Profile-Guided Partial Redundancy Elimination Using Control Speculation: a Lifetime Optimal Algorithm and an Experimental Evaluation

Jingling Xue and Qiong Cai
Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
{jxue,qiongc}cse.unsw.edu.au UNSW-CSE-TR-0420

25 July 2004

THE UNIVERSITY OF
NEW SOUTH WALES

Abstract

A lifetime optimal algorithm, called MC-PRE, is presented for the first time that performs partial redundancy elimination (PRE) by combining code motion and control speculation based on an edge profile. An edge profile provides an approximation of the actual edge frequencies with the predicted edge frequencies. MC-PRE is developed so that the optimality results it achieves are reasoned about with respect to a given edge profile. MC-PRE is computationally optimal since the total number of dynamic computations for an expression in the transformed code is minimized. If the predicted frequencies of all flow edges are nonzero, MC-PRE is also lifetime optimal since the lifetimes of introduced temporaries in the transformed code are also minimized. Otherwise (if some flow edges are zero-weighted), MC-PRE yields a practical transformation (as validated by extensive experiments) from the perspective that the predicted zero frequencies are (or should be) interpreted as the least frequently rather than never executed at all. The computational and lifetime optimality results are rigorously proved. This algorithm works on CFGs with standard basic blocks and is conceptually simple. First, it performs two standard bit-vector data-flow analyses, availability and partial anticipability, to transform a given CFG to an *s-t* flow network. It then relies on a min-cut solver to find a unique minimum cut on the flow network. Finally, it performs a third standard live range analysis to avoid making unnecessary code insertions and replacements for isolated computations. We have implemented the MC-PRE algorithm in `gcc 3.4` and evaluated its performance against Knoop, Rüthing and Steffen's profile-independent LCM, the built-in PRE pass at `gcc`'s O2 optimization level and above. We report and analyze our experimental results for all 22 C, C++ and FORTRAN 77 benchmarks from SPECcpu2000 on two computer architectures, Intel Xeon and UltraSPARC-III. Our results show that MC-PRE is both effective (in terms of the extra redundancies eliminated and performance improvements achieved) and practical (in terms of the relatively small compilation and space overheads introduced).
Keywords: Partial redundancy elimination, code motion, common subexpressions, computational optimality, lifetime optimality, control speculation, data flow analysis

1. INTRODUCTION

Partial redundancy elimination (PRE) is a powerful and widely used optimization technique aimed at removing computations that are redundant due to recomputing previously computed values [Morel and Renvoise 1979]. As a code motion transformation, the technique inserts and deletes computations in a control flow graph (CFG) in order to reduce the total number of remaining such computations in the transformed code. Such a reduction (if any) is achieved by storing the values of computations for later reuse in temporaries that are initialized correctly at appropriate points in a CFG. PRE is attractive because by targeting computations that are redundant only along some paths in a CFG, it encompasses global common subexpression elimination (GCSE), loop-invariant code motion (LICM), and more. As a result, PRE is an important component in global optimizers.

Traditionally, PRE has been implemented as a profile-independent optimization. For example, `gcc 3.4` [GNU Software 2003] consists of a pass for performing PRE, which is based on the well-known algorithm called lazy code motion (LCM) [Knoop et al. 1994]. As another example, `Open64` [Open64] conducts PRE in SSA form (Static Single Assignments [Cytron et al. 1991]) using the well-known SSAPRE algorithm described in [Kennedy et al. 1999]. These classic PRE algorithms guarantee both computationally and lifetime optimal results: the number of computations cannot be reduced any further by safe code motion [Kennedy 1972] and the lifetimes of introduced temporaries are minimized. Under such a safety constraint, they insert an expression π at a point p in a CFG only if all paths emanating from p must evaluate π before any operands of π are redefined. The expression π is known as *fully anticipatable* at p. In other words, they remove partial redundancies along some paths but never introduce additional computations along any path that did not contain them in the original code. These safety-based formulations have two consequences. First, the transformed code cannot cause exceptions that do not exist in the original code. If evaluating π can throw an exception, the exception — which is inevitable — would have occurred a bit earlier in the transformed code than in the original code. Second, due to not exploiting profiling information, they guarantee (albeit conservatively) that the transformed code cannot evaluate π more times than before in *any* program execution under *any* input.

In real programs, some points (nodes or edges) in a CFG are executed more frequently than others. If we have their execution frequencies available and if we know that an expression cannot cause an exception, we can perform code motion transformations missed by the classic PRE algorithms. The central idea is to use *control speculation* (i.e., unconditional execution of an expression that is otherwise executed conditionally) to enable the removal of partial redundancies along some more frequently executed paths at the expense of introducing additional computations along some less frequently executed paths. Such a speculative PRE may potentially insert computations on paths that did not execute them in the original program. As a result, the safety criterion [Kennedy 1972] that is enforced in the classic (profile-independent) PRE is relaxed.

Several researchers have developed profile-guided algorithms to perform the speculative PRE and discussed its potential benefits [Bodik et al. 1998; Bodik 1999; Gupta et al. 1997; Horspool and Ho 1997]. This paper, however, is the first to

present an algorithm for solving the code-motion-based speculative PRE problem that can achieve both computational and lifetime optimality simultaneously. We will review and compare with these and other related earlier research efforts in detail in Section 2. Presently, many existing compiler frameworks have incorporated and used profiling information to support control speculation (and data speculation [Lin et al. 2003]). As increasingly more aggressive profile-guided optimizations are being employed, the profiling overhead can be better amortized (or shared). However, profile-guided speculative PRE algorithms are yet to be incorporated into existing compiler frameworks. As mentioned above, `gcc` supports LCM (which is non-speculative) while `Open64` embraces SSAPRE (which, combined with the extension described in [Lo et al. 1998], can promote register reuse by performing speculative loads and stores).

This work makes the case that (a) a conceptually simple algorithm for performing the speculative PRE can be developed using the standard bit-vector data-flow framework that is available in almost any compiler system, (b) both computational optimality (in terms of the dynamic number of eliminated computations) and lifetime optimality can be achieved from an edge profile by finding a unique s-t minimum cut for each PRE problem, (c) such optimal results can be obtained practically (as validated using the SPECcpu2000 benchmark suite) at small extra compilation overheads compared to an optimal profile-independent PRE algorithm, and finally, (d) such an optimal profile-guided algorithm can generally eliminate more redundancies in dynamic terms and achieve performance improvements than an optimal profile-independent PRE algorithm (as also validated using SPECcpu2000).

Specifically, this paper makes the following contributions:

Computational and Lifetime Optimality. We present a new algorithm for performing the speculative PRE (by combining code motion and control speculation) from an edge profile. This algorithm is called MC-PRE (the "MC" stands for Min-Cut) and uses edge insertions for code motion. MC-PRE is conceptually simple, proceeding essentially in two phases on a given CFG:

(1) First, we solve the two standard data-flow problems, availability and partial anticipability, to remove from a given CFG the so-called non-essential edges (and consequently, all non-essential nodes) to obtain a reduced graph (i.e., a sub-graph of the CFG). We will give the data-flow equations for standard basic blocks so that they are directly and efficiently implementable.

(2) Second, we find a unique s-t minimum cut in a flow network (called *essential flow graph (EFG)*) that is derived from the reduced graph obtained in the first step. The PRE transformation derived from this unique s-t minimum cut is always computationally optimal with respect to the given edge profile. But it may not yet be lifetime optimal. In order to avoid making unnecessary code motion for isolated computations [Knoop et al. 1994], a third standard live variable analysis is performed for the single temporary introduced for each PRE problem. The final PRE transformation is provably lifetime optimal with respect to the given edge profile if the predicted frequencies of all control flow edges are nonzero. Otherwise, this transformation represents a practically desirable solution (as validated by extensive experiments) from the perspective

that the predicted zero frequencies are (or should be) interpreted as the least frequently rather than never executed at all (as discussed in Section 4.2).

Implementation. We have implemented MC-PRE in `gcc 3.4`. We make use of the bit-vector routines in `gcc` to perform our data-flow analysis passes. This is also the same framework used by the LCM algorithm implemented in `gcc`. In performing the min-cut part of our algorithm, we use Goldberg's *push-relabel* HIPR algorithm [GoldBerg 2003] and his implementation, which is one of the best-engineered and fastest implementations available [Chekuri et al. 1997].

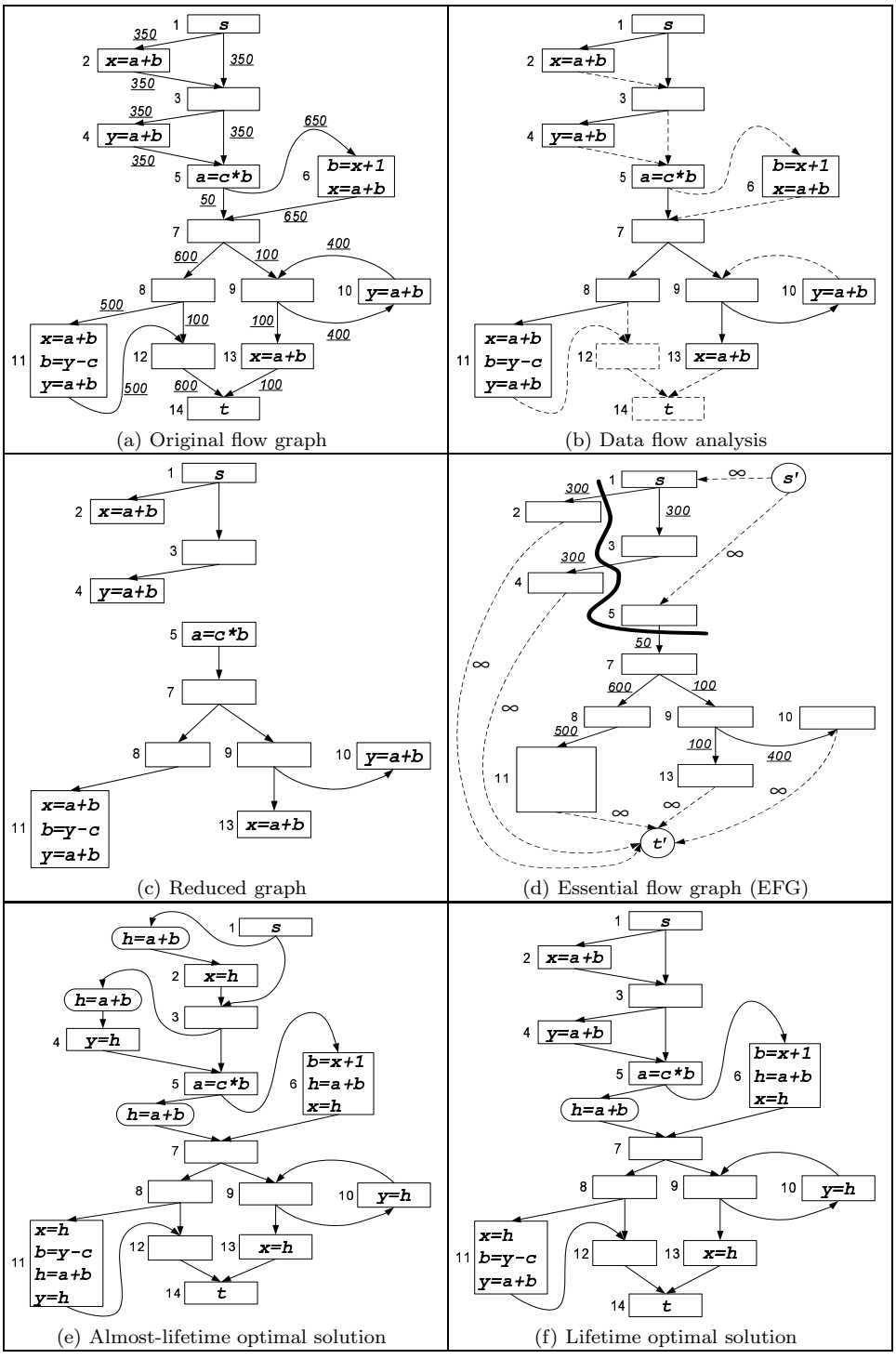
Redundancy Elimination. We compare MC-PRE and LCM in terms of the dynamic number of redundant computations removed for 22 SPECcpu2000 benchmarks on two computer architectures. The benchmark suite used in our experiments include the 15 C programs, the 1 C++ program, and the 6 FORTRAN 77 programs from SPECcpu2000. The two computer platforms used are architecturally different: Intel Xeon and UltraSPARC-III. In the case of Xeon, MC-PRE eliminates between 26.92% and 170.67% (an average of 76.34%) more redundancies than LCM for SPECint2000 and between 0.11% and 361.69% (an average of 10.29%) than LCM for SPECfp2000. In the case of UltraSPARC-III, the increases in eliminated redundancies range from 15.35% to 82.15% with an average of 38.36% for SPECint2000 and from 0.46% to 128.01% with an average of 7.09% for SPECfp2000.

Performance Speedups. Our experiments compare MC-PRE and LCM in terms of the execution times of the 22 benchmark programs on the two computer architectures. MC-PRE achieves better performance results in 19 out of 22 benchmarks on Intel Xeon and 20 out of 22 on UltraSPARC-III. The performance speedups on Xeon range from -0.71% to 4.62% with an average of 1.03%. The speedups on UltraSPARC-III range from -0.61% to 5.08% with an average of 1.44%.

Low Compilation Overhead. By finding a minimum cut efficiently from an EFG, which is derived from a CFG but significantly smaller in general, MC-PRE incurs only small extra compilation overheads relative to LCM. By replacing LCM with MC-PRE, `gcc` has only moderate increases in compile times across 22 SPECcpu2000 benchmark programs on the two computer architectures used. These increases range from -2.52% to 10.51% with an average of 4.29% on Xeon and from -3.79% to 16.29% with an average of 6.98% on UltraSPARC-III. We will provide a detailed analysis for the low computational cost of our algorithm in Section 6.4.2.

Low Space Overhead. A PRE algorithm works by inserting and deleting computations in a CFG. As a result, it may cause the static code sizes to be increased or decreased. In comparison with LCM, MC-PRE has resulted in small increases in code sizes across all the 22 benchmarks. The overall increases for all benchmarks are 2.37% on Xeon and 1.65% on UltraSPARC-III, respectively.

In summary, this paper introduces a new algorithm for performing the speculative PRE. To the best of our knowledge, this is the first lifetime optimal algorithm of its kind and the first implementation and experimental evaluation of such an algorithm in a production-quality compiler. Our work suggests that this algorithm can be employed as a global PRE pass in a profile-driven compiler: it is both effective (in terms of extra redundancies eliminated and performance improvements achieved) and practical (in terms of small compilation and space overheads incurred).



(a) Original flow graph (b) Data flow analysis

(c) Reduced graph (d) Essential flow graph (EFG)

(e) Almost-lifetime optimal solution (f) Lifetime optimal solution

Fig. 1. A running example illustrating our optimal algorithm.

We illustrate the basic idea of our algorithm using an example given in Figure 1 while highlighting its simplicity and optimality. In Section 4, we shall revisit this example when we describe our algorithm in greater detail. In the CFG shown in Figure 1(a), s represents the unique entry node and t the unique exit node. The underlined numbers alongside the flow edges represent the frequencies of execution. Our PRE candidate expression is $a+b$. By convention, all statements that modify either a or b are depicted explicitly and the other statements omitted.

It is assumed that all the local redundancies have been removed using standard techniques such as local common subexpression elimination (LCSE). Thus, we distinguish two kinds of candidate computations in our global redundancy elimination:

$$\begin{array}{rcl} UB & = & \{2, 4, 10, 11, 13\} \\ DB & = & \{6, 11\} \end{array} \tag{1}$$

where UB is the set of blocks where $a+b$ is upwards exposed and DB the set of blocks where $a+b$ is downwards exposed and preceded by assignments to a or b.

The candidate computation in a block from DB needs some special treatment. Consider block 11. Due to the presence of assignment $b = x+1$, the insertion point for the computation $a+b$ in that block cannot possibly be on a flow edge. To minimize the lifetimes of introduced temporaries, the insertion point in this case is clearly the point just before the computation $a+b$. In fact, if we split block 11 into two blocks with the top part containing the first two statements, the bottom part containing the last statement, and finally, add an imaginary flow edge in between, we will be able to transform all such insertion points into flow edges. However, we have deliberately chosen not to perform such a graph transformation since we want to give an algorithm that works directly on standard basic blocks.

Given the CFG shown in Figure 1(a), our mission consists of finding a set of insertion points, inserting $h = a+b$ at these points, where h is a new temporary, and replacing some candidate computations of $a+b$ in the original CFG by h so that both computational and lifetime optimality criteria are satisfied. In such an optimal solution, the dynamic number of expression evaluations for $a+b$ is kept to an absolute minimum with respect to the given edge profile, and so is the lifetime range of the introduced temporary h in every execution path.

The (unique) lifetime optimal transformation, $\mathcal{LO}$, found by MC-PRE is:

$$\begin{array}{rcl} \textit{U-Ins}_{\mathcal{LO}} & = & \{(5, 7)\} \\ \textit{U-Rep}_{\mathcal{LO}} & = & \{10, 11, 13\} \\ \textit{D-Ins}_{\mathcal{LO}} & = & \{6\} \\ \textit{D-Rep}_{\mathcal{LO}} & = & \{6\} \end{array} \tag{2}$$

The first two sets prescribe the insertions and replacements for some upwards exposed computations in UB, respectively. The last two sets specify the insertions and replacements for some downwards exposed computations in DB, respectively. Performing this code motion on the graph in Figure 1(a) leads to the transformed code shown in Figure 1(f). The dynamic number of computations for $a+b$ has been reduced optimally from 2850 to 1900.

For this same example, the profile-independent LCM would have produced the code in Figure 2, requiring a total of 2450 computations for the same expression.

Let us examine our algorithm to see how the optimal transformation $\mathcal{LO}$ is derived. Invoked on this example, MC-PRE proceeds in the following two phases:

—First, we perform two standard data-flow analyses — the forward availability and the backward partial anticipability — on the CFG to identify all non-essential edges. Figure 1(b) duplicates the CFG given in Figure 1(a) with the non-essential edges (and nodes) being depicted in dashes. A node becomes non-essential when all its incident edges are non-essential. A non-essential edge cannot be an insertion point since either the value of $a+b$ is always available on it or an insertion of $h = a + b$ on the edge would never cause the value in h to be reused or both. By removing these non-essential edges and nodes from the given CFG, we obtain the reduced graph as shown in Figure 1(c). This reduced graph is then transformed into the so-called essential flow graph (EFG), which is an s-t flow network given in Figure 1(d). In general, as we will illustrate in Figure 12, such a transformation may cause some nodes in the reduced graph to be split (conceptually).

—Second, we apply a min-cut algorithm to the EFG to obtain the unique minimum cut as illustrated in Figure 1(d). Let C_1 denote this minimum cut:

$$C_1 \ = \ \{(1,2),(3,4),(5,7)\}$$

At this stage, the following transformation, $\mathcal{CO}_1$, is computationally optimal:

$$\begin{array}{rcl} \textit{U-Ins}_{\mathcal{CO}_1} & = & C_1 \ = \ \{(1,2),(3,4),(5,7)\} \\ \textit{U-Rep}_{\mathcal{CO}_1} & = & \textit{UB} \ = \ \{2,4,10,11,13\} \\ \textit{D-Ins}_{\mathcal{CO}_1} & = & \textit{DB} \ = \ \{6,11\} \\ \textit{D-Rep}_{\mathcal{CO}_1} & = & \textit{DB} \ = \ \{6,11\} \end{array} \tag{3}$$

In fact, this solution is our analogue of the so-called almost-lifetime optimal solution coined in [Knoop et al. 1994]. For illustration purposes, Figure 1(e) depicts the resulting transformed code, which is not actually generated. In this transformed program, there are three isolated computations. The computation $a + b$ in block 2 is isolated since the definition $h = a + b$ inserted on the edge $(1, 2)$ is used only in block 2 and becomes dead at its exit. The computation $a + b$ in block 4 is isolated similarly. The downwards exposed computation $a + b$ in block 11 is isolated because the definition $h = a + b$ inserted inside the block is dead at its exit. To avoid making these unnecessary insertions, a live variable data-flow analysis pass for h is performed. The lifetime optimal solution found by MC-PRE is the one already given in (2), which results in Figure 1(f).

It is important to point out at the outset that in our implementation, we do not actually modify a CFG at all to produce the required EFG. Instead, we generate a graph specification for the EFG and feed it to a min-cut solver so that the desired minimum cut can be found very efficiently.

In general, it is safe to perform speculative code motion only on non-excepting expressions. But a speculative evaluation of an expression that may throw runtime exceptions can change the semantics of the program. Fortunately, hardware support available in modern processors (e.g., speculative instructions as in IA-64) allows this constraint to be relaxed. For architectures that do not provide such advanced features, our algorithm can only be applied safely to non-excepting expressions.

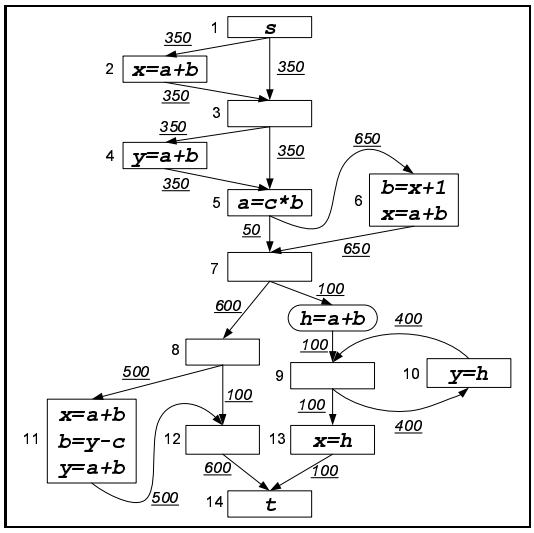


Fig. 2. Transformation of the CFG given in Figure 1(a) by LCM.

The rest of this paper is organized as follows. Section 2 provides a detailed review of the related work. Section 3 gives some background information and introduces the notions of computational optimality and lifetime optimality relevant to the speculative code motion. Section 4 presents the MC-PRE algorithm in a style that can be directly and efficiently implemented in a compiler, while stating some lemmas with proofs. In Section 5, we discuss some theoretical aspects of the algorithm and prove rigorously its correctness and optimality. In Section 6, we present and analyze our experimental results on evaluating the efficiency and effectiveness of the MC-PRE algorithm against LCM using SPECcpu2000 on two architectures. Section 7 concludes the paper and discusses some future work.

2. RELATED WORK

PRE originated from the seminal work of Morel and Renvoise [Morel and Renvoise 1979] and was soon realized as an important optimization technique that subsumes GCSE and LICM. Morel and Renvoise's data-flow framework is imperfect: it is bidirectional, provides no assurance for lifetime optimality and is profile-independent. In the past two decades, their work has undergone a number of refinements and extensions [Briggs and Cooper 1994; Chow 1983; Click 1995; Dhamdhere 1991; Dhamdhere et al. 1992; Drechsler and Stadel 1993; Hosking et al. 2001; Kennedy et al. 1999; Knoop et al. 1992; 1994; Rosen et al. 1988; Simpson 1996]. In particular, Knoop et al. [1994] describe a uni-directional bit-vector formulation that is optimal by the criteria of computational optimality and lifetime optimality, and more recently, Kennedy et al. [1999] present an SSA-based framework that shares the same two optimality properties. In addition, Knoop et al. [2000] extend the LCM algorithm to handle predicated code. Kennedy et al. [1998] extend the SSAPRE algorithm to perform strength reduction and linear function test replacement. By combining global reassociation and global value numbering [Alpern et al. 1988],

Briggs and Cooper [1994] can also eliminate redundancies for certain semantically-identical expressions. However, these previous research efforts are restricted by the safety code motion criterion [Kennedy 1972] and insensitive to the execution frequencies of a program point in a given CFG.

Horspool and Ho [1997] and Gupta et al. [1997] introduce the first two algorithms on performing the profile-guided speculative PRE (by combining code motion and control speculation); both assume single statement blocks only. The former algorithm analyzes edge insertions based on an edge profile; each data-flow variable takes the form of a bit vector whose size equals the number of flow edges in the CFG being analyzed. The latter algorithm analyzes node insertions based on a path profile, requiring bit vectors whose size equals the number of paths in the CFG with non-zero frequencies. In contrast, the bit vectors used by our MC-PRE algorithm requires only one bit each. Each of these two earlier algorithms relies on a cost-benefit analysis to solve the speculative PRE problem (which is shown here to be equivalent to a single-commodity network flow problem after a graph transformation). Due to the *local* heuristics used, the two earlier algorithms are not computationally optimal in terms of the dynamic number of eliminated redundant computations. Finally, these two algorithms are not experimentally evaluated.

Figure 3 compares Horspool and Ho's algorithm (HH) [Horspool and Ho 1997] and MC-PRE. In the CFG given in Figure 3(a), blocks 3, 6 and 7 are known as candidate blocks since they each contain an upwards exposed computation of $a+b$. The HH algorithm has two phases. The first phase conducts a forward data-flow analysis on the given CFG to find the set CIN_B of the least-cost insertion edges for every candidate block B. The second phase selects iteratively the insertion points from CIN_B for every candidate block B according to the overall net benefit achieved. One major problem with HH (as also noted by Horspool and Ho) is that the union of CIN_B for all candidate blocks B may not necessarily contain all optimal insertion points required. In the CFG given in Figure 3(a), the computations $a+b$ in blocks 6 and 7 are partially redundant with respect to the $a+b$ in block 3. MC-PRE produces the optimal solution as shown Figure 3(b), where the insertion points are $(1,3)$ and $(2,4)$. As a result, we have reduced the total number of computations for $a+b$ optimally from 298 to 200. However, the HH algorithm would find $\mathsf{CIN}_3 = \{(1,3)\}$, $\mathsf{CIN}_6 = \{(5,6)\}$ and $\mathsf{CIN}_7 = \{(5,7)\}$ and then choose all these three edges as the insertion points as shown in Figure 3(c). While CIN_3, CIN_6 and CIN_7 are locally optimal for the candidate blocks 3, 6, and 7, respectively, the global optimal insertion edge $(2,4)$ is absent in their combined set. Such a solution has failed to eliminate any redundancies available for this particular example.

Figure 4 compares Gupta, Berson and Fang's algorithm (GBF) [Gupta et al. 1997] and the MC-PRE algorithm. The GBF algorithm also has two phases. The first phase performs availability and anticipability analyses to compute the cost and benefit information at all branch nodes from a path profile. The second phase determines iteratively a subset of branch nodes at which speculation should be enabled. This algorithm cannot always detect situations in which enabling speculation is beneficial (as also recognized by its authors). The CFG shown in Figure 4(a) is taken from their Figure 4(a), where the frequency of the path P6 has been modified to be 60 (from 40 originally). The GBF algorithm would fail to remove any redundancies, because it considers the paths P1 and P4 individually in isolation:

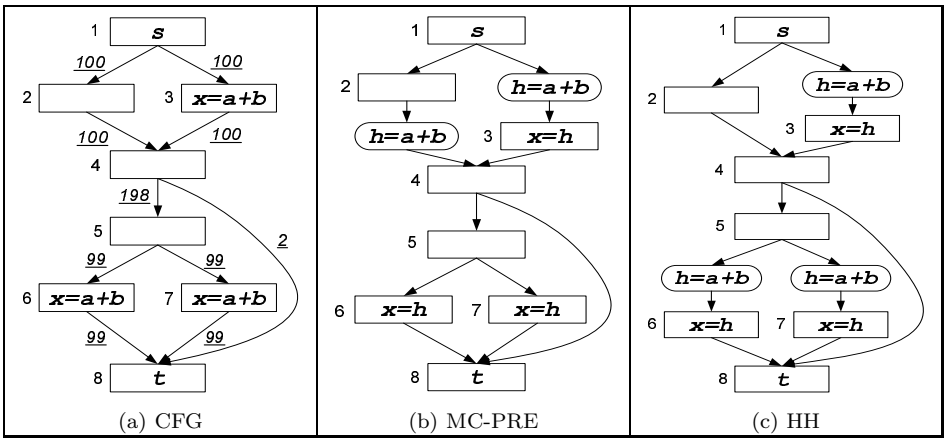


Fig. 3. A comparison of Horspool and Ho's algorithm [Horspool and Ho 1997] and MC-PRE.

executing $a + b$ speculatively on the edge $(4, 5)$ cannot benefit P1 and executing $a+b$ speculatively on both $(4, 5)$ and $(6, 12)$ cannot benefit P4. However, MC-PRE produces the optimal solution as shown in Figure 4(b), reducing the total number of computations for $a+b$ by 39. Finally, GBF relies on a path file, and consequently, has to handle loops as a special case by adding special nodes to break them into acyclic paths. In contrast, MC-PRE achieves optimal code motion directly on any given CFG (containing possibly irreducible components) from an edge profile.

Bodik et al. [1998] apply control flow restructuring as an alternative to speculation to enable code motion. Their algorithm (denoted comPRE) can completely remove all partial redundancies in a CFG at the cost of duplicating its CMP (code-motion-preventing) region without having to rely on profiling information. Figure 5 compares comPRE and MC-PRE. For the CFG in Figure 5(a), MC-PRE will choose not to transform the program since enabling speculation yields no benefit. In Section 4, we will see how this desirable result is obtained in our framework automatically. On the other hand, comPRE will proceed as follows. First, it identifies the CMP region in the CFG by solving the problems of availability and anticipability on a special lattice. The four values in this lattice are encoded using two bits. In the example, the CMP region consists of block 4 (which can be larger, if, for example, it is replaced by an `if` statement). Second, the CMP region is duplicated as illustrated in Figure 5(b) so that all code-motion obstacles are eliminated. Finally, comPRE applies a CMP-based but LCM-equivalent technique to remove all partial redundancies in the restructured CFG. This gives rise to Figure 5(c).

In theory, comPRE removes all redundancies removable by MC-PRE plus some more removable by any control flow restructuring. However, the extra benefit obtained is uncertain unless it is guided by a path profile or something more precise. For the CFG given in Figure 5(a), the frequency of the path $\langle 1, 2, 4\rangle$ determines exactly the amount of overall benefit that can be achieved by the comPRE solution given in Figure 5(c). There are two extreme cases. If the frequency of this path is 0, comPRE has offered no benefit at the expense of code duplication. If the path frequency is 350, comPRE has succeeded in removing all 350 redundant computations

Path	Freq.	Path	Freq.
P1: 1-2-5-7-9-15	100	P6: 1-3-4-5-7-10-13-15	50
P2: 1-2-5-7-10-13-15	1	P7: 1-3-4-5-8-12-11-13-15	50
P3: 1-2-5-8-12-11-13-15	1	P8: 1-3-4-5-8-12-14-15	1
P4: 1-2-5-8-12-14-15	50	P9: 1-3-6-12-11-13-15	1
P5: 1-3-4-5-7-9-15	50	P10: 1-3-6-12-14-15	50

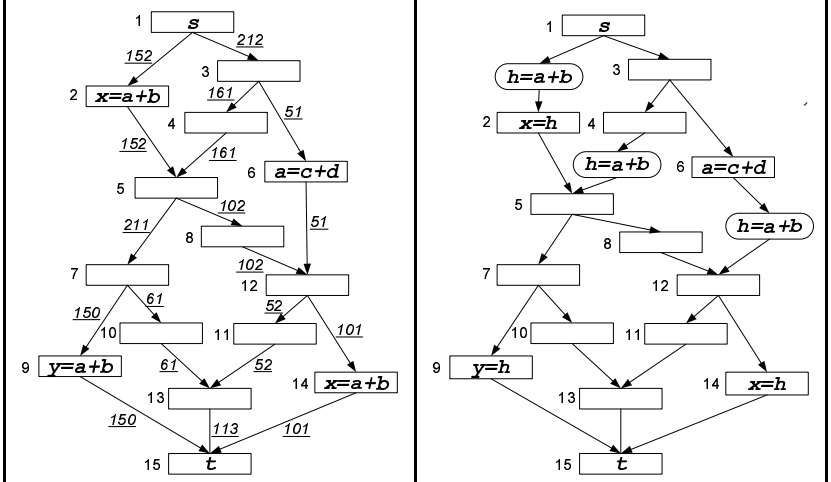


Fig. 4. A comparison of Gupta, Berson and Fang's algorithm [Gupta et al. 1997] and MC-PRE.

for $a+b$. In order to improve the cost-benefit ratio, Bodik et al. [1998] suggested to resort to selective restructuring (by duplicating only the hot paths containing redundancies in the CMP) and/or selective speculation. Unfortunately, this requires the power of expensive path profiles in order to establish precisely the benefit of selective restructuring. If only edge profiles are available, they attempt to improve the frequency analysis [Ramalingam 1996] to estimate on demand the probability that a path may execute. Bodik et al. [1998] implemented comPRE in the IMPACT compiler [Chang et al. 1991] and evaluated its effectiveness using SPEC95. They found that the amount of redundancies that *cannot* be removed by using speculation alone is negligible. In addition, such a negligible amount of redundancies is eliminated at the cost of over 30% code explosion for most benchmarks. In this paper, MC-PRE achieves a complete removal of all partial redundancies removable by using control speculation from an edge profile at small space overheads.

Based on the concept of CMP, Bodik [1999] gives a computationally optimal algorithm for solving the speculative PRE problem from an edge profile. This algorithm, which is denoted PRE(MS) and presented for single statement blocks, operates in four steps. First, the CMP in a given CFG is identified by solving the problems of availability and anticipability on the special lattice discussed above. Second, a minimum cut on the CMP is found to obtain some insertion edges, which may not be all insertion edges required. Third, the insertions on these cut edges

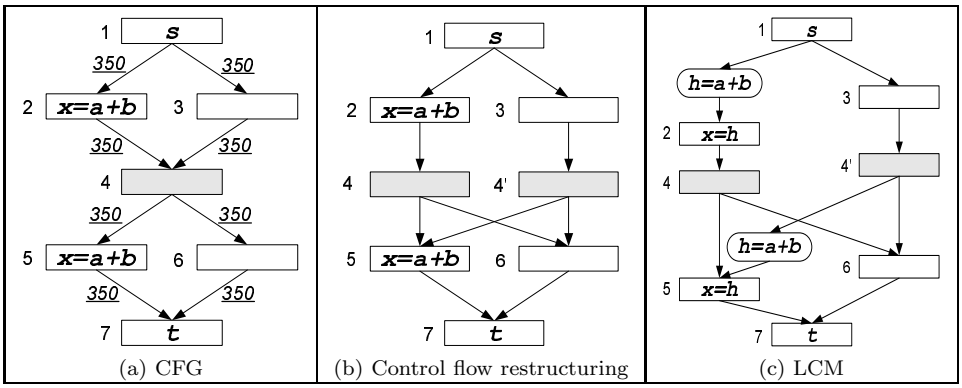


Fig. 5. A comparison of Bodik, Gupta and Soffa's algorithm [Bodik et al. 1998] and MC-PRE.

are made and the availability analysis is repeated so that all insertions required are found and performed. Finally, a CMP-based but LCM-equivalent technique is applied to remove all partial redundancies. Bodik [1999] did not implement this algorithm. Instead he used a non-optimal algorithm discussed in [Bodik et al. 1998, §3.2] in his experiments to perform speculative code motion.

All these existing PRE algorithms on speculative code motion rely on specialized flow analysis techniques to locate potential insertion points. In [Cai and Xue 2003], we present a new algorithm that achieves computational optimality from an edge profile based on standard data-flow analyses and a min-cut algorithm. We gave the algorithm for single statement blocks in that paper but implemented a version for standard basic blocks in Intel's ORP [Adl-Tabatabai et al. 1998; Cierniak et al. 2000]. We reported performance results for the SPECjvm98 benchmarks. Recently, Scholz et al. [2004] map the speculative PRE problem for single statement blocks into Stone's problem and solve the latter also using a min-cut solver. In this paper, we extend our earlier algorithm so that the new algorithm, MC-PRE, can also achieve lifetime optimality for the first time. Compared with our earlier algorithm, MC-PRE introduces temporary registers only when necessary, with the shortest lifetimes possible. The lifetime optimality is achieved by finding a special minimum cut and then performing one additional standard live range analysis to avoid making insertions and replacements for isolated computations. By implementing this algorithm in `gcc` and evaluating it using SPECcpu2000, we show that our algorithm can be potentially used as a practical pass in a profile-guided compiler. Table I summarizes how MC-PRE is compared and contrasted against the existing code-motion-based speculative PRE algorithms we have just reviewed above.

Finally, Lo et al. [1998] extend the SSAPRE algorithm to handle control speculation and register promotion. Their algorithm performs no worse than no speculation but is not optimal. Lin et al. [2003] incorporate alias profiling information to support data speculation in the SSAPRE framework.

3. PRELIMINARIES

To make this paper self-contained, Section 3.1 recalls some concepts and results about directed graphs, flow networks and make precise about the notion of cut used

Algorithm	HH	GBF	PRE(MS)	CX	SHK	This Work
Basic Blocks	single statement blocks					standard
Data-Flow Passes	1	2	5	2	n/a	3
Bit-Vector Size	X	Y	2	1	n/a	1
Insertion Points	heuristics	heuristics	min-cut	min-cut	min-cut	min-cut
Comp. Optimality	×	×	√	√	√	√
Lifetime Optimality	×	×	×	×	×	√

X: #flow edges Y: #paths with non-zero frequencies

Table I. A comparison of MC-PRE with code-motion-based speculative PRE algorithms. CX denotes our earlier work [Cai and Xue 2003] while SHK the recent work [Scholz et al. 2004].

in this paper. In particular, the notion of minimum cut is required to establish computationally and lifetime optimal results. In Section 3.2, we introduce the control flow graphs (CFGs) and the local predicates in our data-flow analysis. In Section 3.3, we formulate the speculative PRE as a code motion transformation and define precisely its correctness, computational optimality and lifetime optimality.

3.1 Directed Graphs and Flow Networks

Let $G = (N, E)$ be a directed graph with the node set N and the edge set E. The notation $\mathsf{pred}(G, n)$ represents the set of all *immediate predecessors* of a node n and $\mathsf{succ}(G, n)$ the set of all its *immediate successors* in G:

$$\begin{aligned} \mathsf{pred}(G, n) &= \{m \mid (m, n) \in E\} \\ \mathsf{succ}(G, n) &= \{m \mid (n, m) \in E\} \end{aligned}$$

A node m is a *predecessor* of a node n if m is an immediate predecessor of n or if m is a predecessor of an immediate predecessor of n. A *path* of G is a finite sequence of nodes $n_1, \ldots, n_k$ such that $n_i \in \mathsf{succ}(n_{i-1})$, for all $1 < i \leqslant k$, and the *length* of the path is k. In this case, we write $\langle n_1, \ldots, n_k \rangle$ and refer to the sequence as a path from n_1 to n_k (inclusive). Any subsequence of a path is referred to as a *subpath*.

A directed graph $G = (N, E)$ is a *flow network* if it has two distinguished nodes, a *source* s and a *sink* t, and a nonnegative *capacity* (or *weight*) $c(n, m) \geqslant 0$ for each edge $(n, m) \in E$. If $(n, m) \notin E$, it is customary to assume that $c(n, m) = 0$ [Cormen et al. 1990]. For convenience, every node is assumed to lie on some path from the source s to the sink t. Thus, a flow network is connected (from s to t).

Let S and $T = N - S$ be a partition of N such that $s \in S$ and $t \in T$. We denote by (S, T) the set of all (directed) edges with tail in S and head in T:

$$(S, T) = \{(n, m) \in E \mid n \in S, m \in T\} \tag{4}$$

A *cut* separating s from t is any edge set $(C, \overline{C})$, where $s \in C$, $\overline{C} = N - C$ is the complement of C and $t \in \overline{C}$. The *capacity* of this cut, denoted by $cap(C, \overline{C})$, is the sum of the capacities of all the edges in the cut (called *cut edges*):

$$cap(C, \overline{C}) = \sum_{(m,n) \in (C, \overline{C})} c(m, n) \tag{5}$$

By a *minimum cut*, we mean a cut separating s from t with minimum capacity.

The minimum cuts are not necessarily unique. This non-uniqueness translates to the non-uniqueness of computationally optimal solutions to a PRE problem. By finding a unique minimum cut for a PRE problem, the corresponding computationally optimal solution is lifetime optimal when the frequencies of all flow edges are positive.

The *max-flow problem* consists of finding a flow of maximum value from the source s and the sink t. The max-flow min-cut theorem of Ford and Fulkerson [1962] dictates that such a flow exists and has a value equal to the capacity of a minimum cut.

3.2 Control Flow Graphs

A control flow graph (CFG) is a directed graph annotated with an edge profile. We represent a CFG (reducible or not) as a weighted graph $G = (N, E, W)$, where

—the nodes in the node set N represent basic blocks,
—the edges in the edge set E represent potential (or nondeterministic) flow of control between basic blocks,
—$s \in N$ represents the unique *entry block* without any predecessors,
—$t \in N$ represents the unique *exit block* without any successors,
—every node in N is on some path from s to t, and
—W is a *weight function*: $W : E \mapsto \mathbb{N}$, where $\mathbb{N}$ denotes the set of natural numbers (starting from 0).

In particular, the weight $W(u, v)$ attached to edge $(u, v) \in E$ is a nonnegative integer representing the frequency of its execution. The edge profiling information required can be gathered via code instrumentation [Ball and Larus 1994] or statistic sampling of the program counter (PC) [Anderson et al. 1997]. An edge profile has less runtime overhead to collect than a path profile [Ball et al. 1998]. The information contained in an edge profile, while less accurate than a path profile, is sufficient to guarantee computationally and lifetime optimal results for the speculative PRE.

For convenience, we write $W(n)$ to represent the execution frequency of a basic block $n \in N$. In this paper, nodes and (basic) blocks are interchangeable.

Assumption 3.1. For every basic block n in a CFG G, the following assumption about an edge profile is made:

$$W(n) \;=\; \sum_{m \in \mathsf{pred}(G,n)} W(m,n) \;=\; \sum_{m \in \mathsf{succ}(G,n)} W(n,m) \tag{6}$$

In all example CFGs used for illustrations, variables with distinct names are consistently meant to be distinct (i.e., not aliased to each other).

3.2.1 *Basic Blocks.* As is customary, a *basic block* is a sequence of consecutive statements (or instructions) in which flow of control enters only at the beginning and leaves only at the end. (This work applies even if such a sequence is not the longest possible.) For convenience and without loss of generality, we assume that each statement has the form $v = e$, where v is a variable and e an expression built in terms of variables, constants and operators. Following [Morel and Renvoise 1979; Knoop et al. 1994], we further assume that the right-hand side (RHS) of

every assignment contains at most one operator. Therefore, by an expression in a program, we always mean the RHS of some assignment.

To avoid using a highly parameterized notation, we present our algorithm for an arbitrary but fixed CFG $G = (N, E, W)$ and a generic expression π. An assignment to some operands of π is called a *modification* to π. It is understood that a PRE problem is defined by both a CFG and an expression. Thus, the two PRE problems on the same CFG are distinct if the two corresponding expressions are distinct.

The concepts of partial and full redundancies are recalled below.

Definition 3.2. Let E_1 and E_2 be two occurrences of expression π (which are not necessarily distinct). If there is a control flow path from E_1 to E_2 that contains no modification to π throughout the path, E_2 is said to be *partially redundant with respect to* E_1. Furthermore, if E_2 is partially redundant along all possible incoming paths leading to the blocking containing E_2, then E_2 is said to be *fully redundant*.

In Figure 1(a), there are several partially redundant computations of $a + b$. For example, the $a + b$ in block 4 is partially redundant with respect to the $a + b$ in block 2. But no computation of $a + b$ is fully redundant.

3.2.2 *Local Redundancies and Local Predicates.* Like LCM, MC-PRE is mainly an iterative bit-vector algorithm operating on a non-SSA form. Therefore, our algorithm conducts a global optimization that aims at removing partial redundancies across the basic blocks. It is assumed that standard techniques such as local common subexpression elimination (LCSE) has been carried out on basic blocks. As a result, every two consecutive occurrences of a given expression π in the same block must be separated by at least one modification to π.

For each block n, we define below four local predicates for a given expression π and state some salient properties. Our illustrating example is given in Figure 6.

$\mathsf{ANTLOC}(n)$	=	true iff π is *locally anticipatable* on entry of block n, i.e., block n contains a unique upwards exposed computation of π (which is the computation of π that is not preceded by any modification to π).
$\mathsf{AVLOC}(n)$	=	true iff π is *locally available* on exit from block n, i.e., block n contains a unique downwards exposed computation of π (which is the computation of π that is not followed by any modification to π).
$\mathsf{TRANSP}(n)$	=	true iff block n is transparent to π, i.e., block n does not contain any modification to π.
$\mathsf{KILL}(n)$	=	$\neg\mathsf{TRANSP}(n)$; n is called a *kill node* if $\mathsf{KILL}(n) = \mathsf{true}$.

To highlight the fact that MC-PRE shares the same bit-vector routines as LCM in our implementation, these predicates are named identically as in `gcc`.

Both the upwards and downwards exposed computations are called the *PRE candidate computations*. All the others (e.g., the second $a + b$ in Figure 6(d)) are irrelevant for global redundancy elimination. Thus, a block contains at most two PRE candidate computations for a given expression π.

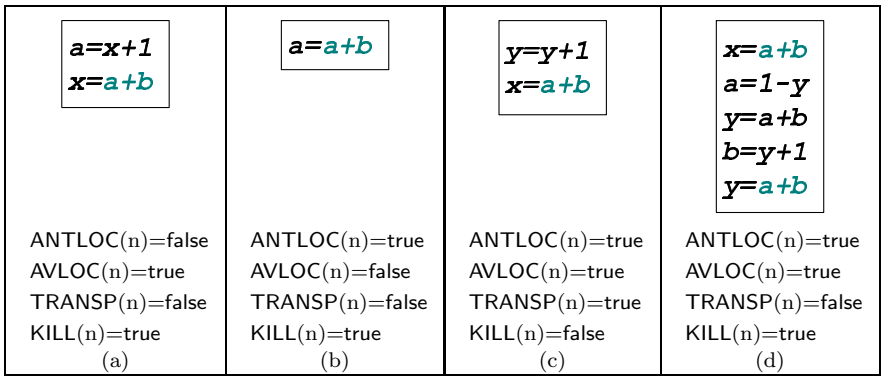


Fig. 6. Basic blocks, local predicates and PRE candidate computations (in gray) for expression $a+b$. In (a), the block is a D-block, where the $a+b$ is downwards but not upwards exposed. In (b), the block is a U-block, where the $a+b$ is upwards but not downwards exposed. In (c), the block is a U-block (but not a D-block), where $a+b$ is both upwards and downwards exposed. In (d), the block is a U-block with the first $a+b$ being upwards exposed. It is also a D-block with the last $a+b$ being downwards exposed. But the second $a+b$ is not a PRE candidate computation.

We have decided to present our algorithm directly for standard basic blocks. Therefore, it becomes necessary to distinguish two kinds of blocks as follows.

Definition 3.3. A block n is called a *U-block* if $\mathsf{ANTLOC}(n) = \mathsf{true}$. The set of all U-blocks in a CFG $G = (N, E, W)$ is denoted by:

$$UB = \{n \in N \mid \mathsf{ANTLOC}(n)\} \tag{7}$$

Definition 3.4. A block n is called a *D-block* if $\mathsf{AVLOC}(n) \wedge \mathsf{KILL}(n) = \mathsf{true}$. The set of all D-blocks in a CFG $G = (N, E, W)$ is denoted by:

$$DB = \{n \in N \mid \mathsf{AVLOC}(n) \wedge \mathsf{KILL}(n)\} \tag{8}$$

Note that a block n can be identified as both a U-block and a D-block. In this case, n must be a kill block, and in addition, n contains exactly two PRE candidate computations of π — the block shown in Figure 6(d) is one such an example. If a single computation of π is both upwards and downwards exposed in a block n, then n cannot be a kill block. In this case, n is classified as a U-block but not also a D-block. One such an example can be found in Figure 6(c).

Finally, we assume that the entry block s has an (imaginary) definition for every variable, which precedes all existing statements, to represent whatever value the variable may have when s is entered. Technically, this prevents an expression from being hoisted past s "accidentally" during speculative code motion.

Assumption 3.5. For the entry block s, it is assumed that $\mathsf{KILL}(s) = \mathsf{true}$ and $\mathsf{ANTLOC}(s) = \mathsf{false}$ for every PRE candidate expression.

Note that our algorithm does not require the entry and exit blocks to be empty.

3.2.3 *Critical Edges.* Our algorithm *reasons about* insertions on edges speculatively. As a result, it is directly applicable to any CFG even if it contains *critical edges*, i.e., the edges leading from nodes with more than one immediate successor

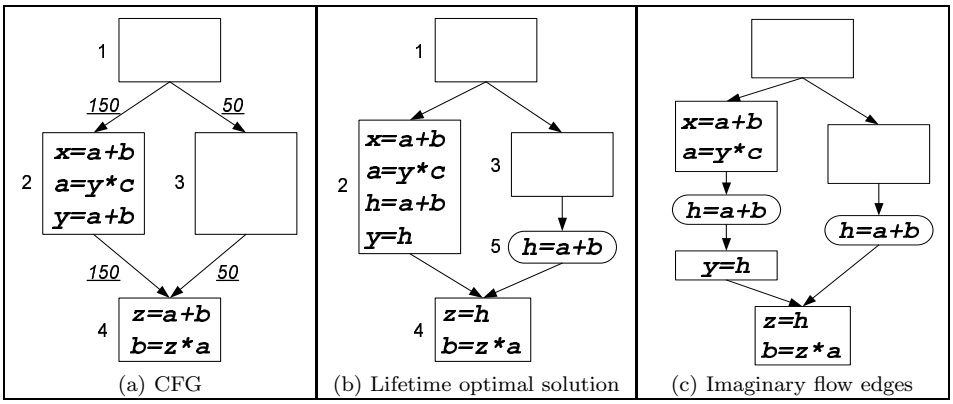


Fig. 7. Insertion points for D-blocks.

to nodes with more than one immediate predecessor [Knoop et al. 1994]. Thus, there is no need to split them before our algorithm is applied.

3.3 Speculative PRE

A speculative PRE optimization is realized as a code motion transformation with all insertions and deletions of computations performed in such a way that the semantics of the program is preserved. In this study, we analyze edge insertions based on an edge profile. Given an expression π, we aim at minimizing its dynamic number of computations as well as the lifetimes of introduced temporaries in a CFG.

In code motion transformations, we should make it clear about the points where insertions are made. The insertion point for a D-block (if required) cannot possibly be on any flow edge. Figure 7 illustrates this problem. For the CFG given in Figure 7(a), our MC-PRE algorithm will produce the solution shown in Figure 7(b). In this simple case, a visual inspection reveals that our solution is both computationally and lifetime optimal. However, the insertion point for the second $a + b$ in the D-block 2 is the point just before the assignment $y = a + b$. The required insertion cannot be performed on a flow edge since the computation $a + b$ is preceded by the assignment $a = y * c$. We could turn such an insertion point into a flow edge once a simple transformation as demonstrated in Figure 7(c) had been done. But our algorithm does not use this transformation. Instead, it will operate directly on standard CFGs. (Of course, it can be more convenient to talk about only edge insertions if we postulate the existence of such imaginary flow edges.)

3.3.1 *Speculative Code Motion.* Figure 8 summarizes an algorithm for implementing a speculative code motion. A *speculative PRE transformation*, denoted T, for a generic expression π in a CFG $G = (N, E, W)$ is characterized by four sets:

—$U\text{-}Ins_T \subseteq E$ is a set of flow edges where insertions are to be made. The *insertion point* for a flow edge is inside a new block created on the edge.

—$D\text{-}Ins_T \subseteq DB$ is a set of D-blocks where insertions are to be made. The *insertion point* in a D-block is the point just before the assignment containing the down-

1. Introduce a new temporary variable h_π for π.

2. (a) Insert $h_\pi = \pi$ at every flow edge in $U\text{-}Ins_T$.
   (b) Insert $h_\pi = \pi$ at the insertion point of every D-block in $D\text{-}Ins_T$.

3. (a) Replace the upwards exposed computation of π appearing in the assignment $\cdots = \pi$ contained in every U-block of $U\text{-}Rep_T$ by h_π.
   (b) Replace the downwards exposed computation of π appearing in the assignment $\cdots = \pi$ contained in every D-block of $D\text{-}Rep_T$ by h_π.

Fig. 8. Code motion for a PRE transformation T.

wards exposed computation of π in the block. This ensures that the lifetimes of all introduced temporaries within basic blocks are kept locally minimal.

—$U\text{-}Rep_T \subseteq UB$ is a set of U-blocks whose upwards exposed computations of π (called *replacement computations*) are to be replaced.

—$D\text{-}Rep_T \subseteq DB$ is a set of D-blocks whose downwards exposed computations of π (also called *replacement computations*) are to be replaced.

If $DB = \emptyset$, we always have $D\text{-}Ins_T = D\text{-}Rep_T = \emptyset$. In other words, the absence of D-blocks implies that all insertion points are on flow edges. Note that in the special case when all blocks contain single assignments only, $DB = \emptyset$ always holds.

For convenience, we define $W(S) = \sum_{x \in S} W(x)$, where S is a set of flow edges or blocks. Let $W(T)$ be the dynamic number of computations of π in the transformed code by a transformation T with respect to the edge profile W:

$$W(T) \;=\; W(UB) + W(DB) - \mathit{benefit}(T) \tag{9}$$

where

$$\mathit{benefit}(T) \;=\; W(U\text{-}Rep_T) + W(D\text{-}Rep_T) - W(U\text{-}Ins_T) - W(D\text{-}Ins_T) \tag{10}$$

Here, $W(UB) + W(DB)$ represents the number of computations of π in the original program and $\mathit{benefit}(W)$ the number of eliminated computations of π. Hence, $W(T)$ gives rise to the total number of computations in the transformed code.

Consider the CFG depicted in Figure 7(a) with $\pi = a + b$, where:

$$\begin{aligned} UB &= \{2, 4\} \\ DB &= \{2\} \end{aligned}$$

Let us write R to denote the optimal transformation performed on Figure 7(a) to obtain Figure 7(b), where R is specified by the following four sets:

$$\begin{aligned} U\text{-}Ins_R &= \{(3, 4)\} \\ U\text{-}Rep_R &= \{4\} \\ D\text{-}Ins_R &= \{2\} \\ D\text{-}Rep_R &= \{2\} \end{aligned} \tag{11}$$

where $U\text{-}Ins_R$ tells us to insert $h = a + b$ on the edge $(3, 4)$, $D\text{-}Ins_R$ tells us to insert $h = a + b$ just before the last statement in the D-block 2, $U\text{-}Rep_R$ asks us to replace the upwards exposed (i.e., the first) $a + b$ in block 4 by h and $D\text{-}Rep_R$ asks us to replace the downwards exposed (i.e., the second) $a + b$ in the D-block 2 by h.

In the original CFG given in Figure 7(a), the number of computations required for $a+b$ is $W(UB)+W(DB) = W(2)+W(4)+W(2) = 500$. By applying R given in (11), the number of eliminated computations is $benefit(R) = W(U\text{-}Rep_R) + W(D\text{-}Rep_R) - W(U\text{-}Ins_R) - W(D\text{-}Ins_R) = 150+200-50-150 = 150$. Hence, the number of remaining computations in the transformed code given in Figure 7(b) is $W(R) = W(UB)+W(DB) - benefit(R) = 500-150 = 350$.

For our running example, we discussed two PRE transformations earlier in Section 1. The reader is now invited to verify that $\mathcal{LO}$ given in (2) specifies the transformation from Figure 1(a) to Figure 1(f) and $\mathcal{CO}_1$ given in (3) specifies the transformation from Figure 1(a) to Figure 1(e).

3.3.2 *Correctness.* A PRE transformation is correct if h_π is initialized on every path leading to a replacement computation in such a way that no modification to π occurs afterwards. Then h_π always represents the same value as π at every replacement computation. So the semantics of the program is preserved.

Definition 3.6. Consider an expression π in a CFG $G = (N, E, W)$. A PRE transformation T is *correct* if it satisfies the following two properties, each of which means that every replacement is *correct*:

P1. If $n \in D\text{-}Rep_T$, then $n \in D\text{-}Ins_T$.

P2. If $n \in U\text{-}Rep_T$, then for every path $p(s,n)$ from the entry block s to the block n, either Statement *S1* or *S2* given below must be true:

- *S1.* (a) $p(s,n)$ includes at least one insertion point $v \in D\text{-}Ins_T$, and
  (b) no block in the subpath of $p(s,n)$ from v to n (excluding both v and n) is a kill block of π.
- *S2.* (a) $p(s,n)$ includes at least one insertion edge $(u,v) \in U\text{-}Ins_T$, and
  (b) no block in the subpath of $p(s,n)$ from v to n (excluding n) is a kill block of π.

For the example given in Figure 7, we can verify that the transformation R given in (11) is correct. Basically, R is correct since each use of h is identified with a definition of h along each incoming path. For our running example, we can also verify similarly that $\mathcal{LO}$ given in (2) generates the correct code given in Figure 1(f) and $\mathcal{CO}_1$ given in (3) generates the correct code given in Figure 1(e).

We denote by $\mathcal{CM}_{Cor}$ the set of all correct transformations for a generic expression π in $G = (N, E, W)$.

3.3.3 *Computational Optimality.* The primary goal is to minimize the dynamic number of computations, which is reflected by the following criterion.

Definition 3.7. Consider an expression π in a CFG $G = (N, E, W)$. A PRE transformation T is said to be *computationally optimal* if

(1) T is correct, i.e., $T \in \mathcal{CM}_{Cor}$, and
(2) $W(T) \leqslant W(T')$ for all $T' \in \mathcal{CM}_{Cor}$.

For the CFG given in Figure 7(a), the transformation R given in (11) is computationally optimal. The transformed code is shown in Figure 7(b). For our running example, where the CFG is depicted in Figure 1(a), the transformations $\mathcal{LO}$ and

$\mathcal{CO}_1$ given in (2) and (3) are both computationally optimal, which result in the transformed programs shown in Figures 1(e) and (f), respectively.

We denote by $\mathcal{CM}_{CompOpt}$ the set of all computationally optimal transformations for a generic expression π in $G = (N, E, W)$.

3.3.4 *Lifetime Optimality.* Of all computationally optimal transformations in $\mathcal{CM}_{CompOpt}$, we want to find one such that the lifetime ranges of introduced temporaries in the transformed code are minimized.

The lifetime ranges of temporaries are defined in terms of program points similarly as in [Rüthing 1998]. A *program point* in a block is a point before the first instruction, between two adjacent instructions or after the last instruction. By making the convention that the last program point in a block is a predecessor of the first program point of any of its successor blocks, we can conveniently talk about paths of program points in a CFG [Aho et al. 1986].

The lifetime range of h_π for a particular insertion h_π is the set of program points where the value of h_π needs to be kept in the temporary h_π and cannot be released. Without loss of generality, the insertion made on an edge $(m, n) \in \textit{U-Ins}_T$ is regarded as being associated with the first program point of block n.

Let $\langle p_1, \ldots, p_k \rangle$ be an *insertion-replacement* path of program points such that

—there is an insertion of $h_\pi = \pi$ at p_1,

—there are no insertions at $p_2, \ldots, p_k$, and

—the instruction after p_k contains a replacement computation.

The *lifetime range* of this path is defined to be $\{p_1, \ldots, p_k\}$.

Definition 3.8. Consider an expression π in a CFG $G = (N, E, W)$. Let $T \in \mathcal{CM}_{Cor}$. Let IRP_T be the set of all insertion-replacement paths in the transformed code obtained by T. The *lifetime range* of T is given by:

$$\mathsf{LTRange}(T) = \bigcup_{\langle p_1,\ldots,p_k\rangle \in \mathsf{IRP}_T} \{p_1, \ldots, p_k\} \tag{12}$$

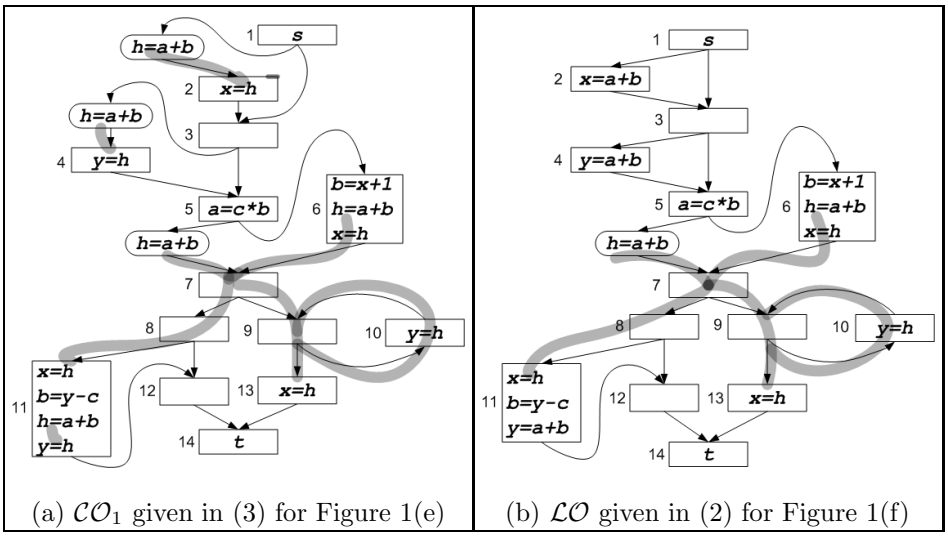
Let $T, T' \in \mathcal{CM}_{CompOpt}$. T is said to be *lifetime better* than T' if $\mathsf{LTRange}(T) \subset \mathsf{LTRange}(T')$. A PRE transformation $T \in \mathcal{CM}_{CompOpt}$ is *lifetime optimal* if

(1) $\mathsf{LTRange}(T) \subseteq \mathsf{LTRange}(T')$ for all $T' \in \mathcal{CM}_{CompOpt}$, and

(2) Every insertion in $\textit{U-Ins}_T \cup \textit{D-Ins}_T$ must be used by some existing computations of π, i.e., be the first program points of some insertion-replacement paths.

The second condition is not necessary for LCM since the execution frequencies of all edges are assumed to be positive. In this paper, an edge or block can have a zero execution frequency. Thus, an insertion at such a point will not affect the computational optimality of a PRE transformation. But such an insertion (i.e., dead assignment) should not be allowed in a lifetime optimal solution.

Let $\mathcal{CM}_{LifeOpt}$ be the set of all lifetime optimal transformations for a generic expression π in $G = (N, E, W)$. Although "lifetime better" is a partial rather than total order, we shall find a solution and show that the solution is the unique lifetime optimal transformation when the frequencies of all flow edges are nonzero.



(a) $\mathcal{CO}_1$ given in (3) for Figure 1(e) (b) $\mathcal{LO}$ given in (2) for Figure 1(f)

Fig. 9. Lifetime ranges for Figures 1(e) and (f).

Figure 9 illustrates the concept of lifetime ranges for $\mathcal{LO}$ given in (2) and $\mathcal{CO}_1$ given in (3), which result in the transformed codes depicted in Figures 1(e) and (f), respectively. In this running example, $\mathcal{LO}$ is lifetime optimal but $\mathcal{CO}_1$ is not.

4. THE MC-PRE ALGORITHM

This section develops our algorithm in two stages. In Section 4.1, we present an initial algorithm, called MC-PRE$_{\text{comp}}$, for finding computationally optimal transformations. By refining this algorithm, Section 4.2 presents our final algorithm, called MC-PRE, that finds a lifetime optimal transformation for a PRE problem. In Section 4.3, we analyze the time and space complexity of the MC-PRE algorithm.

This two-stage presentation has several advantages. First, the development of our lifetime optimal algorithm is parallel to that of the LCM algorithm [Knoop et al. 1994]), with which our algorithm will be evaluated against in Section 6. Second, MC-PRE$_{\text{comp}}$ works for standard basic blocks while our earlier algorithm described in [Cai and Xue 2003] assumes single statement blocks. Finally, such an incremental development allows the issues of computational optimality and lifetime optimality to be dealt with incrementally since the latter is a refinement of the former.

4.1 Computationally Optimal Transformations

The MC-PRE$_{\text{comp}}$ algorithm presented in Figure 10 takes a CFG and returns a computational optimal transformation denoted by $\mathcal{CO}$ (for any edge profile regardless whether some flow edges are zero-weighted or not). The first three steps find *D-Rep*$_{\mathcal{CO}}$, *D-Ins*$_{\mathcal{CO}}$ and *U-Rep*$_{\mathcal{CO}}$ trivially based on the local predicates only. The last step constructs *U-Ins*$_{\mathcal{CO}}$ by relying on two global data-flow analysis passes. The key idea is to remove from the given CFG all non-essential flow edges and nodes so that finding *U-Ins*$_{\mathcal{CO}}$ amounts to finding a minimum cut in a flow network (called essential flow graph (EFG)) thus obtained.

Algorithm MC-PRE$_{\text{comp}}$
INPUT: a CFG $G = (N, E, W)$ and an expression π
OUTPUT: a computationally optimal transformation $\mathcal{CO}$

1 $\mathit{D\text{-}Rep}_{\mathcal{CO}} = \mathit{DB}$.
2 $\mathit{D\text{-}Ins}_{\mathcal{CO}} = \mathit{DB}$.
3 $\mathit{U\text{-}Rep}_{\mathcal{CO}} = \mathit{UB}$.
4 Construct $\mathit{U\text{-}Ins}_{\mathcal{CO}}$ below.
4.1 Perform the availability and partial anticipability analyses.
(a) Solve the forward availability system (initialized to true):

$$\begin{array}{lcl} \mathsf{N\text{-}AVAL}(n) & = & \begin{cases} \mathsf{false} & \text{if } n \text{ is the entry block } s \\ \bigwedge_{m \in \mathsf{pred}(G,n)} \mathsf{X\text{-}AVAL}(m) & \text{otherwise} \end{cases} \\ \mathsf{X\text{-}AVAL}(n) & = & \mathsf{AVLOC}(n) \vee (\mathsf{N\text{-}AVAL}(n) \wedge \mathsf{TRANSP}(n)) \end{array}$$

(b) Solve the backward partial anticipability system (initialized to false):

$$\begin{array}{lcl} \mathsf{X\text{-}PANT}(n) & = & \begin{cases} \mathsf{false} & \text{if } n \text{ is the exit block } t \\ \bigvee_{m \in \mathsf{succ}(G,n)} \mathsf{N\text{-}PANT}(m) & \text{otherwise} \end{cases} \\ \mathsf{N\text{-}PANT}(n) & = & \mathsf{ANTLOC}(n) \vee (\mathsf{X\text{-}PANT}(n) \wedge \mathsf{TRANSP}(n)) \end{array}$$

4.2 Obtain a reduced graph $G_{rd} = (N_{rd}, E_{rd}, W_{rd})$ from G.
(a) Define the four predicates on the flow edges $(m, n) \in E$ (no flow analysis):

$$\begin{array}{rcl} \mathsf{INS\text{-}REDUND}(m,n) & =_{df} & \mathsf{X\text{-}AVAL}(m) \\ \mathsf{INS\text{-}USELESS}(m,n) & =_{df} & \neg\, \mathsf{N\text{-}PANT}(n) \\ \mathsf{NON\text{-}ESS}(m,n) & =_{df} & \mathsf{X\text{-}AVAL}(m) \vee \neg \mathsf{N\text{-}PANT}(n) \\ \mathsf{ESS}(m,n) & =_{df} & \neg \mathsf{NON\text{-}ESS}(m,n) \equiv \neg \mathsf{X\text{-}AVAL}(m) \wedge \mathsf{N\text{-}PANT}(n) \end{array}$$

(b) G_{rd} is defined as follows:

$$\begin{array}{lcl} N_{rd} & = & \{n \in N \mid \exists\, m \in N : \mathsf{ESS}(m,n) \vee \exists\, m \in N : \mathsf{ESS}(n,m)\} \\ E_{rd} & = & \{(m,n) \in E \mid \mathsf{ESS}(m,n)\} \\ W_{rd} & = & W \text{ restricted to the domain } E_{rd} \end{array}$$

4.3 Obtain a multi-source, multi-sink graph $G_{mm} = (N_{mm}, E_{mm}, W_{mm})$ from G_{rd}.
(a) Split a node such that its top part becomes a sink and bottom part a source:
$\mathsf{TOP}(n) =_{df} \mathsf{ANTLOC}(n) \wedge (\exists\, m \in \mathsf{pred}(G_{rd}, n) : (m,n) \in E_{rd})$
$\mathsf{BOT}(n) =_{df} \mathsf{KILL}(n) \wedge (\exists\, m \in \mathsf{succ}(G_{rd}, n) : (n,m) \in E_{rd})$
Let $\mathit{top\_part}(n) =$ **if** $\mathsf{TOP}(n) \wedge \mathsf{BOT}(n)$ $\rightarrow$ $n+$ $\square$ **else** $\rightarrow$ n **fi**
Let $\mathit{bot\_part}(n) =$ **if** $\mathsf{TOP}(n) \wedge \mathsf{BOT}(n)$ $\rightarrow$ $n-$ $\square$ **else** $\rightarrow$ n **fi**
Let $\Gamma(m,n) = (\mathit{bot\_part}(m), \mathit{top\_part}(n))$
(b) G_{mm} is defined as follows:

$$\begin{array}{lcl} N_{mm} & = & \{\mathit{bot\_part}(n) \mid n \in N_{rd}\} \cup \{\mathit{top\_part}(n) \mid n \in N_{rd}\} \\ E_{mm} & = & \{\Gamma(m,n) \mid (m,n) \in E_{rd}\} \\ W_{mm} & = & E_{mm} \mapsto \mathbb{N}, \text{ where } W(e) = W(\Gamma^{-1}(e)) \end{array}$$

(c) $S_{mm} = \{n \in N_{mm} \mid |\mathsf{pred}(G_{mm}, n)| = 0\}$
$T_{mm} = \{n \in N_{mm} \mid |\mathsf{succ}(G_{mm}, n)| = 0\}$
4.4 Obtain a single-source, single-sink EFG $G_{st} = (N_{st}, E_{st}, W_{st})$ from G_{mm}.
(a) Let s' be a new entry block and t' a new exit block.
(b) G_{st} is defined as follows:

$$\begin{array}{lcl} N_{st} & = & N_{mm} \cup \{\{s', t'\} \mid N_{mm} \neq \emptyset\} \\ E_{st} & = & E_{mm} \cup \{(s', n) \mid n \in S_{mm}\} \cup \{(n, t') \mid n \in T_{mm}\} \\ W_{st} & = & W_{mm} \text{ (extended to } E_{st}) \text{ such that } \forall\, e \in (E_{st} - E_{mm}) : W_{st}(e) = \infty. \end{array}$$

4.5 Find a minimum cut.
(a) $\mathcal{C} = \mathrm{MIN\_CUT}(G_{st})$.
(b) $\mathit{U\text{-}Ins}_{\mathcal{CO}} = \Gamma^{-1}(\mathcal{C})$ // maps the edges in $\mathcal{C}$ back to flow edges in G

Fig. 10. An algorithm that guarantees computational optimal results.

As discussed earlier in Section 1, we do not actually create such an EFG directly on the original CFG. Instead, we generate a specification of the EFG and feed it to a min-cut solver so that a minimum cut can be found very efficiently.

We describe below Steps 4.1 – 4.5 for constructing $U\text{-}Ins_{\mathcal{CO}}$ and illustrate them with two example CFGs. The first CFG does not use Step 4.3(a) while the second CFG is designed specifically to illustrate the necessity of this step.

Our first example is the CFG discussed earlier in Figure 1(a). The expression π under consideration is $a + b$. For convenience, let us duplicate (1) as follows:

$$\begin{array}{rcl} UB & = & \{2, 4, 10, 11, 13\} \\ DB & = & \{6, 11\} \end{array} \tag{13}$$

By executing the first three steps of MC-PRE$_{\text{comp}}$, we get trivially:

$$\begin{array}{rcccl} D\text{-}Rep_{\mathcal{CO}} & = & DB & = & \{6, 11\} \\ D\text{-}Ins_{\mathcal{CO}} & = & DB & = & \{6, 11\} \\ U\text{-}Rep_{\mathcal{CO}} & = & UB & = & \{2, 4, 10, 11, 13\} \end{array} \tag{14}$$

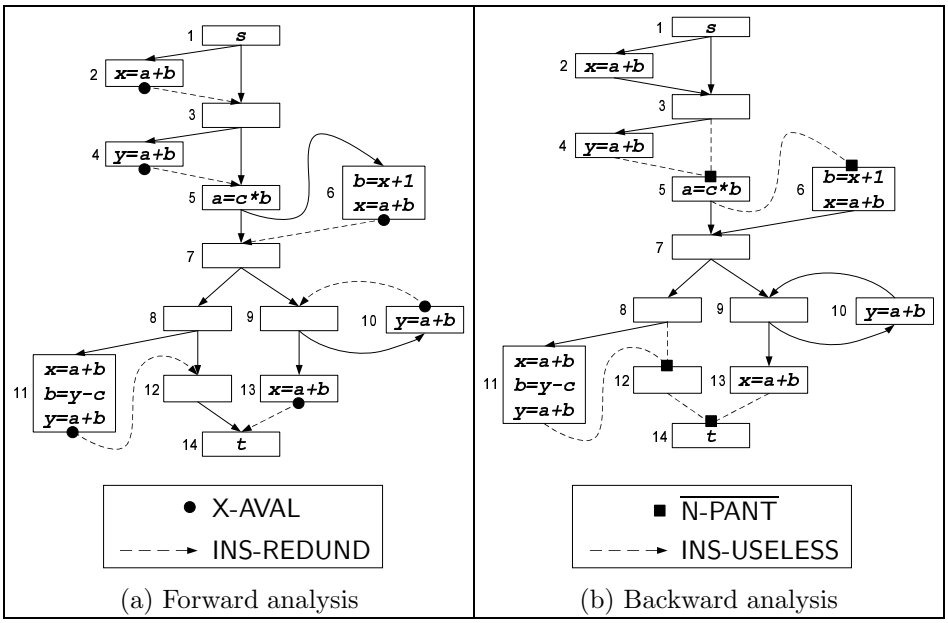
The construction of $U\text{-}Ins_{\mathcal{CO}}$ for this example is done incrementally below.

4.1.1 *Step 4.1: Perform the Availability and Partial Anticipability Analyses.* These are the standard data-flow analyses used in compiler optimizations. In Step 4.1(a), the availability system for an expression π is solved based on the two local predicates AVLOC and TRANSP (Section 3). The two global predicates N-AVAL and X-AVAL on blocks are defined as follows. N-AVAL(n) denotes the availability of π on entry of a block n and X-AVAL(n) the same property at the exit of the same block. An expression is available on entry to a block n if it is available on exit from each predecessor of the block. An expression is available on exit from a block n if it is locally available or if it is available on entry of and transparent at the block.

In Step 4.1(b), the partial anticipability system for an expression π is solved based on the two local predicates ANTLOC and TRANSP (Section 3). There are also two global predicates N-PANT and X-PANT defined on blocks. N-PANT(n) denotes the partial anticipability of π on entry of a block n and X-PANT(n) the same property at the exit of the same block. An expression is partially anticipatable on exit from the block n if it is partially anticipatable on entry of at least one successor of the block. An expression is partially anticipatable on entry to a block n if it is locally anticipatable or if it is partially anticipatable at the exit of and transparent at the block.

4.1.2 *Step 4.2: Obtain a Reduced Graph.* Once the two global flow analyses have been carried out, the four global predicates are defined on the flow edges of G in Step 4.2(a). These predicates classify the flow edges in their order of appearance into *insertion-redundant*, *insertion-useless*, *non-essential* and *essential* edges. The forward availability analysis detects the insertion-redundant edges while the backward partial anticipability analysis detects the insertion-useless edges.

The concept of essentiality for flow edges induces a similar concept for nodes. A node n in G is *essential* if at least one of its incident edges is essential and *non-essential* otherwise. In Step 4.2(b), the reduced graph G_{rd} consists of simply all essential edges in E and all essential nodes in N from the original graph G.



(a) Forward analysis (b) Backward analysis

Fig. 11. Identification of non-essential (and essential) edges in a CFG.

Figure 11 explains this step using our running example. Figure 11(a) illustrates the exit-availability and the insertion-redundant predicate, where the six insertion-redundant edges $(2,3),(4,5),(6,7),(10,9)$, $(11,12)$ and $(13,14)$ are depicted in dashes. Figure 11(b) illustrates the entry-partial-anticipability and the insertion-useless predicate, where the seven insertion-useless edges $(3,5),(4,5),(5,6),(8,12)$, $(11,12),(12,14)$ and $(13,14)$ are depicted in dashes. Note that some non-essential edges, such as $(4,5),(11,12)$ and $(13,14)$, are both insertion-redundant and insertion-useless. It is not difficult to see from this example that an insertion-redundant edge is so named because an insertion $h_\pi = \pi$ on the edge is redundant since the value of h_π is already available on the edge. Similarly, an insertion-useless edge is so named because an insertion $h_\pi = \pi$ on the edge can never make the value of h_π available to any computation of π in the program.

By combining Figure 11(a) and Figure 11(b), we get Figure 1(b), where all non-essential edges and nodes are depicted in dashes. The reduced graph for the running example can be found in Figure 1(c).

The following lemmas are immediate from the construction of G_{rd}.

LEMMA 4.1. *A U-block $n \in UB$ is in G_{rd} if the upwards exposed computation of π in block n is not fully redundant, i.e., $\exists\ m \in \mathsf{pred}(G,n) : \mathsf{X\text{-}AVAL}(m) = \mathsf{false}$.*

LEMMA 4.2. *A D-block $n \in DB - UB$ cannot be contained in G_{rd}.*

Both lemmas can be verified for our running example by noting the *UB* and *DB* given in (1) and examining the reduced graph shown in Figure 1(c). By Lemma 4.1, all U-blocks are in G_{rd}. By Lemma 4.2, the D-block 6 is not in G_{rd}.

THEOREM 4.3. *$G_{rd} = (N_{rd}, E_{rd}, W_{rd})$ is empty iff $\forall\, n \in UB : \forall\, m \in \mathsf{pred}(G, n) : \mathsf{X\text{-}AVAL}(m) = \mathsf{true}$, i.e., all upwards exposed computations are fully redundant.*

PROOF. To prove "$\Longrightarrow$", we note that by Lemma 4.1, if $\exists\, n \in UB : \exists\, m \in \mathsf{pred}(G, n) : \mathsf{X\text{-}AVAL}(m) = \mathsf{false}$, then n must be in G_{rd}. This means that $G_{rd} \neq \emptyset$, contradicting the proof hypothesis. To prove "$\Longleftarrow$", we assume to the contrary that $G_{rd} \neq \emptyset$. Then there must exist one essential edge (u, v) such that $\mathsf{X\text{-}AVAL}(u) = \mathsf{false} \wedge \mathsf{N\text{-}PANT}(v) = \mathsf{true}$. This implies immediately that $\exists\, n \in UB : \exists\, m \in \mathsf{pred}(G, n) : \mathsf{X\text{-}AVAL}(m) = \mathsf{false}$, contradicting the proof hypothesis. □

4.1.3 *Step 4.3: Obtain a Multi-Source, Multi-Sink Graph.* In this step, the objective is to create from the reduced graph G_{rd} a multi-source, multi-sink flow network, where the set of sources is *disjoint* from the set of sinks [Cormen et al. 1990]. One minor complication is that some nodes of G_{rd} with both incoming and outgoing edges must be split to function as both sources and sinks, respectively.

Based on the predicates TOP and BOT defined in Step 4.3(a), a node n in G_{rd} is split when both $\mathsf{TOP}(n) = \mathsf{true}$ and $\mathsf{BOT}(n) = \mathsf{true}$. A block is split only if it is a U-block that is not also a D-block. ($\forall\, n \in DB : \mathsf{BOT}(n) = \mathsf{false}$.) Such a node contains some modification to π, which effectively "kills" or "blocks" the value reuse of π across the node. In this step, we simply split every such a node n into two new nodes $n+$ and $n-$ so that $n+$ will serve as a sink (without outgoing edges) and $n-$ as a source (without incoming edges). After the splitting, the incoming edges of n are all directed into $n+$ and the outgoing edges of n are all directed out of $n-$. There are no edges between $n+$ and $n-$. This splitting process leads to the resulting graph G_{mm} defined in Step 4.3(b). If a node n is not split, then $\mathit{top\_part}(n) = \mathit{bot\_part}(n) = n$. In 4.3(c), S_{mm} consists of all *source nodes* (without incoming edges) and T_{mm} all *sink nodes* (without outgoing edges) in G_{mm}.

The following three lemmas are immediate from the construction of G_{rd} and G_{mm}. Lemma 4.4 asserts that G_{mm} is a multi-source, multi-sink flow network. Lemmas 4.5 and 4.6 expose the structure of its source and sink nodes, respectively.

LEMMA 4.4. $S_{mm} \cap T_{mm} = \emptyset$.

LEMMA 4.5. $\forall\, n \in N_{rd} : \mathsf{KILL}(n) \iff \mathit{bot\_part}(n) \in S_{mm}$.

LEMMA 4.6. $\forall\, n \in N_{rd} : \mathsf{ANTLOC}(n) \iff \mathit{top\_part}(n) \in T_{mm}$.

This step is irrelevant for our running example since no splitting as described in Step 4.3(a) takes place. Thus, $\mathit{top\_part}(n) = \mathit{bot\_part}(n) = n$ holds for every node n. This means that $G_{mm} = G_{rd}$. That is, the resulting multi-source, multi-sink graph is exactly the same as the reduced graph shown in Figure 1(c). In Step 4.3(c), we obtain $S_{mm} = \{1, 5\}$ and $T_{mm} = \{2, 4, 10, 11, 13\}$. By noting Assumption 3.5, the facts stated in Lemmas 4.4 – 4.6 can be easily verified.

This step will be illustrated in Section 4.1.6 by an example given in Figure 12.

4.1.4 *Step 4.4: Obtain a Single-Source, Single-Sink EFG.* This is a standard transformation [Cormen et al. 1990]. In 4.4(a), the new entry node s' and new exit node t' are added. In 4.4(b), we obtain the single-source, single-sink EFG G_{st}, in which all new edges introduced have the weight ∞. Hence, G_{st} is a s-t flow network.

In our running example, the multi-source, multi-sink graph G_{mm} is the same as the reduced graph G_{rd} depicted in Figure 1(c). Thus, $S_{mm} = \{1, 5\}$ and $T_{mm} = \{2, 4, 10, 11, 13\}$. Figure 1(d) depicts the resulting EFG G_{st}.

4.1.5 *Step 4.5: Find a Minimum Cut.* *U-Ins*$_{\mathcal{CO}}$ is chosen to be *any* minimum cut on the EFG G_{st} by applying a min-cut algorithm. The G_{st} for our running example is depicted in Figure 1(d). There are two minimum cuts. If we choose

$$C_1 \;=\; \{(1,2),(3,4),(5,7)\} \tag{15}$$

we find that

$$\textit{U-Ins}_{\mathcal{CO}} \;=\; \Gamma^{-1}(C_1) \;=\; C_1 = \{(1,2),(3,4),(5,7)\} \tag{16}$$

since $G_{mm} = G_{rd}$. The resulting transformation $\mathcal{CO}$ defined by (14) and (16) is the same as $\mathcal{CO}_1$ given in (3). This code motion results in the transformed code shown in Figure 1(e). The number of computations required for $a + b$ is $W(\mathcal{CO}) = 1900$, which is the smallest possible with respect to the profile W. If we choose

$$C_2 \;=\; \{(1,2),(1,3),(5,7)\} \tag{17}$$

instead, *U-Ins*$_{\mathcal{CO}}$ will become:

$$\textit{U-Ins}_{\mathcal{CO}} \;=\; \Gamma^{-1}(C_2) \;=\; C_2 = \{(1,2),(1,3),(5,7)\} \tag{18}$$

Let us combine (14) and (18) and denote this transformation by $\mathcal{CO}_2$:

$$\begin{array}{rcccl} \textit{U-Rep}_{\mathcal{CO}2} & = & \textit{UB} & = & \{2,4,10,11,13\} \\ \textit{U-Ins}_{\mathcal{CO}_2} & = & C_2 & = & \{(1,2),(2,3),(5,7)\} \\ \textit{D-Rep}_{\mathcal{CO}_2} & = & \textit{DB} & = & \{6,11\} \\ \textit{D-Ins}_{\mathcal{CO}_2} & = & \textit{DB} & = & \{6,11\} \end{array} \tag{19}$$

The transformed code is the same as in Figure 1(e) except the insertion $h = a + b$ made on the edge $(3, 4)$ before should now be made on the edge (1,3).

4.1.6 *One More Example.* We illustrate Step 4.3 of MC-PRE$_{\text{comp}}$ using a simple example presented in Figure 12. For the CFG given in Figure 12(a), $(2, 3)$ and $(6, 7)$ are the only non-essential edges. So the reduced graph G_{rd} is the one displayed in Figure 12(b). In Step 4.3(a), we obtain *top_part*(n) = *bot_part*$(n) = n$ for $n \in \{1, 2, 3, 4, 6\}$, *top_part*$(5) = 5+$ and *bot_part*$(5) = 5-$. That is, block 5 is split into 5+ and 5− to produce in Step 4.3(b) the multi-source, multi-sink graph G_{mm} depicted in Figure 12(c). The corresponding single-source, single-sink EFG G_{st} is shown in Figure 12(d). In Step 4.3(c), we obtain $S_{mm} = \{1, 5-\}$ and $T_{mm} = \{2, 5+, 6\}$. In this simple case, the unique minimum cut found in Step 4.5(a) for the EFG G_{st} is $\mathcal{C} = \{(1, 2), (1, 3), (5-, 6)\}$. In Step 4.5(b), we map these edges back to the flow edges in the original CFG: *U-Ins*$_{\mathcal{CO}} = \Gamma^{-1}(\mathcal{C}) = (1, 2), (1, 3), (5, 6)\}$. This gives rise to the following transformation:

$$\begin{array}{rcl} \textit{U-Ins}_{\mathcal{CO}} & = & \{(1,2),(1,3),(5,6)\} \\ \textit{U-Rep}_{\mathcal{CO}} & = & \{2,5,6\} \\ \textit{D-Ins}_{\mathcal{CO}} & = & \emptyset \\ \textit{D-Rep}_{\mathcal{CO}} & = & \emptyset \end{array} \tag{20}$$

It is not difficult to verify that this solution is both computationally optimal (and lifetime optimal). The transformed code is omitted, achieving *benefit*$(\mathcal{CO}) = 200$.

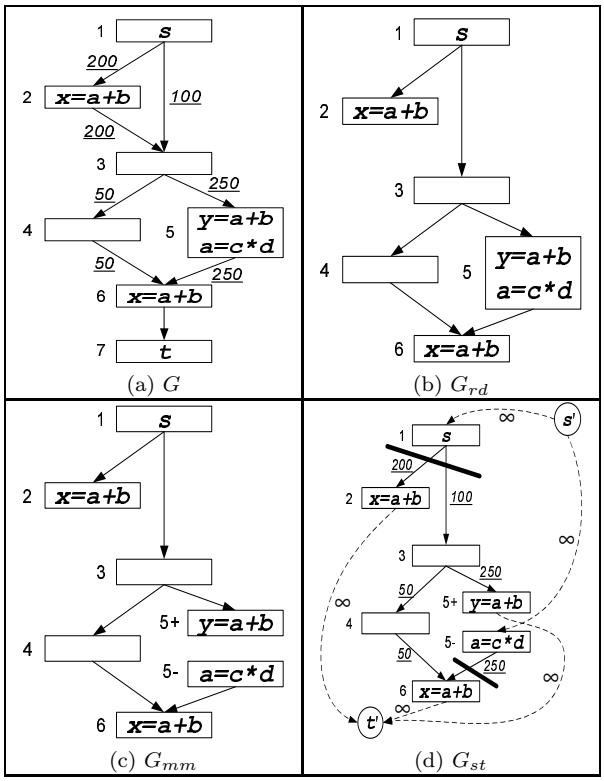


Fig. 12. An example illustrating Step 4.3 of MC-PRE$_{\text{comp}}$.

Finally, the reader may care to verify Lemmas 4.4 – 4.6 for this example.

4.2 Lifetime Optimal Transformations

This section describes our MC-PRE algorithm for finding a lifetime optimal transformation, denoted $\mathcal{LO}$, in $G = (N, E, W)$. It turns out that we must design a practical strategy to deal with the flow edges with zero frequencies. It is understood that the profiling information input to our algorithm provides only an approximation of the actual edge execution frequencies. When constructing $\mathcal{LO}$, we do not make any insertions on the insertion-redundant edges (since they are not in the reduced graph G_{rd}). Effectively, we interpret an insertion-redundant edge with a zero frequency as the least frequently rather than never executed. By Theorem 5.6, no insertions can be made on these edges in any computationally optimal transformation. As a consequence, every partially redundant computation (made redundant along an insertion-redundant incoming edge) is eliminated. We use two examples to motivate such a strategy behind the development of MC-PRE.

Consider the first example given in Figure 13. For the CFG shown in Figure 13(a), the edge $(2, 4)$ is insertion-redundant but $W(2, 4) = 0$. The $\mathcal{LO}$ solution found by MC-PRE is illustrated in Figure 13(b), which is also the one found by LCM. So both algorithms aim at eliminating all the redundancies available when the actual

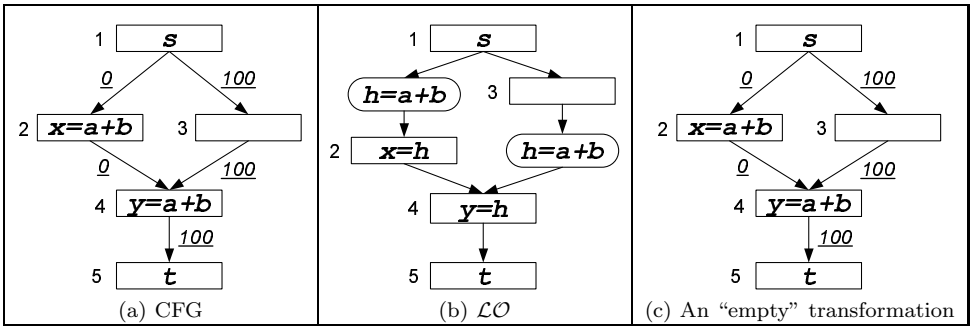


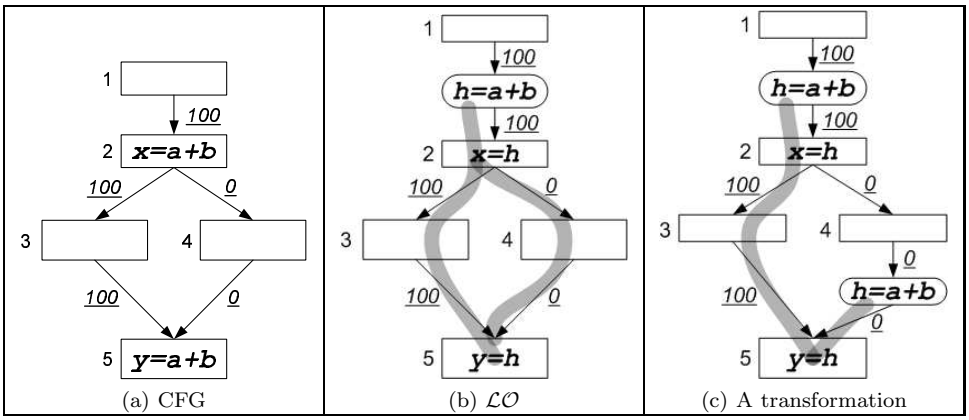
Fig. 13. An illustration about practical handling of zero-weighted insertion-redundant edges.

frequency of $(2, 4)$ turns out to be nonzero, If (2,4) is considered as not executed at all, then doing nothing (as shown in Figure 13(c)) would be lifetime optimal.

Consider the second example given in Figure 14. For the CFG shown in Figure 14(a), all edges except (1,2) are insertion-redundant. The $\mathcal{LCO}$ solution is illustrated in Figure 14(b), which is also the one found by LCM. Both algorithms make no insertions on the insertion-redundant edges. However, the solution shown in Figure 14(c) is lifetime optimal (with respect to the given profiling information). But such a solution is undesirable due to the "redundant" insertion made on edge (4,5), which serves only to kill the earlier definition of h along the path $\langle 1, 3, 4\rangle$.

Both examples are designed to demonstrate that it is not practical to find the so-called "true" lifetime optimal transformation for a PRE problem. In this case, the edge frequencies are interpreted as 100% accurate. So the zero-weighted edges or blocks are considered as never being executed at all. That being the case, we can simply remove all the zero-weighted edges and blocks in a CFG. Then the $\mathcal{LCO}$ solution found by our MC-PRE algorithm is guaranteed to be also lifetime optimal (Theorem 5.10). In practice, the profiling information is only approximate. By making no insertions on the insertion-redundant edges, we aim at not only eliminating the partial redundancies that might occur during program execution as illustrated in Figure 13 but also avoiding the kind of undesirable insertions as illustrated in Figure 14. Finally, if the frequencies of all flow edges in $G = (N, E, W)$ are nonzero, i.e., if $W(E) > 0$, we guarantee that $\mathcal{CM}_{LifeOpt} = \{\mathcal{LCO}\}$ (Theorem 5.10). Otherwise, $\mathcal{LCO}$ represents a practically desirable solution that has been validated by extensive experiments (Section 6).

We are ready to present our MC-PRE algorithm. The minimum cuts in a flow network may not be unique. This implies that a PRE problem may have more than one computationally optimal transformations. That is, $|\mathcal{CM}_{CompOpt}| > 1$ in general. By Definition 3.8, different solutions in $\mathcal{CM}_{CompOpt}$ may have different lifetimes. When finding a lifetime optimal solution, we must also avoid making unnecessary code motion for isolated computations as we discussed earlier in Section 1. Let S_{cut} be the set of all minimum cuts that can be possibly found in Step 4.5(a) of MC-PRE$_{\text{comp}}$ for a PRE problem. Let $\mathcal{T}_{cut}$ be the set of all corresponding



(a) CFG (b) $\mathcal{LCO}$ (c) A transformation

Fig. 14. A further illustration about practical handling of zero-weighted insertion-redundant edges.

computationally optimal transformations:

$$\mathcal{T}_{cut} = \left\{ \left(\begin{array}{rcl} \textit{U-Ins}_{\mathcal{CO}} & = & \Gamma^{-1}(C), \\ \textit{U-Rep}_{\mathcal{CO}} & = & \textit{UB}, \\ \textit{D-Ins}_{\mathcal{CO}} & = & \textit{DB}, \\ \textit{D-Rep}_{\mathcal{CO}} & = & \textit{DB} \end{array} \right) \middle| \, C \in S_{cut} \right\} \tag{21}$$

It is possible that $\mathcal{CM}_{LifeOpt} \subseteq \mathcal{CM}_{CompOpt} - \mathcal{T}_{cut}$. So the lifetime best among all transformations in $\mathcal{T}_{cut}$ found by MC-PRE$_{\text{comp}}$ may not be necessarily lifetime optimal – some code motion may have been done unnecessarily. As discussed in Section 4.1.5, our running example has two minimum cuts in the EFG G_{st}:

$$\mathcal{T}_{cut} = \{\mathcal{CO}_1, \mathcal{CO}_2\} \tag{22}$$

where $\mathcal{CO}_1$ is defined in (3) and $\mathcal{CO}_2$ in (19). For this example, the lifetime optimal solution $\mathcal{LCO}$ is the one given in (2) but $\mathcal{LCO} \notin \mathcal{T}_{cut}$.

Figure 15 gives our final algorithm, called MC-PRE, for finding a lifetime optimal transformation, denoted $\mathcal{LCO}$, for a program. MC-PRE has two main parts:

(1) First, we refine MC-PRE$_{\text{comp}}$ to find a unique minimum cut in G_{st}, by applying the "Reverse" Labelling Procedure of [Ford and Fulkerson 1962]. The corresponding computationally optimal transformation is the lifetime best among all transformations in $\mathcal{T}_{cut}$, but it may not be lifetime optimal.
(2) Second, we perform a third data-flow analysis *in the original CFG* to identify all isolated computations so as to avoid making unnecessary code motion.

Based on the results from these two parts, the lifetime optimal transformation $\mathcal{LCO}$ is found. Unlike MC-PRE$_{\text{comp}}$, MC-PRE requires global data-flow analyses to find not only $\textit{U-Ins}_{\mathcal{LCO}}$ but also the other three sets $\textit{D-Ins}_{\mathcal{LCO}}$, $\textit{U-Rep}_{\mathcal{LCO}}$ and $\textit{D-Rep}_{\mathcal{LCO}}$.

We explain our algorithm using our running example shown in Figure 1. MC-PRE runs in eight steps. By executing Steps 1 – 4 exactly as in MC-PRE$_{\text{comp}}$, we obtain the EFG G_{st} as before. Below we describe its Steps 5 – 8 only and explain how the lifetime optimal solution $\mathcal{LCO}$ in (2) for our running example is derived. In

Algorithm MC-PRE
INPUT: a CFG $G = (N, E, W)$ and an expression π
OUTPUT: a lifetime optimal PRE transformation $\mathcal{LO}$

1. Perform the two data-flow analyses as in Step 4.1 of MC-PRE$_{\text{comp}}$.
2. Obtain $G_{rd} = (N_{rd}, E_{rd}, W_{rd})$ from G as in Step 4.2 of MC-PRE$_{\text{comp}}$.
3. Obtain $G_{mm} = (N_{mm}, E_{mm}, W_{mm})$ from G_{rd} as in Step 4.3 of MC-PRE$_{\text{comp}}$.
4. Obtain $G_{st} = (N_{st}, E_{st}, W_{st})$ from G_{mm} as in Step 4.4 of MC-PRE$_{\text{comp}}$.
5. Find a unique minimum cut in G_{st}.
   (a) Apply any min-cut algorithm to find a maximum flow f in G_{st}.
   (b) Let $G^f_{st} = (N_{st}, E^f_{st}, W^f_{st})$ be the residual network induced by the flow f [Cormen et al. 1990, p. 588], where
   $E^f_{st} = \{(u, v) \in E_{st} \mid W_{st}(u, v) - f(u, v) > 0\}$
   $W^f_{st} = E^f_{st} \mapsto \mathbb{N}$, where $W^f_{st}(u, v) = W_{st}(u, v) - f(u, v)$
   (c) Let $\overline{\Lambda} = \{n \in N_{st} \mid$ there exists a path from n to the sink t' in $G^f_{st}\}$.
   (d) Let $\Lambda = N_{st} - \overline{\Lambda}$.
   (e) Let $\mathcal{C}_\Lambda = (\Lambda, \overline{\Lambda})$.
   (f) Let $\mathcal{C}'_\Lambda = \Gamma^{-1}(\mathcal{C}_\Lambda)$ // the edges in $\mathcal{C}_\Lambda$ are mapped back to flow edges in G
6. Solve the backwards "live range analysis for h_π" in G (see Figure 8):

$$\mathsf{X\text{-}LIVE}(n) = \begin{cases} \mathsf{false} & \text{if } n \text{ is the exit block } t \\ \bigvee_{m \in \mathsf{succ}(G,n)} \mathsf{N\text{-}LIVE}(m) \wedge ((n,m) \notin \mathcal{C}'_\Lambda) & \text{otherwise} \end{cases}$$
$$\mathsf{N\text{-}LIVE}(n) = \mathsf{ANTLOC}(n) \vee (\mathsf{X\text{-}LIVE}(n) \wedge \mathsf{TRANSP}(n))$$

7. Construct *D-Ins*$_{\mathcal{LO}}$ and *D-Rep*$_{\mathcal{LO}}$ as follows:
   (a) $\mathsf{D\text{-}ISOLATED}_\Lambda(n) =_{df} \neg\mathsf{X\text{-}LIVE}(n)$
   (b) Let *D-Ins*$_{\mathcal{LO}} = \{n \in DB \mid \neg\mathsf{D\text{-}ISOLATED}_\Lambda(n)\}$.
   (c) Let *D-Rep*$_{\mathcal{LO}} = \{n \in DB \mid \neg\mathsf{D\text{-}ISOLATED}_\Lambda(n)\}$.
8. Construct *U-Ins*$_{\mathcal{LO}}$ and *U-Rep*$_{\mathcal{LO}}$ as follows:
   (a) $\mathsf{U\text{-}ISOLATED}_\Lambda(n) =_{df} \big(\mathsf{KILL}(n) \vee \neg\mathsf{X\text{-}LIVE}(n)\big) \wedge \forall\, m \in \mathsf{pred}(G, n) : (m, n) \in \mathcal{C}'_\Lambda)$
   (b) Let *U-Rep*$_{\mathcal{LO}} = \{n \in UB \mid \neg\mathsf{U\text{-}ISOLATED}_\Lambda(n)\}$.
   (c) Let *U-Ins*$_{\mathcal{LO}} = \{(m, n) \in \mathcal{C}'_\Lambda \mid \neg\mathsf{U\text{-}ISOLATED}_\Lambda(n)\}$.

Fig. 15. An algorithm that guarantees lifetime optimal results.

Section 4.2.5, we discuss why MC-PRE chooses not to perform any code motion for Figure 5. The proofs for optimality and others are deferred to Section 5.2.

4.2.1 *Step 5: Find a Unique Minimum Cut.* By applying essentially the "Reverse" Labelling Procedure of Ford and Fulkerson [1962] in Steps 5(b) – 5(e), we find the unique minimum cut $\mathcal{C}_\Lambda = (\Lambda, \overline{\Lambda})$ in G_{st}. In Lemma 5.9, we show that $\mathcal{C}_\Lambda$ is unique and thus invariant of the maximum flow f found in Step 5(a). In Step 5(f), $\mathcal{C}'_\Lambda$ contains these cut edges in $\mathcal{C}_\Lambda$ but mapped back to the original CFG.

Let $\mathcal{ALO}$ be the following computationally optimal transformation:

$$\begin{aligned} \textit{U-Ins}_{\mathcal{ALO}} &= \mathcal{C}'_\Lambda = \Gamma^{-1}(\mathcal{C}_\Lambda) \\ \textit{U-Rep}_{\mathcal{ALO}} &= UB \\ \textit{D-Ins}_{\mathcal{ALO}} &= DB \\ \textit{D-Rep}_{\mathcal{ALO}} &= DB \end{aligned} \tag{23}$$

$\mathcal{ALO}$ is the lifetime best among all transformations in S_{cut}. This result is not

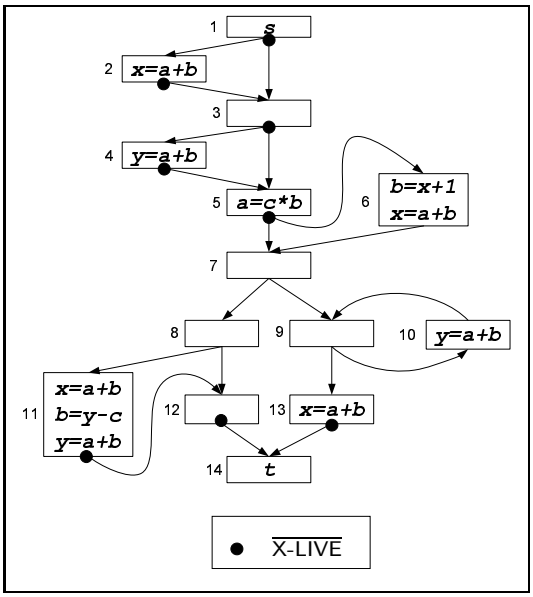


Fig. 16. Live range analysis for h when interpreted as being done in the transformed CFG shown in Figure 1(e).

directly proved in this paper since $\mathcal{ALCO}$ is not actually used; but it is implied by the proof of Theorem 5.10 (when the stated condition there is met).

In the case of our running example, there are only two minimum cuts C_1 and C_2, which are given in (15) and (17), respectively. This step will set $\mathcal{C}_\Lambda = C_1$. Hence,

$$\mathcal{C}'_\Lambda = \Gamma^{-1}(\mathcal{C}_\Lambda) = \mathcal{C}_\Lambda = C_1 = \{(1,2),(3,4),(5,7)\}$$

The computationally optimal transformations $\mathcal{CO}_1$ and $\mathcal{CO}_2$ corresponding to the two minimum cuts C_1 and C_2 can be found in (3) and (19), respectively. Accordingly, $\mathcal{ALCO} = \mathcal{CO}_1$ since $\mathcal{CO}_1$ is lifetime better than $\mathcal{CO}_2$.

Note that $\mathcal{ALCO}$ defined above is the analogue of the almost-lifetime optimal transformation introduced in [Knoop et al. 1994]).

4.2.2 *Step 6: Solve the "Backward Live Range Analysis for h_π" in the Original CFG G.* We perform a standard data-flow analysis *in the original CFG* in order to identify all the isolated computations in Steps 7 – 8. Equivalently, we were actually performing a backward live range analysis for the introduced temporary h_π in the transformed CFG obtained by applying $\mathcal{ALCO}$ to the original CFG.

For our running example, the transformed CFG according to $\mathcal{ALCO} = \mathcal{CO}_1$ can be found in Figure 1(e). Figure 16 illustrates the exit-liveness predicate for the temporary h introduced in this transformed CFG.

4.2.3 *Step 7: Construct D-Ins$_{\mathcal{LCO}}$ and D-Rep$_{\mathcal{LCO}}$.* The downwards exposed computation for π in a D-block n is *isolated* if $\mathsf{D\text{-}ISOLATED}_\Lambda(n) = \mathsf{true}$. i.e. $\mathsf{X\text{-}LIVE}(n) = \mathsf{false}$. The insertion $h_\pi = \pi$ before such an isolated computation is unnecessary since the saved value in h_π is reused only by this computation. In comparison with

Steps 1 – 2 of MC-PRE$_{\text{comp}}$, MC-PRE performs the insertions and associated replacements only for non-isolated computations. In our running example, there are two D-blocks: $DB = \{6, 11\}$. From Figure 16, we see that $\mathsf{X\text{-}LIVE}(6) = \mathsf{true}$ and $\mathsf{X\text{-}LIVE}(11) = \mathsf{false}$. Thus, this this step produces the following two sets:

$$\textit{D-Ins}_{\mathcal{LCO}} \;=\; \textit{D-Rep}_{\mathcal{LCO}} \;=\; \{6\} \tag{24}$$

4.2.4 *Step 8: Construct U-Ins$_{\mathcal{LCO}}$ and U-Rep$_{\mathcal{LCO}}$.* The upwards exposed computation for π in a U-block $n \in UB$ is said to be *isolated* if $\mathsf{U\text{-}ISOLATED}_{\Lambda}(n) = \mathsf{true}$, where the predicate $\mathsf{U\text{-}ISOLATED}_{\Lambda}$ is defined in Step 8(a). In this case, the insertions on all the incoming edges of n should be avoided since the saved value on each edge will be reused only by the upwards exposed computation in block n.

Continuing our running example, where $UB = \{2, 4, 10, 11, 13\}$, we observe from Figure 16 that $\mathsf{U\text{-}ISOLATED}_{\Lambda}$ holds for blocks 2 and 4. Since the $a + b$ in block 2 and the $a + b$ in block 4 are isolated, we obtain in Steps 8(b) and (c):

$$\begin{array}{rcl} \textit{U-Ins}_{\mathcal{LCO}} & = & \{(5, 7)\} \\ \textit{U-Rep}_{\mathcal{LCO}} & = & \{10, 11, 13\} \end{array} \tag{25}$$

By combining (24) and (25), the lifetime optimal solution $\mathcal{LCO}$ in (2) is obtained. The transformed code that we have examined a few times is shown in Figure 1(f).

4.2.5 *One More Example.* We shall prove in Theorem 5.11 that MC-PRE performs some insertions and replacement iff the overall benefit is positive (when the frequencies of all flow edges are nonzero), which cannot be guaranteed by MC-PRE$_{\text{comp}}$. Thus, eliminating unnecessary insertions has a practical consequence. Consider the example in Figure 5. We mentioned in Section 1 that MC-PRE will not perform any code motion for the example. It is not difficult to see that the almost-lifetime optimal transformation is:

$$\begin{array}{rcl} \textit{U-Ins}_{\mathcal{ALCO}} & = & \{(1, 2), (4, 5)\} \\ \textit{U-Rep}_{\mathcal{ALCO}} & = & \{2, 5\} \\ \textit{D-Ins}_{\mathcal{ALCO}} & = & \emptyset \\ \textit{D-Rep}_{\mathcal{ALCO}} & = & \emptyset \end{array} \tag{26}$$

The two computations of $a + b$ are both isolated. Hence, we have:

$$\textit{U-Ins}_{\mathcal{LCO}} = \textit{U-Rep}_{\mathcal{LCO}} = \textit{D-Ins}_{\mathcal{LCO}} = \textit{D-Rep}_{\mathcal{LCO}} = \emptyset \tag{27}$$

4.3 Time and Space Complexity

We are concerned only with MC-PRE since it is the algorithm that we have implemented in `gcc`. Its overall time complexity is dominated by the three uni-directional data-flow analysis passes performed in Steps 1 and 6 and the min-cut algorithm employed in Step 5. The three passes can be done in parallel using bit vectors for all expressions in a CFG. However, the min-cut algorithm operates on each expression separately (at least so in our current implementation). When MC-PRE is applied to each expression in a CFG $G = (N, E, W)$ individually, the worst-case time complexity for each bit-vector pass is $O(|N| \times (d+2))$, where d is the maximum number of back edges on any acyclic path in G and typically $d \leqslant 3$ [Muchnick 1997].

The min-cut step of MC-PRE operates on the EFG $G_{st} = (N_{st}, E_{st}, W_{st})$. There are a variety of polynomial min-cut algorithms in the literature with different time

complexities [Chekuri et al. 1997]. In our implementation, we have used Goldberg's *push-relabel* HIPR algorithm since it has been reported to be efficient with its worst-time complexity being $O(|N_{st}|^2\sqrt{|E_{st}|})$ [GoldBerg 2003]. Hence, MC-PRE has a polynomial time complexity overall.

In our implementation, we do not actually modify the original CFG G at all in order to obtain G_{rd}, G_{mm} and G_{st}. Rather, we generate a graph description for G_{st} and feed it to the min-cut solver to find a minimum cut efficiently. The space requirement for representing G_{st} in the min-cut solver is $O(|N_{st}|)$.

In Section 6, we present experimental data using benchmark programs to show that MC-PRE is both efficient and effective in dealing with real applications.

5. THEORETICAL RESULTS

We develop our proofs (rigorously) also in two stages. Section 5.1 proves the computational optimality of the transformation $\mathcal{CO}$ found by MC-PRE$_{\text{comp}}$. Section 5.2 proves the lifetime optimality of the transformation $\mathcal{LO}$ found by MC-PRE.

We continue to focus on a PRE problem consisting of a CFG $G = (N, E, W)$ with respect to an expression π. Recall that $\mathcal{CM}_{Cor}$ denotes the set of correct transformations, $\mathcal{CM}_{CompOpt}$ the set of computationally optimal transformations and $\mathcal{CM}_{LifeOpt}$ the set of lifetime optimal transformations for the PRE problem.

5.1 Computational Optimality

Definition 5.1. A path $p(n_1, n_k) = \langle n_1, n_2, \ldots, n_k\rangle$ in $G = (N, E, W)$ is called a *kill-comp path* if the following three statements are true:

(1) n_1 is the only predecessor of n_k in the path $p(n_1, n_k)$ that is a kill block of π,

(2) n_1 is not a D-block, and

(3) n_k is the only successor of n_1 in the path $p(n_1, n_k)$ that is a U-block of π.

Statement (1) implies that $n_2, \ldots, n_{k-1}$ are not D-blocks. This definition can be better visualized as shown in Figure 17. In other words, $p(n_1, n_k)$ is a kill-comp path if, from the last modification to π in block n_1 to the upwards exposed computation of π in block n_k, there are no other modifications to and computations of π in the same path. Note that $n_1 = n_k$ is possible if $p(n_1, n_k)$ represents a cycle in G.

Lemma 5.2 shows that *U-Ins*$_{\mathcal{CO}}$ constructed by MC-PRE$_{\text{comp}}$ must contain one edge in every kill-comp path of π. Based on this fact, Lemma 5.3 proves the correctness of $\mathcal{CO}$ found by MC-PRE$_{\text{comp}}$. Lemma 5.4 shows that for every computationally optimal transformation $T \in \mathcal{CM}_{CompOpt}$, we can always derive from it another computationally optimal transformation $T' \in \mathcal{T}_{cut}$, where $\mathcal{T}_{cut}$ is defined in (21). A combination of these two lemmas leads to the establishment of the computational optimality of $\mathcal{CO}$ in Theorem 5.5.

LEMMA 5.2. *If $p(n_1, n_k) = \langle n_1, n_2, \ldots, n_k\rangle$ is a kill-comp path in $G = (N, E, W)$, then U-Ins$_{\mathcal{CO}}$ found by MC-PRE$_{\text{comp}}$ must include one edge from $p(n_1, n_k)$.*

PROOF. Figure 17 illustrates the exit-availability and entry-partial-anticipability of the end points of all the edges in the path $p(n_1, n_k)$. Note that the values of the two predicates X-AVAL and N-PANT are completely defined by examining *only* the nodes and edges in the path $p(n_1, n_k)$. By the definition of ESS in Step 4.2(a) in MC-PRE$_{\text{comp}}$, all the edges on this path are essential. As a consequence, all the

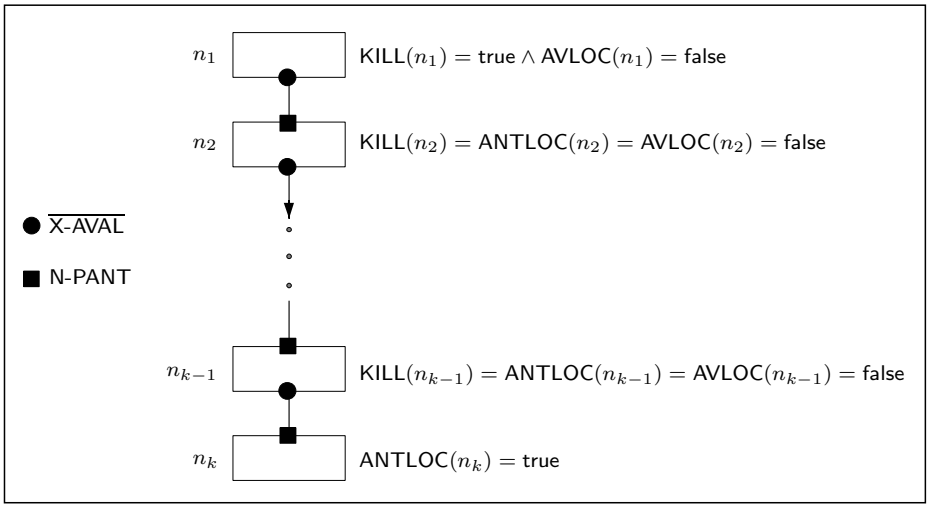


Fig. 17. An illustration of Definition 5.1 and the proof of Lemma 5.2.

nodes on the path are also essential. Hence, this path must be completely contained in the reduced graph G_{rd} constructed in Step 4.2(b) of MC-PRE$_{\text{comp}}$.

In G_{mm} constructed in Step 4.3 of MC-PRE$_{\text{comp}}$, the path $p(n_1, n_k)$ becomes $\langle bot\_part(n_1), n_2, \ldots, n_{k-1}, top\_part(n_k)\rangle$. To see why, we apply Definition 5.1 to find that $\mathsf{BOT}(n_1) = \mathsf{true}$, $\mathsf{BOT}(n_i) = \mathsf{false}$, for all $1 < i < k$, and $\mathsf{TOP}(n_k) = \mathsf{true}$, where TOP and BOT are defined in 4.3(a) of MC-PRE$_{\text{comp}}$. Clearly, in Step 4.3(a), the nodes $n_2, \ldots, n_{k-1}$ are not split, i.e., $top\_part(n_i) = bot\_part(n_i) = n_i$, for all $1 < i < k$. We show below that $bot\_part(n_1) \in S_{mm}$ and $top\_part(n_k) \in T_{mm}$.

For reasons of symmetry, we prove $bot\_part(n_1) \in S_{mm}$ only. If $bot\_part(n_1) = n_1-$, then $bot\_part(n_1) \in S_{mm}$ by construction. On the other hand, if $bot\_part(n_1) = n_1$, we must also have $|\mathsf{pred}(G_{mm}, n_1)| = 0$, which implies that $bot\_part(n_1) \in S_{mm}$. Assume to the contrary that $|\mathsf{pred}(G_{mm}, n_1)| \neq 0$. There are two cases. Case (a) $\mathsf{ANTLOC}(n_1) = \mathsf{true}$. Then $\mathsf{TOP}(n) = \mathsf{true}$, which contradicts the fact that $bot\_part(n_1) = n_1$ since $\mathsf{TOP}(n_1) \wedge \mathsf{BOT}(n_1) = \mathsf{true}$. Case (b) $\mathsf{ANTLOC}(n_1) = \mathsf{false}$. Then $\mathsf{N\text{-}PANT}(n_1) = \mathsf{false}$. Hence, all the incoming edges of n_1 are insertion-useless (i.e., non-essential). This contradicts the assumption that $|\mathsf{pred}(G_{mm}, n_1)| \neq 0$.

In Step 4.4 of MC-PRE$_{\text{comp}}$, we convert G_{mm} into the EFG G_{st} in the standard manner [Cormen et al. 1990, p. 584]. Thus, the minimum cut $\mathcal{C}$ found in Step 4.5(a) for G_{st} (that separates s' from t') is also a minimum cut in G_{mm} (that separates all the sources in S_{mm} from all the sinks in T_{mm}). This means that $\mathcal{C}$ contains an edge in the path $\langle bot\_part(n_1), n_2, \ldots, n_{k-1}, top\_part(n_k)\rangle$ of G_{mm}. By Step 4.5(b), *U-Ins*$_{\mathcal{CO}}$ must contain an edge in the path $p(n_1, n_k)$ of G. □

LEMMA 5.3. *$\mathcal{CO}$ found by MC-PRE*$_{\text{comp}}$ *is correct, i.e., $\mathcal{CO} \in \mathcal{CM}_{Cor}$.*

PROOF. We establish the correctness of $\mathcal{CO}$ by showing that it satisfies Properties P1 and P2 stated in Definition 3.6. By Steps 1 and 2 of MC-PRE$_{\text{comp}}$, Property P1 is trivially satisfied. Note that by Step 3 of MC-PRE$_{\text{comp}}$, we have *U-Rep*$_{\mathcal{CO}}$ = *UB*. To prove that $\mathcal{CO}$ also satisfies Property P2, we let $n \in UB$ be an arbitrary but

fixed U-block for expression π. Let $p(s,n)$ be an arbitrary but fixed path from the entry block s to the node n. By Assumption 3.5, $p(s,n)$ must contain at least one kill block for π, which could be the entry s. Let n_1 be the last kill block in the path $p(s,n)$ such that $n_1 \neq n$. Let n_k be the first U-block immediately after n_1 in the path $p(s,n)$, where $n_k = n_1$ is possible (in the presence of a loop). Such a U-block n_k must exist since the node n, which is a U-block, appears at the end of the path $p(s,n)$. Let $p(n_1,n_k) = \langle n_1, n_2, \ldots, n_k \rangle$ be the subpath of $p(s,n)$ from n_1 to n_k. There are two cases depending on whether n_1 is a D-block or not:

Case (a) n_1 is a D-block. By Definition 3.6, it suffices to show that Statement S1 in its Property P2 holds. By Step 2 of MC-PRE$_{\text{comp}}$, we have $n_1 \in \textit{D-Ins}_{\mathcal{CO}}$. P2.1(a) is true immediately. By construction, the subpath $p(n_k,n)$ of $p(s,n)$, i.e., its subpath starting from n_k to n (excluding both n_k and n), does not contain any kill nodes of π. Hence, P2.1(b) is also satisfied.

Case (b) n_1 is *not* a D-block. By Definition 3.6, we show that Statement S2 in its Property P2 holds. Since n_1 is not a D-block, the construction of $p(n_1,n_k)$ above ensures that $p(n_1,n_k)$ is a kill-comp path by Definition 5.1 (see Figure 17 again). By Lemma 5.2, $\textit{U-Ins}_{\mathcal{CO}}$ must include one edge from $p(n_1,n_k)$. Hence, Property P2.2(a) is satisfied. By construction, the subpath $p(n_k,n)$ of $p(s,n)$, i.e., its subpath starting from n_k to n (excluding n), does not contain any kill nodes of π. Hence, Property P2.2(b) is also satisfied.

The correctness of $\mathcal{CO}$ is established once the above two cases are combined. □

The following lemma shows that $\mathcal{T}_{cut}$ given in (21) must contain a computationally optimal transformation. Apart from this main result, the proof of this lemma justifies why the non-essential flow edges are not chosen as insertion points when constructing computationally optimal transformations. The idea behind the proof is that if we are given a computationally optimal solution in $\mathcal{CM}_{CompOpt}$, we can always derive from it another computationally optimal solution in $\mathcal{T}_{cut}$.

LEMMA 5.4. *Let $T \in \mathcal{CM}_{CompOpt}$. Then there always exists a PRE transformation, denoted $\mathcal{C}(T)$, such that $\mathcal{C}(T) \in \mathcal{T}_{cut}$. In addition, if $W(E) > 0$, i.e., $\forall\, e \in E : W(e) > 0$, then $\textit{U-Ins}_T \subseteq \textit{U-Ins}_{\mathcal{C}(T)}$ holds.*

PROOF. If $T \notin T_{cut}$, then all the following three properties for T may be true: (1) some computations in $\textit{U-Rep}_T \cup \textit{D-Rep}_T$ are not replaced, (2) some insertion edges in $\textit{U-Ins}_T$ are non-essential, and (3) $\Gamma(\textit{U-Ins}_T)$ is not a minimum cut in G_{st} that can be expressed in the form of $(C, \overline{C})$ (due to the existence of zero-weighted flow edges). The proof has three parts, Part 1 – 3. We modify T progressively so that the modified transformation is always computationally optimal along the way. In addition, at the end of Part k, Properties (1) – (k) will no longer hold so that at the end of Part 3, the modified transformation is contained in T_{cut}.

Part 1.. Let $G = (N, E, W)$ be the CFG under consideration. We set:

$$\begin{array}{rcl} \textit{U-Ins}_S & = & \textit{U-Ins}_T \cup \{(m,n) \in E \mid n \in \textit{UB} - \textit{U-Rep}_T\} \\ \textit{U-Rep}_S & = & \textit{U-Rep}_T \cup \{n \mid n \in \textit{UB} - \textit{U-Rep}_T\} \;=\; \textit{UB} \\ \textit{D-Ins}_S & = & \textit{D-Ins}_T \cup \{n \mid n \in \textit{DB} - \textit{D-Rep}_T\} \;=\; \textit{DB} \\ \textit{D-Rep}_S & = & \textit{D-Rep}_T \cup \{n \mid n \in \textit{DB} - \textit{D-Rep}_T\} \;=\; \textit{DB} \end{array} \tag{28}$$

By construction, $U\text{-}Rep_S = UB$ and $D\text{-}Ins_S = D\text{-}Rep_S = DB$ hold. In addition, $S \in \mathcal{CM}_{Cor}$. By Assumption 3.1, we find that $W(S) = W(T)$. Hence, $S \in \mathcal{CM}_{CompOpt}$. By construction, $U\text{-}Ins_S$ must include at least one edge in every kill-comp path $p(n_1, n_k) = \langle n_1, \ldots, n_k \rangle$ of G as defined in Definition 5.1. From the proof of Lemma 5.2, we know that this path becomes (due to node splitting) $\langle bot\_part(n_1), n_2, \ldots, n_{k-1}, top\_part(n_k) \rangle$ in G_{mm} such that $bot\_part(n_1) \in S_{mm}$ and $top\_part(n_k) \in T_{mm}$. Hence, $\Gamma(U\text{-}Ins_S)$ must include a minimum cut in G_{st}.

Part 2. Let $\mathcal{X}$ be the set of all non-essential flow edges in $U\text{-}Ins_S$. We define:

$$\begin{array}{rcl} U\text{-}Ins_{S'} & = & U\text{-}Ins_S - \mathcal{X} \\ U\text{-}Rep_{S'} & = & U\text{-}Rep_S = UB \\ D\text{-}Ins_{S'} & = & D\text{-}Ins_S = DB \\ D\text{-}Rep_{S'} & = & D\text{-}Rep_S = DB \end{array} \tag{29}$$

That is, S' is derived from S with all the non-essential edges in $\mathcal{X}$ being removed from $U\text{-}Ins_S$. Let us now prove that $S' \in \mathcal{CM}_{CompOpt}$. By Definition 3.6 and the definitions of INS-REDUND and INS-USELESS given in Step 4.2(a) in MC-PRE$_{\text{comp}}$, we find that $S' \in \mathcal{CM}_{Cor}$. We assert that $W(e) = 0$ for all $e \in \mathcal{X}$. Otherwise, $W(S') < W(S) = W(T)$, which contradicts the hypothesis that $T \in \mathcal{CM}_{CompOpt}$. Since $W(S') = W(S) = W(T)$, we have $S' \in \mathcal{CM}_{CompOpt}$.

Part 3. If all the (essential) edges in G_{rd} have positive weights, then $\Gamma(U\text{-}Ins_{S'})$ is a minimum cut in G_{st}. Otherwise, let $\mathcal{Z}$ be the set of zero-weighted (essential) edges in $U\text{-}Ins_{S'}$ such that $\Gamma(U\text{-}Ins_{S'}) - \mathcal{Z}$ is a minimum cut in G_{st}. We set:

$$\begin{array}{rcl} U\text{-}Ins_{\mathcal{C}(T)} & = & U\text{-}Ins_{S'} - \mathcal{Z} \\ U\text{-}Rep_{\mathcal{C}(T)} & = & U\text{-}Rep_{S'} = UB \\ D\text{-}Ins_{\mathcal{C}(T)} & = & D\text{-}Ins_{S'} = DB \\ D\text{-}Rep_{\mathcal{C}(T)} & = & D\text{-}Rep_{S'} = DB \end{array} \tag{30}$$

By the definition of $\mathcal{T}_{cut}$ given in (21), $\mathcal{C}(T) \in \mathcal{T}_{cut}$ holds. In addition, if $W(e) > 0$, then $U\text{-}Ins_T \subseteq U\text{-}Ins_{\mathcal{C}(T)}$ holds trivially since $\forall\ e \in \mathcal{X} \cup \mathcal{Z} : W(e) = 0$. □

THEOREM 5.5. *$\mathcal{CO} \in \mathcal{CM}_{CompOpt}$, where $\mathcal{CO}$ is found by MC-PRE$_{\text{comp}}$.*

PROOF. Follows directly from Lemmas 5.3 and 5.4. □

The proof of Theorem 5.5 leads to the following important result, which will be used in establishing the lifetime optimality of $\mathcal{LCO}$ found by our MC-PRE algorithm. In a computationally optimal solution, no insertion takes place on an insertion-redundant edge with a nonzero execution frequency.

THEOREM 5.6. *Consider an expression π in $G = (N, E, W)$. If $(u, v) \in E$ such that* X-AVAL$(u) =$ true *and $W(u, v) > 0$, then $(u, v) \notin U\text{-}Ins_T$ for any $T \in \mathcal{CM}_{CompOpt}$.*

PROOF. Proceeding exactly as in the proof of Theorem 5.5, we obtain a transformation $\mathcal{C}(T) \in \mathcal{CM}_{CompOpt}$ such that $U\text{-}Ins_{\mathcal{C}(T)} \cap \mathcal{X} = \emptyset$, where $\mathcal{X}$ is the set of all non-essential edges contained in $U\text{-}Ins_T$. In the proof of Theorem 5.5, we showed that $\forall\ (m, n) \in \mathcal{X} : W(m, n) = 0$. For an edge $(u, v) \in E$ such that X-AVAL$(u) =$ true and $W(u, v) > 0$, we must have $(u, v) \notin U\text{-}Ins_T$. □

5.2 Lifetime Optimality

In this section, we prove that $\mathcal{LO}$ is the unique lifetime optimal solution when $W(E) > 0$. Theorem 5.7 shows that $\mathcal{LO}$ is computationally optimal, i.e., $\mathcal{LO} \in \mathcal{CM}_{CompOpt}$. Lemma 5.8 recalls a classic result from [Hu 1970] that exposes the structure of all minimum cuts in a flow network. Based on this result, Lemma 5.9 shows that, among all minimum cuts in the EFG G_{st}, the minimum cut $\mathcal{C}_\Lambda = (\Lambda, \overline{\Lambda})$ found in Step 5 of MC-PRE must be such that $\overline{\Lambda}$ is the *smallest*. Finally, Theorem 5.10 establishes $\mathcal{CM}_{LifeOpt} = \{\mathcal{LO}\}$ if $W(E) > 0$.

The following theorem implies that $\mathcal{LO}$ is also a correct PRE transformation.

THEOREM 5.7. $\mathcal{LO} \in \mathcal{CM}_{CompOpt}$.

PROOF. Let us consider the special minimum cut $\mathcal{C}_\Lambda = (\Lambda, \overline{\Lambda})$ found in Step 5 of MC-PRE. By Theorem 5.5, the corresponding transformation $\mathcal{ALO}$ given in (23) is computationally optimal, i.e., $\mathcal{ALO} \in \mathcal{CM}_{CompOpt}$. Note that $\mathcal{LO}$ is derived from $\mathcal{ALO}$ in Steps 6 – 8 of MC-PRE. This construction ensures that *benefit*$(\mathcal{ALO})$ = *benefit*$(\mathcal{LO})$, where *benefit* is defined in (10). In addition, $\mathcal{ALO} \in \mathcal{CM}_{Cor}$. Hence, $W(\mathcal{LO}) = W(\mathcal{ALO})$. This means that $\mathcal{LO} \in \mathcal{CM}_{CompOpt}$. □

Next, we recall Lemma 10 from [Hu 1970] on the structure of all minimum cuts.

LEMMA 5.8. *If $(A, \overline{A})$ and $(B, \overline{B})$ are minimum cuts in an s-t flow network, then $(A \cap B, \overline{A \cap B})$ and $(A \cup B, \overline{A \cup B})$ are also minimum cuts in the network.*

This lemma implies immediately that a unique minimum cut $(C, \overline{C})$ exists such that $\overline{C}$ is the *smallest*, i.e., that $\overline{C} \subset \overline{C'}$ for every other minimum cut $(C', \overline{C'})$. Note that $\subset$ is strict. In addition, this lemma is valid independently of any maximum flow that one may use to enumerate all maximum cuts for the underlying network.

In fact, for the minimum cut $(\Lambda, \overline{\Lambda})$ found by MC-PRE, $\overline{\Lambda}$ is the smallest.

LEMMA 5.9. *Suppose $W(E) > 0$. Let S_{cut} be the set of all cuts in $G_{st} = (N_{st}, E_{st}, W_{st})$ whose capacities are equal to a maximum flow. Consider the minimum cut $(\Lambda, \overline{\Lambda})$ in G_{st} found by MC-PRE. Then the following statement is true:*

$$\overline{\Lambda} \subseteq \overline{C} \text{ for all } (C, \overline{C}) \in S_{cut} \tag{31}$$

where the equality in $\subseteq$ holds iff $\overline{\Lambda} = \overline{C}$.

PROOF. If $W(E) > 0$, then $W(E_{rd}) > 0$. So G_{st} is an s-t flow network with positive edge capacities only. Thus, a cut whose capacity is equal to a maximum flow must be a minimum cut of the form $(C, \overline{C})$, and S_{cut} is the set of all minimum cuts in G_{st} [Hu 1970]. In Step 5 of MC-PRE, we find the minimum cut $(\Lambda, \overline{\Lambda})$ by applying essentially the "Reverse" Labelling Procedure of [Ford and Fulkerson 1962]. Its construction ensures that the statement stated in (31) holds with respect to the maximum flow f used. Lemma 5.8 implies that this "smallest minimum cut" is independent of the maximum flow f. Hence, the validity of (31) is established. □

We prove that $\mathcal{LO}$ is the unique lifetime optimal transformation if $W(E) > 0$.

THEOREM 5.10. *If $W(E) > 0$, then $\mathcal{LO} \in \mathcal{CM}_{LifeOpt}$.*

PROOF. We consider a PRE problem for an expression π in $G = (N, E, W)$. By Theorem 5.7, $\mathcal{LO} \in \mathcal{CM}_{CompOpt}$. We proceed in two steps progressively:

(1) *Every computation of π replaced by $\mathcal{LO}$ must also be replaced by every $T \in \mathcal{CM}_{CompOpt}$.* We prove $U\text{-}Rep_{\mathcal{LO}} \subseteq U\text{-}Rep_T$ only. We can prove $D\text{-}Rep_{\mathcal{LO}} \subseteq D\text{-}Rep_T$ similarly. Let $T \in \mathcal{CM}_{CompOpt}$. Note that $W(E) > 0$. By Lemma 5.4, let $\mathcal{C}(T) \in T_{cut}$ be constructed from T such that $U\text{-}Ins_T \subseteq U\text{-}Ins_{\mathcal{C}(T)}$ and $\Gamma(U\text{-}Ins_{\mathcal{C}(T)})$ is a minimum cut, denoted $(C, \overline{C})$, in G_{st}. By Step 8 of the MC-PRE algorithm, $\mathsf{U\text{-}ISOLATED}_\Lambda(n) = \mathsf{false}$. There are two cases:
   (a) $\mathsf{KILL}(n_1) = \mathsf{true}$. Suppose $n \notin U\text{-}Rep_T$. By the construction of $\mathcal{C}(T)$, $n \in U\text{-}Ins_{\mathcal{C}(T)}$ and $\forall\, m \in \mathsf{pred}(G, n) : (m, n) \in U\text{-}Ins_{\mathcal{C}(T)}$. Since $W(E) > 0$, we must have $\forall\, m \in \mathsf{pred}(G, n) : \mathsf{X\text{-}AVAL}(m) = \mathsf{false}$ by Theorem 5.6. This implies immediately that n_1 is in G_{rd} by Lemma 4.1. Note that $(n_i, n_{i+1}) \in (C, \overline{C})$ but $(n_i, n_{i+1}) \notin (\Lambda, \overline{\Lambda})$. This implies that $\overline{\Lambda} \not\subset \overline{C}$, which is impossible according to Lemma 5.9. Thus, we must have $n \in U\text{-}Rep_T$.
   (b) $\mathsf{KILL}(n_1) = \mathsf{false} \wedge \mathsf{X\text{-}LIVE}(n_1) = \mathsf{true}$. Therefore, there must exist a path $\langle n_1, \ldots, n_k \rangle$ in G such that (a) $n = n_1$, (b) $n_2, \ldots, n_{k-1}$ are neither U-blocks nor D-blocks nor kill blocks, (c) $n_k \in UB$, and (d) $\forall\, 1 \leqslant i < k : (n_i, n_{i+1}) \notin U\text{-}Ins_{\mathcal{LO}}$. Let us show that $\forall\, 1 \leqslant i < k : (n_i, n_{i+1}) \notin U\text{-}Ins_T$. Suppose that $(n_i, n_{i+1}) \in U\text{-}Ins_T$ for some $1 \leqslant i < k$. Since $W(E) > 0$, we must have $\mathsf{X\text{-}AVAL}(n_i) = \mathsf{false}$ by Theorem 5.6. This implies immediately that $\mathsf{X\text{-}AVAL}(n_{k-1}) = \mathsf{false}$. By Lemma 4.1, n_k is contained in G_{rd}. Note that $(n_i, n_{i+1}) \in (C, \overline{C})$ but $(n_i, n_{i+1}) \notin (\Lambda, \overline{\Lambda})$. Thus, $\overline{\Lambda} \not\subset \overline{C}$, which is impossible according to Lemma 5.9. By construction, $n_k \in U\text{-}Rep_{\mathcal{C}(T)}$. Hence, the value of $h_\pi = \pi$ must be available on entry of n. Since $W(n) = W(n_1) > 0$. If $n \notin U\text{-}Ins_T$, then $T \notin \mathcal{CM}_{CompOpt}$.

(2) *All insertions specified by $\mathcal{LO}$ induce the strictly smallest lifetime range.* Let $T \in \mathcal{CM}_{LifeOpt}$. By Property (2) of Definition 3.8, $U\text{-}Ins_{\mathcal{LO}}$ cannot contain any insertion-useless edges. Since $W(E) > 0$, $U\text{-}Ins_T$ cannot contain any insertion-redundant edges (Theorem 5.6). By construction, $U\text{-}Ins_{\mathcal{LO}}$ contains only essential edges. Let us examine the two kind of insertions of $h_\pi = \pi$:
   (a) $D\text{-}Rep_{\mathcal{LO}} = D\text{-}Ins_{\mathcal{LO}} \subseteq D\text{-}Ins_T = D\text{-}Rep_T$. The point for each of these insertions is unique by the definition of our speculative PRE. So the overall lifetime range for the insertions by $D\text{-}Ins_{LO}$ is better than T if $T \neq \mathcal{LO}$.
   (b) $\overline{\Lambda} \subseteq \overline{C}$ by Lemma 5.9. By using $U\text{-}Rep_{\mathcal{LO}} \subseteq U\text{-}Rep_T$ proved in (1), we conclude that the overall lifetime range for the edge insertions specified by $U\text{-}Ins_{LO}$ is better than T if $T \neq \mathcal{LO}$.
   By combining 2(a) and 2(b), we conclude that $\mathcal{CM}_{LifeOpt} = \{\mathcal{LO}\}$. □

Finally, the isolation analysis performed in MC-PRE has one nice implication. The following theorem was illustrated earlier in Section 4.2.5.

THEOREM 5.11. *If $W(E) > 0$, then the following statement about $\mathcal{LO}$ is true:*

$$benefit(\mathcal{LO}) = 0 \iff U\text{-}Ins_{\mathcal{LO}} \cup D\text{-}Ins_{\mathcal{LO}} = \emptyset$$

PROOF. Note that $U\text{-}Ins_{\mathcal{LO}} \cup D\text{-}Ins_{\mathcal{LO}} = \emptyset \iff U\text{-}Rep_{\mathcal{LO}} \cup D\text{-}Rep_{\mathcal{LO}} = \emptyset$. So "$\Longleftarrow$" is obvious. To prove "$\Longrightarrow$", we note that $\forall\, n \in UB : (\forall\, m \in \mathsf{pred}(G, n) : \mathsf{X\text{-}AVAL}(m) = \mathsf{false})$. Otherwise, since $W(E) > 0$, $benefit(\mathcal{LO}) > 0$ must hold. By Lemma 4.1, all blocks in UB are in G_{rd}. By Lemma 4.6, $\forall\, n \in UB : top\_part(n) \in T_{mm}$. Let $\overline{\Lambda} = \{top\_part(n) \mid n \in UB\}$. Obviously, $(\Lambda, \overline{\Lambda})$ is the minimum cut

	Intel	Sun
CPU	Xeon	UltraSPARC-III
CPU Speed	2.4GHz	900MHz
General-Purpose Registers	8	32
L1 I-cache	12K uops, on-chip	32KB/32B/4, on-chip
L1 D-cache	8KB/64B/4, on-chip	64KB/32B/4, on-chip
L2 Unified cache	512KB/64B/8, sectored, on-chip	8MB/512B/2, off-chip
Memory	2GB	3GB
Disk subsystem	73GB	73GB
OS	Redhat Linux 8.0 (2.4.20)	SunOS 5.8
gcc version	3.4	3.4
gcc switches	-O2 -fomit-frame-pointer	-O2

Table II. Machine configurations.

associated with $\mathcal{ALO}$. In addition, $\forall\ n \in UB$: $\mathsf{U\text{-}ISOLATED}_\Lambda(n) = \mathsf{true}$ and $\forall\ n \in DB$: $\mathsf{D\text{-}ISOLATED}_\Lambda(n) = \mathsf{true}$. Hence, $U\text{-}Ins_{\mathcal{LO}} \cup D\text{-}Ins_{\mathcal{LO}} = \emptyset$. □

We note that $\mathcal{CM}_{LifeOpt} = \{\mathcal{LO}\}$ under a slightly weaker condition, i.e., $W(E - \{e \in E \mid \mathsf{INS\text{-}USELESS}(e)\}) > 0$. This is because by Property (2) of Definition 3.8, we must have $U\text{-}Ins_T \cap \{e \in E \mid \mathsf{INS\text{-}USELESS}(e)\} = \emptyset$ if $T \in \mathcal{CM}_{LifeOpt}$.

6. EXPERIMENTS

We evaluate this work using all the 22 C, C++ and FORTRAN 77 benchmarks from SPECcpu2000 on two computer architectures as described in Table II. We have implemented MC-PRE in `gcc 3.4`. All the experiments were conducted when the two computers were running in stand-alone mode with us being the only user.

Section 6.1 describes the `gcc` framework in which this work is validated. In Section 6.2, we explain precisely all the details regarding the implementation of MC-PRE in `gcc 3.4`. Section 6.3 presents our benefit and performance numbers and the associated compilation and space costs. Section 6.4 analyses and discusses our experimental results. In particular, we use a representative benchmark to explain how performing speculative PRE has improved its performance. We also give the reasons for MC-PRE's small compilation overhead increases over LCM.

6.1 Experimental Setup

Figure 18 depicts the `gcc` compiler backend in which our algorithm is implemented and evaluated. The backend applies numerous passes to the RTL (Register Transfer Language) representation [GNU Software 2003] of a function, where the LCSE pass, i.e., the local PRE appears before the global PRE pass.

Knoop, Rüthing and Steffen's profile-independent LCM algorithm is the built-in global PRE pass, called `gcse`, invoked at `gcc`'s `O2` optimization level and above. The LCM configuration shown represents exactly how `gcc` works at `O2` when dynamic profiling is not explicitly enabled. In this case, `gcc` compiles a program only once. Due to the absence of dynamic profiling information, the "Branch Probabilities" pass works by predicting the branch probabilities statically.

In our profile-guided framework, MC-PRE is the replacement for LCM. Due to the introduction of the "Edge Profiling" pass before MC-PRE, compiling a program requires `gcc` to be run twice. In the first run, we turn the switch "`-profile-arcs`"

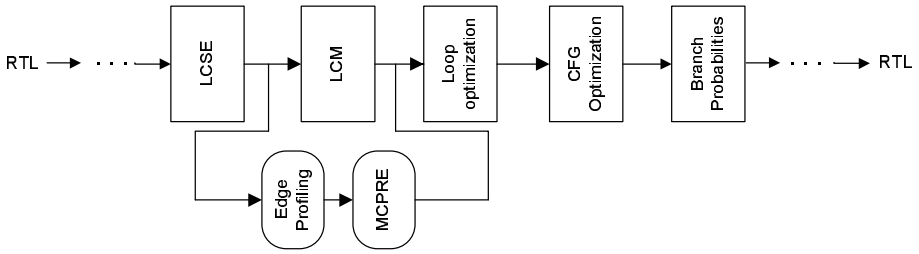


Fig. 18. The `gcc` backend with either LCM or MC-PRE being used as the (global) PRE pass.

on so that `gcc` will instrument a program to gather its dynamic profiling information. The profiling information for all SPECcpu2000 benchmarks is always collected using the train input data sets. In the second run, we invoke `gcc` by turning the switch "`-branch-probabilities`" on. This instructs `gcc` to compute the edge profiles for all the functions in the program from the profiling information gathered in the first run. The edge profiling information can then be used by MC-PRE. In this second run, all benchmarks are executed using the reference input data sets. Immediately after the MC-PRE pass, we ignore the edge profiling information. There are two reasons for doing so. First, the `gcc` passes such as "Loop Optimization" and "CFG Optimization" as shown in Figure 18 do not update profiling information when performing some control flow restructuring transformations. This is because in `gcc`, the "Edge Profiling" pass is positioned after these passes and just before the "Branch Probabilities" pass. Such a phrase ordering cannot be used for us since MC-PRE is profile-guided. Second, even if the profiling information is correctly updated, the "Branch Probabilities" pass will take advantage of this information to compute the branch probabilities, giving MC-PRE an unfair advantage over LCM.

Therefore, both the LCM and MC-PRE configurations differ *only* in the PRE algorithm used. In both cases, the same passes are applied in exactly the same order. The only difference is that the "Edge Profiling" pass is needed to supply the edge profiles required by MC-PRE. Note once again that the LCM configuration is exactly the one from `gcc 3.4` (when dynamic profiling is not enabled). This provides an ideal setting for MC-PRE to be evaluated against LCM [GNU GCC Developers 2004].

In our experiments, both the LCM and MC-PRE configurations use exactly the same set of PRE candidate expressions for a function. These are the expressions identified by `gcc` for its LCM pass. In `gcc`, a PRE candidate expression is always the RHS of an assignment, where the LHS is a virtual register. The RHS expressions that are constants or virtual registers are excluded (since no computations are involved). So are any expressions such as call expressions with side effects.

We make use of the bit-vector library used by LCM to perform the three dataflow analysis passes required by MC-PRE. In both algorithms, all the passes are performed on a given CFG (for its distinct PRE candidate expressions in parallel).

6.2 Implementation Details

We discussed in Section 3.2.3 that MC-PRE works in the presence of critical edges since edge insertions are used. The LCM pass in `gcc` is an implementation of the LCM algorithm described in [Knoop et al. 1994] except that it can reason about edge insertions [Morgan 1998]. Thus, the critical edges are not split in `gcc`.

As we mentioned earlier, we do not modify a CFG G to obtain its G_{rd}, G_{mm}, and finally, the EFG G_{st}. In our implementation, we use Goldberg's *push-relabel* HIPR algorithm [GoldBerg 2003] and his implementation as our min-cut solver. This solver is one of the fastest implementations available [Chekuri et al. 1997]. For every G, we generate a graph specification for the EFG G_{st} and feed it to the min-cut solver to find the unique minimum cut as defined in Step 5 of MC-PRE. Therefore, the solver operates separately on distinct PRE candidate expressions for a CFG, i.e., distinct PRE problems sequentially.

As we shall see in Section 6.4, a large number of reduced graphs G_{rd} in a benchmark program are empty. If $G_{rd} = \emptyset$, then $G_{mm} = G_{st} = \emptyset$. The minimum cut on an empty EFG G_{st} is empty. In this case, Steps 4.2 – 4.4 and 4.5(a) of MC-PRE$_{\text{comp}}$ serve only to set $\mathcal{C} = \emptyset$, and similarly, Steps 2 – 5 of MC-PRE serve only to set $\mathcal{C}_\Lambda = \mathcal{C}'_\Lambda = \emptyset$. In our implementation, all these steps are ignored if $G_{rd} = \emptyset$ and the required minimum cuts are simply set to be empty. By Theorem 4.3, we detect the emptiness of G_{rd} by relying on the information from the availability and partial anticipability analyses. The `gcc` compiler maintains, for each PRE candidate expression π, a list of all blocks in which π is upwards (downwards) exposed. In our terminology, π is associated with a list of the U-blocks in UB (the D-blocks in DB). So Theorem 4.3 can be applied in a straightforward manner.

Step 4.2(a) of MC-PRE$_{\text{comp}}$ builds G_{rd} when it is not empty. This involves traversing G and testing if the outgoing edges of a node are essential or not. In `gcc 3.4`, `edge_list` is an array of pointers pointing to all the `edge_def` structs in a CFG. Then an edge is simply identified by an index into `edge_list`. We have introduced a new field called `edge_index` into `gcc`'s `edge_def` struct as follows:

```
typedef struct edge_def {
  // gcc's existing fields
  int edge_index;
} *edge;
```

When `edge_list` is created, the `edge_index` in an `edge_def` struct is set such that `edge_list`[edge_index] is pointing to that struct. Such a simple change allows Step 4.2(a) of MC-PRE$_{\text{comp}}$ to be carried out more efficiently than otherwise. When visiting a node, denoted `src`, in a CFG, we test whether each of its outgoing edges is essential or not as follows:

```
[1]  edge e;  // a pointer to an edge_def node
[2]  for (e = src->succ; e != NULL; e = e->succ_next) {
[3]      int edge_index = e->edge_index;
[4]      if (ESS(edge_index, expr_index))
[5]          ...
[6]  }
```

where the `for` loop goes through a list of pointers `e` pointing to the outgoing edges of `src` and `e->edge_index` simply gives us the index that identifies the edge e. If we had not introduced `edge_index`, we would have replaced line 3 above with:

```
[3a]  basic_block dest = e->dest;
[3b]  int edge_index = EDGE_INDEX(edge_list, src, dest);
```

where `EDGE_INDEX` finds `edge_index` by performing a linear search in `edge_list`. Such an inefficient operation has been used for debugging purposes only in `gcc`.

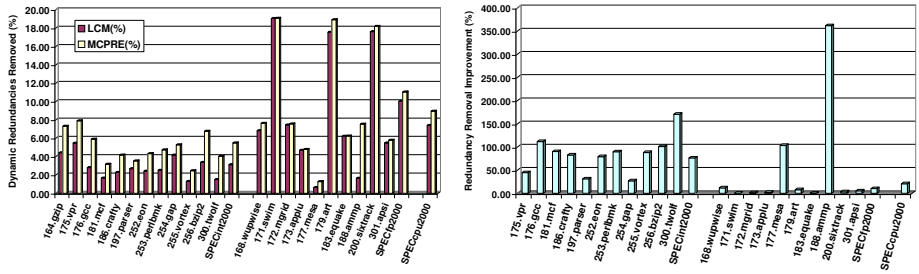
6.3 Benefits and Costs

Our programs are the 22 C, C++ and FORTRAN 77 benchmarks from SPECcpu2000. Our two computer platforms are detailed in Table II. We report the extra redundancies removed and execution time improvements of MC-PRE over LCM and the associated compilation and space costs for achieving these benefits. Recall that in the MC-PRE configuration, `gcc` is invoked twice. All the reported numbers for MC-PRE are those obtained in the second invocation.

The dynamic amount of redundant computations removed by MC-PRE for a benchmark is calculated based on (10). This quantity for LCM is computed using the same formula. The size of a compiled benchmark is given by the size of the text section of its binary code obtained using the UNIX command `size` as suggested in [Beszedes 2003]. The execution time of a benchmark is taken as the median of 11 runs. The compile time of a benchmark is measured using the built-in mechanism in `gcc`.

6.3.1 *Xeon.* Let us examine the experimental results on Xeon. Figure 19 compares the benefits of MC-PRE and LCM in terms of the dynamic amount of redundant computations removed. Figure 19(a) gives the amount of computations removed by MC-PRE and LCM, respectively. Figure 19(b) shows how much more redundancies MC-PRE can eliminate than LCM for each benchmark. MC-PRE is more effective for the SPECint2000 benchmarks since they generally exhibit more complex control flow restructures than the SPECfp2000 benchmarks. Due to its computational optimality, MC-PRE always removes no fewer redundant computations than LCM. The percentage increases range from 26.92% to 170.67% with an average of 76.34% for SPECint2000 and from 0.11% to 361.69% with an average of 10.29% for SPECfp2000. The average increase for SPECcpu200 is 20.83%. The two benchmarks that benefit the most from MC-PRE are `ammp` from SPECfp2000 with an increase of 361.69% and `twolf` from SPECint2000 with an increase of 170.67%.

Figure 20 shows the execution time speedups of MC-PRE over LCM. The performance speedups for SPECint2000 range from 0.00% to 4.62% with an average of 1.24% while those for SPECfp2000 range from −0.71% to 2.59% with an average of 0.89%. The average speedup for the entire SPECcpu2000 benchmark suite is 1.03%. The three best speedups are obtained for `crafty`, `gzip` and `swim` at 4.62%, 2.84% and 2.59%, respectively. The slight performance degradations are observed in `mgrid`, `sixtrack` and `apsi`. It is well-known that the impact of a PRE algorithm on the performance of a program comes from not only the amount of computations removed but also its complex interactions with various later optimization passes (which are quite difficult to quantify). However, compared to LCM, MC-PRE achieves the same or better performance in 19 out of 22 benchmarks.



(a) Redundancies removed by MC-PRE and LCM (b) Improvement of MC-PRE over LCM

Fig. 19. Dynamic redundancies removed by MC-PRE and LCM on Xeon.

Table III compares the code sizes of the compiled benchmarks on Xeon (as well as UltraSPARC-III). Column "NO-PRE" for each architecture refers to the `gcc` configuration in which no global PRE is used. A PRE algorithm both inserts and deletes computations. So it may cause the code sizes to be decreased in some benchmarks and increased in others. Large code size increases generally happen to small programs as in the case of `mcf` and `swim` on Xeon. In comparison with LCM, the code size increases caused by MC-PRE on Xeon are calculated to be 3.01%, 0.84% and 2.37% for SPECint2000, SPECfp2000 and SPECcpu2000, respectively.

Table IV compares the compilation times of LCM and MC-PRE for all the benchmarks on Xeon (and UltraSPARC-III). The compilation overheads for achieving the performance improvements on Xeon are relatively small. The overheads for the SPECint2000 benchmarks are no larger than 9.50%, giving an average of 3.58%. The SPECfp2000 benchmarks except `sixtrack` have small compilation times, giving an average of 6.58%, which will drop to 1.09% if `sixtrack` is excluded. The overhead for the entire SPECcpu2000 benchmark suite is 4.29%. In the case of `bzip2`, `applu` and `ammp` with small compilation times, MC-PRE compiles slightly faster than LCM. The reasons for this phenomenon will be explained in Section 6.4.

6.3.2 *UltraSPARC-III.* Let us examine the results on UltraSPARC-III. Figure 21 compares the dynamic amount of redundant computations eliminated by MC-PRE and LCM. In comparison with LCM, MC-PRE removes between 15.35% and 82.15% (an average of 38.36%) more redundant computations for SPECint2000 and between 0.09% and 128.01% (an average of 7.09%) more redundant computations for SPECfp2000. MC-PRE improves over LCM by 13.98% on the average for the entire SPECcpu2000 benchmark suite.

Figure 22 shows the execution time speedups of MC-PRE over LCM for all the benchmarks. Among the 12 SPECint2000 benchmarks, 11 benchmarks run up to 2.49% faster while `gap` suffers a small performance loss of −0.61%. Among the 10 SPECfp2000 benchmarks, nine benchmarks run up to 5.08% faster while `applu` is −0.29% slower. On the average, the performance speedups for SPECint2000 and SPECfp2000 are 1.17% and 1.58%, respectively. The overall performance improvement for the entire SPECcpu2000 benchmark suite is 1.44%.

As shown in Table III, MC-PRE has caused relatively smaller code size increases

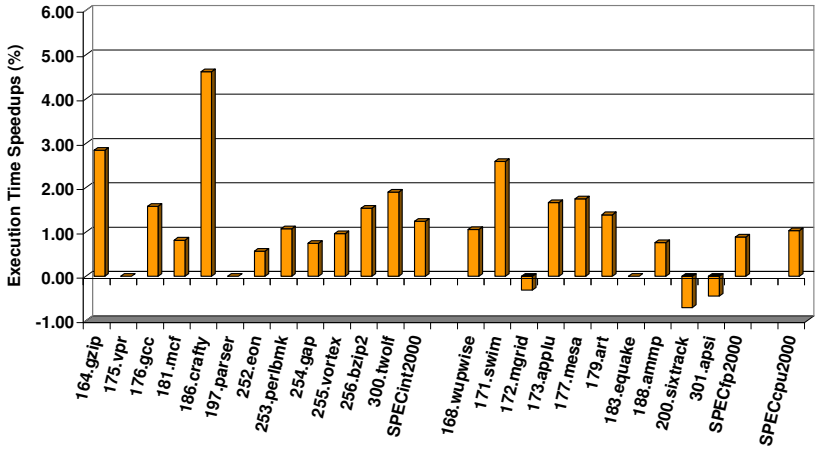


Fig. 20. Execution time speedups of MC-PRE over LCM on Xeon.

Benchmark	Xeon			UltraSPARC-III		
	NO-PRE	Space Increases (%)		NO-PRE	Space Increases (%)	
		LCM	MC-PRE		LCM	MC-PRE
164.gzip	32240	0.84	4.22	47633	−2.76	−0.95
175.vpr	141326	−2.73	0.25	145206	−2.58	−1.10
176.gcc	1267589	0.16	5.45	1475988	−0.77	1.90
181.mcf	9750	1.64	13.46	12918	−0.56	0.28
186.crafty	206354	2.67	9.85	224278	−1.13	0.10
197.parser	87041	0.35	0.33	114300	−0.12	0.82
252.eon	504636	1.40	5.82	471101	−1.56	−0.48
253.perlbmk	471007	1.78	2.45	599598	−1.11	−0.03
254.gap	436772	0.96	0.00	525623	−0.94	−0.04
255.vortex	609303	−0.53	−0.60	592225	−1.05	0.03
256.bzip2	27773	−0.17	2.25	37837	−2.45	−0.78
300.twolf	187119	−1.58	4.03	223189	−2.37	1.23
SPECint2000	3980910	0.45	3.47	4469896	−1.13	0.61
168.wupwise	27979	−5.49	−3.26	25363	−3.55	−0.22
171.swim	6286	14.76	14.76	9720	−0.86	−0.33
172.mgrid	11567	−10.93	−7.47	11990	−2.50	−6.47
173.applu	64917	−9.32	−8.66	64553	−17.28	−17.42
177.mesa	467461	−0.40	−2.96	568433	−0.88	−0.82
179.art	13167	−4.16	−4.53	17644	−3.42	−1.07
183.equake	17012	2.26	5.36	21233	−1.17	−0.75
188.ammp	112304	0.97	1.30	132140	−0.55	1.34
200.sixtrack	815308	1.83	4.30	887480	−1.20	0.87
301.apsi	126261	−6.08	−3.38	118575	−6.41	−3.86
SPECfp2000	1662262	−0.10	0.74	1857131	−2.01	−0.66
SPECcpu2000	5643172	0.29	2.66	6327027	−1.39	0.24

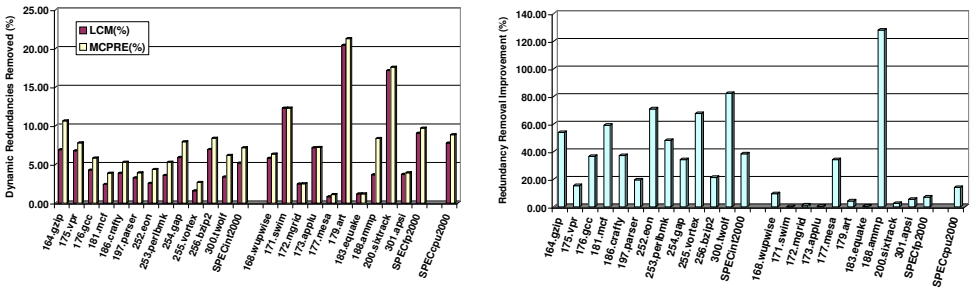
Table III. The code sizes of all benchmarks on Xeon and UltraSAPRC-III (in bytes).

Benchmark	Xeon			UltraSPARC-III		
	LCM	MCPRE	Overhead (%)	LCM	MCPRE	Overhead(%)
64.gzip	1.84	1.92	4.35	5.27	5.42	2.85
175.vpr	6.93	7.13	2.89	16.75	17.77	6.09
176.gcc	61.13	66.94	9.50	171.39	194.69	13.59
181.mcf	1.20	1.24	3.33	2.14	2.23	4.21
186.crafty	11.66	12.27	5.23	33.40	36.38	8.92
197.parser	4.97	5.07	2.01	12.06	12.49	3.57
252.eon	165.56	167.56	1.21	436.66	438.98	0.53
253.perlbmk	29.64	31.24	5.40	82.19	92.39	12.41
254.gap	22.40	22.90	2.23	60.93	63.53	4.27
255.vortex	24.82	25.43	2.46	60.16	62.01	3.08
256.bzip2	1.19	1.16	−2.52	3.30	3.47	5.15
300.twolf	13.76	14.55	5.74	34.81	37.36	7.33
SPECint2000	345.10	357.45	3.58	919.06	966.72	5.19
168.wupwise	1.12	1.18	5.36	2.92	3.08	5.48
171.swim	0.25	0.25	0.00	1.36	1.34	−1.47
172.mgrid	0.48	0.49	2.08	1.32	1.27	−3.79
173.applu	3.86	3.82	−1.04	12.30	12.04	−2.11
177.mesa	25.41	25.76	1.38	74.63	77.94	4.44
179.art	0.54	0.54	0.00	1.46	1.48	1.37
183.equake	0.71	0.72	1.41	1.99	2.05	3.02
188.ammp	7.33	7.23	−1.36	16.33	16.62	1.78
200.sixtrack	62.40	68.96	10.51	228.85	266.13	16.29
301.apsi	5.06	5.26	3.95	13.80	14.26	3.33
SPECfp2000	107.16	114.21	6.58	354.96	396.21	11.62
SPECcpu2000	452.26	471.66	4.29	1274.02	1362.93	6.98

Table IV. The compile times of all benchmarks on Xeon and UltraSAPRC-III (in seconds).

on UltraSPARC-III than Xeon. In comparison with LCM, the average increases for SPECint2000, SPECfp2000 and SPECcpu2000 are calculated to be 1.76%, 1.38% and 1.65%, respectively.

As in the case of Xeon, Table IV shows that the compilation overheads for achieving the performance speedups on UltraSPARC-III are also small. In the



(a) Redundancy removed by MC-PRE and LCM (b) Improvement of MC-PRE over LCM

Fig. 21. Dynamic redundancies removed by MC-PRE and LCM on UltraSPARC-III.

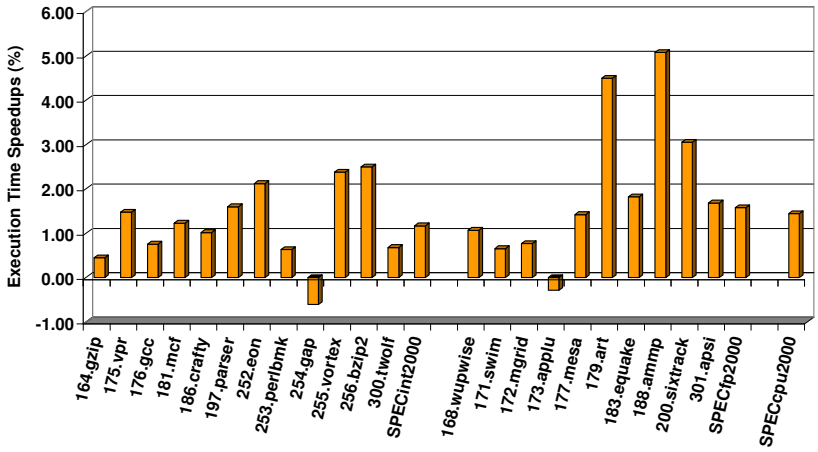


Fig. 22. Execution time speedups of MC-PRE over LCM on UltraSPARC-III.

case of SPECint2000, `gcc` and `perlmk` are the most expensive to compile, attracting an overhead of 13.59% and 12.41%, respectively. The average overhead for SPECint2000 is 5.19%. In the case of SPECfp2000, `sixtrack` incurs the largest overhead of 16.29%. The average overhead for SPECfp2000 is 11.62%; this number will drop to 3.15% if `sixtrack` is excluded. The average overhead for the entire SPECcpu2000 benchmark suite is 6.98%. The reasons why `swim`, `mgrid` and `apply` compiles faster under MC-PRE than LCM will be explained in Section 6.4.

6.4 Analysis and Discussions

We examine the speedups and compilation overhead increases of MC-PRE over LCM. A PRE algorithm affects the execution time of a program in a fairly complex way. There are many contributing factors, including (1) the amount of removed computations, (2) the places where the introduced temporaries are inserted, and (3) the complex interactions of these insertions and deletions with various optimizations carried out later. MC-PRE maximizes (1) and optimizes (2) by means of minimizing the lifetimes of introduced temporaries. Like the existing PRE algorithms, however, MC-PRE is insensitive to (3), which may impact performance in a subtle way. For this reason, Section 6.4.1 uses a representative benchmark to explain how MC-PRE has reduced its execution time by eliminating some redundant computations that can only be eliminated speculatively. In Section 6.4.2, we give exactly the reasons why MC-PRE are not so expensive despite the min-cut component it uses.

Let F_P be the set of functions in a benchmark P and C_f the set of distinct PRE candidate expressions π in a function $f \in F_P$. Thus, the total number of PRE problems to be solved in the benchmark P is given by $\sum_{f \in F_P} |C_f|$. The average number of PRE problems per function in P is $(\sum_{f \in F_P} |C_f|)/|F_P|$. Table V gives these statistics on both Xeon and UltraSPARC-III. Note that the functions here are not meant to be the user functions; they are the RTL-level functions that are processed by a PRE algorithm. Due to the differences in system libraries, macro expansion and function inlining, the four benchmarks, `gzip`, `gcc`, `eon`, and `perlbmk`, do not have the same number of RTL-level functions on both architectures. In

Benchmark	Xeon			UltraSPARC-III		
	#Funs	#BBs	PRE problems (per function)	#Funs	#BBs	PRE problems (per function)
164.gzip	74	1389	1402 (18.95)	80	1519	3788 (47.35)
175.vpr	188	3460	6160 (32.77)	188	3500	9864 (52.47)
176.gcc	1987	58355	48433 (24.37)	1990	58902	97637 (49.06)
181.mcf	26	371	526 (20.23)	26	374	692 (26.62)
186.crafty	109	5638	5450 (50.00)	109	5562	18427 (169.06)
197.parser	324	3841	4011 (12.38)	324	3864	6646 (20.51)
252.eon	1245	12880	39829 (31.99)	1234	13074	45796 (37.11)
253.perlbmk	1026	23612	23846 (23.24)	1025	23844	43381 (42.32)
254.gap	849	21683	26477 (31.19)	849	22193	46032 (54.22)
255.vortex	923	16135	20251 (21.94)	923	16180	25193 (27.29)
256.bzip2	74	951	1010 (13.65)	74	978	2345 (31.69)
300.twolf	191	6721	12337 (64.59)	191	6723	22547 (118.05)
SPECint2000	7016	155036	189732 (27.04)	7013	156713	322348 (45.96)
168.wupwise	22	391	1475 (67.05)	22	383	1826 (83.00)
171.swim	6	65	462 (77.00)	6	65	949 (158.17)
172.mgrid	12	151	942 (78.50)	12	151	1053 (87.75)
173.applu	16	443	4267 (266.69)	16	443	6418 (401.13)
177.mesa	1039	18063	31270 (30.10)	1039	18906	49872 (48.00)
179.art	26	355	609 (23.42)	26	396	1113 (42.81)
183.equake	27	342	995 (36.85)	27	350	1450 (53.70)
188.ammp	179	3575	9382 (52.41)	179	3659	10950 (61.17)
200.sixtrack	241	11149	61302 (254.37)	241	11149	108356 (449.61)
301.apsi	97	1355	8641 (89.08)	97	1361	10854 (111.90)
SPECfp2000	1665	35889	119345 (71.68)	1665	36863	192841 (115.82)
SPECcpu2000	8681	190925	309077 (35.60)	8678	193576	515189 (59.37)

Table V. Statistics about the number of functions, basic blocks and PRE problems in benchmarks on Xeon and UltraSPARC-III. The three columns for a benchmark P on an architecture are computed as follows. Column "#Funs" gives the number of functions, F_P. Column "#BBs" gives the number of basic blocks in all functions of F_P. In Column "PRE problems (per function), the number of "PRE problems" in P is given by $\sum_{f \in F_P} |C_f|$ and the average number of PRE problems "per function" in P is $(\sum_{f \in F_P} |C_f|)/|F_P|$, where C_f is the set of PRE candidate expressions in the function f of the benchmark P.

addition, the RTL representations on both architectures are very different. Note that the number of PRE candidate expressions, i.e., the number of PRE problems in each benchmark on UltraSPARC-III always exceeds that on Xeon.

6.4.1 *Performance Speedups.* Let us look at an example to see how MC-PRE achieves better performance than LCM by removing redundant computations that are only removable speculatively. Our example is `ammp`, which has a speedup of 5.08%, the largest among all the benchmarks, on UltraSPARC-III and 0.76% on Xeon. This benchmark has two most frequently executed functions, called `mm_fv_update_nonbon` and `f_nonbon`, which together consume about 80% of the total execution time of the benchmark on each architecture. Of the two functions, the CFG for `f_nonbon` on UltraSPARC-III provides the unique opportunities for speculative PRE. However, the same opportunities do not exist on Xeon.

Let us analyze how `f_nonbon` is transformed by both MC-PRE and LCM. We

first consider UltraSPARC-III and then Xeon. Figure 23(a) depicts the part of the CFG for `f_nonbon` on UltraSPARC-III, where the usefulness of speculative PRE will be demonstrated below. In the loop identified by the back edge (20,18), there are three PRE expressions in block 18 and three in block 20 . Let m_{18} denote one of the three PRE expressions in block 18 since all the three are handled identically. Similarly, let m_{20} denote one of the three PRE expressions in block 20 since all the three are also handled identically. There are two cases to consider:

PRE for m_{20}. Block 19 is a kill block and X-AVAL(17) = false for m_{20}. Figure 23(b) illustrates how MC-PRE eliminates the computation m_{20} in block 20 via the insertions made on (17,18) and (19,20). There are three PRE expressions like m_{20} in block 20. So the dynamic number of redundant computations removed from block 18 is $3 \times (37008543 - 1168791) = 107519256$. However, LCM cannot carry out this transformation since the insertion on (17,18) is speculative, which introduces the new computation m_{20} that did not exist before along $\langle 17, 18, 19 \rangle$.

PRE for m_{18}. Block 19 is a kill block and X-AVAL(17) = true for m_{18}. In fact, block 14 is a D-block for m_{18} so that an insertion of $h_{m_{18}}$ exists in block 14. Figures 23(c) and (d) compare the MC-PRE and LCM transformations. Note that MC-PRE has eliminated all the recomputations of m_{18} by making a speculative insertion on edge (19,20) with a zero execution frequency. The insertion is speculative since the computation m_{18} did not exist before along any path starting with $\langle 19, 20, 21 \rangle$. Instead, LCM has resorted to an insertion on the back edge (20,18). There are three PRE expressions like m_{18} in block 18. So MC-PRE has eliminated $3 \times (37008543 - 1168791) = 107519256$ more redundant computations than LCM.

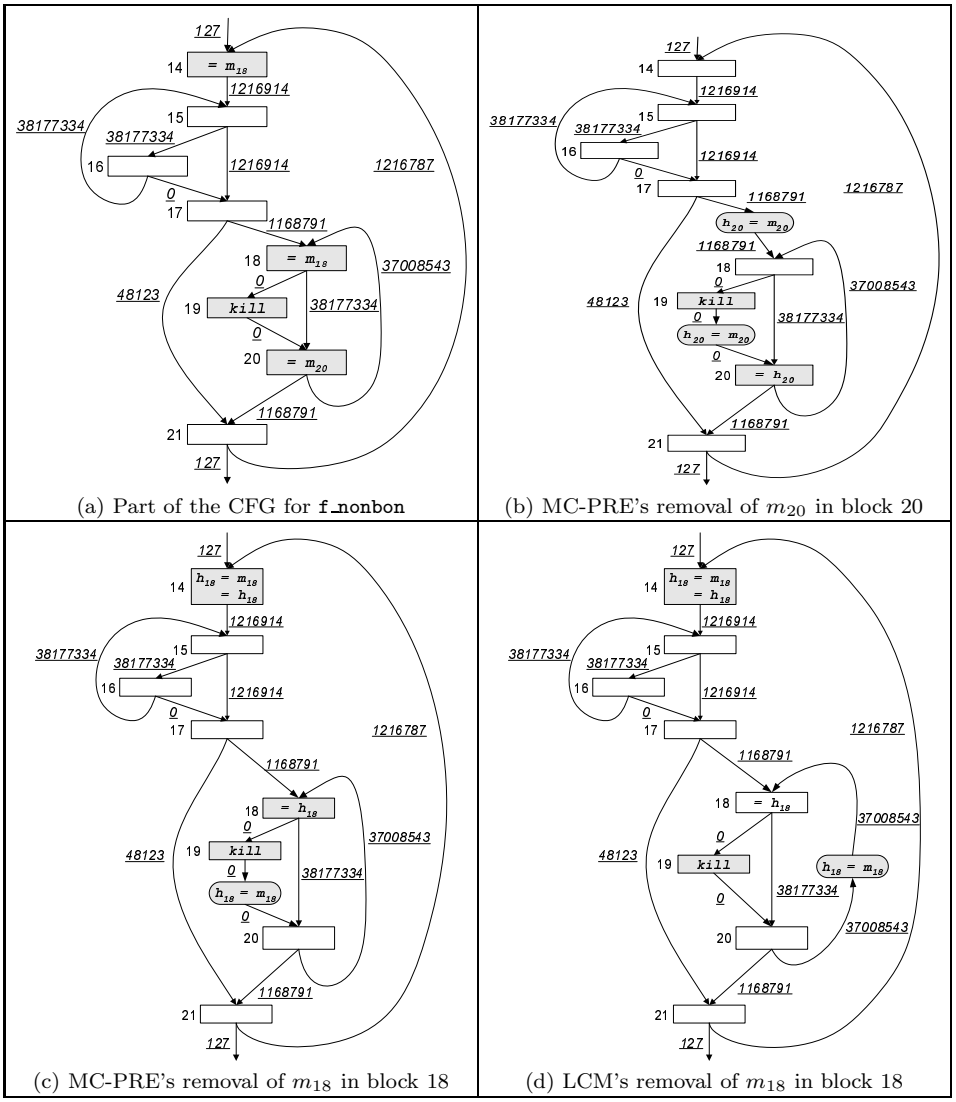
Recall the results of `ammp` given in Figures 21 and 22. In comparison with LCM, MC-PRE eliminates 128.01% (i.e., 249733471) more redundant computations and makes the program run 5.08% (i.e., 41 secs) faster. In `f_nonbon` alone, MC-PRE has eliminated 215038512 more redundant computations than LCM, which represents 86.12% of the overall redundancies eliminated. As a result, `f_nonbon` runs 23 secs faster in the MC-PRE configuration than in the LCM configuration. This represents a 56.10% contribution to the total performance improvement of the benchmark.

On Xeon, however, the `gcc` compiler has managed to show that the branch along the edge (18,19) will never be taken. This causes block 19 to be removed and blocks 18 and 20 merged. Therefore, both MC-PRE and LCM conduct exactly the same transformation in `f_nonbon` for the six PRE expressions discussed above. The small performance improvement by MC-PRE on Xeon is due to the cumulative effects of the speculative PRE performed across all the functions in this benchmark.

6.4.2 *Compilation Overheads.* There are two reasons why MC-PRE is only slightly more expensive than LCM even though MC-PRE relies on a min-cut solver to guarantee both computational and lifetime optimal results. Below we describe the two reasons and support them with ample evidences:

(1) The EFGs Are Significantly Smaller Than the Original CFGs. This fact can be observed clearly in Tables VI and VII. Let us examine Table VI for Xeon only since a similar trend on UltraSPARC-III can be found from Table VII.

Recall that a function f has $|C_f|$ distinct PRE candidate expressions. For each benchmark, Column 2 gives the number of blocks, denoted Col_2, in all the PRE



(a) Part of the CFG for `f_nonbon`

(b) MC-PRE's removal of m_{20} in block 20

(c) MC-PRE's removal of m_{18} in block 18

(d) LCM's removal of m_{18} in block 18

Fig. 23. MC-PRE v.s. LCM solutions to `f_nonbon` in benchmark `ammp` on UltraSPARC-III.

problems in the original CFGs while Column 3 gives the number of blocks, denoted Col_3, in the corresponding nonempty EFGs, i.e., the PRE problems actually solved by the min-cut step of MC-PRE. The values Col_3/Col_2 (i.e., percentage reductions) for all the benchmarks range from 3.71% to 21.59% with an average of only 6.21% for the entire SPECcpu2000 benchmark suite. So the total size measured this way for the original PRE problems in each benchmark has been significantly reduced. From Columns 4 and 5, we observe a significant reduction in the sizes of individual PRE problems. Finally, Columns 6 and 7 show that the largest PRE problems in

Benchmark	#All-BBs		#Avg-BBs		#Max-BBs	
	Original CFGs	Nonempty EFGs $\left(\frac{Col_3}{Col_2}\%\right)$	Original CFGs	Nonempty EFGs	Original CFGs	Nonempty EFGs
164.gzip	59463	10910 (18.35)	42.41	14.19	131	124
175.vpr	432731	32914 (7.61)	70.25	17.57	236	187
176.gcc	9017002	577832 (6.41)	186.17	26.22	1179	949
181.mcf	10802	1960 (18.14)	20.54	9.95	39	32
186.crafty	1199226	99349 (8.28)	220.04	36.21	1078	350
197.parser	115389	16292 (14.12)	28.77	10.63	169	129
252.eon	3296411	324821 (9.85)	82.76	33.33	1343	1244
253.perlbmk	3648014	181340 (4.97)	152.98	21.29	2153	577
254.gap	1630285	110637 (6.79)	61.57	14.08	521	513
255.vortex	859446	151925 (17.68)	42.44	18.36	420	414
256.bzip2	45923	5714 (12.44)	45.47	16.05	138	136
300.twolf	1398389	74432 (5.32)	113.35	24.06	273	261
SPECint2000	21713081	1588126 (7.31)	114.44	23.70	2153	1244
168.wupwise	109877	4079 (3.71)	74.49	15.81	181	148
171.swim	5219	1127 (21.59)	11.30	8.80	17	20
172.mgrid	14586	1732 (11.87)	15.48	8.41	29	32
173.applu	190969	11928 (6.25)	44.75	19.49	81	83
177.mesa	2823267	156251 (5.53)	90.29	15.42	643	373
179.art	14577	2963 (20.33)	23.94	12.94	59	53
183.equake	56416	3728 (6.61)	56.70	16.35	115	105
188.ammp	387945	36565 (9.43)	41.35	11.86	308	138
200.sixtrack	13492816	597225 (4.43)	220.10	68.99	1468	1465
301.apsi	226122	20994 (9.28)	26.17	11.08	84	98
SPECfp2000	17321794	836592 (4.83)	145.14	32.90	1468	1465
SPECcpu2000	39034875	2424718 (6.21)	126.29	26.23	2153	1465

Table VI. Sizes of PRE problems formulated in terms of the original CFGs and nonempty EFGs on Xeon. Let $G^{f,\pi} = (N^{f,\pi}, E^{f,\pi}, W^{f,\pi})$ be the CFG for a function f with respect to an expression π and $G_{st}^{f,\pi} = (N_{st}^{f,\pi}, E_{st}^{f,\pi}, W_{st}^{f,\pi})$ its corresponding EFG. For each benchmark P, Columns 2 and 3 are calculated by $Col_2 = \sum_{f \in F_P} \sum_{\pi \in C_f} |N^{f,\pi}|$ and $Col_3 = \sum_{f \in F_P} \sum_{\pi \in C_f} |N_{st}^{f,\pi}|$, respectively, where F_P and C_f are defined as in Table V. So Column 2 represents the number of blocks in all the original PRE problems in P while Column 3 represents the number of blocks in the nonempty EFGs in P. Their respective averages, calculated by $Col_2 / \sum_{f \in F_P} |C_f|$ and $Col_3 / \sum_{f \in F_P} |\{G_{st}^{f,\pi} \neq \emptyset \mid \pi \in C_f\}|$, are given in Columns 4 and 5. Their respective largest PRE problems, calculated by $\max_{f \in F_P} \max_{\pi \in C_f} |N^{f,\pi}|$ and $\max_{f \in F_P} \max_{\pi \in C_f} |N_{st}^{f,\pi}|$, are given in Columns 6 and 7.

most benchmarks have been made smaller, sometimes quite significantly. However, the largest PRE problems in `swim`, `mgrid`, `applu` and `apsi` are slightly larger. Given a PRE problem on a CFG, the corresponding EFG can be larger than the CFG due to the node splitting performed in Step 4.3 of MC-PRE$_{\text{comp}}$ and the introduction of the source s' and sink t' in Step 4.4 of MC-PRE$_{\text{comp}}$.

Given an EFG $G_{st} = (N_{st}, E_{st}, W_{st})$, finding a minimum cut on the graph using Goldberg's HIPR algorithm takes $O(|N_{st}|^2\sqrt{|E_{st}|})$ [GoldBerg 2003]. Transforming CFGs to their smaller EFGs has reduced the cost of the algorithm significantly.

Recall that in Table V, Column "PRE problems (per function)" for each architecture gives the number of PRE problems to be solved for each benchmark. As shown in Figure 24, the reduced graphs for the majority of the PRE problems in

Benchmark	#All-BBs		#Avg-BBs		#Max-BBs	
	Original CFGs	Nonempty EFGs $\left(\frac{Col_3}{Col_2}\%\right)$	Original CFGs	Nonempty EFGs	Original CFGs	Nonempty EFGs
164.gzip	187584	18605 (9.92)	49.52	17.12	131	127
175.vpr	708229	82886 (11.70)	71.80	28.68	231	200
176.gcc	21493336	983542 (4.58)	220.14	36.50	1177	973
181.mcf	14578	2420 (16.60)	21.07	9.88	39	32
186.crafty	4203577	159669 (3.80)	228.12	40.85	1004	354
197.parser	263012	42479 (16.15)	39.57	18.37	192	167
252.eon	4445834	304187 (6.84)	97.08	35.00	1343	1256
253.perlbmk	12430895	328037 (2.64)	286.55	27.88	2260	1556
254.gap	3445320	219030 (6.36)	74.85	16.53	520	519
255.vortex	1206657	234631 (19.44)	47.90	22.45	420	420
256.bzip2	115477	12654 (10.96)	49.24	19.23	142	144
300.twolf	2761504	146746 (5.31)	122.48	31.78	273	275
SPECint2000	51276003	2534886 (4.94)	159.07	29.20	2260	1556
168.wupwise	122339	5459 (4.46)	67.00	14.40	181	148
171.swim	10919	1404 (12.86)	11.51	8.46	17	20
172.mgrid	16329	2196 (13.45)	15.51	9.42	29	32
173.applu	293381	12981 (4.42)	45.71	18.13	81	83
177.mesa	6388846	148047 (2.32)	128.10	14.82	783	385
179.art	31020	5841 (18.83)	27.87	15.41	59	62
183.equake	88371	8911 (10.08)	60.95	26.21	115	116
188.ammp	478337	51668 (10.80)	43.68	14.65	276	175
200.sixtrack	27265928	2356220 (8.64)	251.63	171.62	1470	1472
301.apsi	288410	30434 (10.55)	26.57	13.85	84	84
SPECfp2000	34983880	2623161 (7.50)	181.41	82.85	1470	1472
SPECcpu2000	86259883	5158047 (5.98)	167.43	43.53	2260	1556

Table VII. Sizes of PRE problems formulated on the original CFGs and nonempty EFGs on UltraSPARC-III. Columns 2 – 6 are computed in exactly the same way as those in Table VI.

nearly all benchmarks are empty. This means that the corresponding EFGs are also empty. The minimum cut for an empty EFG is empty, requiring no invocation to a min-cut algorithm in our implementation (as discussed in Section 6.2).

(2) Nonempty EFGs Are Constructed Efficiently. As discussed in Section 6.2, we have modified `gcc`'s `edge_def` struct in order to facilitate the generation of a graph specification for a nonempty EFG to be fed to the min-cut solver. Thus, a single traversal of a CFG is sufficient to produce the corresponding nonempty EFG.

Finally, we explain why MC-PRE is not so much more costly than LCM in the worst case and why MC-PRE takes slightly less time than LCM in compiling some benchmarks. The compile times of all the benchmarks can be found in Table IV.

In comparison with LCM, `sixtrack`, `gcc` and `perlmk` are the three most expensive benchmarks to compile on two architectures. Figure 25 plots the histograms of the nonempty EFGs for the three benchmarks according to their sizes. The majority of the EFGs for `gcc` and `perlmk` on both architectures have fewer than 100 blocks: 95.15% for `gcc` and 94.96% for `perlmk` on Xeon and 92.00% for `gcc` and 93.69% for `perlmk` on UltraSPARC-III. Due to the existence of some large EFGs, `sixtrack` is the most expensive to compile on both architectures. On Xeon, there are 97.33%

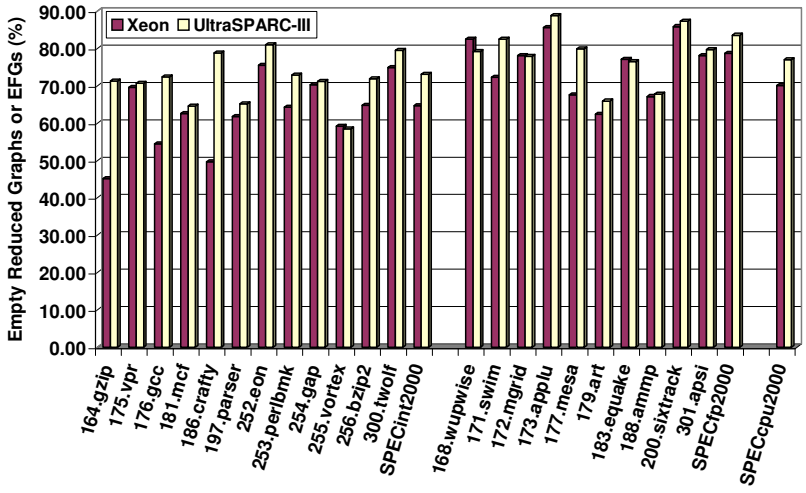


Fig. 24. Statistics about empty reduced graphs (or EFGs) on Xeon and UltraSPARC-III.

EFGs with fewer than 300 blocks. In the remaining EFGs, there are 146 EFGs with 300 – 999 blocks, 79 EFGs with 1200 – 1399 blocks and 6 EFGs with more than 1399 blocks. On UltraSPARC-III, there are 87.64% EFGs with fewer than 300 blocks. In the remaining EFGs, there are 832 EFGs with 300 – 1199 blocks, 762 EFGs with 1200 – 1399 blocks and 103 EFGs with more than 1399 blocks. Despite these relatively large EFGs in `sixtrack`, the compilation time increases of MC-PRE over LCM are only 10.51% on Xeon and 16.29% on UltraSPARC-III.

Let us explain why a few benchmarks, as shown in Table IV, compile faster under MC-PRE than LCM on both architectures. Let us focus on Xeon. The reasons for UltraSPARC-III are the same. In the case of Xeon, `bzip2`, `applu` and `ammp` compile faster under MC-PRE by -2.52%, -1.04% and -1.36%, respectively. As Table V shows, these benchmarks contain 1010, 4267 and 9382 PRE problems to be solved, respectively. LCM performs PRE by conducting four data-flow analysis passes on these problems in parallel. On the other hand, MC-PRE first applies two data-flow analysis passes on these problems in parallel to reduce them into the smaller PRE problems on EFGs. As shown in Figure 24, `bzip2`, `applu` and `ammp` have only 356, 612 and 3083 nonempty EFGs, which require the min-cut step of MC-PRE to be invoked. As Table VI shows, these nonempty EFGs are rather small. So the min-cut step completes very quickly. Finally, MC-PRE performs a third data-flow analysis on all original PRE problems in parallel to avoid insertions and deletions for isolated computations. Therefore, MC-PRE can compile a program faster than LCM if a large number of PRE problems in the program have empty EFGs (which require only three data-flow analysis passes) and the nonempty EFGs are small.

7. CONCLUSION

We have presented the first lifetime optimal algorithm, MC-PRE, that performs speculative PRE by combining code motion and control speculation. We have proved rigorously the optimality of this algorithm. MC-PRE works for the CFGs

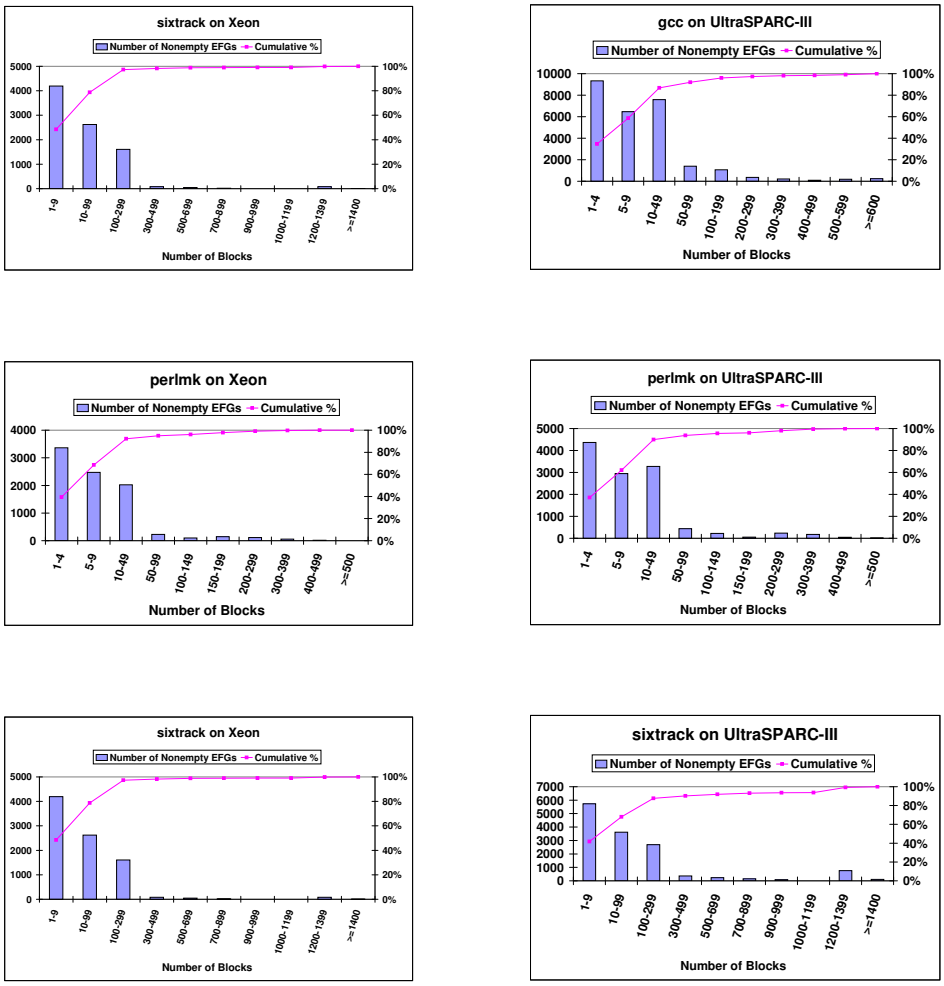


Fig. 25. Nonempty EFGs of `gcc`, `perlmk` and `sixtrack` on Xeon and UltraSPARC-III.

consisting of standard basic blocks so that it can be readily implemented by researchers in a compiler framework. The algorithm is conceptually simple since it is centered around the three standard bit-vector data-flow analyses and a standard min-cut algorithm. Given a PRE problem consisting of a CFG and a PRE candidate expression, we first perform the availability and partial anticipability analyses on the CFG to obtain a single-source, single-sink flow graph (EFG), which tends to be significantly smaller than the original CFG. According to MC-PRE$_{\text{comp}}$, finding any minimum cut on the EFG enables us to find a computational optimal solution. To find the unique lifetime optimal solution, MC-PRE finds a unique minimum cut on the EFG and invokes a third data-flow analysis pass, i.e., the live range analysis to remove all unnecessary insertions and deletions for isolated computations.

We have implemented MC-PRE in `gcc 3.4` and evaluated MC-PRE against LCM using all the 22 C, C++ and FORTRAN 77 benchmarks in SPECcpu2000 on two different architectures. Our experimental results demonstrate that MC-PRE can eliminate more redundant computations in dynamic terms than LCM. This is particularly so for the SPECint2000 benchmarks due to their relatively complex control flow structures. In the performance category, MC-PRE achieves the same or better execution time improvements in 19 benchmarks on Xeon and 20 benchmarks on UltraSPARC-III and causes slight performance degradations in the remaining few benchmarks. The overall performance improvements for all the benchmarks on both architectures are positive. We have analysed some benchmarks to show how MC-PRE improves performance due to redundant computations that can only be eliminated speculatively. We have also conducted a thorough analysis to explain the reasons why MC-PRE has only relatively small compilation overheads even though it contains a min-cut step.

There are some interesting research issues worth pursuing in the future. First, it will be useful to perform PRE while also considering the complex interactions between PRE and some later optimization passes such as register allocation and instruction scheduling. This will, of course, make the PRE optimization become "more" machine-dependent. Second, MC-PRE achieves the computational and lifetime optimal results by considering all expressions separately. Dealing with multiple expressions simultaneously to optimize a certain objective function (rather than merely lifetime ranges) is a natural extension of this work.

REFERENCES

Adl-Tabatabai, A., Cierniak, M., Lueh, G., Parikh, V., and Stichnoth, J. 1998. Fast, effective code generation in a just-in-time Java compiler. In *ACM SIGPLAN' 98 Conference on Programming Language Design and Implementation*. 280–280.

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Alpern, B., Wegman, M. N., , and Zadeck, F. K. 1988. Detecting equality of variables in programs. In *ACM Symposium on Principles of Programming Languages*. 1–11.

Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. L., Leung, S.-T., Sites, R. L., Vandevoorde, M. T., Waldspurger, G. A., and Weihl, W. E. 1997. Continuous profiling: Where have all the cycles gone? In *10th Symposium on Operating System Principles*.

Ball, T. and Larus, J. H. 1994. Optimally profiling and tracing systems. *ACM Transactions on Programming Languages and Systems 16,* 4 (July), 1319–1360.

Ball, T., Mataga, P., and Sagiv, M. 1998. Edge profiling versus path profiling: The showdown. In *ACM Symposium on Principles of Programming Languages*.

Beszedes, A. 2003. Optimizing for space : Measurements and possibilities for improvement. In *GCC Summit 2003*.

Bodik, R. 1999. Path-sensitive value-flow optimizations of programs. Ph.D. thesis, University of Pittsburgh.

Bodik, R., Gupta, R., and Soffa, M. L. 1998. Complete removal of redundant computations. In *ACM SIGPLAN' 98 Conference on Programming Language Design and Implementation*. 1–14.

Briggs, P. and Cooper, K. D. 1994. Effective partial redundancy elimination. In *ACM SIGPLAN' 94 Conference on Programming Language Design and Implementation*. 159–170.

Cai, Q. and Xue, J. 2003. Optimal and efficient speculation-based partial redundancy elimination. In *1st IEEE/ACM International Symposium on Code Generation and Optimization*. 91–102.

Chang, P. P., Mahlke, S. A., Chen, W. Y., Warter, N. J., and Hwu, W. W. 1991. IMPACT: An architectural framework for multiple-instruction-issue processors. In *18th International Symposium on Computer Architecture*. ACM Press, New York, NY, 266–275.

Chekuri, C., Goldberg, A. V., Karger, D. R., Levine, M. S., and Stein, C. 1997. Experimental study of minimum cut algorithms. In *ACM/SIAM Symposium on Discrete Algorithms*. 324–333.

Chow, F. 1983. A portable machine-independent global optimizer — design and measurements. Ph.D. thesis, Computer Systems Laboratory, Stanford University.

Cierniak, M., Lueh, G.-Y., and Stichnoth, J. M. 2000. Practicing JUDO: Java under dynamic optimizations. In *ACM SIGPLAN' 00 Conference on Programming Language Design and Implementation*. 13–26.

Click, C. 1995. Global code motion/global value numbering. In *ACM SIGPLAN' 95 Conference on Programming Language Design and Implementation*. La Jolla, California, 246–257.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms*. Cambridge, Mass.: MIT Press.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (October), 451–490.

Dhamdhere, D. M. 1991. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems 13,* 2, 291–294.

Dhamdhere, D. M., Rosen, B. K., and Zadeck, F. K. 1992. How to analyze large programs efficiently and informatively. In *ACM SIGPLAN '92 Conference on Programming language design and implementation*. ACM Press, 212–223.

Drechsler, K. and Stadel, M. 1993. A variation of Knoop, Rüthing and Steffen's lazy code motion. *ACM SIGPLAN Notices 28,* 5, 29–38.

Ford, L. R. and Fulkerson, D. R. 1962. *Flows in Networks*. Princeton University Press.

GNU GCC Developers. 2004. Private Communication.

GNU Software. 2003. GCC. http://www.gnu.org.

GoldBerg, A. 2003. Network Optimization Library. http://www.avglab.com/andrew/soft.html.

Gupta, R., Berson, D. A., and Fang, J. Z. 1997. Path profile guided partial redundancy elimination using speculation. In *IEEE International Conference on Computer Languages*. 230–239.

Horspool, R. and Ho, H. 1997. Partial redundancy elimination driven by a cost-benefit analysis. In *8th Israeli Conference on Computer System and Software Engineering*. 111–118.

Hosking, A. L., Nystrom, N., Whitlock, D., Cutts, Q., and Diwan, A. 2001. Partial redundancy elimination for access path expressions. *Software Practice and Experience 31,* 6, 577–600.

Hu, T. C. 1970. *Integer Programming and Network Flows*. Addison-Wesley.

Kennedy, K. 1972. Safety of code motion. *International Journal of Computer Mathematics 3,* 2–3, 117–130.

Kennedy, R., Chan, S., Liu, S.-M., Lo, R., and Tu, P. 1999. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems 21,* 3, 627–676.

Kennedy, R., Chow, F. C., Dahl, P., Liu, S.-M., Lo, R., and Streich, M. 1998. Strength reduction via SSAPRE. In *7th International Conference on Compiler Construction*. Lisbon, Portugal, 144–158.

Knoop, J., Collard, J.-F., and Ju, R. 2000. Partial redundancy elimination on predicated code. In *7th International Static Analysis Symposium (SAS 2000)*. Lecture Notes in Computer Science (LNCS). Springer-Verlag, Santa Barbara, CA.

Knoop, J., Rüthing, O., and Steffen, B. 1992. Lazy code motion. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*. San Francisco, California, 224–234.

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

Lin, J., Chen, T., Hsu, W.-C., Yew, P.-C., Ju, R. D.-C., Ngai, T.-F., and Chan, S. 2003. A compiler framework for speculative analysis and optimizations. In *ACM SIGPLAN' 03 Conference on Programming Language Design and Implementation*.

Lo, R., Chow, F., Kennedy, R., Liu, S.-M., and Tu, P. 1998. Register promotion by sparse partial redundancy elimination of loads and stores. 17–19.

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

Morgan, R. 1998. *Building an Optimizing Compiler*. Butterworth-Heinemann.

Muchnick, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc.

Open64. http://open64.sourceforge.net/.

Ramalingam, G. 1996. Data flow frequency analysis. In *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*. 267–277.

Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1988. Global value numbers and redundant computations. In *ACM Symposium on Principles of Programming Languages*. San Diego, California, 12–27.

Rüthing, O. 1998. *Interacting Code Motion Transformations: Their Impact and Their Complexity*. Springer-Verlag.

Scholz, B., Horspool, N., and Knoop, J. 2004. Optimizing for space and time usage with speculative partial redundancy elimination. In *ACM SIGPLAN '04 Conference on Languages, Compilers, and Tools for Embedded Systems*.

Simpson, L. T. 1996. Value-driven redundancy elimination. Ph.D. thesis, Rice University.