, 14, 12, 18]), none mention the deficiencies discussed in Sections 2.2 and 2.3. Rosen *et al.* recognize the naming problem and propose a complex alternative to PRE; however, they do not consider reordering complex expressions [23].

The idea of exploiting associativity and distributivity to rearrange expressions is well known [17, 1]; however, early work concentrated on simplifying individual expressions. We know of two prior approaches to reassociation with the goal of exposing loop-invariant expressions, both discovered within IBM and published the same year. Scarborough and Kolsky describe a front-end discipline for generating an array address expression as a sum of products and associating the sum to expose the loop-invariant parts [25]. Cocke and Markstein also mention the idea of reassociation, this time within the optimizer instead of the front end [9].

In a chapter for an upcoming book, Markstein *et al.* describe a sophisticated algorithm for strength reduction that includes a form of reassociation [20]. Their algorithm attacks the problem on a loop-by-loop basis, working from inner to outer loops. In each loop, they perform some forward propagation and sort subexpressions into loop-variant and loop-invariant parts, hoisting the invariant parts. We presume their approach is a development of earlier work within IBM. Other work by O'Brien *et al.* and Santhanam briefly describe what are apparently further developments of the Cocke and Markstein approach [22, 24].

It is difficult to compare our approach directly to these earlier methods. We were motivated by a desire to separate concerns. We already had solutions to hoisting loop invariants and strength reduction; therefore, we looked for a way to reassociate expressions. We also prefer our global approach to loop-by-loop alternatives since it can make improvements in loop-free code and may admit simpler implementation.

Recent work by Feigen *et al.* and by Knoop *et al.* describe alternative approaches to the problem of eliminating partially-dead expressions [15, 19]. While an adequate comparison of the alternatives would require trial implementations and empirical measurements, it is clear that they solve a similar class of problems in radically different ways. In our case, the elimination of some partially-dead expressions is an unexpected benefit of forward propagation.

7 Summary

In this paper, we show how to use global reassociation and global value numbering to reshape code in a way that improves the effectiveness and applicability of partial redundancy elimination. The effect of these transformations is to expose new opportunities for optimization. In particular, more expressions are shown to be redundant or loop-invariant; partial redundancy elimination optimizes these newly exposed cases. Additionally, some partially-dead expressions are eliminated.

We showed experimental results that demonstrate the effectiveness of partial redundancy elimination. The data also shows that applying our transformations before partial redundancy elimination can produce significant further improvements.

We introduced an algorithm for global reassociation. It efficiently reorders the operands of associative operations to expose loop-invariant expressions. Its simplicity should make it easy to add to an existing compiler.

Acknowledgements

We owe a debt of gratitude to our colleagues on the compiler project: Tim Harvey, Rob Shillingsburg, Taylor Simpson, Lisa Thomas, and Linda Torczon. Without their support and implementation efforts, this work would have been impossible. We also appreciate the thorough reviews provided by the program committee; they significantly improved both the form and content of this paper.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] 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, January 1988.

[3] Marc A. Auslander and Martin E. Hopkins. An overview of the PL.8 compiler. *SIGPLAN Notices*, 17(6):22–31, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*.

[4] Robert L. Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, July 1986.

[5] Steve Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In Jack Dongarra, Paul Messina, Danny C. Sorensen, and Robert G. Voight, editors, *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 400–405, 1990.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[7] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, Orlando, Florida, January 1991.

[8] Fred C. Chow. *A Portable Machine-Independent Global Optimizer – Design and Measurements*. PhD thesis, Stanford University, December 1983.

[9] John Cocke and Peter W. Markstein. Measurement of program improvement algorithms. In *Proceedings of Information Processing 80*. North Holland Publishing Company, 1980.

[10] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[12] Dhananjay M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

[13] Dhananjay M. Dhamdhere and Uday P. Khedker. Complexity of bidirectional data flow analysis. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Charleston, South Carolina, January 1993.

[14] Karl-Heinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[15] Lawrence Feigen, David Klappholz, Robert Casazza, and Xing Xue. The revival transformation. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 421–434, Portland, Oregon, January 1994.

[16] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[17] Dennis J. Frailey. Expression optimization using unary complement operators. *SIGPLAN Notices*, 5(7):67–85, July 1970. *Proceedings of a Symposium on Compiler Optimization*.

[18] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

[19] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *SIGPLAN Notices*, 29(6), June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[20] Peter W. Markstein, Victoria Markstein, and F. Kenneth Zadeck. Reassociation and strength reduction. In *Optimization in Compilers*. ACM Press, *to appear*.

[21] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[22] Kevin O'Brien, Bill Hay, Joanne Minish, Hartmann Schaffer, Bob Schloss, Arvin Shepherd, and Matthew Zaleski. Advanced compiler technology for the RISC System/6000 architecture. In *IBM RISC System/6000 Technology*. IBM Corporation, Armonk, New York, 1990.

[23] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, California, January 1988.

[24] Vatsa Santhanam. Register reassociation in PA-RISC compilers. *Hewlett-Packard Journal*, pages 33–38, June 1992.

[25] Randolph G. Scarborough and Harwood G. Kolsky. Improved optimization of FORTRAN object programs. *IBM Journal of Research and Development*, pages 660–676, November 1980.

[26] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[27] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.