Parallel Processing Letters

Predicated Partial Redundancy Elimination using a Cost Analysis

Bernhard Scholz
Vienna University of Technology
Vienna, Austria

and

Eduard Mehofer
University of Vienna
Vienna, Austria

and

Nigel Horspool
University of Victoria
Victoria, BC, Canada

Received February 16, 2003
Revised September 26, 2003
Communicated by Harald Kosch

ABSTRACT

Partial redundancy elimination (PRE) is a key technology for modern compilers. However traditional approaches are conservative and fail to exploit many opportunities for optimization. New PRE approaches which greatly increase the number of eliminated redundancies have been developed. However, they either cause the code size to explode or they cannot handle statements with side-effects. In this paper we describe a predicated partial redundancy elimination (PPRE) approach which can potentially remove all partial redundancies. To avoid performance overheads caused by predication, PPRE is applied selectively based on a cost model. The cost analysis presented in the paper utilizes probabilistic data-flow information to decide whether PPRE is profitable for each instance of a partially redundant computation. Refinements of the basic PPRE transformation are described in detail. In contrast to some other approaches our transformation is strictly semantics preserving.

Keywords: Partial Redundancy Elimination (PRE), Probabilistic Data-Flow Analysis (PDFA), Predication

1. Introduction

Partial redundancy elimination (PRE) is an important optimization technique used in compilers to improve the efficiency of a program. The objective of PRE is to avoid unnecessary re-computations at runtime. The PRE transformation replaces

the computations by accesses to temporary variables and initializes them at suitable program points such that the number of evaluations is reduced. In addition to classical expression optimizations, several optimizations can be found in the literature which employ PRE as underlying technique. For example, PRE has been successfully applied in compilers for high-performance systems where communication optimizations [6] and dynamic redistributions [7] use PRE as the underlying optimization technique. PRE is also used for optimizations in RISC compilers. An important example of a RISC optimization is the load-reuse analysis [2].

It has been observed that traditional approaches [11,8] are too cautious and fail to take advantage of many opportunities for optimization. To improve the effectiveness of PRE, new approaches have been developed that use speculation [5,4] and code duplication [14]. Speculation uses profile information and inserts additional computations (i.e. speculative computations) in the program. In contrast, 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 code size.

In this paper we introduce a novel transformation for PRE that can potentially achieve a complete removal of all partially redundant computations without duplicating code. Our new PRE approach is based on predication. A predicate controls whether the value of a computation stored in a temporary variable is valid or not. If the value of the computation is not valid any more, the predicate is updated. Computations are conditionally executed depending on the predicate. If the temporary variable does not contain a valid value, the value is re-computed.

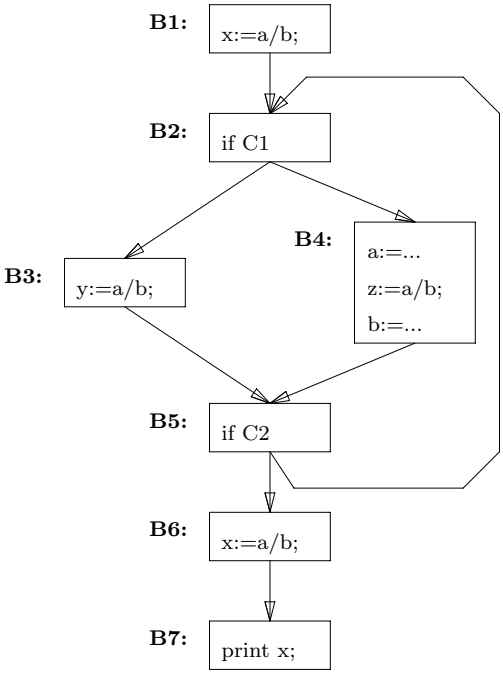
Many recent computer architectures, including the Intel Itanium and the ARM, provide predicated instructions. These instructions can, in some circumstances, be used to implement the predication described in this paper. We discuss how our PRE approach can take advantage of Itanium predicated instructions and provide a brief code example.

Our transformation is guided by a cost-analysis based on probabilistic data-flow analysis (PDFA) [10,13] that decides whether it is profitable to predicate a computation. Important features of our predicated partial redundancy elimination (PPRE) 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.

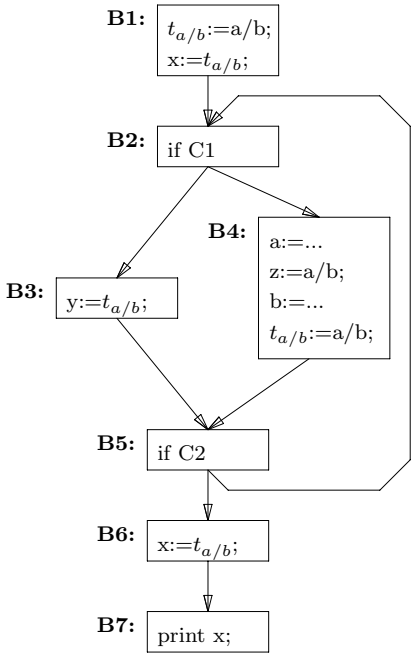
The paper is organized as follows. In Section 2 we motivate PRE with predication. In Section 3 the optimization is shown in detail. The basic transformation is given and a cost analysis determines whether the transformation is beneficial to the program performance or not. Section 4 surveys related work.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

2. Motivation

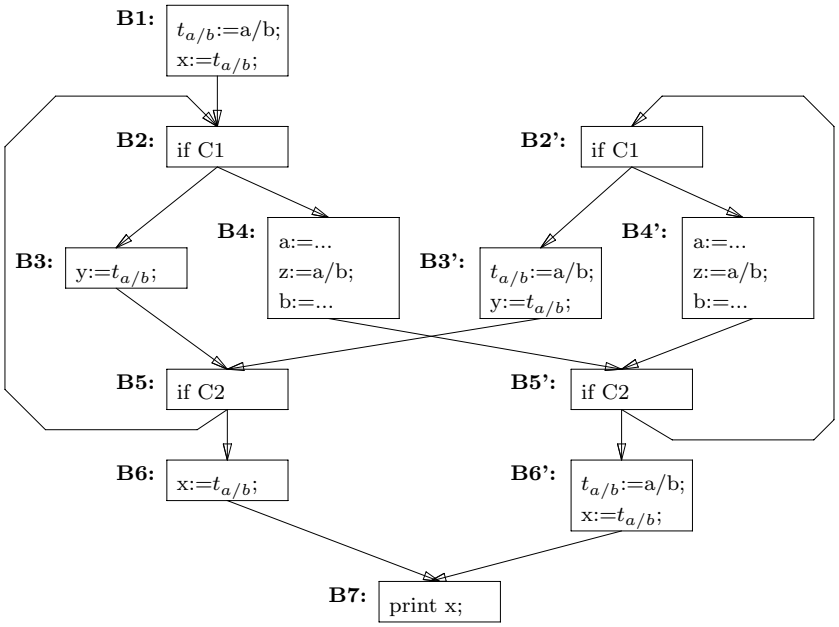
Consider our motivating example in Figure 1(a). The control flow graph consists of a branching statement inside a loop and four occurrences of expression a/b at nodes **B1**, **B3**, **B4**, and **B6**. The evaluation of expression a/b at node **B3** is partially redundant, since the computation is redundant with respect to node **B1**, but not redundant within the loop if the right branch with node **B4** is executed. Similarly,



(a) Example



(b) Speculative PRE



(c) Complete Removal by Code Explosion

Figure 1: Motivating Example.

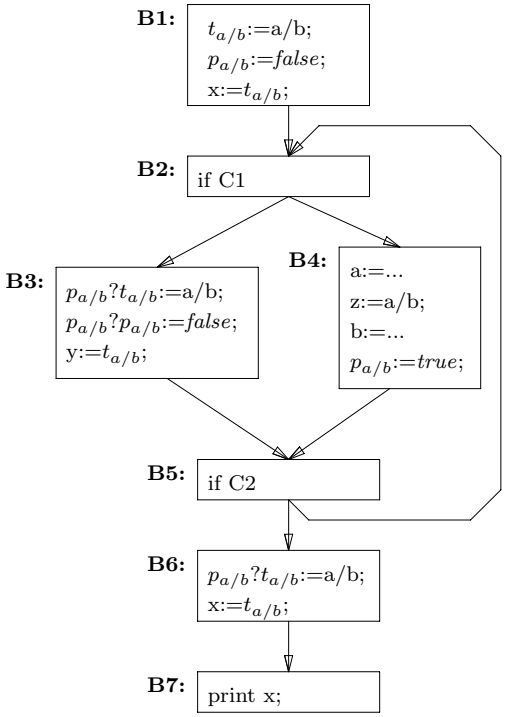


Figure 2: PRE with Predication.

evaluation of expression a/b at node **B6** is partially redundant, since it is redundant with respect to node **B3**, but not redundant with respect to node **B4**. The two remaining occurrences of expression a/b at nodes **B1** and **B4** are not (partially) redundant and their evaluation is required.

Traditional PRE approaches like [8] fail to eliminate the partial redundancies of expression a/b at nodes **B3** and **B6**. This deficiency has been addressed by the newer approaches to PRE. In Figure 1(b) an approach is illustrated that uses speculation [5,4]. Let us assume that the left-branch of the conditional statement within the loop is executed more often than the right one. Then the computation of expression a/b can be speculatively eliminated at node **B3** at the price of introducing a new computation of a/b at the end of node **B4**. Note that the assignment $t_{a/b} := a/b$ does not exist in the original program and might influence program semantics. Expression a/b is not a safe computation, since it can raise a "division by zero" exception. If b should happen to be zero in some execution of the program, an exception would occur. Hence, insertion of the expression on the right branch might destroy program semantics and has to be dealt with in some way.

The flow graph of Figure 1(c) shows another approach which completely removes all partial redundancies of the example in Figure 1(a) by employing code duplication as introduced in [14]. The transformation duplicates nearly all nodes in the control flow graph. After applying the transformation, two versions of most

nodes exist in the graph. One node represents the state where the computation is not available, while the other one represents the node where the computation 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(c), the expression a/b is computed and stored in $t_{a/b}$. 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 node **B4**, 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 $t_{a/b}$ again, and the loop on the left-hand side is re-entered. Not only does the control flow graph become irreducible,[a] but the number of nodes has nearly doubled. In the worst case, code growth is exponential in the number of expressions and the approach is not viable in practice. To alleviate this problem, Bodik et al. [1] have introduced an approach that limits code growth under the guidance of profile information. However, much code would still be duplicated.

The key idea of PPRE is to save the value of the computation 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 evaluations of the expression can be skipped and the saved value is loaded. In this way a complete removal of all partial redundancies can be achieved without duplicating the code. In comparison to speculative PRE, our approach is safe. No additional computations (other than predicate tests and predicate assignments) are inserted into the program.

Consider Figure 2. Predicate $p_{a/b}$ records the availability or unavailability of expression a/b. If the predicate $p_{a/b}$ holds, expression a/b is not available in temporary $t_{a/b}$ and must be re-computed, otherwise the value stored in $t_{a/b}$ can be re-used. Occurrences of expression a/b as well as statements that may block a re-use of the computed value have to be handled properly. The transformation for blocking statements is simple: Whenever a statement may block a re-use of an expression, the corresponding predicate must be invalidated. For example, assignments to variable a and b at node **B4** in Figure 2 block the re-use of the stored value of a/b and, hence, use of temporary $t_{a/b}$ is prohibited by setting predicate $p_{a/b}$ to true. Since the assignment inserted for the first blocking statement with variable a is dead, it can be removed and the second one for blocking statement with variable b is sufficient, as seen in node **B4**.

The basic transformation is shown at node **B3** in Figure 2. The original assignment $y := a/b$ is replaced with three statements, as follows. The first statement $p_{a/b}?t_{a/b} := a/b$ evaluates expression a/b and stores the value in temporary $t_{a/b}$, if predicate $p_{a/b}$ is true; otherwise the assignment is skipped. The second statement $p_{a/b}?p_{a/b} := false$ sets the value of predicate $p_{a/b}$ to false. And finally the third statement $y := t_{a/b}$ assigns the temporary to the original variable.

The basic transformation as shown at node **B3** can be optimized in several ways. If the expression is not partially available, the predicates can be omitted, as is illustrated at node **B1**. Since expression a/b is not partially available at node **B1**,

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

the assignments to temporary $t_{a/b}$ and predicate variable $p_{a/b}$ have not been predicated. Further, if the predicate variable is dead, the assignment to the predicate variable can be omitted. This is shown at node **B6** where the assignment to $p_{a/b}$ has been removed. And finally, if both above conditions hold, i.e. the expression is not partially available and the predicate variable is dead, no transformation is performed. This situation is illustrated by node **B4** where the assignment $z := a/b$ remains unchanged in Figure 2.

The transformation is not free of cost and neither are all computations of the expression replaced by predicated evaluations. Consider again our motivating example in Figure 1(a) and a program run that performs 10 loop iterations taking right branch (node **B4**), left branch (node **B3**), right branch, left branch, and so on, starting with right branch and terminating with left branch. Since variables a and b are modified within the loop, the expression has to be evaluated each time, i.e. there is no redundancy at all. Since the left branch and the right branch are executed 5 times, we get a total number of 20 executed assignments. In Figure 2, however, we get a total number 35 executed assignments. Thus for the above program run, PPRE does not pay off. On the other hand, if we consider a program run which always takes the left branch within the loop, PPRE succeeds in completely removing all evaluations of expression a/b within the loop. Therefore, the transformation needs to be guided by cost analysis based on profile information that decides whether a predicated computation improves program performance or not.

3. Predicated PRE (PPRE)

In this section, we develop our PPRE 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 basic transformation is shown in Figure 3. Basically every assignment $u := exp$ has to be replaced by the sequence $t_{exp} := exp; u := t_{exp}$ where t_{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 p_{exp} which is associated with the term exp as well and which indicates whether temporary t_{exp} can be reused, or the term exp has to be recomputed. The important point is that $t_{exp} := exp$ is not executed each time. The term exp is evaluated only if necessary, i.e. it is evaluated if the predicate p_{exp} is true as indicated by the predicated assignment $p_{exp}?t_{exp} := exp$, as shown in case 1 of Figure 3. Next p_{exp} is set to false to suppress unnecessary re-computations. However, if a variable occurring in exp is modified, the value held in t_{exp} cannot be reused any more, and the term exp needs to be recomputed. This situation is described in case 2 of Figure 3. There it is assumed that variable v occurs in term exp. Thus assigning a new value to v requires a re-computation of term exp, and this is enforced by setting the predicate p_{exp} to true.

Case 1: Compute	$\vdots$ u := *exp*; $\vdots$	$\vdots$ p_{exp} ? t_{exp} := *exp*; p_{exp} ? p_{exp} := **false**; u := t_{exp}; $\vdots$
Case 2: Block	$\vdots$ v := ... ; $\vdots$	$\vdots$ v := ... ; p_{exp} := **true**; $\vdots$

(a) Original Code (b) Transformed Code

Figure 3: Basic Transformation

It is important to stress that, contrary to some 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, PPRE guarantees that the exception is raised at exactly the same program point. 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.

Modern CPU architectures feature predicated execution, which facilitates conditional execution of statements. The flags are implemented as hardware predicates. Setting and resetting flags have very cheap costs and the re-computation of an expression can be conditionally executed. For example, the Intel Itanium [15] provides 64 predicate registers, numbered $p0$ through $p63$, which contain either 0 or 1. Nearly all Itanium instructions can be prefixed with a predicate register. The instruction is executed only if the predicate register holds 1; otherwise execution of the instruction is skipped. For example, the instruction

```
(p2) add r3=r1,r2,1
```

will add the contents of registers $r1$ and $r2$ and the constant 1, storing the result in $r3$, but only if the predicate register $p2$ contains 1. Unfortunately this instruction will use one CPU cycle whether it is executed or not and there would be no benefit from our technique. However, if the expression to be evaluated references memory or if the expression requires a call to a subroutine (for example, an integer division on the Itanium requires 16 instructions and would be a plausible operation to implement via a subroutine call), then use of predicated instructions would be beneficial. If, for example, the partially redundant expression is $a + b$ where a and b are both floating-point values held in memory, then the Itanium code to evaluate the expression and set the predicate flag (implemented as predicate register $p7$ here) would be similar to the following:

```
(p7) ldf.fill f8=a     // load a
(p7) ldf.fill f9=b     // load b
;;
(p7) fadd.s1 f10=f8,f9 // compute f10 = a+b
     p7=0              // suppress reevaluation of a+b
```

Finally, at a point where the expression is invalidated by a store to either of the memory locations a or b, the instruction

```
p7=1
```

should be inserted.

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

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

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 re-computation. By combining Equation (1) and (2) we obtain

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

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 before terminating the loop. Let us further assume that the execution times for term a/b are given as follows: $Recompute = 110ns$, $OrigComp = 100ns$, and $Reuse = 10ns$. Thus we get:

$$p > \frac{110ns - 100ns}{110ns - 10ns} \;=\; \frac{1}{10} \tag{4}$$

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:

$$p_n \;=\; \frac{nr.\ of\ times\ a/b\ is\ available\ at\ n}{nr.\ of\ times\ n\ occurs\ in\ \pi} \tag{5}$$

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 (5) 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 [12,10]. 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 re-computation 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 [10] 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 4. Profiling information is easily obtained with our modified 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 [10]).

4. Related Work

Traditional partial redundancy elimination techniques [8] cannot always remove redundant expressions since static analysis approaches are too conservative. In [1]

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

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

where

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

Figure 4: Partial Availability Analysis

it is reported that the number of dynamically eliminated expressions can be doubled by employing more sophisticated approaches. However, although a simple algorithm [14] 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 [5,4] 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 [1] 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. In [3], a full implementation of memoization for assembly code is proposed. However, the approach requires a hardware lookup mechanism and hardware buffers, which seems to be costly for a stock CPU design.

With our approach, a probabilistic data flow analysis is required to determine whether a transformation is profitable or not. Ramalingam [12] 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 [10], 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 [9] 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 [13] 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. However, since a complete removal of all partial redundancies might not always be profitable, we introduced a cost analysis and apply the transformation selectively. The cost analysis is based on probabilistic data flow analysis (PDFA) and utilizes profile information. Moreover, refinements of the basic transformation to improve the efficiency of the program have been presented. 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.

References

[1] R. Bodík, R. Gupta, and M.L. Soffa. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, Montreal, Canada, 17–19 June 1998.

[2] R. Bodik, R. Gupta, and M.L. Soffa. Load-reuse analysis: Design and evaluation. *ACM SIGPLAN Notices*, 34(5):64–76, May 1999.

[3] D.A. Connors and W.W. Hwu. Compiler-directed dynamic computation reuse: Rationale and initial results. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 158–169, Haifa, Israel, November 16–18, 1999. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[4] R. Gupta, D.A. Berson, and J.Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 230–239. IEEE Computer Society Press, 1998.

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

[6] 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(6):1251–1297, November 1999.

[7] J. Knoop and E. Mehofer. Distribution assignment placement: Effective optimization of redistribution costs. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):628 – 647, June 2002.

[8] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[9] 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(7):65 – 72, July 2000.

[10] E. Mehofer and B. Scholz. A novel probabilistic data flow framework. In *International Conference on Compiler Construction (CC 2001)*, Lecture Notes in Computer Science (LNCS), Vol. 2027, pages 37 – 51, Genova, Italy, April 2001. Springer.

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

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

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

[14] B. Steffen. Property oriented expansion. In *Proc. Int. Static Analysis Symposium (SAS'96), Aachen (Germany)*, volume 1145 of *Lecture Notes in Computer Science (LNCS)*, pages 22–41, Heidelberg, Germany, September 1996. Springer-Verlag.

[15] W. A. Triebel. *Itanium Architecture for Software Developers.* Intel Press, July 2000.