Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination

Bernhard Scholz
School of Information Technologies
Madsen Building F09
University of Sydney
NSW 2006, Australia
scholz@it.usyd.edu.au

Nigel Horspool
Department of Computer Science
University of Victoria
Victoria, BC
Canada V8W 3P6
nigelh@uvic.ca

Jens Knoop
Technische Universität Wien
Institut für Computersprachen
Argentinierstrasse 8
1040 Wien, Austria
knoop@complang.tuwien.ac.at

Abstract

Speculative partial redundancy elimination (SPRE) uses execution profiles to improve the expected performance of programs. We show how the problem of placing expressions to achieve the optimal expected performance can be mapped to a particular kind of network flow problem and hence solved by well known techniques. Our solution is sufficiently efficient to be used in practice. Furthermore, the objective function may be chosen so that reduction in space requirements is the primary goal and execution time is secondary. One surprising result that an explosion in size may occur if speed is the sole goal, and consideration of space usage is therefore important.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**] Compilers, Optimization.

General Terms

Algorithms, Measurement, Theory

Keywords

code motion, common subexpressions, partial redundancy, profile-guided optimization, speculation

1. INTRODUCTION

Partial redundancy elimination (PRE) removes redundant computations in a program [13]. It is a standard optimization technique in modern compilers. Most PRE algorithms involve Code Motion [12] and do not take profile information into account. They are therefore conservative in nature, not inserting a new computation of an expression if there is the risk of increasing the dynamic number of computations performed by the program on some execution path.

Speculative partial redundancy elimination (SPRE) inserts new computations on low frequency paths and removes them from high frequency paths, with the goal of minimizing the expected number of computations [9,6]. However, not all computations are suitable for SPRE. If a computation is inserted at some program point where the computation is not *anticipable* (i.e. it is not computed on all paths which originate at that program point), then the computation should be free of side-effects. For example, an expression that may cause a divide by zero exception should not be inserted on a path where that same expression did not occur in the original program.

Figure 1 shows a running example used throughout this paper; the expression to be optimized is `a+b`. Nodes in this control flow graph are labelled **B1** through **B9**, and their execution frequencies (obtained by profiling or other means) are shown as an italicized number alongside each block. This initial version of the program contains 3 static occurrences of `a+b` and 106 dynamic computations of `a+b`.

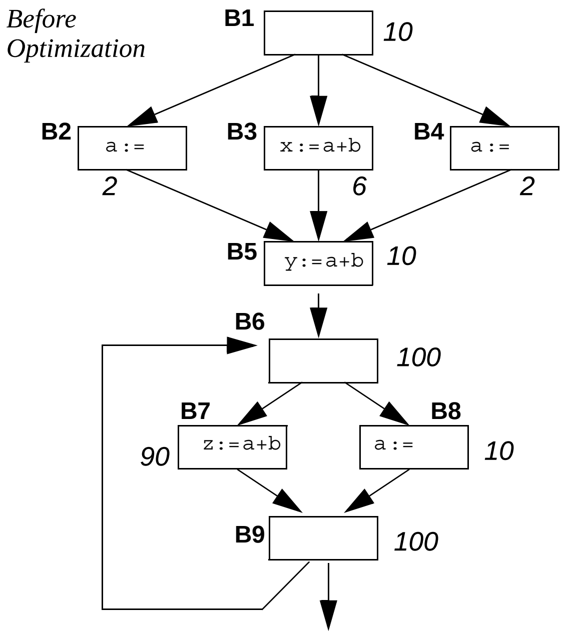


Figure 1. The running example

If SPRE were to be applied in its normal way, optimizing for speed and therefore minimizing the expected number of computations of a+b, we would obtain the optimized version of the program shown in Figure 2. The key improvement is to make the value of `a+b` available in a new variable `h` before the loop is entered. This allows the computation in node **B7** to be eliminated, replaced by a reference to `h`. However node **B8** invalidates (kills) the value held in `h`; to make `a+b` available

LCTES'04, June 11–13, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-806-7/04/0006…$5.00.

on the next iteration we had to insert a new computation of `a+b` at the end of **B8**. To make `a+b` available in `h` ready for entry to the loop, we had to insert new computations of `a+b` in nodes **B2** and **B4**, and we had to modify the code in node **B3**. The result is that the program of Figure 2 has 4 static occurrences and 20 dynamic computations of `a+b`. Note that although the program should be significantly faster than the original, the program would also be larger because it contains more static occurrences. This result may not be desirable for an embedded system application where space is often a critical resource, more so than execution time.

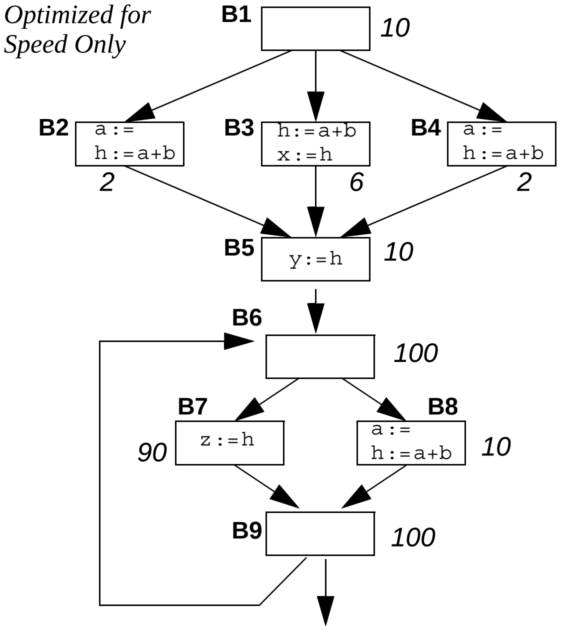


Figure 2. Running example – optimized for speed

A different transformation of the running example is shown in Figure 3. This version makes a+b available on loop entry in a different way. The result is that the new program contains 3 static occurrences and 26 dynamic computations of a+b. Although this version of the program would be a little slower than the one shown in Figure 2, it should also be a little smaller. After compilation and after the usual compiler optimizations of copy propagation, etc., it should be close in size to the original program of Figure 1 and yet significantly faster. Depending on the relative importance of space versus time for the application, this version may well be optimal for an embedded system application.

Our new SPRE algorithm allows an objective function to be used which accounts for memory usage or for execution time or for any linear combination of the two. By giving space a non-zero weighting, we can achieve reductions in size while simultaneously making the program run faster – the result shown in Figure 3.

Although our technique is not the first to offer an optimal solution to the SPRE problem [3], we believe it has these significant advantages.

- With our formulation, it is easy to choose between different cost models, thus allowing us to optimize for speed, code size, power consumption or any linear combination of them. Both code size and power use are especially relevant for embedded systems applications.

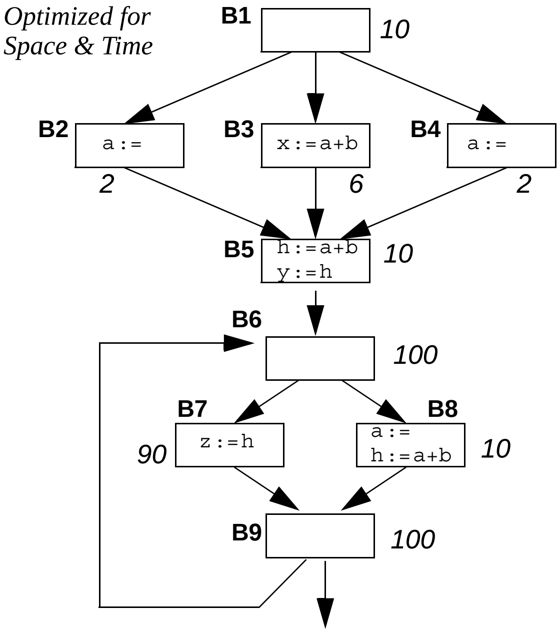


Figure 3. Running example – optimized for both space and time

- We map the problem to a form of network flow problem, known as Stone's Problem, for which optimal solutions can be *efficiently* found in polynomial time.

2. BACKGROUND

2.1 Control Flow Graph

A control flow graph G < N,E,s,f> is a directed rooted graph with the node set N, an edge set $E \subseteq N \times N$, and two distinguished nodes $s \in N$, a unique start (or entry) node, and $f \in N$, a unique final (or exit) node.

Edges $(u,v) \in E$ represent the (possibly non-deterministic) branching structure of G. The functions $succ(u) = \{ v \mid (u,v) \in E \}$ and $pred(u) = \{ v \mid (v,u) \in E \}$ represent the *immediate successors* and *immediate predecessors* of node u. A *finite path* of G is a sequence $\pi = <\upsilon_1, u_2, \ldots, u_k>$ of nodes such that $u_{i+1} \in succ(u_i)$ for all $1 \le u < k$. Symbol ε denotes the empty path. The notation $Path(u,v)$ denotes the set of paths starting at node u and ending at node v. A graph $G <N,E,s,f>$ is *well formed* if, for all nodes $u \in N$, there exists a path from the start node s to node u and from u to the final node f.

Without any loss of generality, we make two assumptions which simplify our presentation of the new SPRE algorithm. First, we assume that each node *n* represents a single simple statement, *not* a basic block. Generalization of the SPRE algorithm to use basic blocks is straightforward, but multiplies the number of cases which would need to be explained later on. Second, we assume that statements of the form `a:=`*`exp`* where `a` is an operand of *`exp`*, are expanded to two statements `t:=`*`exp`*`;  a:=t` which compute the right-hand side expression and modify the operand separately, where `t` is a new temporary variable. Forbidding "recursive" assignments like `a:=a+b` again reduces the number of cases which need to be explained. Removing this restriction in an implementation of the PRE algorithm is straightforward.

3. LOCAL TRANSFORMATION

In this section we consider properties of nodes in the CFG and a local transformation. Partial redundancy elimination is normally performed for all computations in a set. However, the optimization can be performed for each computation separately, achieving the same final result. Therefore, to simplify our exposition, we detail the PRE algorithm for just one computation e and we refer to it as *the computation*.

As already stated, we simplify the explanation further by using CFGs in which nodes represent simple statements. It is easy to generalize our technique to basic blocks.

For our program transformation, we introduce a new temporary variable h_e. This variable is used to hold the value of computation e for later reuse. For every node u in the CFG, we identify two program points – the entry to the node, labelled by i_u, and the exit from the node, which is labelled by o_u. Our analysis technique will determine whether we wish the computation e to be available at each of these program points. And, if the analysis does decide that e is to be available at the entry to a node u then the code inside u can potentially be simplified to access the value of e from variable h_e instead of recomputing e.

Given the set of $2|N|$ entry and exit labels, our algorithm will partitition that set into two subsets, A_e and $\overline{A_e}$. The former subset, A_e, is a set of program points where we wish to ensure that e is available in the variable h_e. The latter subset, $\overline{A_e}$, contains all the other labels. At these places, we do not care whether e is available after the transformation.

Therefore, if $i_u \in A_e$ and if our code transformations are performed correctly so that e is actually available at all these program points, then a computation of e within node u can be deleted – replaced with a use of h_e instead, thus reducing the cost of executing that node. However, putting i_u in A_e may incur costs elsewhere in the program, because it may be necessary to insert code of the form $h_e := e$ in one or more other nodes. Note that the labels i_u and o_u that we associate with a node u are not to be confused with data-flow predicates. They are not predicates at all. We use the notion of “labels” because the labels can be easily mapped to elements of Stone’s problem, to which we are going to reduce the SPRE problem.

The problem of performing partial redundancy elimination in an optimal manner thus reduces to the problem of making a globally optimal determination of A_e and $\overline{A_e}$, such that the program semantics is not destroyed and the total program cost is minimized.

For a node u, there are four combinations of choices to consider: the entry label is a member of either A_e or $\overline{A_e}$, and the exit label is also a member of either A_e or $\overline{A_e}$. We call these combinations *scenarios*.

Because of our simplifying assumptions that CFG nodes are simple statements and that no statement is “recursive” (of the form `a:=a+b`), we have just three kinds of program nodes to consider. They are as follows.

1. a node that neither computes nor kills e (*NULL* case),
2. a node that computes e and does not subsequently kill it (*COMP* case),
3. a node that does not compute e and kills e (*MOD* case).

We diagram the three cases when e is the expression `a+b` in Figure 4.

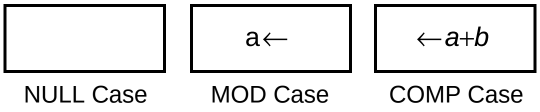


Figure 4. Three cases for nodes

Each combination of a scenario with the local properties of a node gives rise to different costs. In some scenarios, we are forced to compute e and assign it to h_e; in some others, we can substitute h_e for a computation of e. Intuitively, re-computing is more expensive than re-using. We introduce a cost function to model these costs.

DEFINITION 1. *Let* $cst_u : \{A_e, \overline{A_e}\} \times \{A_e, \overline{A_e}\} \rightarrow Z_0$ *be the cost function* $cst_u(i_u, o_u)$ *of a CFG node u, which gives the costs for a scenario.*

A scenario has either a zero cost if the computation is not performed or one cost unit if the computation has to be performed. For each scenario and each kind of node, we can enumerate which code transformations are required — both to make e available on exit when $o_u \in A_e$ and to take advantage of e being available on entry if $i_u \in A_e$. For each combination we specify a cost and a (local) code transformation to perform, as shown in Table 1.

Table 1. Costs and actions for scenarios

Case	Scenario		Cost	Action
MOD	$\overline{A_e}$	$\overline{A_e}$	0	–
	$\overline{A_e}$	A_e	1	insert$_X$
	A_e	$\overline{A_e}$	0	–
	A_e	A_e	1	insert$_X$
COMP	$\overline{A_e}$	$\overline{A_e}$	1	–
	$\overline{A_e}$	A_e	1	insert$_N$ (and delete)
	A_e	$\overline{A_e}$	0	delete
	A_e	A_e	0	delete
NULL	$\overline{A_e}$	$\overline{A_e}$	0	–
	$\overline{A_e}$	A_e	1	insert
	A_e	$\overline{A_e}$	0	–
	A_e	A_e	0	–

The code transformation actions are listed in Table 2.

This way, each partitioning of labels into the sets A_e and $\overline{A_e}$ specifies a (global code motion) transformation. In the following section, we will discuss when such a transformation is correct in the context of speculative PRE.

Before doing so, note that we implicitly assume `t=a+b;` and `h=a+b; t=h;` to have the same execution cost. The cost of an extra move instruction in the second version could be taken into account, if we so desired. However, copy propagation is performed after PRE and move instructions are often eliminated.

Table 2. Transformation actions

–	No transformation is performed
insert$_X$	Insert computation $h_e := e$ at exit from node
insert$_N$	Insert computation $h_e := e$ at entry to node and replace the computation of e with a use of h_e
delete	Replace the computation of e with a use of h_e

4. PROGRAM TRANSFORMATION

As already stated, the partitioning of the set of labels $\{i_u, o_u \mid u \in N\}$ between A_e and $\overline{A_e}$ represents a program transformation for speculative partial redundancy elimination. However, not all partitions preserve program semantics. Intuitively, there cannot be a node u for which $o_u \in \overline{A_e}$ but where it has a successor node v for which $i_v \in A_e$.

Conceptually, a PRE transformation for e can be considered a three-step procedure.

1. Introduce a temporary variable h_e to store the value of e.

2. Insert computations $h_e := e$ at some program points.

3. Replace some source code computations of e by h_e.

Conservative PRE considers a transformation matching this pattern *admissible*, if (1) insertions never introduce a computation of e on a path from s to f yielding a new value on that path, and (2) if a computation of e always yields the same value which is stored in h_e, when h_e is used to replace a source code computation of e at some program point.

It is the first constraint, which prevents conservative PRE in the tradition of the seminal work of Morel and Renvoise [13] from achieving the optimization transformation displayed in Figure 2. For speculative PRE, however, we can drop the first constraint, and only keep the second one, which is indispensable. This leads to the following more formal definition of a correct speculative PRE transformation. It relies on the predicates *store$_N$*, *store$_X$*, and *replace* defined for nodes. Here, the former two predicates indicate whether the value of e has to be stored at the entry or exit of the argument node, respectively, and the latter, whether a source code computation of e is to be replaced at the argument node. These correspond to the actions listed in Table 2.

4.1 Correctness

Denoting the set of all program transformations matching the three-step pattern of a PRE-transformation by *PRE*, we can define:

DEFINITION 2. *[Correctness] A program transformation of PRE is correct (in the context of speculative PRE) iff*

$$\begin{aligned}
&\forall p = <u_1, \ldots, u_k> \in Path(s,f)\text{: } \forall i\ (1 \le i \le k)\text{:}\\
&\quad replace(u_i) \Rightarrow\\
&\quad\ ((\exists j\ (1 \le j \le i)\text{: } store_N(u_j)\text{: } \forall k\ (j \le k < i)\text{: } COMP(u_k) \vee NULL(u_k))\\
&\quad\ \vee (\exists j\ (1 \le j < i)\text{: } store_X(u_j)\text{: } \forall k\ (j < k < i)\text{: } COMP(u_k) \vee NULL(u_k))
\end{aligned}$$

In the following we denote the subset of correct program transformations of *PRE* by *SPRE*. We are now ready to focus on optimal speculative PRE transformations.

4.2 Optimality

DEFINITION 3. *[Optimality] A program transformation of SPRE is optimal for a given profile iff the objective function*

$$f = \sum_u \mathrm{cst}_u(i_u, o_u)(\alpha\,\mathrm{frq}(u) + \beta\,\mathrm{spc}(u)) \qquad (1)$$

is minimal, where $frq(u)$ denotes the frequency of execution of node u, i.e. it represents dynamic cost, and $spc(u)$ denotes a space factor cost that would be incurred for making an insertion into node u. We would usually choose to have $spc(u)$ be a constant, independent of which node contains the insertion (though the constant may depend on which expression is the subject of the analysis and represent the number of instruction bytes needed for its evaluation). We denote the set of all optimal transformations of *SPRE* by $SPRE_{opt}$. The parameters α and β determine the trade-off between space and time.

If $\alpha > 0$ and $\beta = 0$, we optimize purely for speed. Conversely, if $\alpha = 0$ and $\beta > 0$, we optimize purely for space. Other combinations of non-negative values make any trade-off we like between time and space.

For any choice of a cost function, we want an optimal program transformation of *SPRE*. Note that by definition this implies correctness of this transformation as well.

5. STONE’S PROBLEM

We will map our formulation of SPRE to Stone’s Problem [16], which we use as a means for obtaining an optimal program transformation in polynomial time. We give a brief overview of Stone’s problem outlining, where appropriate, the relationship of the SPRE problem to this problem. We will explain the mapping of the SPRE problem to Stone’s Problem in more detail in Section 6.

Stone has solved the process allocation problem for two processors in polynomial time by using s-t min-cut reduction. Stone’s problem assumes a set of processes $P = \{p_1, \ldots, p_k\}$ with different execution time requirements, and two processors A and B with different execution speeds.[1] The total execution time will depend on which processes are assigned to which processors. In addition, communication might occur between two processes. If there is communication between two processes and they have been assigned to different processors, we have to take communication costs into account.

More formally, we introduce cost functions which reflect the execution and communication costs of Stone’s problem. First, the cost functions $wA(p)$ and $wB(p)$ give the execution costs of process p if executed on processor A or processor B, respectively.[2] Second, the communication costs between two processes p_1 and p_2 are given by $wAB(p_1,p_2)$. Note that communication costs are only taken into account if the two processes are assigned to different processors; otherwise the communication costs are implicitly zero.

[1] In our application the entry and exit labels will play the role of processes, and the sets A_e and $\overline{A_e}$ the role of the processors.

[2] Considering our application, the execution costs will be given by the frequency information of the program profile.

Then, Stone's problem is defined as follows.

DEFINITION 4. *[Stone's Problem] Given a set P, find two partitions A and B of P (i.e. P=A∪B and A∩B=∅) such that the objective function*

$$f = \sum_{p \in A} wA(p) + \sum_{p \in B} wB(p) + \sum_{p \in A \wedge q \in B} wAB(p, q) \qquad (2)$$

is minimal.

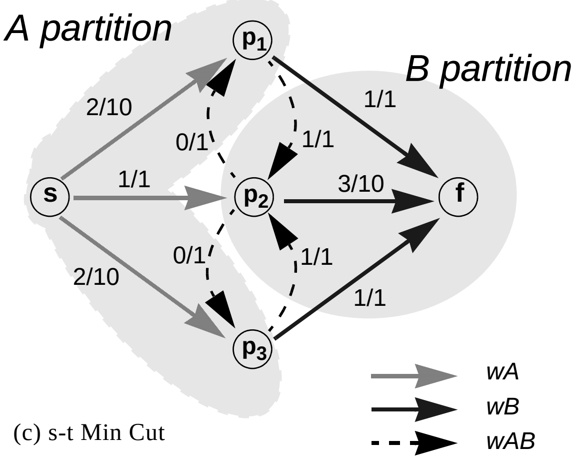
Consider the example in Figure 5. In this example we have three processes. The execution times of the processes on both processors are given in Figure 5(a). Column *wA* gives the execution time for processor *A* and column *wB* gives the execution time for processor *B*. The communication costs are shown in Figure 5(b). In this example, the table is symmetric (i.e. $wAB(p_1,p_2)$ is equal to $wAB(p_2,p_1)$), though this is not a necessity.

P	wA	wB
p_1	1	10
p_2	10	1
p_3	1	10

(a) Execution Costs

wAB	p_1	p_2	p_3
p_1	0	1	0
p_2	1	0	1
p_3	0	1	0

(b) Communication Costs



(c) s-t Min Cut

Figure 5. Example of Stone's Problem

Without employing the s-t min-cut reduction, it is quite clear that processes p_1 and p_3 should be executed on processor *A*. Only p_2 should be performed on processor *B*. For this process allocation, the overall execution time has three cost units. Other allocations have significantly higher execution costs. Though there are communication costs between processes p_1 and p_2, and communication costs between processes p_2 and p_3, the overall communication costs are lower than letting the processes run on the same processor.

Stone's problem can be algorithmically solved by employing a s-t min-cut reduction. A s-t network is constructed such that every process of Stone's problem is represented by a node in the network. In addition, there are two artificial nodes, i.e. the start and target node of the network.

For execution costs of processor *A*, edges between the processes and a special sink node *f* are added. The capacity of each edge reflects the execution time of the corresponding process on processor *A*. Correspondingly, edges are added between a special source node *s* and the processes which reflect the execution time of Processor *B*. Finally, edges are inserted which model the communication costs between processes. The capacity of an edge between processes p_1 and p_2 is given by the communication costs *wAB*. After constructing the network, the set of cut edges with minimal costs disconnects the graph into two partitions. Stone [16] showed that the objective function of Definition 4 is identical to the minimum cut-set of the constructed s-t network. The cut-set can be found by computing the max-flow of the s-t network [4].

For our example, the s-t min cut reduction is shown in Figure 5(c). All three processes appear as nodes in the s-t network. The execution costs for processor *A* are modelled as edges between processes and the target node. Similarly, the edges between source node and the processes reflect the execution costs on processor *B*. Communication occurs between processes p_1 and p_2 and processes p_2 and p_3. Since we have to take both directions into account, edges in both directions are added. In Figure 5(c) edges are labelled with two numbers. For example the label 3/10 appears between p_2 and *f*. The first number, 3, gives the flow that was computed by the max-flow problem; the second number, 10, gives the capacity of the edge as determined from the tables in parts (a) and (b) of the figure. Based on the flow, the cut-edges can be determined (for a cut edge it is necessary that the flow is equal to the capacity). For our example the cut-edges are $\{(s,p_2), (p_1,p_2), (p_2,p_3), (p_1,t), (p_3,t)\}$ which disconnect the set of processes into two partitions. The sum of the weights of all cut edges yields the minimum of Stone's objective function which is 5 in our case, comprising 3 cost units for execution and 2 cost units for communication.

6. MAPPING

In this section, we show the mapping of SPRE to Stone's problem. First, we convert the cost functions of the local transformations summarized in Table 1 into a mathematical notation as shown in Table 3. Basically, the three cases for a node *u* can be written as conditional functions. For the COMP case, a computation (the original computation in that node) is performed if the entry label i_u is not in A_e (otherwise the computation in the node would be deleted). The MOD case requires the introduction of a computation if the exit label o_u specifies that the computation must be available at the exit. The NULL case is more complex: a computation of *e* must be inserted in the node if the entry label i_u is not in A_e but the exit label o_u is a member of A_e.

Constructing the network. The entry and exit labels correspond to the processes of Stone's problem, and the processors correspond to the two possible sets (A_e and $\overline{A_e}$) between which the labels are to be partitioned. Different costs are imposed for placing a label in the A_e set or in the $\overline{A_e}$ set.

Table 3. Cost functions for local transformations

COMP	$cst_u(i_u,o_u)$ =	$\begin{cases} 1, & i_u \in \overline{A_e} \\ 0, & i_u \in A_e \end{cases}$
MOD	$cst_u(i_u,o_u)$ =	$\begin{cases} 1, & o_u \in A_e \\ 0, & o_u \in \overline{A_e} \end{cases}$
NULL	$cst_u(i_u,o_u)$ =	$\begin{cases} 1, & i_u \in \overline{A_e} \wedge o_u \in A_e \\ 0, & \text{otherwise} \end{cases}$

We can transform the objective function into an objective function of Stone's problem as follows. Given the cost formula

$$f = \sum_u \mathrm{cst}_u(i_u, o_u)(\alpha\mathrm{frq}(u) + \beta\mathrm{spc}(u))$$

then for all nodes $\upsilon \in$ COMP, we use $\mathrm{cst}_u(i_u, o_u) = 1$ if $i_u \in \overline{A_e}$ and 0 otherwise; for all nodes u ∈ MOD we use $\mathrm{cst}_u(i_u, o_u) = 1$ if $o_u \in \overline{A_e}$ and 0 otherwise; finally for all nodes u in NULL we use $\mathrm{cst}_u(i_u, o_u) = 1$ if $i_u \in \overline{A_e} \wedge o_u \in \overline{A_e}$, and 0 otherwise.

The mapping to Stone's problem is straightforward in the above formulation. When we are minimizing the expected number of computations of the expression *e*, the costs of *wA* and *wB* are either zero or the execution frequency of node *u*. When we are minimizing for space, the costs of *wA* and *wB* are either zero or one. (Cost measures which combine space and time can be specified in the obvious manner.) The costs for the transparent case can be mapped to communication costs between two processes. In contrast to Stone's original formulation, the mapping does not consider symmetric communication costs. This means, that the reverse case (i.e. the exit label is in $\overline{A_e}$ while the entry label is in A_e) is not needed to obtain an optimal solution. Intuitively, an optimal solution that contains an instance of the reverse case can always be transformed to an equally optimal solution by moving the exit label from $\overline{A_e}$ to A_e (as we can do if the node does not kill *e*).

However, correctness constraints need to be considered; we can incorporate them into the cost model with a simple transformation. We add a term Δ to the objective function *f*. The value of Δ is zero if the correctness constraints are fulfilled and infinite otherwise.

LEMMA 1.

$$h = f + \Delta \qquad (3)$$

The new objective function *h* is equivalent to *f* iff the constraints hold – otherwise it equals ∞.

The Δ term is chosen as infinity in two cases. First, the entry label for the start node *s* is required to be an element of $\overline{A_e}$ so that no computations are assumed to be available when the program starts; thus we assign infinite cost to the entry label i_s being in A_e. The second case arises if one of the correctness constraints along a program edge is violated.

The first constraint sets the costs for i_s to infinity if it is member of A_e. The second constraint is mapped to the communication costs of Stone's problem. The correctness constraint is violated iff, for an edge $(u,v) \in E$, the label i_v is in A_e and the label o_u is in $\overline{A_e}$. In other words, correctness is violated if the succeeding node expects the computation but the preceding node does not deliver the computation. It is the only case when the constraint is violated

In Stone's problem this can be simply modelled with communication costs, i.e. for every edge $(u,v) \in E$ in the control flow graph, we insert an edge in the network between o_u and i_v and attach infinite weight to this edge.

The Running Example. For our running example, we construct the s-t min-cut network as demonstrated in Figure 6. The nodes in the network are the entry and exit labels associated with the CFG nodes. We draw these labels inside a box for each node, where we used a white box for a NULL node case, a solid grey box for a COMP case, and a striped box for a MOD case. For example, node **B1** represents a NULL case, **B2** is a MOD case, and **B3** is a COMP case.

The edges in the network either model computational costs or correctness constraints. Each edge is labelled with an annotation like *a/n*. The second number *n* denotes the maximum capacity of the edge, as determined by the constraint rules. The first number, *a*, denotes the flow along that edge when the max-flow is determined for the network with the given edge capacities.

To solve SPRE finding the optimal number of *dynamic computations*, we construct edges as follows.

- For each edge $(u,v) \in E$ in the control flow graph, insert an edge (o_u, i_v) in the network with infinite capacity.
- For each NULL node *u*, create an edge (i_u, o_u) with capacity frq(*u*).
- For each COMP node *u*, add an edge (i_u, f) with capacity frq(*u*).
- For each MOD node *u*, add an edge (s, o_u) with capacity frq(*u*).
- Create an edge (s, i_1) with infinite capacity, where block B1 is assumed to be the entry node of the CFG.

The max-flow of the example is 20 cost units, which corresponds to the min-cut shown as the dotted line in Figure 6. The capacities of the edges crossed by the min-cut line are 2, 6, 2 and 10 which sum to 20. This means that the lowest dynamic cost for the running example is 20. Using the min-cut solution, we divide the set of labels into the "not needed" partition ($\overline{A_e}$) and the "computed" partition (A_e) as desired. The not needed partition is {s, i_1, o_1. i_2, i_3, i_4}; the needed partition contains all the other labels, namely {o_2, o_3, o_4, i_5, o_5, i_6, o_6, i_7, o_7, i_8, o_8, i_9, o_9, f}. Applying the local transformations to each block as determined by the label partitioning leads to the optimized program presented in Figure 2

6.1 The Program Transformation

After partitioning the labels, we can perform the local transformations on the nodes, which were described in Section 3. As desired, this results in the global transformation which has already been shown in Figure 2.

Use of a cost function which incorporates a space component would produce the transformation shown in Figure 3.

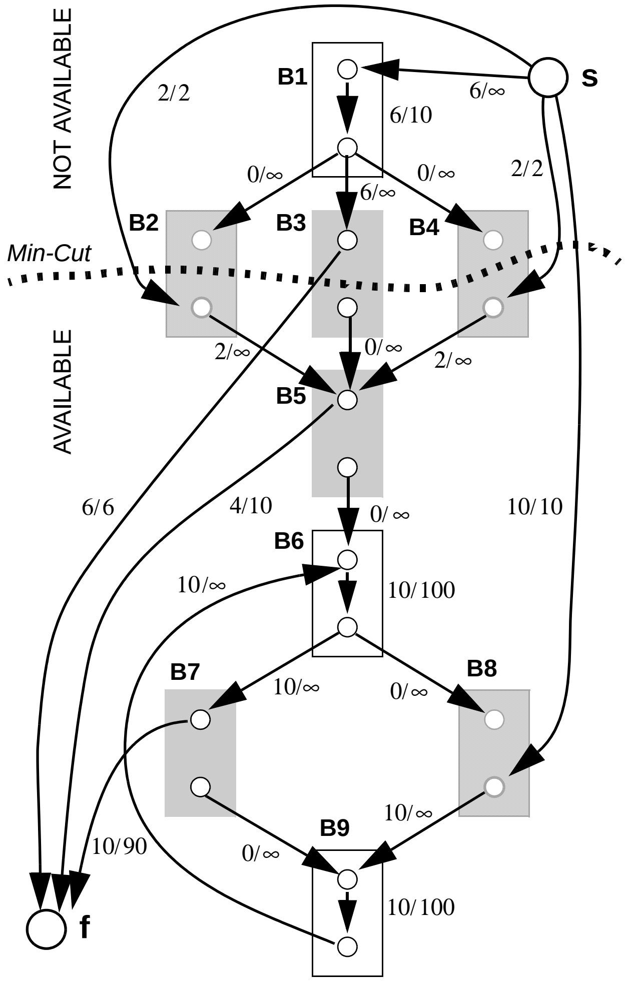


Figure 6. Network for running example

(using edge capacities for speed optimization)

6.2 Correctness and Optimality

The following two lemmas help prove the correctness and optimality of the transformation resulting from our analysis.

LEMMA 2. Let A_e and $\overline{A_e}$ be a partitioning of labels, and let τ be the program transformation induced by this partitioning. If $\forall (u,v) \in E: i_v \in A_e \Rightarrow o_u \in A_e$, then τ is correct.

Then we have:

LEMMA 3. Let A_e and $\overline{A_e}$ be the partitioning resulting from our analysis. Then we have: $\forall\ (u,v) \in E : i_v \in A_e \Rightarrow o_u \in A_e$.

Together with the minimality of the cost function ensured by the solution of the network problem, this implies the desired optimality of the program transformation of our approach. In fact, denoting the program transformation induced by the partitioning resulting from our algorithm by *SPRE,* we have as desired:

THEOREM 1 [Optimality] $SPRE \in SPRE_{opt}$.

7. FASTER ANALYSIS

Reducing the number of nodes in the network constructed for Stone's Problem has a major impact on performance. We show how to do that below, while still obtaining the same optimal result for the SPRE problem.

One simple approach is suggested by the network created for the running example, as shown in Figure 6. There are several dead-end nodes, such as those labelled i_2 and o_8 (the input node for block **B2** and the output node for **B8** respectively). These nodes and their adjacent edges may be safely deleted from the network without affecting the min-cut solution. A more sophisticated mapping from the control flow graph to the network could avoid creating these nodes initially.

Alternatively, and this is the approach we chose to implement, the number of nodes in the network can be significantly reduced by changing the implication relation that reflects the correctness constraints on program edges into an equivalence relation. The equivalence relation amalgamates several labels. Then several nodes in the original network can be represented by one node in the reduced network. The solution remains optimal and does not violate any correctness constraints, as argued below.

First note that changing the implication relation to an equivalence relation narrows the set of correct program transformations. This is because an implication allows the computation to be discarded along a program path, while the equivalence relation does not. However, the cost functions fulfil the following two (monotonicity-like) constraints:

LEMMA 4.

$$\forall u \in N\colon \forall X \in \{A_e, \overline{A_e}\} : \mathrm{cst}_u(X, \overline{A_e}) \le \mathrm{cst}_u(X, A_e) \tag{4}$$

$$\forall u \in N\colon \forall X \in \{A_e, \overline{A_e}\} : \mathrm{cst}_u(A_e, X) \le \mathrm{cst}_u(\overline{A_e}, X) \tag{5}$$

The first constraint states that requiring the computation at the end of a node is more expensive than not requiring it. The second one says that having the computation at the entry is cheaper than not having it (i.e. a locally better solution for the node in the original CFG may be permitted). Based on these two constraints, we can construct two cases. The first case is a portion of a network where there are several predecessor nodes $u_1, \dots, u_k$ and one successor node *v*. The predecessor nodes do not have any successor nodes other than *v*. A situation where $i_v \in \overline{A_e}$ while $o_{u_i} \in A_e$ holds for some *i* would violate the equivalence relation. That is, the equivalence relation would force $o_{u_i} \in \overline{A_e}$ for all *i*. But this is not a problem because the first constraint says we can discard the computation without increasing the cost.

The second case can be constructed similarly. Assume we have one predecessor node *u* and several successor nodes $v_1, \dots, v_k$, each of which has a unique predecessor – namely *u*. Assume that *u* provides the computation but one of the successor nodes does not need the computation on entry. According to the second constraint, we know that keeping the computation at the entry does not destroy the optimality.

Partitioning the control flow graph into these two cases is straightforward. However not all edges can be classified into either of these two cases and they are called *critical edges,* which impose problems even for classical PRE algorithms. Some formulations of PRE remove critical edges by inserting new (empty) nodes into the CFG, however that is not necessary for our approach. For our formulation of SPRE, we simply have to keep critical edges and their associated labels unreduced.

8. EXPERIMENTS

We conducted experiments with the SpecInt95 benchmark suite and the Gnu Compiler Collection to compare our speculative approach with the classical approach.

Though the performance gains achieved by pure PRE are very limited for modern computer hardware [2], we want to stress that PRE techniques are applicable to other areas of optimization such as load/store elimination, communication optimization [11], etc. From these optimizations, significant performance gains can be expected, leveraging the benefits of a more advanced PRE technique.

We used the production run of the SpecInt95 benchmark suite for profiling. We had several questions about it.

1. What is the problem size of SpecInt95?
2. What fraction of compile time does SPRE account for?
3. How does our new SPRE approach compare to the classical PRE approach?
4. What percentage of nodes can be eliminated by applying the techniques of Section 7?

The relevant parameters of the problem size are given in Tables 4 and 5. The first table shows the quantitative numbers (num) of how many control flow graphs (CFGs), number of basic blocks (Blocks), number of edges (Edges) and number of different expressions (Exprs) are in the SpecInt95 programs. Table 5 shows the percentages of the CFGs, basic blocks and edges that are actually executed during the test runs used for gathering profile information. The numbers indicate that only a fraction (about 40%) of the basic blocks in the programs are executed in those test runs. This means that classical PRE methods are likely to spend significant time optimizing large portions of code which is hardly ever executed.

Table 4. Sizes of the SpecInt95 programs

Benchmark	CFGs	Blocks	Edges	Exprs
099.go	360	16013	21483	12657
124.m88ksim	221	7063	9433	3448
126.gcc	1554	88875	121881	39737
129.compress	20	332	409	218
130.li	167	2825	3573	1302
132.ijpeg	350	7533	9566	6059
134.perl	234	16041	22235	7257
147.vortex	834	26664	35692	13029
TOTALS	3740	165346	224272	83707

Table 6 shows the overall compile time (Total Time) and the overhead for solving the SPRE networks. The overhead is, on average, a little over 4% of total compilation time, which we consider to be acceptable. Better algorithms [5, 10] than Ford-Fulkerson could be used for max-flow and reduce the compile time overhead.

Table 5. Fractions of SpecInt95 programs executed

Benchmark	% CFGs	% Blocks	% Edges
099.go	96.94	90.72	7.67
124.m88ksim	19.91	8.88	6.77
126.gcc	65.89	48.89	43.39
129.compress	75.00	77.11	73.11
130.li	64.07	43.65	36.61
132.ijpeg	38.00	29.09	24.65
134.perl	55.13	23.00	18.94
147.vortex	3.96	1.73	1.30
OVERALL	49.04	40.18	36.11

In Tables 7 and 8, we compare our SPRE approach for three different objective functions with the classical approach. Table 7 shows the figures for the numbers of static computations of expressions in the program, while Table 8 shows dynamic figures. The comparisons are given as ratios, comparing the number of computations in the optimized program against the number in the original program. The best ratios in each row are shown in **bold**.

Table 6. Compile time with SPRE

Benchmark	Total Time	SPRE Time
099.go	43.41 secs	3.25%
124.m88ksim	38.11 secs	0.30%
126.gcc	253.14 secs	5.98%
129.compress	2.02 secs	0.79%
130.li	14.05 secs	0.38%
132.ijpeg	36.17 secs	0.46%
134.perl	66.24 secs	9.42%
147.vortex	106.41 secs	0.39%
TOTAL	559.55 secs	4.21%

Table 7. Static ratios (space)

	SPRE + Cost Model			
Benchmark	speed	mix	space	PRE
099.go	2.72	0.87	**0.85**	0.92
124.m88ksim	2.17	**0.91**	**0.91**	0.99
126.gcc	23.04	0.96	**0.92**	0.98
129.compress	2.01	**0.94**	**0.94**	0.97
130.li	2.83	0.94	**0.93**	0.97
132.ijpeg	2.35	0.97	**0.96**	0.99
134.perl	56.51	0.96	**0.90**	0.99
147.vortex	1.15	**0.91**	**0.91**	1.04

Table 8. Dynamic ratios (time)

	SPRE + Cost Model			
Benchmark	**speed**	**mix**	**space**	**PRE**
099.go	**0.81**	**0.81**	0.88	0.84
124.m88ksim	**0.97**	**0.97**	1.00	0.98
126.gcc	**0.93**	**0.93**	1.23	0.95
129.compress	**0.90**	**0.90**	0.98	0.92
130.li	**0.96**	**0.96**	1.11	0.97
132.ijpeg	**0.98**	**0.98**	1.03	0.99
134.perl	**0.97**	**0.97**	1.53	0.98
147.vortex	**0.95**	**0.95**	1.14	0.96

Table 9. Reduction in network size for SpecInt95

	Nodes		Edges	
Benchmark	**num**	**c%**	**num**	**c%**
099.go	1946724	31.43	2251804	38.71
124.m88ksim	383864	30.76	444013	37.21
126.gcc	20787804	29.21	24941666	38.28
129.compress	11372	34.57	12238	39.03
130.li	65852	32.70	72019	38.31
132.ijpeg	400708	33.55	442407	40.24
134.perl	9088578	28.17	11041344	36.38
147.vortex	2221096	29.90	2616346	38.35
OVERALL	34905998	29.19	41821837	37.81

Our three objective functions were: **speed** which optimizes the dynamic number of computations, **space** which optimizes the static number of computations, and **mix** which is similar to *speed* but has a small weighting for space costs.

A ratio of, say, 0.88 in Table 7 indicates that the optimized program contains only 88% of the number of computations as compared to the original program and hence should require less memory. The same ratio in Table 8 would indicate that the optimized program executes only 88% of the computations executed by the original program.

As can be seen in the two tables, the *speed* cost model does produce the smallest numbers of dynamic computations but can cause the size of the program to explode. (In a couple of cases, that explosion is dramatic.) Conversely, the *space* cost model significantly reduces the static number of computations but can cause the number of dynamic computations to increase much more. Only when both the space and time objectives are taken into account with the *mix* cost model do we achieve excellent results in both dimensions.

The dynamic results show that we eliminate up to 57% more executed computations than with classical PRE. On average, our SPRE approach is 34% better than classical PRE. Both the *mix* and *space* cost models can dramatically reduce the static number of computations in a program. This is important for making good use of instruction cache buffers in modern computers, as well as optimizing for embedded systems architectures.

Table 9 shows the importance of reducing the number of network nodes. The table shows the total number of edges and nodes (num) and the percentage (c%) which remains after the reduction techniques described in Section 7. About one third of the original network nodes and 40% of the original edges remain in the reduced network. The time spent in the min-cut algorithm is more than halved by this reduction.

9. RELATED WORK

Classical approaches to PRE [12,13] are conservative and cannot handle a case like that shown in Figure 1, where speculative insertions of computations are needed to improve the expected performance. Hailperin [8] went beyond the classical PRE approach by introducing a cost function to control the overall transformation. His approach blends elements of PRE with constant propagation and strength reduction in a single transformation. However, the impact on code size cannot be controlled by this approach.

Speculative PRE was first introduced by Horspool and Ho [9] who related the problem to network flow and proposed the use of a min-cut algorithm. However their formulation of the algorithm did not always discover optimal solutions. Bodik's Ph.D. thesis [1] covers speculative PRE and describes an algorithm for its solution. A claim is made for the optimality of the algorithm but no proof and no experimental results are provided. Further work was performed by Gupta with Bodik, Soffa and others [2,6,7]. Steffen [15] and Bodik, Gupta and Soffa [2] replicated code to increase the amount of redundancy which could be eliminated. However code size becomes a major issue and has to be kept under control, with Steffen using a bisimulation approach and Bodik et al. using profiling information to guide the expansion process.

The closest work to that reported here is a recent paper by Cai and Xue [3] which is the first to provide a provably optimal solution to SPRE and include experimental results. However we consider that our approach has some significant advantages over theirs. First, their approach requires edge profiling, whereas we use basic block counters which are easier to implement. Second, our approach uses the s-t min-cut algorithm to find the optimal solution whereas their approach requires two dataflow analyses before getting to that point. Third, and finally, their approach does not take space into account. As our experiments demonstrate, a space explosion can easily occur if space is not considered as a contributor to the cost of a solution.

One paper which considers PRE and the effect on program size was recently published by Rüthing, Knoop and Steffen [14]. It allows a prioritization of speed, size, and register pressure, and automatically generates a proven optimal program with respect to the prioritization chosen. However, in contrast to the approach here, it is conservative, i.e., it never impairs a program path. Hence, speculation as required for our running example is beyond its scope. A trade-off between, say, speed and size is supported by providing either a code-size minimal solution among the computationally best transformations, or, alternatively, the computationally best solution among the minimal code-size solutions. Handling the trade-off flexibly using a linear combination of costs, as here, would be beyond its scope.

10. CONCLUSIONS

We have shown how the SPRE problem can be generalized to optimize for space or for time or for a linear combination of them and solved optimally in an efficient manner.

We have not only shown that the optimal SPRE approach yields significantly better solutions than the classical PRE approach, but we have also shown that an optimal time solution can be undesirable. This is a surprising result. Our experiments show that optimizing for time without regard to space can sometimes lead to an explosion in size. The use of a cost function which combines both space and time is therefore an essential ingredient of a SPRE algorithm and would be very important when compiling for embedded systems.

11. ACKNOWLEDGMENTS

We would like to thank Erik Eckstein who had the original idea for the local transformation of a basic block. Two authors gratefully acknowledge funding received from the Natural Science and Engineering Research Council of Canada.

12. REFERENCES

[1] R. Bodik. Path-Sensitive Value-Flow Optimizations of Programs. Ph.D. thesis, University of Pittsburgh, 1999.

[2] R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant computations. Proceedings of ACM Conference on Programming Language Design and Implementation, vol. 33, 5, pages 1–14, New York, June 1998.

[3] Q. Cai and J. Xue. Optimal and efficient speculation-based partial redundancy elimination. Proceedings of the 1st IEEE/ACM International Symposium on Code Generation and Optimization, pages 91–102, 2003.

[4] L. Ford and D. Fulkerson. Flows in Networks. Princeton University Press, 1962.

[5] A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. Journal of the ACM, vol. 35, 4, pages 921–940, 1988.

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

[7] R. Gupta and R. Bodik. Register pressure sensitive redundancy elimination. Proceedings of International Conference on Compiler Construction (CC99), LNCS, vol. 1175, pages 107–121, Springer-Verlag, March 1999.

[8] M. Hailperin. Cost-optimal code motion. ACM Transactions on Programming Languages and Systems, vol. 20, 6, pages 1297–1322, Nov. 1998.

[9] R. N. 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, pages 111–118, June 1997.

[10] V. King, S. Rao, and R. Tarjan. A faster deterministic maximum flow algorithm. Journal of Algorithms, vol. 17, 3, pages 447–474, 1994.

[11] J. Knoop and E. Mehofer. Distribution assignment placement: effective optimization of redistribution costs. IEEE Transactions on Parallel and Distributed Systems, vol. 13, 6, pages 628 – 647, 2002.

[12] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: theory and practice. ACM Transactions on Programming Languages and Systems, vol. 16, 4, pages 1117–1155, July 1994.

[13] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. Communications of the ACM, vol. 22, 2, pages 96–103, February 1979.

[14] O. Rüthing, J. Knoop, and B. Steffen. Sparse code motion. Proceedings of 27th ACM Symposium on Principles of Programming Languages, pages 170–183, 2000.

[15] B. Steffen. Property-oriented expansion. Proceedings of the 3rd Static Analysis Symposium (SAS'96), LNCS, vol. 1145, pages 22–41, Springer-Verlag, 1996.

[16] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. IEEE Transactions on Software Engineering, vol. SE-3, 1, pages 85–93, January 1977.