Partial Redundancy Elimination Driven by a Cost-Benefit Analysis

R. Nigel Horspool and H. C. Ho
Department of Computer Science
University of Victoria, P.O. Box 3055
Victoria, BC, Canada V8W 3P6
E-mail: {nigelh, cshcho}@csc.uvic.ca

Abstract

Partial redundancy elimination has become a major compiler optimization that subsumes various ad hoc code motion optimizations. However, partial redundancy elimination is extremely conservative, failing to take advantage of many opportunities for optimization. We describe a new formulation of partial redundancy elimination based on a cost-benefit analysis of the flowgraph. Costs and benefits are measured by the number of evaluations of an expression. For that reason, our technique requires estimates for the execution frequency of every edge in the flowgraph. The new technique is much more aggressive, performing more code motion and thereby reducing the number of expression evaluations as compared to the standard optimization.

1. Introduction

Partial redundancy elimination (which we abbreviate to *PRE*) is a technique which first appeared in the dissertation of Étienne Morel and was published in a 1979 CACM article of Morel and Renvoise [13]. PRE sets up a system of dataflow equations for code motion transformations. Solutions to the equations show where new computations of expressions should be inserted and where existing computations should be deleted, thus achieving code motion (as well as subsuming a standard optimization for eliminating redundant expression computations).

A simple example of PRE appears in Figure 1. This example focuses solely on one expression, namely a+b. All extraneous detail has been suppressed. It does not matter for the purposes of PRE optimization what values are assigned to a or b; thus we show an assignment to a as simply

a = …

Similarly, it matter how the a+b expression is used; thus a statement that uses (i.e. needs the value of) a+b is shown as

… = a+b

The use of a+b in block 4 is said to be partially redundant. If control enters block 4 from block 3 it needs to be

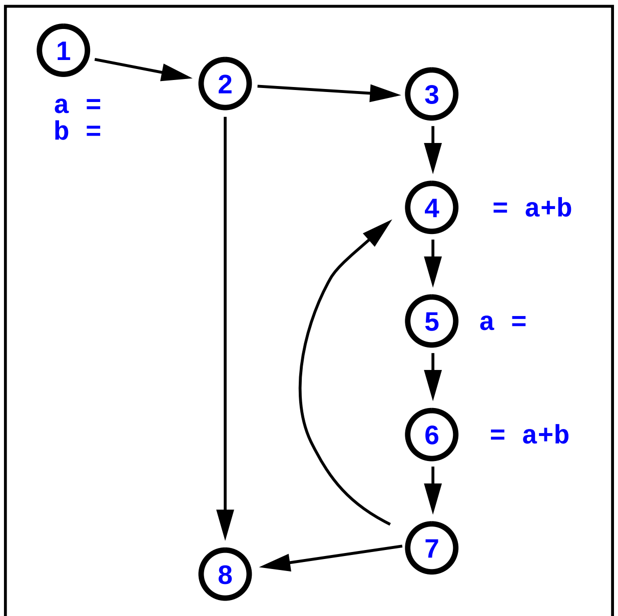

Figure 1. An example of partial redundancy

evaluated (i.e. it is *not* redundant), but if control enters from block 7 then the expression is redundant because the value of a+b computed in block 6 could have been saved in a temporary location and made available for re-use in block 3. Solving the Morel and Renvoise equations leads to these results:

Insert[3]	= { a+b }
Insert[i]	= ϕ, i 3
Redund[4]	= { a+b }
Redund[i]	= ϕ, i 4

The solutions for the *Insert* set instruct us to insert a new computation of a+b at the bottom of block 3; the solutions for the *Redund* set instruct us to delete the first (exposed) use of expression a+b in block 4 because that expression is or has become fully redundant. We can view the Insert and Redund sets as implementing code motion; they tell us, in effect, to move a computation from block 4 to block 3.

The Morel and Renvoise equations are, however, imperfect because they sometimes cause expressions to be moved unnecessarily or to be moved too far. Unnecessary code motion is undesirable because it increases the lifetimes of variables and thus increases register pressure. That, in turn, can have adverse effects on the quality of the generated code. A number of papers have addressed this issue and have proposed improvements to the equations to suppress unnecessary code motion [4], [7], [16]. Other papers have considered generalizing the technique by allowing expressions to be inserted on edges in the flowgraph [6], whereas the original Morel and Renvoise paper only gave equations for insertions at the ends of basic blocks. Still other papers have discussed efficient solution techniques for the equations [3], [5], [15]. (The Morel and Renvoise dataflow equations are somewhat unusual in that they imply bidirectional flow of information, and thus there does not appear to be a natural order in which the equations can be solved efficiently.)

More recently, Knoop, Rüthing and Steffen (KRS) have completely redeveloped the partial redundancy elimination technique from first principles [12]. Their solutions are optimal in the sense that expressions are always evaluated as late as possible, but that is of course subordinate to the primary goal of minimizing the number of evaluations of an expression when the program executes. The KRS approach completely eliminates all unnecessary code motion. Furthermore, the equations are expressed as a series of unidirectional dataflow equations and they therefore can be solved efficiently. Other works by the same authors have applied similar ideas to dead code elimination [11].

2. Classical Partial Redundancy Elimination is Too Conservative

It takes only one example to illustrate that a standard dataflow solution to PRE is, of necessity, conservative. Consider the flowgraph fragment shown in Figure 2. In this figure, the underlined numbers written alongside the control flow edges represent frequencies of execution. If the expression `a+b` were to be inserted into block 2 (or were to be inserted on the edges 1-2 or 2-4), then the occurrence of `a+b` in block 6 would become fully redundant and could be deleted. From a cost-benefit viewpoint, such a code transformation would be a clear win. For a cost of two extra evaluations of `a+b`, we would eliminate 9999 evaluations for a net gain of 9997 evaluations.

However, none of the existing PRE formulations performs such a code transformation. There are two reasons.

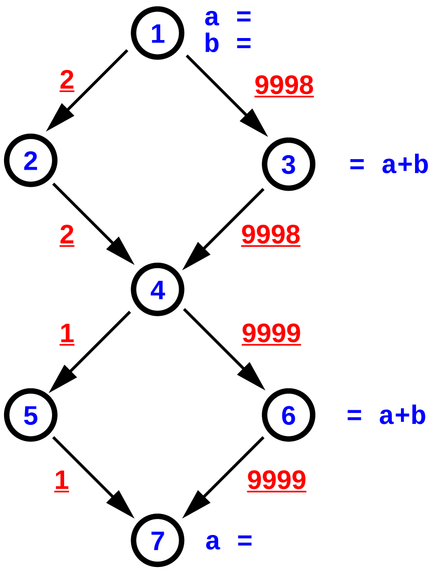

Figure 2. Example of a missed opportunity for optimization

The first reason is that current PRE methods are independent of execution frequencies. Without such information, there is a risk that an insertion of a new computation on an edge might increase the total number of evaluations of the expression. The second reason is that an optimizer must be careful not to introduce side-effects into a program that did not exhibit them before. If we were to insert the expression `a/b` in block 3 of the flowgraph of Figure 2 and if division by zero could cause an error interrupt on the target computer, then there is a risk that we have just caused the program to fail. (For example, block 1 might end with a test of variable `b` and might transfer control to block 2 when `b` is zero and to block 3 when `b` is non-zero.)

Current PRE methods insert an expression *e* at a point *p* only if all control flow paths emanating from *p* must evaluate *e* before any operands of *e* are redefined. The expression is said to be *anticipable* at point *p*. If *e* is anticipable then it does not matter if evaluating *e* can cause an error interrupt – at worst, the optimizer would have made an error occur a little earlier than would otherwise have been the case. The error would have been inevitable. Standard PRE methods also guarantee that the optimized program cannot evaluate *e* more times than in the original program.

If, however, we are sure that an expression cannot cause an error interrupt or some other side-effect and if we have execution frequency information available, we can perform code motion transformations that would be missed by standard PRE techniques.

3. A Cost-Benefit Formulation of Partial Redundancy Elimination

We consider insertions and deletions of expressions which are inherently safe – i.e., expressions which cannot cause an error interrupt or have any side-effect. On many modern computer architectures, most expressions appear to fall into this category. Comparisons and logical operations are almost always free of side-effects. Unless overflow checking is specifically enabled on hardware that supports such checking, most integer operations are inherently safe. Floating-point operations on computer systems that follow the IEEE standard are similarly safe unless interrupts are specifically enabled. (Using the IEEE format, floating point calculations with invalid operands produce results of infinity or NaN – *not-a-number.*)

We also assume that we have complete information regarding the execution frequency of every edge in the flowgraph. This information could be obtained through a prior run of the program on representative test data with profiling enabled or it could be obtained via static estimation techniques [17].

For our measure of cost, we will use the number of evaluations of an expression. Our goal will be to minimize the number of evaluations in the program execution(s) that supplied the frequency information. It is reasonable to assume that any progress towards accomplishing this goal will improve the program's efficiency in subsequent runs.

Our analysis has two phases. The first phase determines the lowest cost of making an expression fully redundant at various program points throughout the flowgraph and records how to achieve that lowest cost. The second phase is complementary. It checks each basic block containing a computation of the expression of interest and determines what net benefit would be achieved, in terms of the number of evaluations, if that expression were to be eliminated. The results of the second phase are two sets: a set of blocks containing expressions that should be deleted and a set of edges where new computations of expressions should be inserted.

3.1. Cost Analysis

To simplify the analysis, we consider occurrences of one expression *e* in a flowgraph. We assume that *e* is a safe expression.

We not only wish to determine the lowest cost (measured as number of evaluations) of making *e* fully redundant at a program point *p*, but we would also like to know how that lowest cost can be achieved. Making *e* fully redundant may require performing insertions of *e*, either at point *p* itself or in the paths that lead to *p*. We will analyze edge insertions only – insertions of *e* along flowgraph edges because this is more general than the alternative of permitting insertions only in basic blocks. (It would, however, be easy to formulate a block-insertions version of our analysis if that is desired.)

We use sets of flowgraph edges to represent costs. A set `{ (1,2), (4,5) }` would represent the possibility of inserting new computations of *e* on edges 1-2 and 4-5 in the flowgraph; the numeric cost of that set measured as the number of expression evaluations would be the sum of the execution frequencies of edges 1-2 and 4-5.

The givens (i.e. the initial data) on which our analysis will be based are as follows:

$\mathsf{Freq}_{i\,j}$	=	the execution frequency of edge *i-j*.
TRANSP_i	=	true iff block *i* is transparent to *e* – i.e. block *i* does not contain any assignments to operands of *e*.
AVLOC_i	=	true iff expression *e* is locally available on exit from block *i* – i.e. block *i* contains a computation of *e* that is not subsequently killed by an assignment to an operand of *e*.

The function *Cost* maps a cost set to its numeric cost, measured as the number of evaluations of *e* that must be added to the flowgraph:

$$\mathsf{Cost}(S) = \sum_{(i,\,j)\,\in\,S} \mathsf{Freq}_{ij}$$

The algebra for our dataflow analyses requires a cost set that corresponds to $\perp$ – a set that represents a complete lack of information about a solution. We will use the following set for this purpose:

$$\perp \;=\; \{\, \texttt{<i,j>} \mid \texttt{<i,j>} \text{ is a flowgraph edge } \}$$

It represents an upper bound on all cost estimates. We informally use the notation ∞ to represent $\mathsf{Cost}(\perp)$.

We wish to determine the following cost sets associated with nodes (basic blocks) in the flowgraph:

CIN_j	=	the cost set on entry to node *j*
COUT_j	=	the cost set on exit from node *j*

A solution for a set CIN_j can be interpreted as meaning that the minimum cost of making expression *e* fully available on entry to block *j* is $\mathsf{Cost}(\mathsf{CIN}_j)$ and that lowest cost can be realized by making insertions of *e* on every edge in the set CIN_j. Similarly for the COUT_j set. Note that if block *j* contains a computation of *e* that is available on exit from *j* (i.e. AVLOC_j is true) then COUT_j will be an empty set. This is because no insertions of *e* are needed to make *e* fully available at this program point – and the incremental cost of making it fully available is zero.

The CIN and COUT sets are related by a system of dataflow equations as follows:

1.

$$\mathrm{COUT}_j = \begin{cases} \mathrm{CIN}_j & \text{if } \mathrm{TRANSP}_j \bullet \overline{\mathrm{AVLOC}_j} \\ \varnothing & \text{if } \mathrm{AVLOC}_j \\ \perp & \text{otherwise} \end{cases}$$

The first part of the equation says that if block *j* is transparent to *e* and does not contain a locally available computation of *e*, then we can make *e* available on exit from *j* by making it available on all incoming edges to *j*. However, if block *j* contains a computation of *e* that reaches the end of *j*, then there is no additional cost of making *j* available – and that explains the second line. Finally, the third line means, in effect, that no insertions anywhere could make *e* available at this point. (This is a consequence of only permitting insertions on edges.)

2.

$$\mathrm{CIN}_j = \begin{cases} \perp & \text{if } j \text{ is the entry node} \\ \bigcup\limits_{i \in \mathrm{pred}(j)} C_{ij} & \text{otherwise} \end{cases}$$

where

$$C_{ij} = \begin{cases} \{(i,j)\} & \text{if } \mathrm{Freq}_{ij} \le \mathrm{Cost}(\mathrm{COUT}_i) \\ \mathrm{COUT}_i & \text{otherwise} \end{cases}$$

The main part of the definition for $\mathrm{CIN_j}$ says that *e* can be made fully available on entry to block *j* if we pay the price of making it available on each incoming edge to *j*. The lowest cost of making it available on an edge (*i*,*j*) is either the least cost of making *e* available at exit from *i* (i.e. the solution to $\mathrm{COUT_i}$) or it is the cost of inserting *e* on the edge (*i*,*j*) – whichever has the lower cost. The use of a *less-or-equal* comparison, as opposed to a *less-than* comparison, in the definition of $\mathrm{C_{i\,j}}$ is quite deliberate. It encourages insertions to occur at the latest possible point and will therefore avoid unnecessary code motion.

An iterative approach to solving the system of dataflow equations is guaranteed to converge to a fixpoint. If we start by initializing all the CIN and COUT sets to empty, except for the CIN set of the entry node which should be initialized to ⊥, then the total cost of the sets (as measured by the Cost function) can only grow monotonically while the iteration proceeds. The sets themselves can both increase and decrease in cardinality, but the computed cost for a set can never decrease. Since there is an upper bound for each cost set and because the number of possibilities for an increment in cost is finite, convergence in a finite number of steps is assured.

We give an example of cost analysis for the flowgraph of Figure 3. The initial givens are:

$\mathrm{TRANSP_i}$	=	true for *i* = 2, 4, 5, 6, 7, 8 and false for all other blocks.
$\mathrm{AVLOC_i}$	=	true for *i* = 2, 4, 7, 8 and false for all other blocks.
$\mathrm{Freq_i}$	=	the underlined numbers shown alongside the edges in Figure 3.

We proceed by initializing all the CIN and COUT sets to ϕ, except for $\mathrm{CIN_1}$ which is initialized to ⊥. Iteratively applying the dataflow equations rapidly converges to the solution shown in Table 1.

As a foretaste of the benefits analysis to be given in the next section, we can take the Table 1 solutions and use them to immediately discover some computations of a+b that can be profitably moved. For example, the a+b in block 7 is evaluated 101,000 times. That cost exceeds the $\mathrm{CIN_7}$ cost we have just determined; therefore we should insert a+b on every edge in $\mathrm{CIN_7}$, namely on edge (3,5) and eliminate it from block 7 for a net saving of 99,000

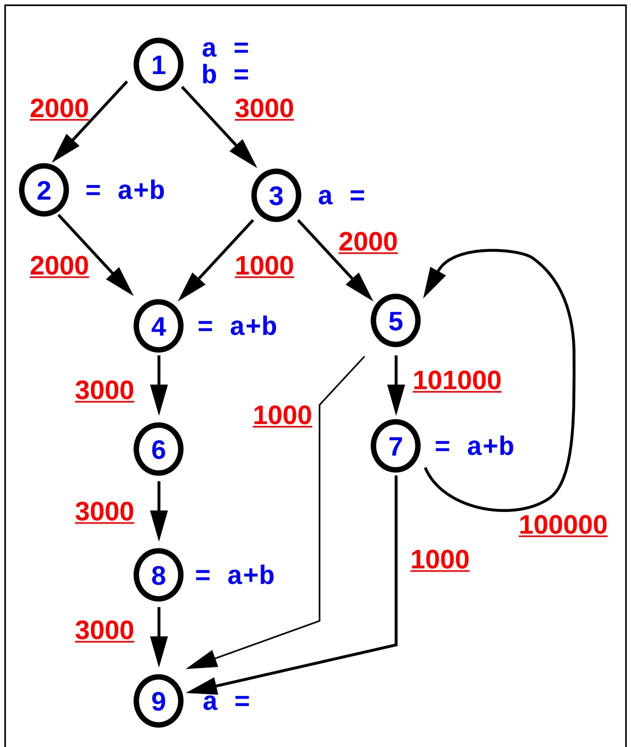

Figure 3. A flowgraph with a cycle

Table 1. Cost analysis results for Figure 3

block i	CIN_i	Cost (CIN_i)	$COUT_i$	Cost ($COUT_i$)
1	$\perp$	∞	$\perp$	∞
2	{ (1,2) }	2000	ϕ	0
3	{ (1,3) }	3000	$\perp$	∞
4	{ (3,4) }	1000	ϕ	0
5	{ (3,5) }	2000	{ (3,5) }	2000
6	ϕ	0	ϕ	0
7	{ (3,5) }	2000	ϕ	0
8	ϕ	0	ϕ	0
9	{ (5,9) }	1000	$\perp$	∞

evaluations of a+b. Note that the presence of the (5,9) edge in the flowgraph prevents current PRE techniques from performing the proposed code motion, because this code motion would insert a computation of a+b on the path 1-3-5-9 where there had been no computation before.

3.2. Simple Benefit Analysis

As we shall explain later, a full version of benefit analysis is combinatorial in nature. We believe that the full algorithm must have exponential running time. However, a simple analysis can be applied immediately and it finds some obvious places where a speed-up can be achieved.

If a block *b* contains an upwards exposed computation of expression *e* (i.e a computation of *e* that is not preceded by assignments to any operands of *e*), then *e* is a candidate for elimination. The benefit to be derived from eliminating *e* from *b* would be $Freq_b$, where $Freq_b$ denotes the frequency of execution of block *b*. The set CIN_b represents the cost of making *e* fully redundant at entry to *b*. If we compute the quantity

$NetBenefit = Freq_b - Cost(CIN_b)$

and if that quantity is greater than zero, we would speed up the program by inserting *e* on all the edges in the set CIN_b and deleting *e* from *b*.

Note that there may be another block *b´* where the net benefit would be negative but $CIN_{b'}$ contains some edges in common with CIN_b. If we have already decided to insert *e* on the edges in CIN_b then the *additional* cost of making *e* fully redundant at entry to *b´* may be less than $Freq_{b'}$. Therefore, we should apply an algorithm like the one below to find as many blocks as possible where computations of *e* can be profitably eliminated.

```
Candidates := { b | block b contains an
                        upwards exposed use of e }
Insert := $\phi$
Redund := $\phi$
while $\exists$ b (b $\in$ Candidates):
        $Freq_b$ - Cost($CIN_b$ - Insert) > 0 do
begin
    Insert := Insert $\cup$ $CIN_b$
    Redund := Redund $\cup$ {b}
    Candidates := Candidates - {b}
end
```

We claim that our benefits algorithm will find the major code motion opportunities in the flowgraph. However it is deficient in two respects. Firstly, there could be a subset of the candidate blocks which yields a net benefit if processed together, but where no individual block yields a net benefit if processed separately. Secondly, if two candidate blocks b1 and b2 contain computations of *e* that should be eliminated, there may be more profitable places to insert new occurrences of *e* than on the edges in the combined set $CIN_{b1} \cup CIN_{b2}$.

The first of these two deficiencies can be solved by transforming the dependencies between candidate blocks and insertion edges into a network flow problem. (This transformation appears as problem 27-3 in [2].) Then the problem of finding the optimal Redund set can be solved using a min-cut algorithm.The best min-cut algorithms run in time $O(m\, n \log(n^2/m))$ where *m* is the number of edges and *n* is the number of vertices in the network [2] [8] [10]. Relating this back to our PRE problem, *n* would be bounded by the sum of the number of edges in the flowgraph and the number of candidate blocks in the flowgraph, while *m* is bounded by their product. However, we believe that the second deficiency renders any approach that solves just the first deficiency almost useless.

4. More Examples

4.1. A Larger Example

First, we demonstrate the code motions generated for the larger example flowgraph shown in Figure 4.

The results of the cost set analysis for this flowgraph appear in Table 2. Using the cost sets shown in that table, our benefit analysis algorithm would determine:

Insert	= { (1,2), (3,6) }
Redund	= { 7, 8, 11, 14 }

Making these insertions and deletions would, if the program were to be rerun with the same input data, reduce the total number of evaluations of a+b from 5400 to 1407.

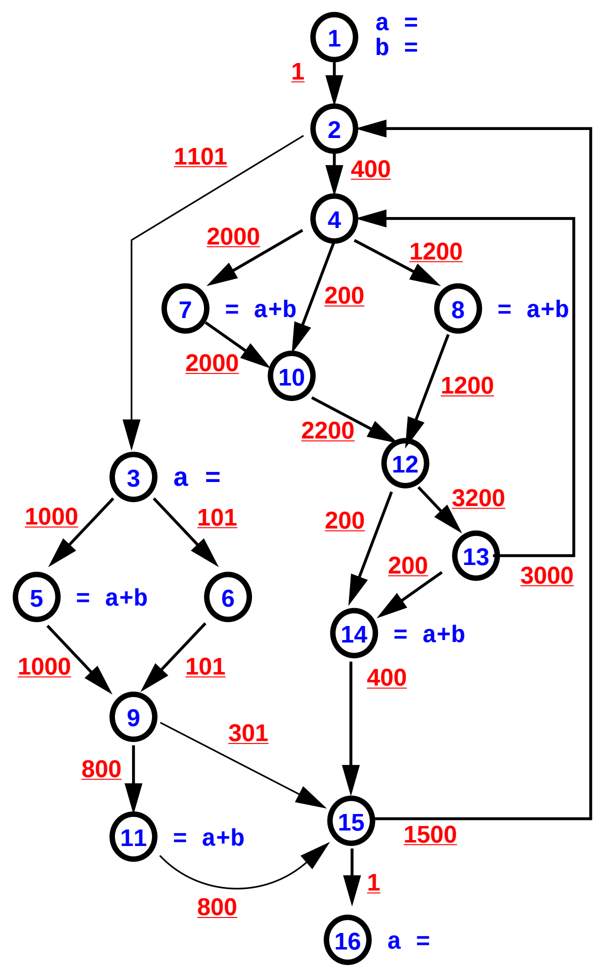

Figure 4. A larger example flowgraph

Table 2. Cost analysis for flowgraph of Figure 4

block i	CIN_i	Cost
1	$\perp$	∞
2, 3, 4	{ (6,9), (1,2) }	102
5	{ (3,5) }	1000
6	{ (3,6) }	101
7, 8	{ (6,9), (1,2) }	102
9	{ (6,9) }	101
10	{ (6,9), (1,2) }	102
11	{ (6,9) }	101
12, 13, 14	{ (6,9), (1,2) }	102
15	{ (6,9) }	101
16	{ (15,16) }	1

4.2. An Example Requiring Two Passes

Next, we illustrate why further development of out cost-benefit analysis technique might be needed. Figure 5 shows a relatively small flowgraph where the code motion that the algorithm prescribes is not what a simple visual inspection would have produced.

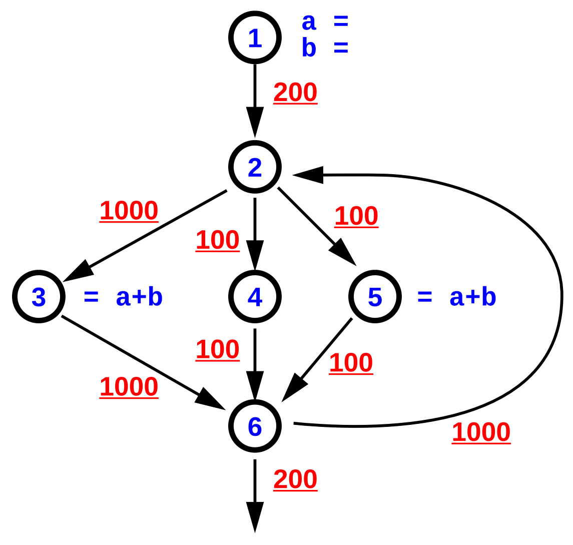

Figure 5. An example illustrating a need for repeated analyses

Our analysis algorithm produces the following table of costs.

block i	CIN_i	$Cost_i$
1	$\perp$	∞
2, 3	{(1,2), (4,6)}	300
4	{(2,4)}	100
5	{(2,5)}	100
6	{(4,6)}	100

The insertions and deletions generated by the analysis are as follows:

Insert	= { (1,2), (4,6) }
Redund	= { 3 }

The obvious improvement to the program is to insert a computation on edge (1,2) only. However, if we perform the prescribed code motion, we obtain the flowgraph shown in Figure 6. Note that we have shown the insertion on edge (1,2) as an insertion at the bottom of block 1 and the insertion on edge (4,6) as an insertion at the bottom of block 4. Re-application of our cost-benefit analysis algorithm will eliminate the fully redundant computations in blocks 4 and 5, yielding the optimal solution.

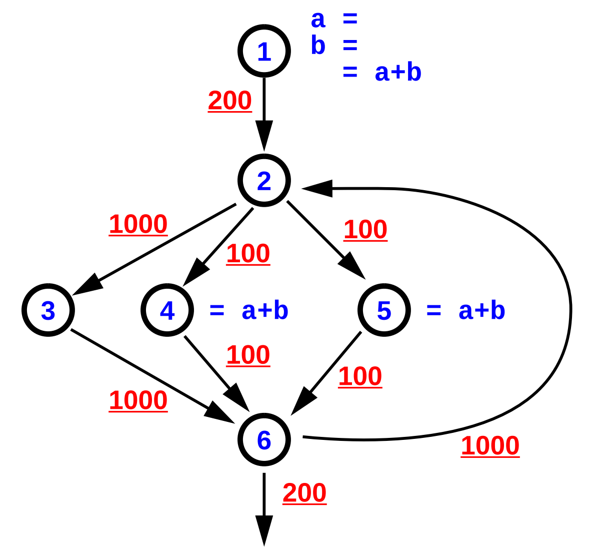

Figure 6. After applying code motion to flowgraph of Figure 5

5. Discussion and Future Directions

We are not the first to have proposed treating safe expressions differently for code motion optimizations. An interval-based approach using that idea was published as long ago as 1972 by Ken Kennedy [9]. Nor are we the first to have proposed using program profile information to assist the code optimizer. Probably the most significant of the previous work in this area concerns the use of profile information to determine which function calls can be profitably inlined [14]. The closest previously published work to our own is an examination by Chang et al. of how profiles can be used in 'classical' code optimizations [1]. Indeed, Chang et al. gave one example where a partial redundancy elimination was performed. However, they did not formalize the technique or give a systematic algorithm for finding the opportunities for code motion.

We believe that our approach is the first formalization of code motion optimization as a cost-benefit analysis based on hard numbers. It is a promising technique that warrants further investigation. It finds opportunities for code motion that are inappropriate for 'classical' PRE methods, producing significant improvements to flowgraphs that contain heavily executed loops. Our approach favours late insertions of code over early insertions, avoiding unnecessary code motion – a problem that flawed the original Morel and Renvoise approach.

Some interesting work lies ahead. In addition to building a test implementation in a compiler and gathering experimental statistics, we also plan to pursue the following directions:

- Finding a more efficient implementation of the cost analysis algorithm.
- Analyzing the time complexity of the approach.
- Finding a better heuristic algorithm for selecting which computations should be eliminated after cost analysis has been performed.
- Handling expressions which are safe in some regions of the program and unsafe in other regions. (For example, an array reference expression `a[i]` is safe in those parts of the program where range analysis can determine that the index `i` must be within the array bounds.)
- Extending the technique to unsafe expressions.
- Constructing an interprocedural version of the algorithm.
- Applying a similar approach to partial dead code elimination [11].

6. Acknowledgments

We gratefully acknowledge financial support for this work from the Natural Sciences and Engineering Research Council of Canada. We thank Valerie King for recognizing that our benefits analysis problem was related to problem 27-3 in a textbook on algorithms [2] and that a simplified version of the problem is at least NP-complete.

7. References

[1] P.P. Chang, S.A. Mahlke, W.Y. Chen, and W.W. Hwu. "Using Profile Information to Assist Classic Code Optimizations", *Software – Practice & Experience* 21, 12, Chichester, Dec. 1991, pp. 1301-1321.

[2] Cormen, T.H., C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[3] D.M. Dhamdere. "A Fast Algorithm for Code Movement Optimization", *ACM SIGPLAN Notices* 9, 3, New York, Aug. 1988, pp. 243-273.

[4] D. M. Dhamdere. "Practical Adaptation of the Global Optimization of Morel and Renvoise", *ACM Trans. on Prog. Lang. and Systems* 13, 2, New York, April 1991, pp. 291-294

[5] D.M. Dhamdere, B.K. Rosen, and F.K. Zadeck. "How to Analyze Large Programs Efficiently and Informatively", Proc. of PLDI'92, *ACM SIGPLAN Notices* 27, 7, New York, July 1992, pp. 212-223.

[6] D.M. Dhamdere and H. Patil. “An Elimination Algorithm for Bidirectional Data Flow Problems Using Edge Placement”, *ACM Trans. on Prog. Lang. and Systems* 15, 2, New York, April 1993, pp. 312-336.

[7] K.-H. Drechsler and M.P. Stadel. “A Solution to a Problem with Morel and Renvoise’s ‘Global Optimization by Suppression of Partial Redundancies’”, *ACM Trans. on Prog. Lang. and Systems* 10, 4, New York, Oct. 1988, pp. 635-640.

[8] A.J. Goldberg and R.E. Tarjan. “A New Approach to the Max Flow Problem”, *Journal of the ACM* 35, New York, 1988, pp. 921-940.

[9] K. Kennedy. “Safety of Code Motion”, *Intl. J. of Computer Mathematics* 3, 1972, pp. 117-130.

[10] V. King, S. Rao, and R. E. Tarjan. “A Faster Deterministic Maximum Flow Algorithm”, *Journal of Algorithms* 17, 3, 1994, pp. 447-474.

[11] J. Knoop, O. Rüthing, and B. Steffen. “Partial Dead Code Elimination”, Proc. of PLDI’94, *ACM Sigplan Notice*s 29, 6, New York, June 1994, pp. 147-158.

[12] J. Knoop, O. Rüthing, and B. Steffen. “Optimal Code Motion: Theory and Practice”, *ACM Trans. on Prog. Lang. and Systems* 16, 4, New York, July 1994, pp. 1117-1155.

[13] E. Morel and C. Renvoise. “Global Optimization by Suppression of Partial Redundancies”, *Communications of the ACM* 22, 2, New York, Feb. 1979, pp. 96-103.

[14] K. Pettis and R.C. Hansen. “Profile-Guided Code Positioning”, Proc. of PLDI’92, *ACM SIGPLAN Notices* 27, 7, New York, July 1992, pp. 16-27.

[15] F.W. Schröer. “Districts: A Foundation for the Suppression of Partial Redundancies”, GMD Working Paper 304, Sankt Augustin, Germany, Aug. 1988.

[16] A. Sorkin. “Some Comments on ‘A Solution to a problem with Morel and Renvoise’s ‘Global Optimization by Suppression of Partial Redundancies’’”, *ACM Trans. on Prog. Lang. and Systems* 11, 4, New York, Oct. 1989, pp. 666-668.

[17] T.A. Wagner, V. Maverick, S.L. Graham, and M.A. Harrison. “Accurate Static Estimators for Program Optimization”, Proc. of PLDI’94, ACM SIGPLAN Notices 29, 6, New York, June 1994, pp. 85-96.