Partial Redundancy Elimination with Predication Techniques

Bernhard Scholz[1], Eduard Mehofer[2], and Nigel Horspool[3]

[1] Institute of Computer Languages, Vienna University of Technology
Argentinierstr. 8/4, A-1040 Vienna, Austria
scholz@complang.tuwien.ac.at

[2] Institute for Software Science, University of Vienna
Liechtensteinstr. 22, A-1090 Vienna, Austria
mehofer@par.univie.ac.at

[3] Department of Computer Science, University of Victoria
P.O. Box 3055, Victoria, BC, Canada V8W 3P6
nigelh@uvic.ca

Abstract. Partial redundancy elimination (PRE) techniques play an important role in optimizing compilers. Many optimizations, such as elimination of redundant expressions, communication optimizations, and load-reuse optimizations, employ PRE as an underlying technique for improving the efficiency of a program.

Classical approaches are conservative and fail to exploit many opportunities for optimization. Therefore, new PRE approaches have been developed that greatly increase the number of eliminated redundancies. However, they either cause the code to explode in size or they cannot handle statements with side-effects.

First, we describe a basic transformation for PRE that employs predication to achieve a complete removal of partial redundancies. The control flow is not restructured, however, predication might cause performance overhead. Second, a cost-analysis based on probabilistic data-flow analysis decides whether a PRE transformation is profitable and should be applied. In contrast to other approaches our transformation is strictly semantics preserving.

1 Introduction

Partial redundancy elimination (PRE) is an important optimization technique in compilers for improving the efficiency of programs. The objective of PRE is to avoid unnecessary re-computations of values at runtime. The PRE transformation replaces the computations by accesses to temporary variables, where the temporary variables are initialized at suitable program points.

PRE is a key technology for compilers. In the literature, several optimizations can be found which employ PRE as the underlying technique. For example, PRE has been successfully applied in compilers for high-performance systems. Communication optimizations [KBC$^+$99] and dynamic redistributions [KM02] employ PRE as underlying optimization technique. PRE is also used for optimizations in RISC compilers. An important example of a RISC optimization is the load-reuse analysis [BGS99] that also uses PRE as underlying technique.

However, it has been observed that classical approaches [KRS94] are too cautious and fail to take advantage of many opportunities for optimization. To improve the effectiveness of PRE, new approaches were developed that use speculation [HH97,GBF98] and code duplication [Ste96]. Speculation uses profile information and inserts additional computations (i.e. speculative computations) in the program. In constrast, code duplication copies code and identifies information carrying paths. However, both techniques raise concerns: (1) speculation is not always applicable due to possible side-effects in expressions, and (2) code duplication may cause an explosion in size.

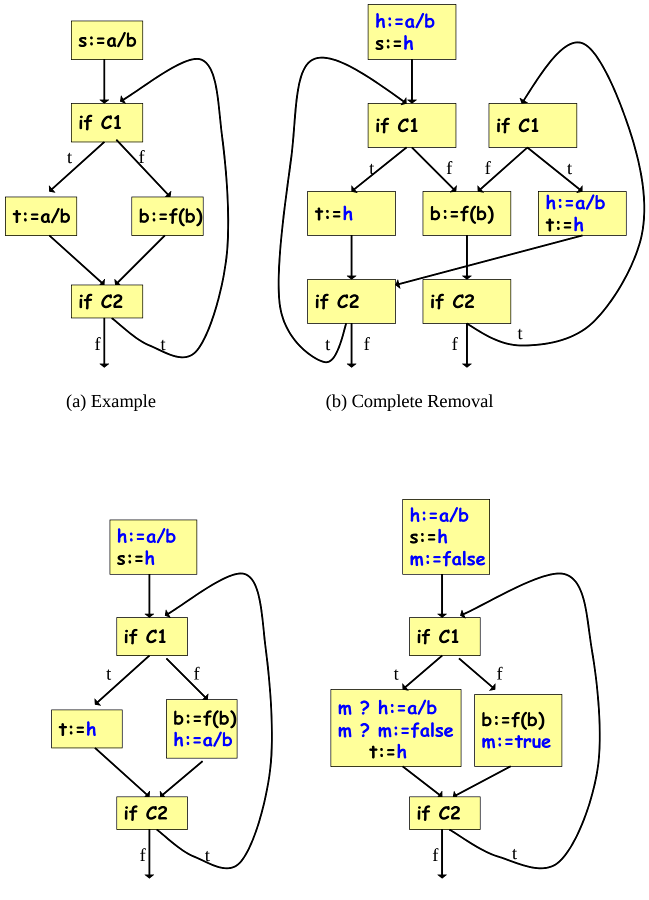
In this paper we introduce a novel transformation for PRE that achieves a complete removal of all partially redundant expressions without duplicating code. Our PRE technique uses predication. A predicate controls whether the computation in the temporary variable is valid or not. If the computation is destroyed, the predicate is updated. Computations are conditionally executed. If the temporary variable does not contain a valid value, re-computation is performed and the predicate is updated again. The transformation is guided by a cost-analysis based on probabilistic data-flow analysis [MS01,SM02] that decides whether a computation needs to be predicated. The features of our PRE approach are that (1) the control flow graph is not re-structured, (2) all partial redundancies can be removed, and (3) our transformation is semantics preserving.

The paper is organized as follows. In Section 2 we motivate PRE with predication. In Section 3 the optimization is shown in detail. The basic transformation is given and a cost analysis determines whether the transformation is beneficial to the program performance or not. Section 4 surveys related work. Finally, a summary is given in Section 5.

2 Motivation

Consider the example in Figure 1. In the first control flow graph in Figure 1(a), the computation on the left-hand branch of the inner-loop is partially redundant. The expression `a/b` is evaluated twice if the left-branch is executed in the first iteration of the loop and if the left-branch is subsequently executed. A traditional PRE algorithm [KRS94] is not able to eliminate this partial redundancy because the statement `b:=f(b)` on the right-branch destroys the computation of the expression.

Now consider the second graph in Figure 1(b), which shows a complete removal of partial redundancies of the example in Figure 1(a) by employing code duplication as introduced in [Ste96]. The transformation nearly duplicates all nodes in the control flow graph. After applying the transformation two versions of a node exist in the graph. One node represents the state where the computation is not available, while the other one represents the node where the computa-



(a) Example

(b) Complete Removal

(c) Speculative PRE

(d) PRE with Predication

Fig. 1. Motivating Example

tion is available. Whenever the computation is destroyed, control is transferred from a node where the computation is assumed to be available to a node where it is not. Conversely, evaluation of the expression causes a control transfer to a node where it is assumed to be available.

In the first node of the example in Figure 1(b), the expression `a/b` is computed and stored in `h`. The loop on the left-hand side does not destroy the computation of `a/b` and therefore can re-use the value of `a/b` inside the loop. Whenever the computation is destroyed by `b:=f(b)`, the loop on the right-hand side is entered. The program stays in the loop on the right-hand side until `a/b` is re-computed. Then, the computation is available in `h` again, and the loop on the left-hand side is re-entered.

Not only does the control flow graph become irreducible[4], but the number of nodes has nearly doubled. In general, the code growth is exponential in the

[4] For some optimizations irreducible control flow graphs have a negative impact.

number of expressions and, therefore, the approach is not viable in practice. To alleviate this problem, Bodik et al. [BGS98] have introduced an approach that limits code growth under the guidance of profile information. However, much code would still be duplicated.

In Figure 1(c) an approach is illustrated that uses speculation [HH97,GBF98]. Let us assume that the left-branch is executed more often than the right one. Expression `a/b` inside the loop can only be hoisted to the first node if the expression is re-computed after destroying the computation in the right-branch. However, the expression `a/b` is not a safe computation since it can raise a "division by zero" exception. For example, along a subsequent execution of the right branch the computation of the expression `a/b` is inserted in the program which does not exist in the original program. If, on some execution of the program `b` happens to be zero, the error would occur. Hence, the insertion on the right branch might destroy program semantics.

The intuition behind our predication approach is straightforward, as demonstrated in Figure 1(d). Instead of duplicating code, our approach achieves complete removal of partial redundancies by introducing a predicate. The predicate reflects the unavailability of the expression in `h`. If the predicate holds, the expression `a/b` is not available in `h` and must be re-computed, otherwise the computation in `h` can be re-used. In comparison to speculative PRE, our approach is safe. No additional computations (other than predicate tests) are inserted in the program.

For re-computing the value of `a/b`, we use predication as shown in the example of Figure 1(d). Predication is a conditional execution of a statement depending on the logical value of a predicate. For example, the assignment in `m ? h:=a/b` on the left-branch is only executed if predicate `m` holds. After recomputing the expression, the predicate is set to `false` with `m ? m:=false;`. When a statement destroys the value of expression `a/b`, the predicate needs to be invalidated as shown in the node on the right branch inside the loop.

The transformation is not free of cost and neither are all computations of the expression replaced by predicated evaluations. E.g., in the first node we do not transform the computation of `a/b` to a predicated one because there is no preceding computation which can be exploited. In addition, the predicated computation inside the loop will only make sense if the left branch is executed more frequently than the right branch. Therefore, the transformation needs to be guided by a cost analysis based on profile information that decides whether a predicated computation would improve program performance or not.

3 PRE With Predication

In this section, we develop our predicated PRE approach. The optimization consists of an analysis part which identifies profitable predication opportunities followed by a subsequent transformation step. We start with a description of the transformation first. Subsequently we develop a cost model and describe how the transformation is guided by it. Finally, we present the analysis required for the cost model.

3.1 Basic Transformation

The key idea of predicated PRE is to save the value of an expression in a temporary and to maintain a predicate which indicates whether the saved value is still valid or not. Whenever the saved value is valid, subsequent computations of the expression can be eliminated by loading the saved value.

The transformation is shown in Fig. 2. Basically every assignment $u := exp$ has to be replaced by the sequence $h_{exp} := exp; u := h_{exp}$ where h_{exp} denotes a temporary which is associated with term *exp* and is used to hold the value of the last computation of *exp*. Additionally, we have predicate m_{exp} which is associated with the term *exp* as well and which indicates whether temporary h_{exp} can be reused or the term *exp* has to be recomputed. The important point is that $h_{exp} := exp$ is not executed each time. The term *exp* is evaluated only if necessary, i.e. it is evaluated if the predicate m_{exp} is true as indicated by the predicated assignment $m_{exp}?h_{exp} := exp$, as shown in case 1 of Fig. 2. Next m_{exp} is set to false to suppress unnecessary recomputations. However, if a variable occurring in *exp* is modified, the value held in h_{exp} cannot be reused any more, and the term *exp* would need to be recomputed. This situation is described in case 2 of Fig. 2. There it is assumed that variable v occurs in term *exp*. Thus assigning a new value to v requires a recomputation of term *exp*, and this is enforced by setting the predicate m_{exp} to true.

Case 1: **Compute** Case 2: **Destroy**

$$\mathrm{u} := \mathit{exp}; \quad \Rightarrow \begin{cases} \mathrm{m}_{exp} \;?\; \mathrm{h}_{exp} := \mathit{exp}; \\ \mathrm{m}_{exp} \;?\; \mathrm{m}_{exp} := \mathbf{false}; \\ \mathrm{u} := \mathrm{h}_{exp}; \end{cases} \qquad \mathrm{v} := \ldots \quad \Rightarrow \begin{cases} \mathrm{v} := \ldots \\ \mathrm{m}_{exp} := \mathbf{true}; \end{cases}$$

Fig. 2. Basic Transformation

It is important to stress that contrary to other PRE approaches our transformation is strictly semantic preserving, i.e. we do not introduce new computations on any path which were not present in the original code. If an evaluation of an expression raises an exception, that exception would be raised at exactly the same point in the computation. However we add additional assignments which have to be executed and may degrade a program's performance. Hence, an analysis of the effect on performance is inevitable.

3.2 Cost Model

For each appearance of a term *exp* in the program we must decide whether it is profitable to perform the transformation or not. A transformation for term *exp* at some program point pays off if

$$OrigComp > PredicatedComp \tag{1}$$

where *OrigComp* denotes the computational costs of *exp* in the original code and *PredicatedComp* denotes the computational costs of *exp* in the transformed code. Obviously, *PredicatedComp* is dominated by the number of times the value stored in the temporary variable can be reused compared to the number of times the term has to be recomputed. If p denotes the probability that the stored value of *exp* is valid at some program point, then *PredicatedComp* is given by

$$\mathit{PredicatedComp} \;=\; p \times \mathit{Reuse} \;+\; (1-p) \times \mathit{Recompute} \tag{2}$$

with *Reuse* denoting the costs if m_{exp} is set to false and the stored value can be reused, and with *Recompute* denoting the costs associated with a recomputation. By combining Equation (1) and (2) we obtain

$$p > \frac{\mathit{Recompute} - \mathit{OrigComp}}{\mathit{Recompute} - \mathit{Reuse}} \tag{3}$$

Equation (3) specifies a lower bound for probability p. The execution times of *Recompute*, *OrigComp*, and *Reuse* can be measured or predicted and the value of p determined. Whenever $p_n > p$ holds at some program point n, it is profitable to apply the transformation. Otherwise, if $p_n \leq p$, the performance of the program may be degraded.

Let us consider again our motivating example with a program run π which enters the loop and 2 times takes the left branch, 1 time the right branch, and finally 6 times the left branch and terminating the loop. Let us further assume that the execution times for term a/b are given as follows: $\mathit{Recompute} = 110ns$, $\mathit{OrigComp} = 100ns$, and $\mathit{Reuse} = 10ns$. Thus we get $p > (110ns - 100ns)/(110ns - 10ns) \;=\; 1/10$. In our motivating example, the term a/b occurs in nodes 1 and 3. Since p_n denotes the probability that term a/b is valid at program point n, p_n can be calculated easily by the ratio:

$$p_n \;=\; \frac{\mathit{nr.\ of\ times\ a/b\ is\ available\ at\ n}}{\mathit{nr.\ of\ times\ n\ occurs\ in\ \pi}} \tag{4}$$

Node 1 is executed once and term a/b is never available, thus we get $p_1 = 0/1 = 0$. Since p_1 is not greater than p, a decision to perform the transformation is negative. On the other hand, node 3 is executed 8 times and term a/b is available and reused 7 times which results in $p_3 = 7/8$. Now $p_3 > p$ holds and the transformation should therefore be performed for node 3.

Calculation of the probabilities p_n is crucial to our cost model. An important observation is that the definition of probability p_n in Equation (4) is identical to the definition of probabilistic partially available expressions.

3.3 Probabilistic Partially Available Expression Analysis

Classical data flow analysis determines whether a data flow fact may hold or does not hold at some program point. Probabilistic data flow systems compute a range, i.e. a probability, with which a dataflow fact will hold at some program point [Ram96,MS01]. In probabilistic dataflow systems, control flow graphs annotated with edge probabilities are employed to compute the probabilities of dataflow facts. Usually, edge probabilities are determined by means of profile runs based on representative input data sets. These probabilities denote heavily and rarely executed branches and are used to weight dataflow facts when propagating them through the control flow graph.

An expression e is called *partially available* at a program point n, if there is at least one path from the entry node to n containing a computation of e and with no subsequent assignments to any variable used in e on that path. Such a path contains an unnecessary recomputation of e which can be avoided by *partial redundancy elimination* techniques.

Central to our cost model is the definition of p_n. Combining probabilistic data flow analysis with partial availability, we arrive at a suitable definition of p_n: p_n represents the probability that an expression e is available at program point n. If n is reached N times during a program's execution, and e is available on A of those N occasions, then p_n is estimated as A/N.

Our probabilistic data flow framework [MS01] takes as input profiles with edge probabilities of the control flow graphs and the dataflow equations for the dataflow problem. The dataflow equations for partial availability are defined in the usual way, and are shown in Figure 3. Profiling information is easily obtained with our GNU gcc environment by specifying appropriate compiler options. Thus we can obtain estimates for p_n values with minimal programming effort and with little extra compilation time (detailed experiments with SPEC95 have been published in [MS01]).

$$\textbf{N-PAVAIL}(n) = \begin{cases} false & \text{if } \; n = \text{start node} \\ \bigvee_{m \in pred(n)} \textbf{X-PAVAIL}(m) & \text{otherwise} \end{cases}$$

$$\textbf{X-PAVAIL}(n) = LocAvail(n) \vee \textbf{N-PAVAIL}(n) \wedge \overline{LocBlock(n)}$$

where

$LocAvail_{exp}(n)$ = There is an expression exp which is available at the end of n.
$LocBlock_{exp}(n)$ = The expression exp is blocked by some instruction of n.

Fig. 3. Partial Availability Analysis

3.4 Refinements of the Basic Transformation

First, the predicate can be omitted from assignments when the expression is known not to be available. More formally, if exp is not partially available at program point n, but on at least one path starting from n there exists a (partially) redundant occurrence of exp, the predicates in front of the assignments can be omitted (cf. Fig. 2, case 1). This simplification has been applied for assignment $s := a/b$ at node 1 in our motivating example.

Second, maintenance of the predicate can be omitted if the predicate is not used. Assume that variable v occurs in term exp and variable v is assigned a new value at program point n. If there is no subsequent use of that new value of v in a predicated assignment of exp on any path emanating from n, the assignment invalidating the reuse of the temporary h_{exp} can be omitted.

4 Related Work

Classical partial redundancy elimination techniques [KRS94] cannot always remove redundant expressions since static analysis approaches are too conservative. In [BGS98] it is reported that the number of dynamically eliminated expressions can be doubled by employing more sophisticated approaches. However, although a simple algorithm [Ste96] can achieve a complete removal of all partial redundancies, the approach causes code growth which is exponential in the number of expressions and is therefore not viable in practice. Speculative approaches [HH97,GBF98] do not restructure the control flow graph and insert additional computations into the control flow graph. They achieve nearly the same optimization results as complete removal. However, a major disadvantage of speculative PRE is that computations with side-effects cannot be handled. In [BGS98] a combination of speculative and code duplication is given. To limit code growth and to select the appropriate PRE technique, profile information is taken into account.

Our approach has the potential to remove all redundancies, though removal is performed only when it would be profitable. Predicates indicate whether the computation is available in temporaries or not. The approach is similar to memoization techniques for functional languages. However, no lookups in a memoization table are required since only the last computation of an expression is stored in a temporary and a predicate controls whether the computation is valid or not.

With our approach, a probabilistic data flow analysis is required to determine whether a transformation is profitable or not. Ramalingam [Ram96] pioneered the field of probabilistic data flow analysis which computes the probability of a dataflow fact. The approach yields an approximate solution which can differ from the accurate solution. In [MS01], that approach was improved by utilizing execution history for estimating the probabilities of the dataflow facts. For calculating the deviations of the probabilistic approaches from the accurate solution, the notion of an abstract run [MS00] was developed. An abstract run accurately calculates the frequencies; however the computational complexity is proportional to the program path length and is thus not feasible in practice. To compute an accurate solution in acceptable time, a novel approach [SM02] based on whole program paths was developed.

5 Summary

We have presented a partial redundancy elimination approach based on probabilistic data flow analysis and predication. Our basic transformation achieves a complete removal of all redundancies. Contrary to other approaches, the control flow graph is not restructured and the optimization is strictly semantics preserving, i.e. computations with possible side effects are handled correctly. However, predication can cause additional costs. Hence, cost-analysis controls the PRE transformation.

References

[BGS98] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa, *Complete removal of redundant expressions*, Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI) (Montreal, Canada), 17–19 June 1998, pp. 1–14.

[BGS99] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa, *Load-reuse analysis: Design and evaluation*, ACM SIGPLAN Notices **34** (1999), no. 5, 64–76.

[GBF98] Rajiv Gupta, David A. Berson, and Jesse Z. Fang, *Path profile guided partial redundancy elimination using speculation*, Proceedings of the 1998 International Conference on Computer Languages, IEEE Computer Society Press, 1998, pp. 230–239.

[HH97] R. Nigel Horspool and H. C. Ho, *Partial redundancy elimination driven by a cost-benefit analysis*, Proceedings of 8th Israeli Conference on Computer Systems and Software Engineering (ICSSE'97) (Herzliya, Israel), IEEE Computer Society, June 1997, pp. 111–118.

[KBC$^+$99] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy, *A global communication optimization technique based on data-flow analysis and linear algebra*, ACM Transactions on Programming Languages and Systems **21** (1999), no. 6, 1251–1297.

[KM02] J. Knoop and E. Mehofer, *Distribution assignment placement: Effective optimization of redistribution costs*, IEEE Transactions on Parallel and Distributed Systems **13** (2002), no. 6, 628 – 647.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen, *Optimal code motion: Theory and practice*, ACM Transactions on Programming Languages and Systems **16** (1994), no. 4, 1117–1155.

[MS00] E. Mehofer and B. Scholz, *Probabilistic data flow system with two-edge profiling.* Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00), ACM SIGPLAN Notices **35** (2000), no. 7, 65 – 72.

[MS01] ———, *A novel probabilistic data flow framework*, International Conference on Compiler Construction (CC 2001) (Genova, Italy), Lecture Notes in Computer Science (LNCS), Vol. 2027, Springer, April 2001, pp. 37 – 51.

[Ram96] G. Ramalingam, *Data flow frequency analysis*, Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI'96) (Philadephia, Pennsylvania), May 1996, pp. 267–277.

[SM02] B. Scholz and E. Mehofer, *Dataflow frequency analysis based on whole program paths*, Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT-2002) (Charlottesville, VA), September 2002.

[Ste96] Bernhard Steffen, *Property oriented expansion*, Proc. Int. Static Analysis Symposium (SAS'96), Aachen (Germany) (Heidelberg, Germany), Lecture Notes in Computer Science (LNCS), vol. 1145, Springer-Verlag, September 1996, pp. 22–41.