Fast Profile-Based Partial Redundancy Elimination

R. Nigel Horspool[1], David J. Pereira[1], and Bernhard Scholz[2]

[1] Department of Computer Science, University of Victoria
P.O. Box 3055, Victoria, BC, Canada V8W 3P6
{nigelh, djp}@cs.uvic.ca
[2] School of Information Technologies, Madsen Building F09
University of Sydney, Sydney, NSW 2006, Australia
scholz@it.usyd.edu.au

Abstract. Partial Redundancy Elimination (PRE) is a standard program optimization which removes redundant computations via Code Motion. It subsumes and generalizes the optimizations of Global Common Subexpression Elimination (GCSE) and Loop Invariant Code Motion (LICM). Recent work has generalized PRE to become Speculative PRE (SPRE), which uses estimates of execution frequencies to find the optimal places in a program to perform computations. However, the analysis performed by the compiler is computationally intensive and hence impractical for just-in-time (JIT) compilers.

This paper introduces a novel approach which abandons a guarantee of optimality in favour of simplicity and speed of analysis. This new approach, called Isothermal SPRE, achieves results which are close to optimal in practice, yet its analysis time is at least as good as current compiler techniques for code motion. It is a technique suitable for use in JIT compilers.

1 Introduction

The simplest computation performed by a program is the evaluation of an expression, say `a+b`. If the program contains a sequence of statements similar to

```
x = a+b;
x = x + c*d;
y = a+b;
```

then (assuming complications involving aliasing of variable names do not occur) the second computation of `a+b` is *fully redundant*, since neither `a` nor `b` is modified between the two computations of `a+b`. A good compiler would translate the code as though it had been written as

```
t1 = a+b;
x = t1;
x = x + c*d;
y = t1;
```

where `t1` is a new temporary variable (and which would be a good candidate for implementing as a register).

D. Lightfoot and C. Szyperski (Eds.): JMLC 2006, LNCS 4228, pp. 362–376, 2006.

The generalization to *partial redundancy* occurs if we have a program that contains multiple control flow paths and where a computation is redundant on some path(s) but not on all paths. An example fragment of program with a partially redundant occurrence of `a+b` is shown in Figure 1(a). In this (meaningless) calculation, the value of `a+b` computed in the loop condition will usually be the same value as was computed on the previous iteration. A compiler which performs *partial redundancy elimination* optimization will go through a two step process of inserting some additional computations of `a+b` to produce the intermediate version of Figure 1(b) and then eliminating those occurrences which have become fully redundant to achieve the result shown in Figure 1(c).

```
while((a+b) > sum) {
   if (sum % 10 == 0)
      a = a + 1;
   sum += b;
}
```

(a) Original while loop

```
t1 = a+b;          // inserted
while((a+b) > sum) {
   if (sum % 10 == 0) {
      a = a + 1;
      t1 = a+b;    // inserted
   }
   sum += b;
}
```

(b) After insertions of `a+b`

```
t1 = a+b;
while(t1 > sum) {  // replaced
   if (sum % 10 == 0) {
      a = a + 1;
      t1 = a+b;
   }
   sum += b;
}
```

(c) After deletion of redundancies

Fig. 1. Application of (classical) PRE to a loop

However, the classical PRE analysis is performed without any knowledge of the relative frequencies of execution of the different paths through the program. Thus PRE is required to be conservative, and will never choose to insert a computation e at a point P in the program unless it is guaranteed that the value of e will be used on every path that continues from point P. After those computations that become fully redundant due to the insertions have been removed, the number of computations of e cannot be greater than in the original program. Usually it will be smaller. Another benefit of the conservative approach is that even *unsafe* expressions can be moved. An unsafe expression is a computation which may cause a run-time exception. For example, an array reference `A[i]` in Java may cause an exception either because the array `A` has not been allocated or because the index `i` is out of range. If the expression does cause an exception

at run-time, then the optimized program will at worst raise that exception at an earlier point in the program.[1] In no case would the transformed program raise an exception that would not be raised in the original program.

The conservatism of PRE causes it to miss optimization opportunities that involve *safe expressions*, i.e. expressions that cannot raise an exception when computed. An enumeration of expressions which should be considered safe depends on the semantics of the programming language and on the platform for which the code is compiled. For example, `a+b` is normally safe in the C language. However `a/b` would be safe at a point P in a C program only if the compiler can prove that `b` is non-zero at P or if the integer division instruction on the target platform does not generate a divide-by-zero interrupt. (The PowerPC architecture provides an example of such a platform.)

The example of Figure 2 shows a loop that PRE cannot optimize, but which SPRE will. The transformation from Figure 2(a) to Figure 2(b) cannot be performed by PRE. Without knowledge of execution path frequencies, an insertion of `t1=a*a` in the *then* clause of the *if* statement might make the program slower. PRE has to consider the possibility, for example, that the *else* clause is never executed. That would introduce 10000 computations in the transformed program that would not have been performed by the original program. However, if SPRE is given profile information which shows that the *else* clause is executed more frequently than the *then* clause, then it will produce the result shown in Figure 2(b) because the total number of computations of `a*a` would be smaller.

```
for(i=0; i<10000; i++) {
   if (A[i]<0) {  //  1% frequency
      a = a+1;
   } else {       // 99% frequency
      sum += a*a;
   }
}
```

(a) Original code

```
t1 = a*a;
for(i=0; i<10000; i++) {
   if (A[i]<0) {
      a = a+1;
      t1 = a*a;
   } else {
      sum += t1;
   }
}
```

(b) Result from SPRE

Fig. 2. A loop that PRE does not optimize

The SPRE approach is restricted to safe expressions because a compiler should never introduce the possibility of an exception that was not present in the original program. However, there is no reason why a dual approach of using SPRE for safe expressions and PRE for unsafe expressions could not be adopted.

A major obstacle to adopting SPRE in a compiler is that the existing analysis algorithms are computationally intensive. For each candidate expression, the current formulations of SPRE construct a network flow problem and then finds a minimum-cut partition of the network. Given that the number of nodes V in

[1] However this may cause other difficulties for Java because it has precise exception semantics and code motion must take this into account.

the network is proportional to the size of the control flow graph of the program being analyzed, and given that standard algorithms for finding the minimum cut have $O(V^3)$ time complexity, finding the solution is costly (even if it finds an optimal solution).

In contrast, PRE uses data flow analyses which can be formulated as *bit-vector* problems. This means that PRE determines solutions for all candidate expressions simultaneously. Furthermore, the worst-case time complexity for solving the data flow equations is quadratic in the size of the control flow graph, with linear time complexity being the norm for almost all control flow graphs that occur in practice.

Although there is undoubtedly scope for implementing faster versions of SPRE, its analysis time is about two orders of magnitude slower than PRE. This may be acceptable for use in a standard optimizing compiler where much effort can be expended to achieve the fastest possible target program. However, it has restricted applicability in a just-in-time compiler where all the analysis must be performed on the fly.

In this paper, we introduce a new formulation of SPRE where the optimality of its final result is sacrificed in order to achieve a very efficient analysis. We call the new formulation *Isothermal Speculative Partial Redundancy Elimination* (ISPRE) for reasons which will be covered later.

ISPRE performs standard data flow analyses which can, again, be implemented as bit vector problems. Furthermore, these analyses are simpler than those performed by PRE. Since ISPRE uses program profile information, it will usually produce results which are better than PRE, though they would usually be a bit worse than those of SPRE. Experimental results included in this paper confirm this expectation. These same results also demonstrate the speed of the implementation of ISPRE, comparing it to the speed of PRE and SPRE.

2 Background and Related Work

Common Subexpression Elimination (CSE) has existed as a standard compiler optimization since the early Fortran compilers [1,2]. The first formulation of Global Common Sub-expression Elimination (GCSE), via an available expressions analysis, is described in [4].

The generalization from GCSE to partial redundancy elimination (i.e. PRE) was first described by Morel and Renvoise [10]. They later extended their analyses to the interprocedural case [11].

There have been several developments to PRE that have both improved its implementation in compilers and the quality of the transformed program. These include the reformulation of PRE as a set of *uni*directional analyses [16], and the establishment of critical-edge splitting [5] as a crucial component in increasing the power of PRE. Finally, Lazy Code Motion (LCM) [8,9] is a PRE formulation which is optimal with respect to lifetimes of the temporary variables introduced to hold expression values. Since these temporaries would often be implemented as registers, LCM has the smallest impact on register pressure. LCM is presently the algorithm of choice in modern optimizing compilers.

The idea of using profiling information to improve the expected performance of PRE is due to Horspool and Ho [6]. Subsequent work has shown that the problem can be mapped to a form of network flow problem known as Stone's Problem [15]. The name *Speculative Partial Redundancy Elimination* (SPRE) has been applied to the problem, and algorithms for finding optimal solutions have been presented [3,13]. These optimal solutions minimize the expected number of computations of the candidate expressions, based on the execution frequencies of the different paths through the program obtained from the program profile. A secondary, but still very important, concern is in minimizing register pressure. Xue and Cai [17] have developed a variation on SPRE which minimizes the lifetimes of the temporary variables while still maintaining optimality.

3 Notation and Terminology

In this paper, we present an *intraprocedural* analysis algorithm. That is, each procedure of a program will be transformed by ISPRE independently of the other procedures. Extension of ISPRE to the interprocedural case should be straightforward and is left for future work [12, sec. 19.2].

We assume that each procedure is translated into an intermediate representation (IR) by the compiler and that machine independent optimizations such as ISPRE are applied to the IR form. For the purposes of this paper, we assume that IR statements have these forms only:

```
L:                      // a label
   x = c                // assign a constant
   x = y                // assign a variable
   x = y op z           // assign a simple expression
   goto L               // unconditional branch)
   if (a op b) goto L   // conditional branch
```

where *op* represents a simple operation like addition or multiplication, or like less-than when used in a conditional branch. The precise details are unimportant when describing ISPRE.

The sequence of IR instructions for a procedure is partitioned into basic blocks. A basic block is a maximal sequence of instructions through which the only flow of control is sequential. This implies that the first instruction in a basic block must be either a labelled instruction or an instruction which follows a conditional branch. It also implies that the last instruction in a basic block is a branch, either conditional or unconditional.

The basic blocks of a procedure form a control flow graph (CFG). A CFG is a directed graph with the node set N, where each node $b \in N$ represents a basic block. The CFG has an edge set $E \subseteq N \times N$, and two distinguished nodes: $s \in N$, a unique start (or entry) node, and $f \in N$, which is a unique final (or exit) node. Edges $(u, v) \in E$ represent the branching structure of the CFG. The functions $succs(u) = \{v|(u, v) \in E\}$ and $preds(u) = \{v|(v, u) \in E\}$ represent the *immediate successors* and *immediate predecessors* of node u.

4 Isothermal Speculative Partial Redundancy Elimination

Isothermal SPRE (ISPRE) is a complete reformulation of SPRE. It is, by design, an approximate technique for performing code motion using information obtained from program profiles. The major part of the performance gains achieved by ISPRE are made in one transformation pass over the flowgraph. Further improvements can be made with additional passes, but a law of diminishing returns apply. We distinguish two versions of ISPRE with the names *Single Pass ISPRE* and *Multipass ISPRE*, according to whether just one pass or several transformation passes over the program are performed. In the following, and in our experiments, we describe single pass ISPRE.

ISPRE initially uses profile information to divide a CFG G into two subgraphs — a **hot** region G_{hot} consisting of the nodes and the edges executed more frequently than a given threshold frequency Θ, and a **cold** region G_{cold} consisting of the remaining nodes and edges. A pictorial representation of a division of a CFG into its hot and cold regions is shown in Figure 3. In this picture, the black region represents G_{hot} and the grey region represents G_{cold}.

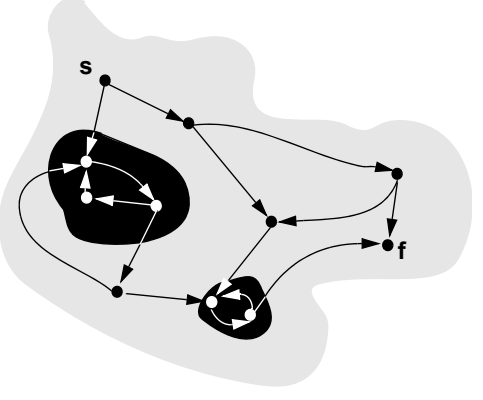


Fig. 3. Dividing a CFG into Hot and Cold Subgraphs

The example illustrates that either subgraph may consist of disconnected components. As shown here, the components of G_{hot} would usually correspond to loops. However, that is not necessarily the case because an isolated node with several predecessors and several successors could be hot while its immediate neighbours are all cold. It is also possible for a cold component to consist of just a single edge, to provide a second example of a degenerate case.

More formally,

$$G_{hot} = \langle N_H, E_H \rangle$$
$$G_{cold} = \langle N_C, E_C \rangle$$

where

$$N_H = \{u \mid u \in N \wedge \text{freq}(u) > \Theta\}$$
$$E_H = \{e \mid e \in E \wedge \text{freq}(e) > \Theta\}$$
$$N_C = N - N_H$$
$$E_C = E - E_H$$

Given that division into hot and cold subgraphs, we define the *Ingress* edges as

$$\text{Ingress} = \{(u, v) \mid u \in N_C \wedge v \in N_H\}$$

That is, the *Ingress* set consists of those edges which transfer control from a cold node to a hot node. Note that Ingress $\subseteq E_C$ must hold (because every edge adjacent to a cold node must be cold).

ISPRE operates by inserting expressions on edges in the *Ingress* set, and thus making some expressions in hot nodes become fully redundant. If those fully redundant expressions are then replaced by references to temporaries which hold saved values of the expressions, we have achieved code motion from G_{hot} to G_{cold}.

The code motion is driven by the results of two analyses:

removability, which deduces instances of computations in the hot region that can be deleted; and

necessity, which deduces edges in the *Ingress* set where computations must be inserted, so as to ensure the correctness of the deletions determined by removability analysis.

Both *removability* and *necessity* are formulated as analyses that fall within the monotone dataflow framework of Kam and Ullman [7]. This implies that they can be implemented as unidirectional analyses using bit-vector representations of sets of expressions. That is, we can efficiently compute removability and necessity for all candidate expressions simultaneously.

4.1 Removability Analysis

An expression e is a possible candidate for removal if (1) e is a *safe* expression and (2) there is an *upwards exposed* use of e in at least one node $u \in N_H$. As mentioned previously, an expression is safe at a particular program point if computing it at that point cannot generate an exception. Exactly which expressions can be considered safe is both language and platform dependent, and is beyond the scope of this paper. An expression a op b, for some operator *op* is upwards exposed in a basic block if it is not preceded in that basic block by any assignments to a or b (or, in the terminology of dataflow analysis, is not preceded by any statements which kill e).

Our removability analysis is based on the assumption that *every* candidate expression is available on *every* edge in the *Ingress* set. An expression e is

available at a point P if it has been computed on every edge leading to P without being subsequently killed (i.e. no operand of e has been modified). The necessity analysis will ensure that our assumption is satisfied.

Given the assumption, removability analysis just becomes available expressions analysis [1]. A candidate expression e is removable from node u if and only if u contains an upwards exposed use of e and if e is available on entry to u. The dataflow equations, with modifications to incorporate our assumption, can now be stated.

First, the following sets are computed for each basic block by processing the intermediate code in the block.

$$\text{XUSES}(b) \stackrel{\text{def}}{=} \{\ e \mid \text{expression } e \text{ occurs in } b \text{ and is not preceded by any redefinitions of operands of } e\ \}$$

$$\text{GEN}(b) \stackrel{\text{def}}{=} \{\ e \mid \text{expression } e \text{ occurs in } b \text{ and is not followed by any redefinitions of operands of } e\ \}$$

$$\text{KILL}(b) \stackrel{\text{def}}{=} \{\ e \mid \text{block } b \text{ contains a statement which may redefine an operand of } e\ \}$$

Then the following dataflow equations are solved by finding a least fixed point solution.

$$\begin{aligned}
&\forall b \in N: \\
&\quad \text{AVOUT}(b) = (\text{AVIN}(b) - \text{KILL}(b)) \cup \text{GEN}(b) \\
&\quad \text{AVIN}(b) = \bigcap_{p \in \mathit{preds}(b)} \begin{cases} \text{Candidates} & \text{if } (p,b) \in \text{Ingress} \\ \text{AVOUT}(p) & \text{otherwise} \end{cases} \\
&\forall b \in N_H: \\
&\quad \text{Removable}(b) = \text{AVIN}(b) \cap \text{XUSES}(b)
\end{aligned}$$

In the above equations, *Candidates* represents the set of all candidate expressions. The solutions to the equations for *Removable* indicate which upwards exposed uses of expressions can be removed from each node in G_{hot}.

Note that the equations for *AVIN* and *AVOUT* are solved for all blocks in the CFG, not just in the hot region. This is because the expression availability within the cold region is useful in completing the necessity analysis.

4.2 Necessity Analysis

The solutions for the *Removable* sets assume that computations of all candidate expressions are available on the *Ingress* edges. That assumption could be satisfied by inserting the computations on all those edges. However, that would be a suboptimal solution because not all the insertions would be needed. There are two reasons why inserting an expression e on an edge (u, v) in the *Ingress* set may be unnecessary.

1. *It is useless:* the expression may not reach any exposed use of e in the hot region which has been deemed to be removable, or
2. *It is redundant:* the expression e may already be available at the end of block u.

The dataflow equations for *NEEDIN* and *NEEDOUT* determine whether insertions of the candidate expressions would be useless or not. When they have been solved, their results are used to construct the *Insert* sets. The calculation of these sets takes into account whether the insertion would be redundant or not.

$$
\begin{aligned}
&\forall b \in N_H : \\
&\quad \text{NEEDIN}(b) \;\; = \;\; (\text{NEEDOUT}(b) - \text{GEN}(b)) \cup \text{Removable}(b) \\
&\quad \text{NEEDOUT}(b) \;\; = \bigcup_{s \in succs(b)} \text{NEEDIN}(s) \\
&\forall (u, v) \in \text{Ingress} : \\
&\quad \text{Insert}(u, v) \;\; = \;\; \text{NEEDIN}(v) \; - \; \text{AVOUT}(u)
\end{aligned}
$$

4.3 An ISPRE Example

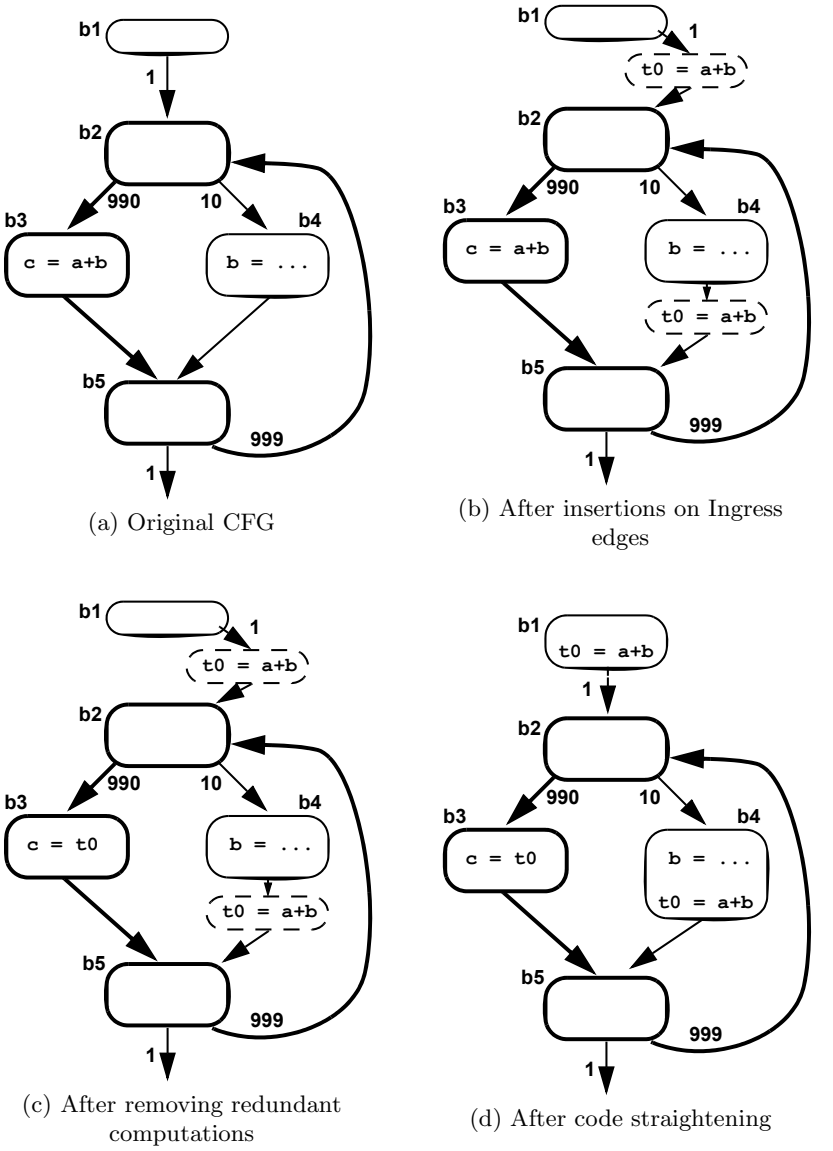
An example CFG to be optimized by Isothermal SPRE is shown in Figure 4(a). For the example, we use a threshold value Θ of 900. Thus the hot region G_{hot} consists of blocks b2, b3, and b4, and edges b2→b3, b3→b5, and b5→b2, while the *Ingress* set consists of edges b1→b2 and b4→b5. ISPRE assumes that the result of the computation of a+b is available in temporary variable t0 on edges b1→b2 and b4→b5. Although ISPRE does not actually transform the CFG at this stage, the removability analysis assumes the existence of the extra computations on the *Ingress* edges, as shown in Figure 4(b).

Removability analysis then finds that the computation of a+b in block b3 would be redundant and can be replaced with t0. The result is shown in Figure 4(c).

Finally, we can clean up the CFG. We should, whenever possible, avoid inserting new code on edges because that implies the creation of new basic blocks and that, in turn, may cause the compiler to generate more branch instructions. In our example, the code to be inserted in edge b1→b2 can be moved to the bottom of block b1; similarly the code to be inserted on b4→b5 can be moved to the bottom of node b4. The result is shown in Figure 4(d).

4.4 Multipass ISPRE

The analysis described above partitions the CFG into two regions: a hot region and a cold regions. Once the code motions implied by that partitioning have been completed, there is no reason why the same process should not be repeated with a smaller threshold value. The smaller value for Θ will select a larger subgraph for G_{hot}, one that contains the previous G_{hot} region. The ISPRE transformations



(a) Original CFG

(b) After insertions on Ingress edges

(c) After removing redundant computations

(d) After code straightening

Fig. 4. Example of ISPRE

for the second pass will again move computations from the hot region to the cold region, improving the overall performance of the program.

We propose, but we do not yet have experimental justification for, the halving of the threshold value on each pass until the expected performance gains become unimportant.

The structure of multipass ISPRE would be as follows:

$\Theta = 0.90$ * *maximum frequency*
repeat
    perform ISPRE on the CFG
    $\Theta = \Theta$ / 2
until Θ *is small*

Much of the analysis for the second and subsequent iterations of ISPRE can reuse the results of the dataflow analyses from the previous iteration, just updating the solutions.

5 Experimental Results

For our experiments, we used the `gcc` compiler (version 4.1.0) when applied to various programs from the SPEC CPU2000 benchmark suite.[14] For these preliminary experiments, the `test` subsection of the suite has been used. (We plan to use the full `ref` suite in future experiments.)

Table 1 compares the effects of different PRE optimization algorithms. The column labelled *LCM* shows the execution times when compiled by `gcc` using its PRE algorithm, which is an implementation of LCM [8]. The two columns labelled *SPRE* show the (a) times when an optimal speculative PRE algorithm is used by the compiler, and (b) those times as compared to LCM. (The implementation of SPRE follows that given in [13].) Finally, the two columns labelled *ISPRE* show the results achieved by the method described in this paper. In the comparisons with LCM, a negative percentage value shows a smaller time than LCM while a positive value shows the converse.

For all the ISPRE experiments reported here, a single partitioning of the CFG was performed. That is, the multipass ISPRE was not tested in these experiments. The threshold parameter Θ was always set to be 90% of the highest node frequency in the CFG. We observe that even a single pass of ISPRE produces execution times which are very similar to those for SPRE. Taken over the set of twelve benchmarks, ISPRE produces slightly better results than SPRE.

One might ask how it is possible that an approximate technique like ISPRE could produce better timings than SPRE which is provably optimal. A partial answer is that SPRE is optimal only with respect to the expected number of evaluations of the candidate expressions when the program is run. The dynamic number of evaluations is not perfectly correlated with execution time because of interactions between PRE and other compiler optimizations, and there are interactions with the code generation phase of the compiler. We suspect that the dominant interaction effect is register pressure. The version of SPRE implemented for these experiments does not take register pressure into account. On the other hand, LCM keeps the lifetimes of the introduced temporary variables to a minimum and is therefore minimizing its effect on register pressure. We also believe that ISPRE naturally chooses insertion points for new computations at places which do not have a severe impact on register pressure.

Table 1. Execution Times of Optimized Programs

Benchmark	LCM time (seconds)	SPRE time (seconds)	SPRE relative to LCM	ISPRE time (seconds)	ISPRE relative to LCM
164.gzip	1.183	1.266	7.02%	1.190	0.59%
181.mcf	0.118	0.119	0.85%	0.115	-2.54%
197.parser	1.418	1.416	-0.14%	1.305	-7.97%
253.perlbmk	4.364	4.363	-0.02%	4.373	0.21%
255.vortex	3.376	3.203	-5.12%	3.194	-5.39%
300.twolf	0.145	0.144	-0.69%	0.146	0.69%
173.applu	0.149	0.150	0.67%	0.150	0.67%
178.galgel	4.930	4.750	-3.65%	4.770	-3.25%
183.equake	0.553	0.532	-3.80%	0.532	-3.80%
188.ammp	5.688	5.281	-7.16%	5.213	-8.35%
189.lucas	7.102	7.094	-0.11%	7.202	1.41%
301.apsi	4.128	4.150	0.53%	4.164	0.87%
Summary	33.154	32.468	-2.07%	32.354	-2.41%

Table 2. Compilation Times

Benchmark	LCM time (seconds)	SPRE time (seconds)	SPRE relative to LCM	ISPRE time (seconds)	ISPRE relative to LCM
164.gzip	2.300	2.460	6.96%	2.330	1.30%
181.mcf	1.200	1.240	3.33%	1.190	-0.83%
197.parser	8.370	9.150	9.32%	8.370	0.00%
253.perlbmk	28.630	33.430	16.77%	28.600	-0.10%
255.vortex	23.150	24.200	4.54%	23.270	0.52%
300.twolf	12.890	14.660	13.73%	12.840	-0.39%
173.applu	2.930	2.920	-0.34%	2.930	0.00%
178.galgel	10.580	10.760	1.70%	10.550	-0.28%
183.equake	1.110	1.330	19.82%	1.050	-5.41%
188.ammp	6.600	7.280	10.30%	6.730	1.97%
189.lucas	1.900	1.940	2.11%	1.940	2.11%
301.apsi	6.230	6.190	-0.64%	6.200	-0.48%
Summary	105.890	115.560	9.13%	106.000	0.10%

One of the claims made in this paper is that the analysis performed by ISPRE is much faster than SPRE and similar to that of the standard implementations of PRE. This claim is supported by the timings shown in Table 2. These timings show the total compilation times for the benchmark programs. In the environment of JIT compilation, all the initial phases of a compiler (lexical analysis, syntactic analysis, semantic analysis and IR code generation) would have been performed before the program begins execution. Thus the time spent on performing code optimization becomes much more significant.

Table 3. Compilation Times for PRE Optimization Phase Only

Benchmark	LCM time (seconds)	SPRE time (seconds)	SPRE relative to LCM	ISPRE time (seconds)	ISPRE relative to LCM
`164.gzip`	0.010	0.140	14.00	0.010	1.00
`181.mcf`	0.020	0.300	15.00	0.010	0.50
`197.parser`	0.040	0.680	17.00	0.040	1.00
`253.perlbmk`	0.230	4.760	20.70	0.340	1.48
`255.vortex`	0.130	1.040	8.00	0.210	1.62
`300.twolf`	0.060	1.740	29.00	0.130	2.17
`173.applu`	0.030	0.050	1.67	ϵ	-
`178.galgel`	0.270	0.270	1.00	0.310	1.15
`183.equake`	ϵ	0.160	-	0.200	-
`188.ammp`	0.060	0.590	9.83	0.020	0.33
`189.lucas`	0.040	0.020	0.50	0.030	0.75
`301.apsi`	0.030	0.040	1.33	0.070	2.33
Summary	0.920	9.630	10.47	1.170	1.27

To further reveal the difference in analysis times between the three different PRE implementations, Table 3 shows just the times spent in performing the PRE optimization during compilation. The columns which show relative performance are displayed as ratios (not as percentage differences) because most ratios are large numbers. The large ratios for SPRE, e.g. 29 for the `300.twolf` benchmark, occur with the benchmarks which contain large CFGs and are a symptom of the cubic time computational complexity of the SPRE analysis. In a couple of cases, the benchmark programs are small and the measured times are negligible. In these cases, the times are shown as ϵ and the ratios between the times are left blank.

The case of `183.equake` shows a negligible compilation time with LCM but a much larger compilation time with ISPRE – even larger than the compilation time with SPRE. It is currently under investigation.

6 Conclusions and Further Work

This paper has introduced a new way to implement partial redundancy elimination in a compiler. Unlike other PRE implementations, there is no claim of optimality for any cost metric (not lifetimes of saved expression values, not expected number of expression computations). However, we do claim that the method is simple to implement, is fast, and produces results that are close to those produced by the optimal SPRE algorithm. We claim the the technique is fast enough to be used by JIT compilers.

We have much further work to do, including: evaluation of multipass ISPRE, selection of threshold values, analysis of register pressure and lifetime issues, incorporating unsafe expressions into the framework, and optimizing in the presence of Java or C# exception handling.

We believe that ISPRE has the potential to become the code motion optimization algorithm of choice in future compilers, especially just-in-time compilers.

Acknowledgements

The authors gratefully acknowledge funding for this research received from the IBM Center for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. M. Breuer. Generation of optimal code for expressions via factorization. *CACM*, 12(6):333–340, 1969.
3. Q. Cai and J. Xue. Optimal and efficient speculation-based partial redundancy elimination. In *CGO '03: Proceedings of the ACM/IEEE 2003 Symposium on Code Generation and Optimization*, pages 91–104, 2003.
4. J. Cocke. Global common subexpression elimination. In *Proceedings of a symposium on compiler optimization*, pages 20–24, 1970.
5. D. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1991.
6. R. Horspool and H. Ho. Partial redundancy elimination driven by a cost-benefit analysis. In *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering*, pages 111–118, 1997.
7. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.
8. J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming Language Design and Implementation*, pages 224–234, 1992.
9. J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion: theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.
10. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *CACM*, 22(2):96–103, 1979.
11. E. Morel and C. Renvoise. Interprocedural elimination of partial redundancies. In S. Muchnik and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 160–188. Prentice Hall, June 1981.
12. S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
13. B. Scholz, N. Horspool, and J. Knoop. Optimizing for space and time usage with speculative partial redundancy elimination. In *LCTES '04: Proceedings of ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 221–230, 2004.
14. Standard Performance Evaluation Corporation. Cpu2000. `http://www.spec.org`, 2006.

15. Harold Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.
16. M. Wolfe. Partial redundancy elimination is not bidirectional. *SIGPLAN Notices*, 34(6):43–46, 1999.
17. Jingling Xue and Qiong Cai. A lifetime optimal algorithm for speculative pre. *ACM Transactions on Architecture and Code Generation*, page (to appear), 2006.