TACO, 3(2):115-155, 2006.

A Lifetime Optimal Algorithm for Speculative PRE

Jingling Xue and Qiong Cai
University of New South Wales

A lifetime optimal algorithm, called MC-PRE, is presented for the first time that performs speculative PRE based on edge profiles. In addition to being computationally optimal in the sense that the total number of dynamic computations for an expression in the transformed code is minimized, MC-PRE is also lifetime optimal since the lifetimes of introduced temporaries are also minimized. The key in achieving lifetime optimality lies not only in finding a unique minimum cut on a transformed graph of a given CFG but also in performing a data-flow analysis directly on the CFG to avoid making unnecessary code insertions and deletions. The lifetime optimal results are rigorously proved. We evaluate our algorithm in `GCC` against three previously published PRE algorithms, namely, MC-PRE$_{\text{comp}}$ (Qiong and Xue's computationally optimal version of MC-PRE), LCM (Knoop, Rüthing and Steffen's lifetime optimal algorithm for performing non-speculative PRE) and CMP-PRE (Bodik, Gupta and Soffa's PRE algorithm based on code-motion preventing (CMP) regions, which is speculative but not computationally optimal). We report and analyze our experimental results, obtained from both actual program execution and instrumentation, for all 22 C, C++ and FORTRAN 77 benchmarks from SPECcpu2000 on an Itanium 2 computer system. Our results show that MC-PRE (or MC-PRE$_{\text{comp}}$) is capable of eliminating more partial redundancies than both LCM and CMP-PRE (especially in functions with complex control flow), and in addition, MC-PRE inserts temporaries with shorter lifetimes than MC-PRE$_{\text{comp}}$. Each of both benefits has contributed to the performance improvements in benchmark programs at the costs of only small compile-time and code-size increases in some benchmarks.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Non-numerical algorithms and problems; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; *network problems*

General Terms: Algorithms, Languages, Experimentation, Performance

Additional Key Words and Phrases: Partial redundancy elimination, classic PRE, speculative PRE, computational optimality, lifetime optimality, 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]. 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 3.4.3` consists of a pass for performing PRE, which is based on the well-known algorithm called lazy code mo-

Authors' address: Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia.

tion (LCM) [Knoop et al. 1994]. As another example, `Open64` conducts PRE in static single assignment (SSA) form [Cytron et al. 1991]) using the 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 remove partial redundancies along some paths but never introduce additional computations along any path that did not contain them originally.

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 enforced in classic PRE is relaxed.

There are three computationally optimal algorithms for solving the speculative PRE problem [Bodik 1999; Cai and Xue 2003; Scholz et al. 2004]. However, none of them is also lifetime optimal. Bodik et al. [1998] have also introduced a speculative PRE algorithm, which is referred to as CMP-PRE in this paper, based on code-motion-preventing (CMP) regions. They show that CMP-PRE can eliminate more partially redundant computations than LCM despite its being non-computationally optimal. This paper presents the first algorithm that achieves both computational and lifetime optimal results simultaneously. 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).

In summary, this paper makes the following contributions:

Lifetime Optimality. We present the first lifetime optimal algorithm, MC-PRE (the "MC" stands for Min-Cut), for performing speculative PRE from edge profiles. In addition to being computationally optimal, MC-PRE is also lifetime optimal. The key in achieving the lifetime optimality lies not only in finding a unique minimum cut on a transformed graph of a given CFG but also in performing a data-flow analysis on the CFG to avoid making unnecessary code insertions and replacements for isolated computations. The lifetime optimal results are rigorously proved.

Algorithm. Our computationally optimal algorithm described in [Cai and Xue 2003] assumes single statement blocks. This paper presents our lifetime optimal algorithm in terms of standard basic blocks so that it is directly implementable.

Implementations. We have implemented MC-PRE in `GCC` (GNU Compiler Collection) 3.4.3. 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 fastest implementations available [Chekuri et al. 1997]. As a result, we have also obtained an implementation of our earlier PRE algorithm [Cai and Xue 2003] in `GCC`. This precursor of MC-PRE, denoted MC-PRE$_{\text{comp}}$, is computationally optimal but not lifetime optimal. For comparison purposes, we have also implemented Bodik, Gupta and Soffa's CMP-PRE algorithm [Bodik et al. 1998], which uses a non-Boolean lattice with three values. The data-flow analyses required by CMP-PRE are carried out using modified versions of `GCC`'s bit-vector routines.

Eliminated Computations. We evaluate MC-PRE against LCM and CMP-PRE using all 22 SPECcpu2000 benchmarks on Itanium 2 in terms of redundant computations eliminated. MC-PRE eliminates more non-full (i.e., strictly partial) redundancies than LCM and more non-full redundancies that are only speculatively removable (i.e., that are not removable by LCM) than CMP-PRE. This is particularly pronounced for functions with complex control flow. In the case of SPECint2000, MC-PRE removes between 31.39% and 147.56% (an average of 90.13%) more non-full redundancies than LCM and between 0.52% and 280.47% (an average of 58.15%) more non-full redundancies that are only speculatively removable than CMP-PRE. In the case of SPECfp2000, these percentage increases are 0.10% – 264.67% (33.84%) over LCM and 0.00% – 89.98% (15.23%) over CMP-PRE.

Lifetimes of Introduced Temporaries. MC-PRE uses insertions with shorter lifetimes than MC-PRE$_{\text{comp}}$ in all 22 benchmarks used. In addition, MC-PRE is comparable with or better than LCM and CMP-PRE in terms of this criterion in these benchmarks despite that these two previous PRE algorithms are not computationally optimal for solving the speculative PRE problem.

Performance Improvements. By eliminating more redundant computations than LCM and CMP-PRE and using insertions with the shortest lifetimes possible (shorter than MC-PRE$_{\text{comp}}$), MC-PRE achieves nearly the same or better performance results than the other three PRE algorithms in all 22 benchmarks. By keeping the lifetimes of introduced temporaries to a minimum, MC-PRE yields faster codes than MC-PRE$_{\text{comp}}$ in 19 of these benchmarks. In the case of SPECint2000, MC-PRE outperforms MC-PRE$_{\text{comp}}$, LCM and CMP-PRE in nearly all its 12 benchmarks. In the case of SPECfp2000, LCM has succeeded in eliminating over 80% of the non-full redundancies in nine out of its 10 benchmarks. However, MC-PRE still achieves nearly the same or better performance results in these benchmarks than the other three algorithms. MC-PRE is optimal since it can eliminate all partial (full and non-full) redundancies by using temporaries with the shortest lifetimes possible as long as they can be eliminated profitably (with respect to a given edge profile). As a result, MC-PRE can potentially achieve better performance than LCM and CMP-PRE even in programs where LCM and CMP-PRE have eliminated most of their non-full redundancies. This has happened to `mesa` and `sixtrack` in the SPECfp2000 suite. For `sixtrack`, MC-PRE achieves a speedup of 7.03% over LCM since all non-full redundancies that MC-PRE eliminates but LCM does not are from its hottest functions. For `mesa`, MC-PRE achieves a speedup of 5.75% (5.73%) over LCM (CMP-PRE) since MC-PRE has eliminated redundant computations from a number of expensive expressions that both LCM and CMP-PRE

have failed to remove in one of its hottest functions.

Low Compilation Overhead. Conventional wisdom suggests that a min-cut solver may be too expensive to be practically useful in our setting. Instead of directly operating on CFGs, a min-cut solver operates on significantly reduced subgraphs of CFGs. As a result, 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. The compile times from MC-PRE, MC-PRE$_{\text{comp}}$ and CMP-PRE do not differ significantly.

Low Space Overhead. A PRE algorithm inserts and deletes computations in a program. As a result, it may cause the static code size in a program to be increased or decreased. MC-PRE are comparable with LCM and CMP-PRE in terms of this criterion while MC-PRE$_{\text{comp}}$ results in slightly larger binaries in some benchmarks.

PRE is almost universally used in optimizing compilers. The experimental results as summarized above show that MC-PRE can be practically employed as a PRE pass in a profile-guided optimizing compiler framework.

We illustrate the basic idea of our algorithm using an example given in Figure 1 by focussing on one single PRE candidate expression, $a + b$. After the local PRE pass has been performed within basic blocks, there are only two kinds of candidate computations of $a + b$ to be dealt with by the global PRE pass:

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

where UB is the set of blocks where $a + b$ is upwards exposed and DB is the set of blocks where $a + b$ is both downwards exposed and preceded by assignments to a or b. (Note that UB and DB are not symmetrically defined in this paper.)

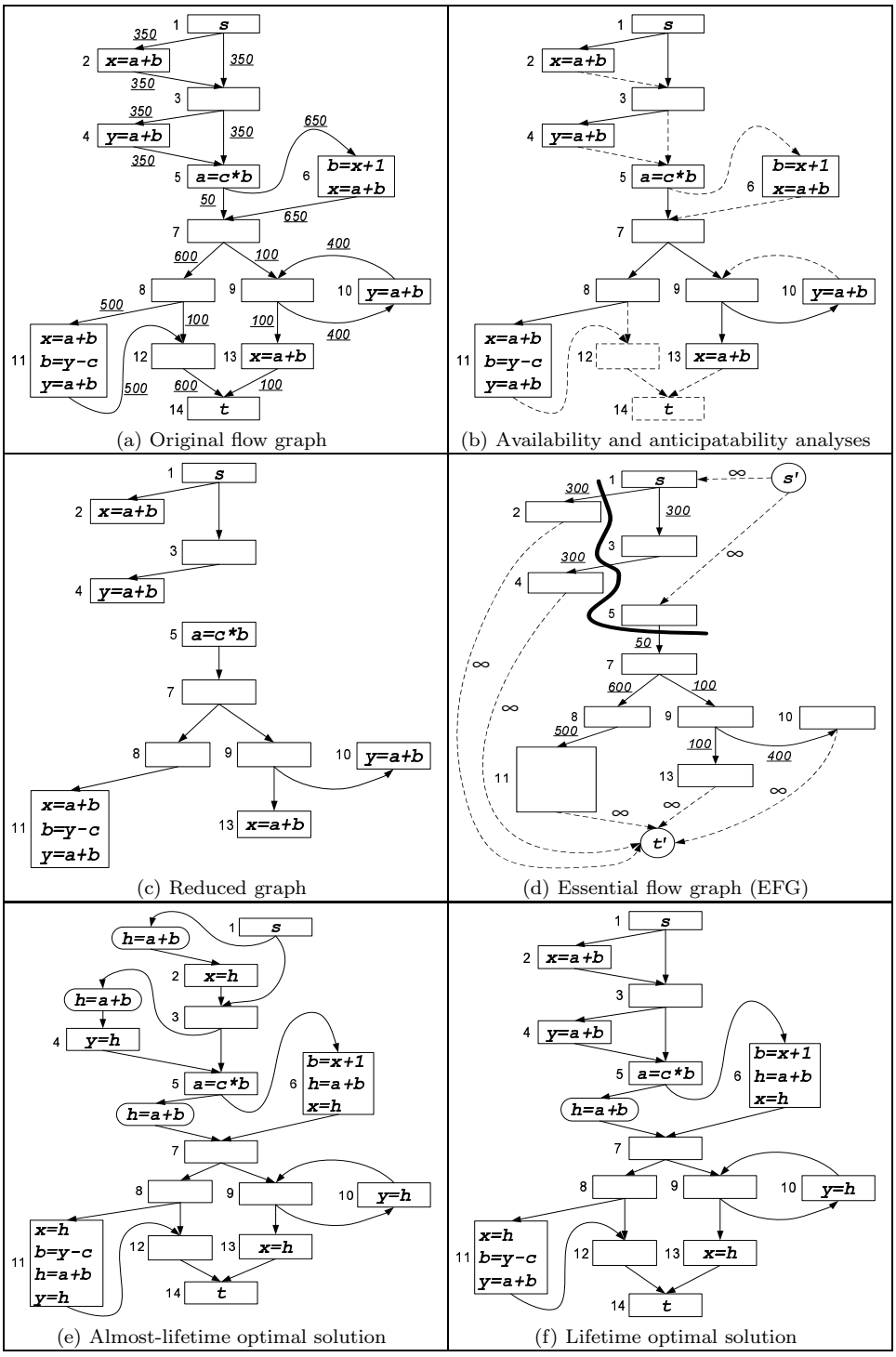
MC-PRE starts with by performing two standard data-flow analyses — the forward availability and the backward partial anticipatability — on the CFG given in Figure 1(a) to identify all non-essential edges and nodes. Figure 1(b) duplicates the CFG given in Figure 1(a) with all the non-essential edges and nodes being depicted in dashes. 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). We apply a min-cut algorithm to the EFG to obtain the 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:

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

This transformation would be lifetime optimal if insertions and deletions for isolated computations were allowed. Thus, such an *almost-lifetime* optimal transformation corresponds to the Almost LCM transformation in classic PRE [Knoop et al. 1992; 1994]. For illustration purposes, Figure 1(e) depicts the transformed code, which is not actually generated. The first two sets prescribe the insertions and replacements



(a) Original flow graph

(b) Availability and anticipatability analyses

(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.

for the upwards exposed computations in UB, respectively. The last set specifies the insertions and replacements for the downwards exposed computations in DB.

This work recognizes that finding a special minimum cut alone is not sufficient to guarantee lifetime optimality for speculative PRE. In the transformed program shown in Figure 1(e), there are three isolated computations. The upwards exposed 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 upwards exposed computation $a + b$ in block 4 is also 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 and associated deletions, MC-PRE performs a third data-flow analysis pass on the original CFG given in Figure 1(a). As a result, the unique lifetime optimal transformation, $\mathcal{LO}$, found by MC-PRE is:

$$\begin{aligned} \mathsf{U\text{-}INS}_{\mathcal{LO}} &= \{(5,7)\} \\ \mathsf{U\text{-}DEL}_{\mathcal{LO}} &= \{10,11,13\} \\ \mathsf{D\text{-}INSDEL}_{\mathcal{LO}} &= \{6\} \end{aligned} \tag{3}$$

Performing this code motion on the CFG in Figure 1(a) leads to the optimally transformed code shown in Figure 1(f). The dynamic number of computations for $a + b$ has been reduced optimally from 2850 to 1900. (For the same expression, the reduction by the profile-independent LCM would be from 2850 to 2450.)

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 gives some background information and introduces the notions of computational optimality and lifetime optimality for speculative PRE. 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 evaluate MC-PRE against MC-PRE$_{\text{comp}}$, LCM and CMP-PRE using SPECcpu2000 on an Itanium 2 computer system. 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 1998; Knoop et al. 1992; 1994; Rosen et al. 1988; Rüthing et al. 2000; Simpson 1996]. In particular, Knoop et al. [1994] describe a uni-directional bit-vector formulation, known as *LCM*, 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. Hailperin [1998] generalizes LCM so that the generalized version also performs constant propagation and strength reduction. Kennedy et al. [1998] extend

the SSAPRE algorithm to perform strength reduction and linear function test replacement. Steffen et al. [1990] present a PRE algorithm achieving computational optimality with respect to the Herbrand interpretation and later [1991] a less powerful but more efficient variant together with an extension to uniformly cover also strength reduction. 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. [1998] introduce the first two algorithms on performing profile-guided speculative PRE. Due to the local heuristics used, both algorithms are not computationally optimal. Later, computationally optimal results are achieved by the three different algorithms described in [Bodik 1999; Cai and Xue 2003; Scholz et al. 2004], which all rely on finding a minimum cut to obtain the required insertion points. However, none of these three algorithms is also lifetime optimal due to the arbitrary minimum cuts chosen in their algorithms. Bodik [1999] suggested to achieve lifetime optimal results by choosing cut edges as close as possible to the sinks of his flow network; but the details of this suggestion are missing. In order to reduce the lifetimes of introduced temporaries, Bodik [1999] used a non-optimal algorithm (presented earlier in [Bodik et al. 1998] and denoted CMP-PRE here) to perform speculative PRE based on code-motion-preventing (CMP) regions. They show that CMP-PRE can eliminate more non-full redundancies than LCM despite its being non-computationally optimal.

In this paper, we extend our earlier algorithm [Cai and Xue 2003] so that the new algorithm, MC-PRE, can also achieve lifetime optimality for the first time. As a result, MC-PRE introduces temporary registers only when necessary, with the shortest lifetimes possible. The lifetime optimality is achieved by finding not only a special minimum cut and but also performing one additional data-flow analysis to avoid making insertions and replacements for isolated computations.

Bodik et al. [1998] apply control flow restructuring as an alternative to speculation to enable code motion. They implemented their algorithm in the IMPACT compiler [Chang et al. 1991] and evaluated its effectiveness using SPEC95. They found that the number of redundancies that *cannot* be removed by speculation alone is negligible. In addition, such a negligible number of redundancies is eliminated at the cost of over 30% code explosion for most benchmarks. Steffen [1996] eliminates all partial redundancies in a program essentially by unrolling the program as far as necessary and then attempting to minimize the potentially exponential code-size explosion by collapsing nodes that are so-called bisimilar. In this paper, MC-PRE achieves a complete removal of all partial redundancies removable by using control speculation from edge profiles at negligible space overheads.

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

3. PRELIMINARIES

Section 3.1 recalls some concepts and results about directed graphs, flow networks and defines the notion of minimum cut used. In Section 3.2, we introduce control flow graphs (CFGs) and the local predicates used in our data-flow analysis. In Section 3.3, we formulate speculative PRE as a code motion transformation and define 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:

$$\begin{aligned} \mathsf{pred}(G, n) &= \{m \mid (m, n) \in E\} \\ \mathsf{succ}(G, n) &= \{m \mid (n, m) \in E\} \end{aligned} \tag{4}$$

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:

$$(S, T) = \{(n, m) \in E \mid n \in S, m \in T\} \tag{5}$$

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*):

$$cap(C, \overline{C}) = \sum_{(m,n) \in (C, \overline{C})} c(m, n) \tag{6}$$

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

3.2 Control Flow Graphs

A control flow graph (CFG) is a directed graph annotated with an edge profile. We represent a CFG as a weighted graph $G = (N, E, W)$, where N is the set of basic blocks, E the set of control flow edges, and $W : E \mapsto \mathbb{N}$ is the set of non-negative integers representing the execution frequencies of all flow edges. In addition, $s \in N$ denotes the unique *entry block* without any predecessors and $t \in N$ the unique *exit block* without any successors. Furthermore, every block is assumed to lie on some path from s to t. For convenience, we also write $W(n)$ to represent the execution frequency of a block $n \in N$. Nodes and (basic) blocks are interchangeable.

The profiling information input to a profile-guided algorithm provides only an approximation of actual execution frequencies. Therefore, an edge or node with a

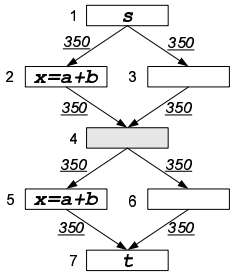


Fig. 2. Profitability in partial redundancy elimination by using control speculation. Although being partially redundant, $a+b$ in block 5 will not be eliminated since it is not profitably removable.

zero frequency is interpreted as least frequently rather than never executed at all.

Assumption 3.1. For every node or edge x in G, its execution frequency is assumed to be nonzero, i.e., $W(x) > 0$.

Assumption 3.2. The following assumption about an edge profile is made:

$$W(n) \;=\; \sum_{m \in \mathsf{pred}(G,n)} W(m,n) \;=\; \sum_{m \in \mathsf{succ}(G,n)} W(n,m) \tag{7}$$

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.* 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. To avoid using a highly parameterized notation, we present our algorithm for a generic CFG $G = (N, E, W)$ and a generic expression π. An assignment to some operand 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, two PRE problems on the same CFG are distinct if their corresponding expressions are distinct.

3.2.2 *Partial, Full and Non-Full Redundancies.* An expression is *partially redundant* if the value computed by the expression is available on some control flow paths reaching that expression. A partially redundant expression becomes *fully redundant* if the value of the expression is available on all control flow paths reaching that expression. The redundancies that are partial but not full are referred to as *non-full redundancies* in this paper (and as *strictly partial redundancies* elsewhere). The proposed PRE algorithm aims at removing all partial (full or non-full) redundancies as long as they can be removed profitably by using control speculation.

A computation of π in $G = (N, E, W)$ is said to be *profitably removable* if it is partially redundant and its elimination can reduce the total number of evaluations of π with respect to W. For example, no code motion should be performed for the example given in Figure 2 since the partially redundant computation $a + b$ in block 5 is not profitably removable. However, a PRE algorithm must eliminate all partial redundancies that are profitably removable in order to be computationally optimal.

3.2.3 *Local Redundancies and Local Predicates.* PRE is a global optimization for removing partial redundancies across the basic blocks. Standard techniques such as local common subexpression elimination (LCSE) are assumed to have been carried out on basic blocks. Thus, 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, the four local predicates for a given expression π are used in the normal manner. $\mathsf{ANTLOC}(n)$ is true iff π is *locally anticipatable* on entry to block n. $\mathsf{AVLOC}(n)$ is true iff π is *locally available* on exit from block n. $\mathsf{TRANSP}(n)$ is true iff block n is transparent to π, i.e., block n does not contain any modification to π. Finally, $\mathsf{KILL}(n) = \neg\mathsf{TRANSP}(n)$. We call n a *kill node* if $\mathsf{KILL}(n) = \mathsf{true}$.

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:

$$UB = \{n \in N \mid \mathsf{ANTLOC}(n)\} \tag{8}$$

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:

$$DB = \{n \in N \mid \mathsf{AVLOC}(n) \wedge \mathsf{KILL}(n)\} \tag{9}$$

By definition, a block n can be both a U-block and a D-block. In this case, n must be a kill block and contains two distinct PRE candidate computations of π. 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 called a U-block but not also a D-block.

The entry block s is assumed to have an (imaginary) definition for every variable, which precedes all existing statements, to represent whatever value the variable may have when s is entered. Note that the entry and exit blocks need not be empty.

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.

3.2.4 *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 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

When eliminating redundant computations for an expression π, code insertions of the form $h_\pi = \pi$, where h_π is a distinct temporary, are introduced so that the saved value of π in h_π can be reused later. A *speculative PRE transformation*, denoted T, for an expression π in a CFG $G = (N, E, W)$ is characterized by three sets:

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

—$\mathsf{D\text{-}INSDEL}_T \subseteq DB$ is a set of D-blocks such that each of these downwards exposed computations of π (also called *replacement computations*) will be replaced by h_π and immediately preceded by an insertion of $h_\pi = \pi$ (see block 6 in Figure 1(f)).

—$\mathsf{U\text{-}INS}_T \subseteq E$ is a set of flow edges (called *insertion edges*) on which $h_\pi = \pi$ will be inserted. (The insertion will take place in a new block created on the edge.)

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 realized by a transformation T with respect to the edge profile W:

$$W(T) \;=\; W(UB) + W(DB) - \mathit{benefit}(T) \tag{10}$$

where $\mathit{benefit}(W)$ gives rise to the total number of eliminated computations of π:

$$\mathit{benefit}(T) \;=\; W(\mathsf{U\text{-}DEL}_T) - W(\mathsf{U\text{-}INS}_T) \tag{11}$$

Definition 3.6. Consider an expression π in $G = (N, E, W)$. A PRE transformation is *correct* if every use of h_π is identified with a definition of h_π on every path reaching the use. $\mathcal{CM}_{Cor}$ denotes the set of all such correct transformations for π.

Definition 3.7. Consider an expression π in $G = (N, E, W)$. A PRE transformation T is said to be *computationally optimal* if the following statements are true:

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

$\mathcal{CM}_{CompOpt}$ denotes the set of all computationally optimal transformations for π.

The *lifetime* (or *live range)* of a variable is the portion of a program in which the variable's value must be preserved. This work uses exactly the same notion of lifetime optimality as defined in classic PRE [Knoop et al. 1994]. Informally, a PRE transformation is lifetime optimal if (1) every computation of π that it deletes must be deleted by every other computationally optimal transformation, and (2) every definition of h_π that it inserts must have the shortest lifetime possible for every use of h_π that the definition reaches. This notion of lifetime optimality is recasted below in terms of our notations for describing a PRE transformation.

Definition 3.8. Consider an expression π in $G = (N, E, W)$. $T \in \mathcal{CM}_{CompOpt}$ is *lifetime better* than $T' \in \mathcal{CM}_{CompOpt}$ if the following statements are true:

(1) $\mathsf{U\text{-}DEL}_T \subseteq \mathsf{U\text{-}DEL}_{T'}$.
(2) $\mathsf{D\text{-}INSDEL}_T \subseteq \mathsf{D\text{-}INSDEL}_{T'}$.
(3) Let $d^n_{u,v}$ be the definition of h_π inserted on an arbitrary but fixed edge $(u, v) \in \mathsf{U\text{-}INS}_T$ that reaches an arbitrary but fixed use of h_π in a U-block, n, along its incoming edge $(m, n) \in E$ according to T. Let $d^n_{u',v'}$ be the definition of h_π inserted on $(u', v') \in \mathsf{U\text{-}INS}_{T'}$ that reaches the same use of h_π in the same U-block n along the same incoming edge (m, n) according to T'. Then the lifetime of $d^n_{u,v}$ is no longer than the lifetime of $d^n_{u',v'}$.

$T \in \mathcal{CM}_{CompOpt}$ is *lifetime optimal* if T is lifetime better than all $T' \in \mathcal{CM}_{CompOpt}$. $\mathcal{CM}_{LifeOpt}$ denotes the set of all such lifetime optimal transformations for π.

Under Assumption 3.1, we shall show constructively that $|\mathcal{CM}_{LifeOpt}| = 1$.

Note that $\mathcal{CO}_1$ given in (2) and $\mathcal{LO}$ in (3) result in the transformed codes depicted in Figures 1(e) and (f), respectively. $\mathcal{LO}$ 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.

4.1 Computationally Optimal Transformations

The MC-PRE$_{\text{comp}}$ algorithm given in Figure 3 takes a CFG and returns a computational optimal transformation denoted by $\mathcal{CO}$. MC-PRE$_{\text{comp}}$ works for standard basic blocks while our earlier algorithm [Cai and Xue 2003] assumes single statement blocks. As a result, Step 3.3(a) of MC-PRE$_{\text{comp}}$ is new and is required to perform a conceptual splitting for some basic blocks as illustrated in Figure 4.

We will focus on describing the parts of MC-PRE$_{\text{comp}}$ that are different from our earlier algorithm and state some lemmas that will be used later for the proofs of our lifetime optimal algorithm. Steps 1 and 2 of MC-PRE$_{\text{comp}}$ find $\mathsf{D\text{-}INSDEL}_{\mathcal{CO}}$ and $\mathsf{U\text{-}DEL}_{\mathcal{CO}}$ trivially based on the local predicates for basic blocks. Step 3 constructs $\mathsf{U\text{-}INS}_{\mathcal{CO}}$ by relying on two global data-flow analysis passes. We describe this step below and illustrate it with two examples. The first example does not use Step 3.3(a) while the second is designed 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 $a + b$. The *UB* and *DB* sets for this example can be found in (1). By executing Steps 1 and 2 of MC-PRE$_{\text{comp}}$, we obtain trivially:

$$\begin{array}{rcccl} \mathsf{D\text{-}INSDEL}_{\mathcal{CO}} & = & DB & = & \{6, 11\} \\ \mathsf{U\text{-}DEL}_{\mathcal{CO}} & = & UB & = & \{2, 4, 10, 11, 13\} \end{array} \tag{12}$$

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

4.1.1 *Steps 3.1 and 3.2: Perform the Availability and Partial Anticipatability Analyses to Obtain a Reduced Graph.* These are the same two steps used in our earlier algorithm [Cai and Xue 2003] for single statement blocks. They serve to remove all non-essential flow edges (and nodes) from a CFG since a PRE transformation that makes code insertions on such edges cannot be computationally optimal under Assumption 3.1. Once the two global flow analyses in Step 3.1 are done, the four global predicates are defined on the flow edges of G in Step 3.2(a). According to these predicates, a flow edge will be either *insertion-redundant* or *insertion-useless* or both. As a result, a flow edge is either *essential* or *non-essential*. The forward availability analysis detects the insertion-redundant edges while the backward partial anticipatability 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 3.2(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.

For the CFG given in Figure 1(a), Figure 1(b) depicts all the non-essential edges and nodes in dashes. Figure 1(c) depicts the reduced graph obtained.

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 G_{rd} shown in Figure 1(c).

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

1 $\mathsf{D\text{-}INSDEL}_{\mathcal{CO}} = DB$.
2 $\mathsf{U\text{-}DEL}_{\mathcal{CO}} = UB$.
3 Construct $\mathsf{U\text{-}INS}_{\mathcal{CO}}$ as follows:
3.1 Perform the availability and partial anticipatability 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 anticipatability system (initialized to false):

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

3.2 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:

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

3.3 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 $\mathit{top\_part}(n) = \textbf{if } \mathsf{TOP}(n) \wedge \mathsf{BOT}(n) \ \rightarrow\ n+ \ [] \ \textbf{else} \ \rightarrow\ n \ \textbf{fi}$
Let $\mathit{bot\_part}(n) = \textbf{if } \mathsf{TOP}(n) \wedge \mathsf{BOT}(n) \ \rightarrow\ n- \ [] \ \textbf{else} \ \rightarrow\ n \ \textbf{fi}$
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) $S_{mm} = \{n \in N_{mm} \mid \mathsf{pred}(G_{mm}, n) = \emptyset\}$
$T_{mm} = \{n \in N_{mm} \mid \mathsf{succ}(G_{mm}, n) = \emptyset\}$
3.4 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:

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

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

Fig. 3. An algorithm that guarantees computationally optimal results.

By Lemma 4.1, all the five U-blocks in the example are contained in G_{rd}. By Lemma 4.2, the D-block 11 is contained in G_{rd} but the D-block 6 is not.

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. □

4.1.2 *Step 3.3: 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 3.3(a), a node n in G_{rd} is split when $\mathsf{TOP}(n) = \mathsf{BOT}(n) = \mathsf{true}$. (Note that $\forall\, n \in DB : \mathsf{BOT}(n) = \mathsf{false}$.) 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 directed into $n+$ and the outgoing edges of n directed out of $n-$. There are no edges between $n+$ and $n-$. This splitting process results in the graph G_{mm} defined in Step 3.3(b). If a node n is not split, then $\mathit{top\_part}(n) = \mathit{bot\_part}(n) = n$. In 3.3(c), S_{mm} consists of all *source nodes* (without incoming edges) and T_{mm} of 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. $\forall\, n \in N_{rd} : \mathsf{KILL}(n) \Longleftarrow \mathit{bot\_part}(n) \in S_{mm}$.

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 3.3(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 3.3(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 4.1.5 by an example given in Figure 4.

4.1.3 *Step 3.4: Obtain a Single-Source, Single-Sink EFG.* This is a standard transformation [Cormen et al. 1990]. In 3.4(a), the new entry node s' and new exit node t' are added. In 3.4(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.

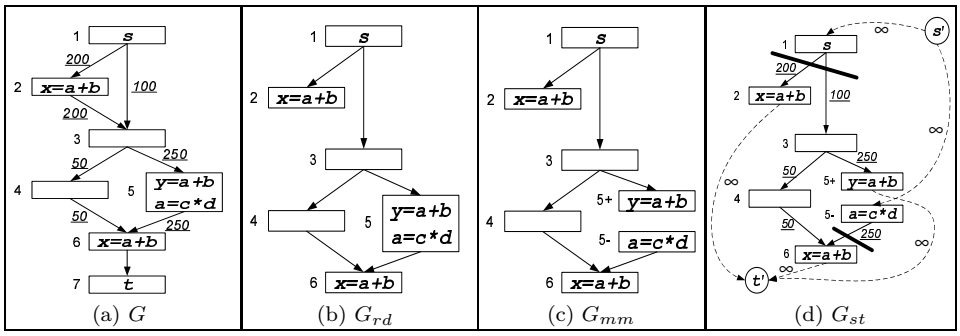


Fig. 4. An example illustrating Step 3.3 of MC-PRE$_{\text{comp}}$.

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}.

4.1.4 *Step 3.5: Find a Minimum Cut.* $\mathsf{U\text{-}INS}_{\mathcal{CO}}$ 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

$$C_1 \;=\; \{(1,2),(3,4),(5,7)\} \tag{13}$$

we find that

$$\mathsf{U\text{-}INS}_{\mathcal{CO}} \;=\; \Gamma^{-1}(C_1) \;=\; C_1 = \{(1,2),(3,4),(5,7)\} \tag{14}$$

since $G_{mm} = G_{rd}$. The resulting transformation $\mathcal{CO}$ defined by (12) and (14) is the same as $\mathcal{CO}_1$ given in (2). 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

$$C_2 \;=\; \{(1,2),(1,3),(5,7)\} \tag{15}$$

$\mathsf{U\text{-}INS}_{\mathcal{CO}}$ will become:

$$\mathsf{U\text{-}INS}_{\mathcal{CO}} \;=\; \Gamma^{-1}(C_2) \;=\; C_2 = \{(1,2),(1,3),(5,7)\} \tag{16}$$

Let us combine (12) and (16) and denote this resulting transformation by $\mathcal{CO}_2$:

$$\begin{array}{rcccl} \mathsf{U\text{-}INS}_{\mathcal{CO}_2} & = & C_2 & = & \{(1,2),(1,3),(5,7)\} \\ \mathsf{U\text{-}DEL}_{\mathcal{CO}2} & = & UB & = & \{2,4,10,11,13\} \\ \mathsf{D\text{-}INSDEL}_{\mathcal{CO}_2} & = & DB & = & \{6,11\} \end{array} \tag{17}$$

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

4.1.5 *One More Example.* We illustrate Step 3.3 of MC-PRE$_{\text{comp}}$ using a simple example presented in Figure 4. For the CFG given in Figure 4(a), $(2, 3)$ and $(6, 7)$ are the only non-essential edges. So the reduced graph G_{rd} is the one displayed in Figure 4(b). In Step 3.3(a), we obtain $\mathit{top\_part}(n) = \mathit{bot\_part}(n) = n$ for $n \in \{1, 2, 3, 4, 6\}$, $\mathit{top\_part}(5) = 5+$ and $\mathit{bot\_part}(5) = 5-$. That is, block 5 is

split into 5+ and 5− to produce in Step 3.3(b) the multi-source, multi-sink graph G_{mm} depicted in Figure 4(c). The corresponding single-source, single-sink EFG G_{st} is shown in Figure 4(d). In Step 3.3(c), we obtain $S_{mm} = \{1, 5-\}$ and $T_{mm} = \{2, 5+, 6\}$. In this simple case, the unique minimum cut found in Step 3.5(a) for the EFG G_{st} is $\mathcal{C} = \{(1,2),(1,3),(5-,6)\}$. In Step 3.5(b), we map these edges back to the flow edges in the original CFG: $\mathsf{U\text{-}INS}_{\mathcal{CO}} = \Gamma^{-1}(\mathcal{C}) = (1,2),(1,3),(5,6)\}$. This gives rise to the following transformation:

$$\begin{array}{rcl} \mathsf{U\text{-}INS}_{\mathcal{CO}} & = & \{(1,2),(1,3),(5,6)\} \\ \mathsf{U\text{-}DEL}_{\mathcal{CO}} & = & \{2,5,6\} \\ \mathsf{D\text{-}INSDEL}_{\mathcal{CO}} & = & \emptyset \end{array} \tag{18}$$

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

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 $G = (N, E, W)$ and shows further that $\mathcal{CM}_{LifeOpt} = \{\mathcal{LO}\}$ (Theorem 5.7). The minimum cuts in a flow network may not be unique. This implies that a PRE problem may have more than one computationally optimal transformation: $|\mathcal{CM}_{CompOpt}| \geqslant 1$. 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 insertions and deletions for isolated computations as we discussed in Section 1. Let S_{cut} be the set of all minimum cuts found in Step 3.5(a) of MC-PRE$_{\text{comp}}$ for a PRE problem. Let $\mathcal{T}_{cut}$ be the set of all corresponding computationally optimal transformations:

$$\mathcal{T}_{cut} = \left\{ \left(\begin{array}{rcl} \mathsf{U\text{-}INS}_{\mathcal{CO}} & = & \Gamma^{-1}(C), \\ \mathsf{U\text{-}DEL}_{\mathcal{CO}} & = & UB, \\ \mathsf{D\text{-}INSDEL}_{\mathcal{CO}} & = & DB \end{array} \right) \middle| C \in S_{cut} \right\} \tag{19}$$

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 lifetime optimal – some code motion may have been done unnecessarily. As discussed in Section 4.1.4, our running example has two minimum cuts in the EFG G_{st}:

$$\mathcal{T}_{cut} = \{\mathcal{CO}_1, \mathcal{CO}_2\} \tag{20}$$

where $\mathcal{CO}_1$ is defined in (2) and $\mathcal{CO}_2$ in (17). For this example, the lifetime optimal solution $\mathcal{LO}$ is the one given in (3) but $\mathcal{LO} \notin \mathcal{T}_{cut}$.

Figure 5 gives our final algorithm, called MC-PRE, for finding a lifetime optimal transformation, denoted $\mathcal{LO}$, 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.

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 3.1 of MC-PRE$_{\text{comp}}$.
2. Obtain $G_{rd} = (N_{rd}, E_{rd}, W_{rd})$ from G as in Step 3.2 of MC-PRE$_{\text{comp}}$.
3. Obtain $G_{mm} = (N_{mm}, E_{mm}, W_{mm})$ from G_{rd} as in Step 3.3 of MC-PRE$_{\text{comp}}$.
4. Obtain $G_{st} = (N_{st}, E_{st}, W_{st})$ from G_{mm} as in Step 3.4 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:

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

$$\mathsf{N\text{-}LIVE}_\Lambda(n) = \mathsf{ANTLOC}(n) \vee (\mathsf{X\text{-}LIVE}_\Lambda(n) \wedge \mathsf{TRANSP}(n))$$

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

Fig. 5. An algorithm that guarantees lifetime optimal results.

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

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 $\mathcal{LO}$ in (3) for our running example is derived. 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 5.6, 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:

$$\begin{aligned} \mathsf{U\text{-}INS}_{\mathcal{ALO}} &= \mathcal{C}'_\Lambda = \Gamma^{-1}(\mathcal{C}_\Lambda) \\ \mathsf{U\text{-}DEL}_{\mathcal{ALO}} &= UB \\ \mathsf{D\text{-}INSDEL}_{\mathcal{ALO}} &= DB \end{aligned} \tag{21}$$

$\mathcal{ALO}$ is the lifetime best in $\mathcal{T}_{cut}$. This result is not directly proved in this paper since $\mathcal{ALO}$ is not actually used; but it is implied by the proof of Theorem 5.7. In fact, $\mathcal{ALO}$ corresponds to ALCM (Almost LCM) [Knoop et al. 1992; 1994].

In the case of our running example, there are only two minimum cuts C_1 and C_2, which are given in (13) and (15), 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 (2) and (17), respectively. Accordingly, $\mathcal{ALO} = \mathcal{CO}_1$ since $\mathcal{CO}_1$ is lifetime better than $\mathcal{CO}_2$.

4.2.2 *Step 6: Solve the "Backward Live Range Analysis for h_π" in the Original CFG.* We perform a third data-flow analysis *in the original CFG* in order to identify all isolated computations in Steps 7 and 8. Equivalently, we were actually performing a backward live range analysis for the temporary h_π in the transformed CFG obtained by applying $\mathcal{ALO}$ to the original CFG. For our running example, the transformed CFG according to $\mathcal{ALO} = \mathcal{CO}_1$ can be found in Figure 1(e).

4.2.3 *Step 7: Construct* $\mathsf{D\text{-}INSDEL}_{\mathcal{LO}}$. The downwards exposed computation of π in a D-block cannot be profitably removable since it is not partially redundant (Section 3.2.2). But it may cause other computations of π to be partially redundant and profitably removable. The downwards exposed computation of π in a D-block n is *isolated* if $\mathsf{D\text{-}ISOLATED}_\Lambda(n) = \mathsf{true}$. i.e. $\mathsf{X\text{-}LIVE}_\Lambda(n) = \mathsf{false}$. 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 Step 1 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: $DB = \{6, 11\}$. We find that $\mathsf{X\text{-}LIVE}_\Lambda(6) = \mathsf{true}$ and $\mathsf{X\text{-}LIVE}_\Lambda(11) = \mathsf{false}$. Thus, this step yields:

$$\mathsf{D\text{-}INSDEL}_{\mathcal{LO}} = \{6\} \tag{22}$$

4.2.4 *Step 8: Construct* $\mathsf{U\text{-}INS}_{\mathcal{LO}}$ *and* $\mathsf{U\text{-}DEL}_{\mathcal{LO}}$. To see if an upwards exposed computation in UB is profitably removable or not, Theorem 4.8 is used.

LEMMA 4.7. *Let $n \in N$. If $\forall\ m \in \mathsf{pred}(G,n) : (m,n) \in \mathcal{C}'_\Lambda$ (i.e., all incoming edges of n in the original CFG are in $\mathcal{C}'_\Lambda$), then top_part$(n) \in T_{mm}$ and $n \in UB$.*

PROOF. If $\forall\ m \in \mathsf{pred}(G,n) : (m,n) \in \mathcal{C}'_\Lambda$, then n must be contained in G_{rd}. By Assumptions 3.1 and 3.2, $\sum_{m \in \mathsf{pred}(G_{st},n)} W(m,n) \geqslant \sum_{m \in \mathsf{succ}(G_{st},n)} W(n,m)$. By Lemma 5.6, *top_part*$(n) \in T_{mm}$ holds. By Lemma 4.6, we have $n \in UB$. □

THEOREM 4.8. *The upwards exposed computation of π in a U-block $n \in UB$ is not profitably removable iff $\forall\ m \in \mathsf{pred}(G,n) : (m,n) \in \mathcal{C}'_\Lambda$.*

PROOF. Lemmas 4.7 and 5.6. □

An upwards exposed computation of π that is not profitably removable may cause other computations of π to be profitably removable. To identify such a computation, the predicate $\mathsf{U\text{-}ISOLATED}_\Lambda$ defined in Step 8(a) is used. Given a U-block $n \in UB$, its upwards exposed computation of π is *isolated* if $\mathsf{U\text{-}ISOLATED}_\Lambda(n) = \mathsf{true}$. An isolated computation cannot cause other computations to be removed profitably. If so, the insertions of the form $h_\pi = \pi$ on the incoming edges of n should be avoided since the saved value h_π on these edges is reused only in block n.

Continuing our running example, where $UB = \{2, 4, 10, 11, 13\}$, we find 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):

$$\begin{array}{rcl} \mathsf{U\text{-}INS}_{\mathcal{LO}} & = & \{(5, 7)\} \\ \mathsf{U\text{-}DEL}_{\mathcal{LO}} & = & \{10, 11, 13\} \end{array} \tag{23}$$

By combining (22) and (23), the lifetime optimal solution $\mathcal{LO}$ in (3) is obtained. The transformed code that we have examined a few times is shown in Figure 1(f).

4.3 Time and Space Complexity

The overall time complexity of MC-PRE 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 with different time complexities [Chekuri et al. 1997]. We have used Goldberg's *push-relabel* HIPR algorithm since it is 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 $O(|N_{st}|)$.

5. THEORETICAL RESULTS

It may sound intuitive that a lifetime optimal solution can be found by finding a special minimum cut and then removing any "unnecessary" code motion introduced by the minimum cut solution. However, the proofs required are quite involved and deserve a formal treatment. We develop our proofs rigorously in two stages. Section 5.1 proves the computational optimality of $\mathcal{CO}$ found by MC-PRE$_{\text{comp}}$. Section 5.2 proves the lifetime optimality of $\mathcal{LO}$ 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

$\mathcal{CO}$ is arbitrarily taken from $\mathcal{T}_{cut}$. So the following theorem implies $\mathcal{T}_{cut} \subseteq \mathcal{CM}_{Cor}$.

THEOREM 5.1. $\mathcal{CO} \in \mathcal{CM}_{Cor}$.

PROOF. In Step 1 of MC-PRE$_{\text{comp}}$, $\mathsf{D\text{-}INSDEL}_{\mathcal{CO}} = DB$. This ensures trivially that every use of h_π in a D-block is always identified with a definition of h_π that is inserted just before the use in that D-block. In Step 2 of MC-PRE$_{\text{comp}}$, $\mathsf{U\text{-}DEL}_{\mathcal{CO}} = UB$. In Step 3 of MC-PRE$_{\text{comp}}$, $\mathsf{U\text{-}INS}_{\mathcal{CO}}$ is found such that $\Gamma(\mathsf{U\text{-}INS}_{\mathcal{CO}})$ is a minimum cut in the EFG G_{st}. Thus, every non-fully redundant computation of π, which must be contained in a U-block and appear in G_{rd} by Lemmas 4.1 and 4.6, is related with a definition of h_π on every control flow path reaching the U-block. This implies immediately that every fully redundant computation of π in a U-block is also related with a definition of h_π on each of its incoming control flow paths. By Definition 3.6, $\mathcal{CO} \in \mathcal{CM}_{Cor}$ holds. □

The following lemma spells out why non-essential edges cannot be insertion edges.

LEMMA 5.2. *Let $T \in \mathcal{CM}_{CompOpt}$. Then there exists a PRE transformation, denoted $\mathcal{C}(T) \in \mathcal{T}_{cut}$ such that $\mathcal{C}(T) \in \mathcal{CM}_{CompOpt}$. In addition, both $\mathsf{U\text{-}INS}_T$ and $\mathsf{U\text{-}INS}_{\mathcal{C}(T)}$ must contain only essential edges such that $\mathsf{U\text{-}INS}_T \subseteq \mathsf{U\text{-}INS}_{\mathcal{C}(T)}$.*

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

$$\begin{array}{rcl} \mathsf{U\text{-}INS}_{\mathcal{C}(T)} & = & \mathsf{U\text{-}INS}_T \cup (\bigcup_{n \in UB - \mathsf{U\text{-}DEL}_T} \{(m,n) \in E \mid m \in \mathsf{pred}(G,n)\}) \\ \mathsf{U\text{-}DEL}_{\mathcal{C}(T)} & = & \mathsf{U\text{-}DEL}_T \cup \{n \mid n \in UB - \mathsf{U\text{-}DEL}_T\} \;=\; UB \\ \mathsf{D\text{-}INSDEL}_{\mathcal{C}(T)} & = & \mathsf{D\text{-}INSDEL}_T \cup \{n \mid n \in DB - \mathsf{D\text{-}INSDEL}_T\} \;=\; DB \end{array} \tag{24}$$

By construction, $\mathsf{U\text{-}DEL}_{\mathcal{C}(T)} = UB$ and $\mathsf{D\text{-}INSDEL}_{\mathcal{C}(T)} = DB$ hold. In addition, $\mathsf{U\text{-}INS}_T \subseteq \mathsf{U\text{-}INS}_{\mathcal{C}(T)}$. Since $T \in \mathcal{CM}_{Cor}$, we must have $\mathcal{C}(T) \in \mathcal{CM}_{Cor}$. By Assumption 3.2, $W(\mathcal{C}(T)) = W(T)$. Hence, $\mathcal{C}(T) \in \mathcal{CM}_{CompOpt}$. By Assumption 3.1, both $\mathsf{U\text{-}INS}_T$ and $\mathsf{U\text{-}INS}_{\mathcal{C}(T)}$ must contain only essential edges in G_{rd}. Otherwise, we would be able to derive $\mathcal{C}(T)' \in \mathcal{CM}_{Cor}$ from $\mathcal{C}(T)$ in such a way that $\mathsf{U\text{-}INS}_{\mathcal{C}(T)'}$ contains all and only the essential edges in $\mathsf{U\text{-}INS}_{\mathcal{C}(T)}$. This implies that $W(\mathcal{C}(T)') < W(\mathcal{C}(T)) = W(T)$, contradicting the facts that $T, \mathcal{C}(T) \in \mathcal{CM}_{CompOpt}$. Hence, $\Gamma(\mathsf{U\text{-}INS}_{\mathcal{C}(T)})$ must be a minimum cut in G_{st} by itself. Otherwise, we would be able to derive $\mathcal{C}(T)'' \in \mathcal{CM}_{Cor}$ from $\mathcal{C}(T)$ in such a way that $\mathsf{U\text{-}INS}_{\mathcal{C}(T)''}$, which is a strict subset of $\mathsf{U\text{-}INS}_{\mathcal{C}(T)}$, must be a minimum cut in G_{st}. This implies that $W(\mathcal{C}(T)'') < W(\mathcal{C}(T))$ under Assumption 3.1, contradicting the fact that $\mathcal{C}(T) \in \mathcal{CM}_{CompOpt}$. By the definition of $\mathcal{T}_{cut}$ given in (19), $\mathcal{C}(T) \in \mathcal{T}_{cut}$ holds. □

Since $\mathcal{CO}$ is not fixed in $\mathcal{T}_{cut}$, the following theorem implies $\mathcal{T}_{cut} \subseteq \mathcal{CM}_{CompOpt}$.

THEOREM 5.3. $\mathcal{CO} \in \mathcal{CM}_{CompOpt}$.

PROOF. By Theorem 5.1, $\mathcal{CO} \in \mathcal{CM}_{Cor}$. By Lemma 5.2, for any $T \in \mathcal{CM}_{CompOpt}$, we can obtain $\mathcal{C}(T) \in \mathcal{T}_{cut}$ such that $\mathcal{C}(T) \in \mathcal{CM}_{CompOpt}$. Thus, $W(T) = W(\mathcal{C}(T))$. $\mathcal{CO} \in \mathcal{T}_{cut}$ is arbitrarily chosen in Step 3.5 of in MC-PRE$_{\text{comp}}$. So $W(\mathcal{C}(T)) = W(\mathcal{CO})$. This implies that $W(T) = W(\mathcal{CO})$. Thus, $\mathcal{CO} \in \mathcal{CM}_{CompOpt}$. □

5.2 Lifetime Optimality

We prove that $\mathcal{LO}$ is the unique lifetime optimal solution. By Theorem 5.4, $\mathcal{LO}$ is computationally optimal, i.e., $\mathcal{LO} \in \mathcal{CM}_{CompOpt}$. Lemma 5.5 recalls a classic result

from [Hu 1970] that exposes the structure of all minimum cuts in a flow network. Based on this result, Lemma 5.6 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 5.7 establishes that $\mathcal{CM}_{LifeOpt} = \{\mathcal{LO}\}$.

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

THEOREM 5.4. $\mathcal{LO} \in \mathcal{CM}_{CompOpt}$.

PROOF. Let us consider the special minimum cut $\mathcal{C}_\Lambda = (\Lambda, \overline{\Lambda})$ found in Step 5 of MC-PRE. By Theorem 5.3, the corresponding transformation $\mathcal{ALO}$ given in (21) 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 $\mathcal{LO} \in \mathcal{CM}_{Cor}$ and *benefit*$(\mathcal{ALO})$ = *benefit*$(\mathcal{LO})$, where *benefit* is defined in (11). 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 5.5. *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 5.6. *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:*

$$\overline{\Lambda} \subseteq \overline{C} \text{ for all } (C, \overline{C}) \in S_{cut} \tag{25}$$

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

PROOF. Under Assumption 3.1, 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 (25) holds with respect to the maximum flow f used. Lemma 5.5 implies that this "smallest minimum cut" is independent of the maximum flow f. Hence, the validity of (25) is established. □

We prove below that $\mathcal{LO}$ is the unique lifetime optimal transformation.

THEOREM 5.7. $\mathcal{CM}_{LifeOpt} = \{\mathcal{LO}\}$.

PROOF. Let $G = (N, E, W)$ be the CFG under consideration. By Theorem 5.4, $\mathcal{LO} \in \mathcal{CM}_{CompOpt}$. We show that $\mathcal{LO}$ is the lifetime best among all transformations in $\mathcal{CM}_{CompOpt}$ based on Properties (1) – (3) stated in Definition 3.8.

$\mathcal{LO}$ is derived from the special minimum cut $\Gamma(\mathcal{C}'_\Lambda) = \mathcal{C}_\Lambda = (\Lambda, \overline{\Lambda})$ found in Step 5 of MC-PRE. Let $T \in \mathcal{CM}_{CompOpt}$ be a computationally optimal transformation. Let $\mathcal{C}(T) \in T_{cut} \subseteq \mathcal{CM}_{CompOpt}$ be as constructed in Lemma 5.2 such that $\Gamma(\mathsf{U\text{-}INS}_{\mathcal{C}(T)})$ is a minimum cut, denoted $(C, \overline{C})$, in G_{st}. By Lemma 5.6, $\overline{\Lambda} \subseteq \overline{C}$.

First, we prove that $\mathcal{LO}$ satisfies Property (1), i.e., $\mathsf{U\text{-}DEL}_{\mathcal{LO}} \subseteq \mathsf{U\text{-}DEL}_T$. Let $n \in UB$ such that $n \in \mathsf{U\text{-}DEL}_{\mathcal{LO}}$. We show that $n \in \mathsf{U\text{-}DEL}_T$. According to Step 8 of MC-PRE, we must have $\mathsf{U\text{-}ISOLATED}_\Lambda(n) = \mathsf{false}$, which happens when either (a) $\exists\ m \in \mathsf{pred}(G, n) : (m, n) \in \mathcal{C}'_\Lambda$ or (b) $\mathsf{KILL}(n) = \mathsf{false} \wedge \mathsf{X\text{-}LIVE}_\Lambda(n) = \mathsf{true}$.

Case (a) $\exists\ m \in \mathsf{pred}(G, n) : (m, n) \in \mathcal{C}'_\Lambda$. Assume to the contrary that $n \notin \mathsf{U\text{-}DEL}_T$. Given the construction of $\mathcal{C}(T)$ from T that is defined in (24), $\mathsf{U\text{-}INS}_{\mathcal{C}(T)}$ includes all incoming edges of n. By Lemma 5.2, all these edges are essential and must thus be contained in G_{rd}. As a result, n, as an essential node, must also be contained in G_{rd}. By Lemma 4.6, $\mathit{top\_part}(n) \in T_{mm}$ since $n \in UB$. This implies that $\overline{\Lambda} \not\subseteq \overline{C}$, which is impossible by Lemma 5.6.

Case (b) $\mathsf{KILL}(n) = \mathsf{false} \wedge \mathsf{X\text{-}LIVE}_\Lambda(n) = \mathsf{true}$. There must exist a path $\langle n_1, \ldots, n_k \rangle$ in G such that (1) $n = n_1$, (2) $n_2, \ldots, n_{k-1}$ are neither U-blocks nor kill blocks (nor D-blocks), (3) $n_k \in UB$, and (4) $\forall\ 1 \leqslant i < k : (n_i, n_{i+1}) \notin \mathcal{C}'_\Lambda$. Due to (4), $\exists\ m \in \mathsf{pred}(G, n_k) : (m, n_k) \in \mathcal{C}'_\Lambda$ holds trivially. Thus, in Step 8 of MC-PRE, we find that $\mathsf{U\text{-}ISOLATED}_\Lambda(n_k) = \mathsf{false}$, and consequently, that $n_k \in \mathsf{U\text{-}DEL}_{\mathcal{LO}}$. Applying the result that we have already proved in Case (a), $n_k \in \mathsf{U\text{-}DEL}_T$ must hold. Let us show that $\forall\ 1 \leqslant i < k : (n_i, n_{i+1}) \notin \mathsf{U\text{-}INS}_T$. Suppose that $(n_i, n_{i+1}) \in \mathsf{U\text{-}INS}_T$ for some $1 \leqslant i < k$. Then (n_i, n_{i+1}) must be essential by Lemma 4.3, implying that $\mathsf{X\text{-}AVAL}(n_i) = \mathsf{false}$. In particular, $\mathsf{X\text{-}AVAL}(n_{k-1}) = \mathsf{false}$. By Lemma 4.1, n_k is contained in G_{rd}. By Lemma 4.6, $\mathit{top\_part}(n_k) \in T_{mm}$ since $n_k \in UB$. This again implies the impossible result that $\overline{\Lambda} \not\subseteq \overline{C}$ by Lemma 5.6. Let us now assume to the contrary that $n \notin \mathsf{U\text{-}DEL}_T$. Since $\forall\ 1 \leqslant i < k : (n_i, n_{i+1}) \notin \mathsf{U\text{-}INS}_T$, the value of $h_\pi = \pi$ must be available on entry of n. Note that $W(n) > 0$ under Assumption 3.1. If $n \notin \mathsf{U\text{-}DEL}_T$, then $T \notin \mathcal{CM}_{CompOpt}$. (Otherwise, including n in $\mathsf{U\text{-}DEL}_T$ would give rise to a computationally better transformation.)

Next, we observe that we can prove Property (2), i.e., $\mathsf{D\text{-}INSDEL}_{\mathcal{LO}} \subseteq \mathsf{D\text{-}INSDEL}_T$ if we proceed similarly as in Case (b) above.

Finally, we prove that $\mathcal{LO}$ satisfies Property (3). This follows immediately from the fact that $\overline{\Lambda} \subseteq \overline{C}$ by Lemma 5.6. The uniqueness of $\mathcal{LO}$ is due to the fact that the equality in $\subseteq$ holds iff $\overline{\Lambda} = \overline{C}$ (also by Lemma 5.6), i.e., iff $T = \mathcal{LO}$. □

6. EXPERIMENTS

We evaluate this work using all the 22 C, C++ and FORTRAN 77 benchmarks from SPECcpu2000 on an Itanium computer system equipped with a 1.0GHz Itanium 2 processor and 4GB of RAM running Redhat Linux 8.0 (2.4.20). We have implemented MC-PRE, and consequently, MC-PRE$_{\text{comp}}$ (our computationally optimal PRE algorithm [Cai and Xue 2003]) in `GCC 3.4.3`. We have also implemented Bodik, Gupta and Soffa's CMP-PRE algorithm [Bodik et al. 1998] in `GCC 3.4.3`. Knoop, Rüthing and Steffen's LCM [Knoop et al. 1994] is the default PRE pass in in `GCC` 3.4.3. All benchmarks are compiled under the optimization level "-O3".

Section 6.1 discusses the implementation details of all four PRE algorithms used in our experiments. Section 6.2 describes the `GCC` framework in which this work is validated. Section 6.3 evaluates MC-PRE against the other three algorithms in terms of eliminated computations, lifetimes of introduced temporaries, execution times, compile times and code sizes for all 22 benchmarks. In particular, we analyse

the performance improvements of MC-PRE over the other three algorithms using `pfmon` and the reasons behind MC-PRE's small compile-time increases over LCM.

6.1 Implementation Details

The LCM pass in `GCC` is a variant of LCM [Knoop et al. 1994] that was described in [Drechsler and Stadel 1993] except that it can reason about edge insertions [Morgan 1998]. Thus, the critical edges in a CFG are not split in `GCC`.

We discussed in Section 3.2.4 that MC-PRE works in the presence of critical edges since edge insertions are used. In our implementations, we do not modify a CFG G to obtain its G_{rd}, G_{mm}, and finally, the EFG G_{st}. The min-cut solver we used is based on Goldberg's *push-relabel* HIPR algorithm [Goldberg 2003]; it is one of the fastest implementations available [Chekuri et al. 1997]. For every G, we generate a graph specification for the EFG G_{st} in terms of the data structure expected by HIPR 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.

MC-PRE$_{\text{comp}}$ is actually the first part of the MC-PRE algorithm. In its Step 3.5, the minimum cut to be found is unspecified since the resulting $\mathcal{CO}$ is always computationally optimal regardless. In order to evaluate the effects of minimizing lifetimes on performance, this step is implemented to return the unique minimum cut $(\Lambda, \overline{\Lambda})$ by applying the (Forward) Labelling Procedure of [Ford and Fulkerson 1962]. As a result, $\overline{\Lambda}$ is the largest possible (Lemma 5.6). The PRE transformation obtained using such a cut will correspond to the Busy Code Motion (BCM) as described in [Knoop et al. 1992; 1994], resulting in the longest lifetimes possible. Note that in all three computationally optimal algorithms for speculative PRE [Bodik 1999; Cai and Xue 2003; Scholz et al. 2004], min-cut cuts are arbitrarily chosen. As a result, the notion of isolated computations does not exist.

As we shall see in Section 6.5, 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 3.2 – 3.4 and 3.5(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 anticipatability 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.

In the actual implementation MC-PRE, we use a standard optimization to replace some edge insertions prescribed by $\mathsf{U\text{-}INS}_{\mathcal{LO}}$ with so-called block insertions as is done for $\mathsf{D\text{-}INSDEL}_{\mathcal{LO}}$. Let $\mathcal{C}'_{\Lambda,\text{copy}} = \{n \in N \mid \forall\, m \in \mathsf{pred}(G,n) : (m,n) \in \mathcal{C}'_\Lambda\}$. By Lemma 4.7, every block n in $\mathcal{C}'_{\Lambda,\text{copy}}$ is a U-block. Instead of making an edge insertion on each of its incoming edges, we will make one single block insertion just before the upwards exposed computation in block n. Hence, $\mathcal{LO}$ becomes:

$$\begin{aligned} \mathsf{INSERT}_{\mathcal{LO}} &= \mathsf{U\text{-}INS}_{\mathcal{LO}} - \{(m,n) \in \mathcal{C}'_\Lambda \mid n \in \mathcal{C}'_{\Lambda,\text{copy}}\} \\ \mathsf{DELETE}_{\mathcal{LO}} &= \mathsf{U\text{-}DEL}_{\mathcal{LO}} - \mathcal{C}'_{\Lambda,\text{copy}} \\ \mathsf{COPY}_{\mathcal{LO}} &= \mathsf{D\text{-}INSDEL}_{\mathcal{LO}} \cup \mathcal{C}'_{\Lambda,\text{copy}} \end{aligned} \tag{26}$$

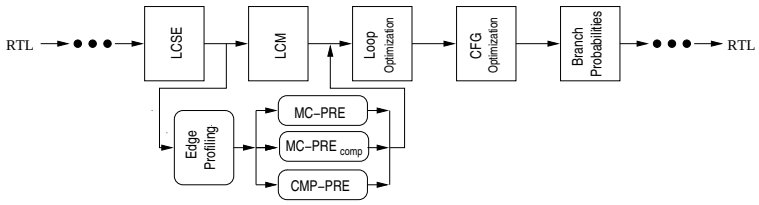


Fig. 6. GCC backend with one of the four PRE algorithms being used as the PRE pass.

where $\mathsf{DELETE}_{\mathcal{LCO}}$ gives all partially redundant computations that can be eliminated profitably, $\mathsf{COPY}_{\mathcal{LCO}}$ gives all computations that are not redundant themselves but cause other computations to be eliminated profitably, and $\mathsf{INSERT}_{\mathcal{LCO}}$ gives all edge insertions required to make all partially redundant computations in $\mathsf{DELETE}_{\mathcal{LCO}}$ to become fully redundant. Such a simplification is not performed for MC-PRE$_{\text{comp}}$ since Lemma 4.7 does not hold for MC-PRE$_{\text{comp}}$.

The CMP-PRE algorithm we have implemented is described in [Bodik et al. 1998]. Given a CFG and a set of PRE candidate expressions in the CFG, CMP-PRE consists of solving three bit-vector data-flow problems simultaneously for all the candidate expressions and performing graph traversals on the CFG separately for these candidate expressions to identify their respective CMP (code-motion-preventing) regions. First, both availability and anticipatability analyses are performed on a CFG over a non-Boolean data-flow lattice with three values, MUST, MAY and NO. Second, the CFG is traversed, once for each PRE candidate expression (i.e., once for each PRE problem), to identify all CMP regions in the CFG. A CMP region for an expression π consists of all connected blocks such that π is both MAY-available and MAY-anticipatable (i.e., strictly partially available and anticipatable) on entry to each of its blocks. As a consequence, some entry edges to a CMP region are MUST-available (NO-available) in the sense that π is fully (not) available on exit from the source blocks of these edges. Similarly, some exit edges from a CMP region are MUST-anticipatable (NO-anticipatable) in the sense that π is fully (not) anticipatable on entry to the target blocks of these edges. Speculative PRE of π for a CMP region is *profitable* if the total execution frequency of MUST-anticipatable exit edges exceeds that of NO-available entry edges. In this case, $h_\pi = \pi$ is inserted on all the NO-available entry edges to the CMP region. Third, availability analysis is re-run on the CFG. Finally, the CFG is transformed based on the information from the original anticipatability analysis and the new availability analysis. No specific algorithm was given in [Bodik et al. 1998] about how to find the CMP regions in a CFG. In our implementation, all CMP regions in a CFG for a PRE problem are found by using graph traversals of the CFG.

6.2 Experimental Setup

Figure 6 depicts the GCC backend in which our algorithm is implemented and evaluated. The backend applies numerous passes to the RTL (Register Transfer Language) representation 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 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`" 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 6 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 phase 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.

Like MC-PRE, MC-PRE$_{\text{comp}}$ and CMP-PRE are also profile-guided. Both are invoked in exactly the same way as MC-PRE as described above.

In our experiments, all the four PRE algorithms use exactly the same set of PRE candidate expressions for a function. These are the expressions identified by `GCC` for its LCM pass. This way, we can have a fair evaluation about the relative strengths and weaknesses of the four PRE algorithms. 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. Therefore, all PRE candidate expressions are exception-free. (In our experiments, signed arithmetic is assumed to wrap around. This has often been the default option for C programs since signed overflow is undefined in C.)

In all the PRE algorithms used, the same optimization 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 three speculative PRE algorithms, MC-PRE, MC-PRE$_{\text{comp}}$ and CMP-PRE. In addition, MC-PRE and MC-PRE$_{\text{comp}}$ use exactly the same bit-vector library used by LCM to perform their required data-flow analysis passes (for all distinct PRE candidate expressions in a CFG in parallel). CMP-PRE performs its data-flow analyses on the modified versions of these bit-vector routines since it works over a non-Boolean lattice. Note that the LCM configuration is the one from `GCC` 3.4.3. This provides an ideal setting for all the four PRE algorithms to compared against each other.

6.3 Redundancy Elimination and Performance Improvements

One important criterion for measuring the effectiveness of a PRE algorithm is to quantify the number of non-fully redundant computations that it removes from a program. Figure 7 evaluates the four PRE algorithms using the SPECcpu2000 benchmarks according to this criterion. MC-PRE and MC-PRE$_{\text{comp}}$ are on par with each other since both are computationally optimal. LCM is the worst performer since it is profile-independent (and thus conservative). CMP-PRE is neither computationally optimal nor lifetime optimal. In principle, CMP-PRE lies between LCM and MC-PRE. MC-PRE eliminates more redundancies than both LCM and CMP-PRE in functions with complex control structures. As a result, the effectiveness of MC-PRE is more pronounced in SPECint2000 than in SPECfp2000. In the case of 12 SPECint2000 benchmarks, MC-PRE eliminates between 31.39% and 147.56% (an average of 90.13%) more non-full redundancies than LCM and between 0.28% and 58.76% (an average of 14.39%) more non-full redundancies than CMP-PRE. This has contributed to the performance improvements of MC-PRE over LCM and CMP-PRE in almost all its 12 benchmarks. In the case of 10 SPECfp2000 benchmarks, LCM has successfully eliminated over 80% of the non-full redundancies in all the benchmarks except `ammp`. CMP-PRE has removed most of the remaining non-full redundancies from most of these benchmarks. However, the non-full redundancies that LCM and CMP-PRE fail to eliminate in a program can be from some of its hottest functions, and furthermore, these redundant computations can be occurrences of expensive PRE candidate expressions (e.g., memory operations) in these hottest functions. In these situations (as is the case for `mesa` and `sixtrack`), MC-PRE, which can eliminate all partial redundancies removable by using code motion and control speculation, can still achieve performance improvements in such programs over LCM and CMP-PRE.

Figure 8 depicts the performance improvements (or degradations) of MC-PRE over MC-PRE$_{\text{comp}}$, LCM and CMP-PRE. The execution time of a benchmark is taken as the arithmetic average of five runs and validated by the CPU cycles obtained using the `pfmon` tool available for the Itanium architecture. By removing more redundant computations and keeping the lifetimes of introduced temporaries to a minimum, MC-PRE can achieve nearly the same performance results as or better performance improvements than the other three algorithms in all 22 benchmarks. The performance degradations observed in some benchmarks are small. In general, MC-PRE improves the performance of a SPECint2000 benchmark over LCM and CMP-PRE due to the combined effects of optimizing a number of its hot functions. In the case of a SPECfp2000 benchmark, however, the performance improvement tends to come from optimizing fewer hot functions (typically one or two) in the benchmark. These results are analyzed in three separate sections below.

6.3.1 *MC-PRE vs. MC-PRE$_{\text{comp}}$.* Figure 9 shows that MC-PRE$_{\text{comp}}$ introduces temporaries with longer lifetimes than MC-PRE. Recall that every computation of π removed by MC-PRE must also be removed by MC-PRE$_{\text{comp}}$. The lifetime of h_π for a replacement computation (i.e., a use) of h_π in a CFG is measured to be the percentage of the basic blocks in which h_π is live in the CFG. Figure 5(a) compares MC-PRE and MC-PRE$_{\text{comp}}$ in terms of the average lifetimes of their introduced temporaries for non-isolated replacement computations (i.e., deletions) in

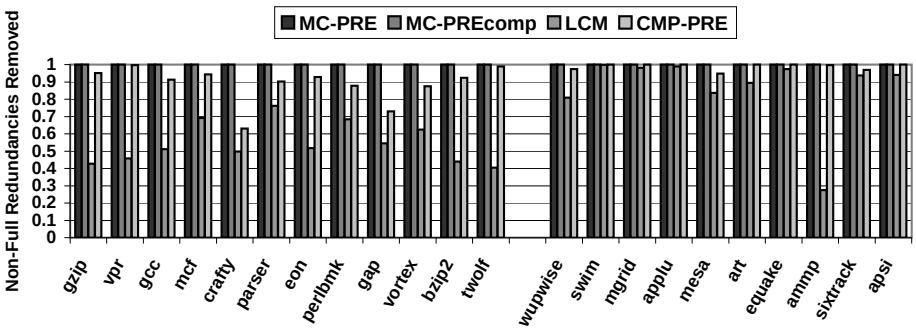


Fig. 7. Non-fully redundant computations eliminated (in dynamic terms).

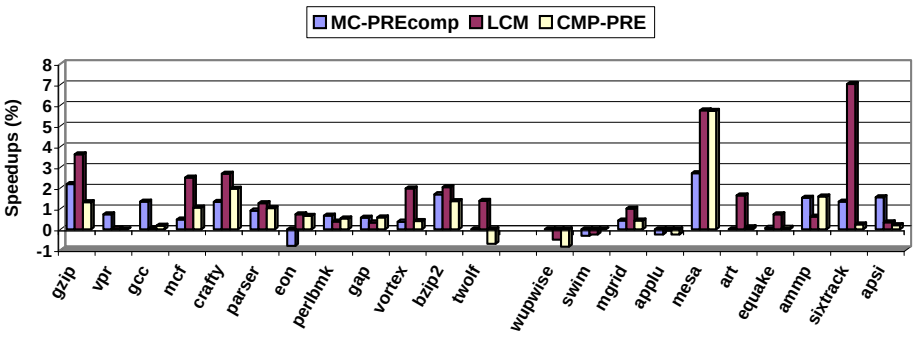


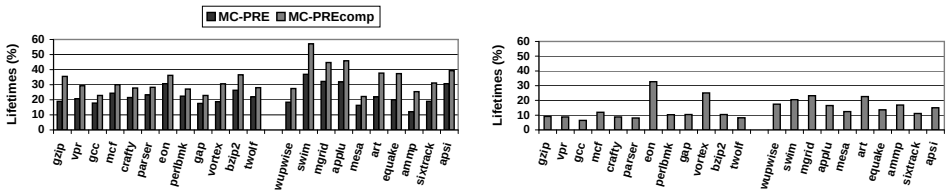
Fig. 8. Percentage speedups of MC-PRE over MC-PRE$_{\text{comp}}$, LCM and CMP-PRE.

a benchmark. The lifetimes for all benchmarks are longer under MC-PRE$_{\text{comp}}$ than MC-PRE. The percentage increases range between 17.97% for `eon` and 110.27% for `ammp`. Figure 5(b) depicts the lifetimes of temporaries for isolated computations that are introduced by MC-PRE$_{\text{comp}}$ but eliminated by MC-PRE.

Comparing MC-PRE and MC-PRE$_{\text{comp}}$ in Figure 8, we see that minimizing the lifetimes of introduced temporaries certainly has an overall positive effect on performance. MC-PRE achieves the same or better performance results than MC-PRE$_{\text{comp}}$ in 19 out of the 22 benchmarks used.

MC-PRE eliminates more redundancies than LCM and CMP-PRE and uses insertions with shorter lifetimes than MC-PRE$_{\text{comp}}$. The combined effects of MC-PRE's achieving both optimal results on performance will be examined below.

6.3.2 *MC-PRE vs. LCM.* As shown in Figure 8, MC-PRE achieves the same or better performance results than LCM in 19 out of 22 benchmarks. Some small performance losses are observed in `wupwise`, `swim` and `applu`. The benchmarks with better speedups are typically those in which MC-PRE has succeeded in eliminating more redundancies than LCM. These include `gzip` (3.63%), `mcf` (2.50%), `crafty` (2.69%), vortex (1.97%), `bzip` (2.04%) and `twolf` (1.37%) in the SPECint2000 benchmark suite and `mesa` (5.75%), `art` (1.64%) and `sixtrack` (7.03%) in the SPECfp2000 benchmark suite. By comparing MC-PRE and MC-PRE$_{\text{comp}}$, we fur-



(a) Non-isolated deletions (b) Isolated deletions by MC-PRE$_{\text{comp}}$

Fig. 9. Lifetimes of introduced temporaries by MC-PRE and MC-PRE$_{\text{comp}}$.

ther observe that minimizing the lifetimes of introduced temporaries has generally contributed to the performance improvements in these benchmarks.

MC-PRE tends to achieve the performance improvement in a SPECInt2000 benchmark by optimizing a number of its hot functions. For example, MC-PRE eliminates 100.84% more partial redundancies than LCM in `crafty` with 97.12% of these redundant computations coming from the following hottest functions in the benchmark: `Evaluate`, `MakeMove`, `Search`, `UnMakeMove`, `GenerateCaptures`, `EvaluatePawns` and `NextMove`. The performance improvements in these individual functions have contributed to the overall speedup of this benchmark.

The two largest speedups, 5.75% and 7.03%, are attained in the two SPECfp2000 benchmarks, `mesa` and `sixtrack`, in which LCM has already removed the majority of their redundant computations. Both benchmarks serve as good examples to demonstrate the benefits for a PRE algorithm to achieve both computationally and lifetime optimal results in some programs. In both cases, the performance improvements (over LCM and CMP-PRE) are obtained because MC-PRE can achieve the two optimal results in one or two of their hot functions with complex control flow.

In `mesa`, about 55.17% of the execution time from the compiled binary under LCM is spent on the three hottest functions, `general_textured_triangle` (21.50%), `gl_texture_pixels` (11.44%) and `sample_1d_linear` (22.23%). MC-PRE removes only 19.55% more non-fully redundant computations than LCM (in dynamic terms). However, 94.7% of these are eliminated from `sample_1d_linear` (whose CFG consists of 83 blocks and 119 edges). This represents 1.77% of the dynamic number of RTL instructions executed in the function. So the performance gain in `mesa` mostly comes from optimizing this function. Note that the minimization of lifetimes of introduced temporaries has also been beneficial (Figure 8).

In `sixtrack`, the execution time from the compiled binary under LCM is mostly spent in its two hottest functions, `thin6d` (98.49%) and `umlauf` (0.93%). MC-PRE removes only 7.75% more non-fully redundant computations than LCM. However, all these redundant computations are removed from the two hottest functions: `thin6d` (52.01%) and `umlauf` (41.77%). Most of the performance gain comes from optimizing `thin6d` (with 196 blocks and 347 edges). Again Figure 8 shows clearly that the minimization of lifetimes of introduced temporaries has also contributed to the performance improvement in this benchmark.

MC-PRE removes more redundancies than LCM. Despite this, the average lifetimes of introduced temporaries in both cases, as shown in Figure 10, are close. It is important to point out that LCM, which is profile-independent, is not computationally optimal for solving the speculative PRE problem. Therefore, LCM is not

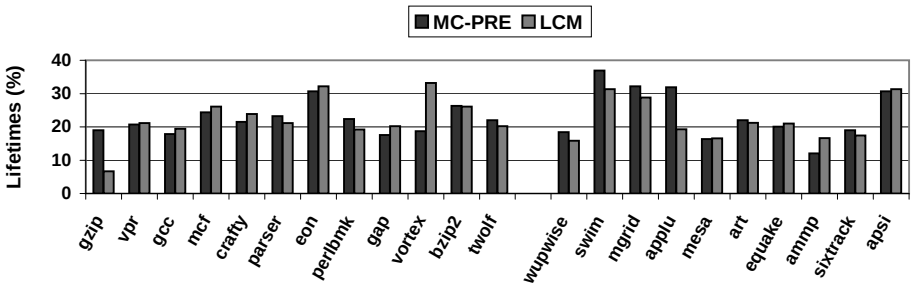


Fig. 10. Lifetimes of introduced temporaries by MC-PRE and LCM.

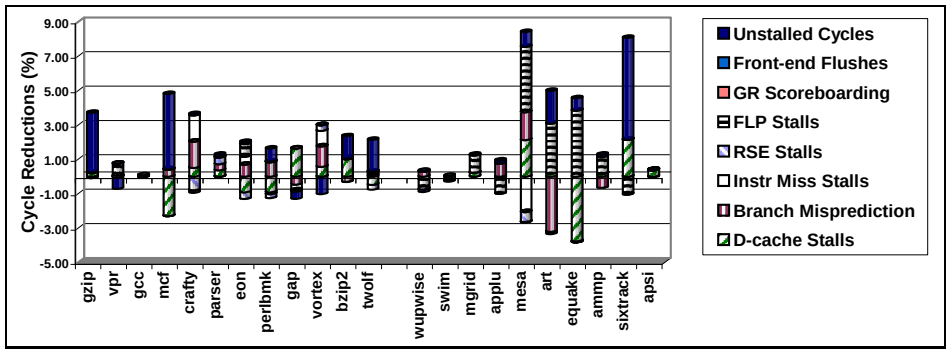


Fig. 11. **Performance improvements of MC-PRE over LCM split into those contributed by the individual reductions in the eight event categories. An event bar for a benchmark represents the performance speedup achieved by MC-PRE over LCM as a result of reducing the cycles in that event category. All the eight event bars for a benchmark add up to the performance speedup given in Figure 8 for that benchmark.**

lifetime better than MC-PRE even though it achieves shorter lifetimes in a program than MC-PRE (Definition 3.8). With this caveat in mind, we see from Figure 10 that MC-PRE introduces temporaries with comparable lifetimes as LCM.

Below we analyze the performance improvements of MC-PRE over LCM using `pfmon`. Figure 11 breaks down the performance speedup (or degradation) of a benchmark achieved by MC-PRE over LCM, which can be found in Figure 8, into eight components resulting from the corresponding cycle reductions (or increases) in the eight event categories. According to [Sverre Jarp 2002], the execution cycles of a program can be broken into stalls and unstalled cycles. The stalls can be further subdivided into seven categories: data cache (D-cache) stalls, branch misprediction, instruction miss stalls, register stack engine (RSE) stalls, floating-point unit (FLP) stalls, general register (GR) scoreboarding stalls and front-end flushes.

As shown in Figure 11, MC-PRE has reduced the unstalled cycles in 16 out of the 22 benchmarks. The percentage reductions are particularly significant for `gzip`, `mcf`, `bzip2`, `twolf` and `sixtrack`, which are benchmarks with relatively large speedups. In the case of `sixtrack`, the one with the largest speedup, the decrease in its unstalled cycles has contributed to 84.34% of its performance improvement.

D-cache stalls and FLP unit stalls are known to be the main contributors to the pipeline stalls for integer and floating-point benchmarks in the SPECcpu2000 benchmark suite, respectively. D-cache stalls are mainly the bubbles caused by the latencies of integer load instructions while FLP unit stalls are the bubbles caused by the latencies of floating-point load instructions and inter-register dependencies.

MC-PRE is applied only to the PRE candidate expressions that do not cause runtime exceptions. These include all RTL-level exception-free memory operations that may be eventually translated into loads. For example, `(mem:m addr)` represents one such a memory operation (i.e., access) at the address `addr`, where `m` specifies the smallest addressable unit for this memory operation. By eliminating more redundant computations – some of which are loads – than LCM, and by minimizing the lifetimes of introduced temporaries, MC-PRE has succeeded in reducing the D-cache stalls and FLP unit stalls in some benchmarks. However, since some of these redundant computations are eliminated speculatively, compensating insertions are made on the paths where those eliminated computations did not exist before. The D-cache and FLP unit stalls in some benchmarks are increased. Looking at all the benchmarks in Figure 11, we observe some relatively large reductions in the D-cache stalls in `gap`, `mesa` and `sixtrack`. On the other hand, `mcf` and `equake` suffer some visible increases. In the case of the SPECfp2000 benchmarks, we observe 3.81%, 2.91% and 3.84% reductions in terms of the FLP unit stalls for `mesa`, `art` and `equake`, respectively. In particular, `mesa` is the benchmark with the second largest speedup (5.75%). Some small increases in the FLP unit stalls are observed in `wupwise`, `applu` and `sixtrack`. If both D-cache and FLP unit stalls are combined, MC-PRE will be more effective than LCM in reducing the stall cycles in this combined category in 15 out of the 22 benchmarks.

Figure 8 shows that MC-PRE results in slightly slower binaries than LCM in `wupwise`, `swim` and `applu`. In `swim` and `applu`, LCM has succeeded in removing almost all redundant computations. In `wupwise`, MC-PRE eliminates 16.62% more redundant computations than LCM with 99.04% of these redundancies being removed from `zgemm`, the hottest function in the benchmark. However, looking at Figure 11, the performance slowdowns in all three benchmarks are mainly caused by some slight increases in their FLP unit stall cycles. One challenging future work is to use a cost model that can also include a quantitative estimate of the impact of such an entity on performance to drive the PRE optimization.

In summary, MC-PRE has reduced more unstalled cycles than LCM across most of the benchmarks used. In addition, the speculative elimination of safe memory operations (using temporaries with the shortest lifetimes as is possible) has an overall positive impact on reducing D-cache and FLP unit stalls. As shown in Figure 11, the decreases in these stall categories are responsible for the performance improvements achieved in the SPECcpu2000 benchmarks.

6.3.3 *MC-PRE vs. CMP-PRE.* All the four PRE algorithms can eliminate all fully redundant computations. MC-PRE removes optimally all non-fully redundant computations that can be removed by using speculative code motion. LCM can only remove some of these redundant computations non-speculatively. We measure the effectiveness of a speculative PRE algorithm by computing how much (in percentage) it can eliminate speculatively from all non-full redundancies that LCM fails to eliminate (non-speculatively). Figure 12 compares MC-PRE and CMP-PRE based

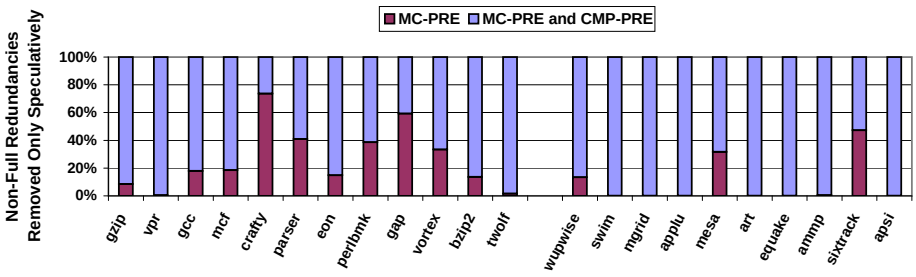


Fig. 12. Non-fully redundant computations eliminated only speculatively by MC-PRE and CMP-PRE(i.e., these redundant computations cannot be eliminated by LCM).

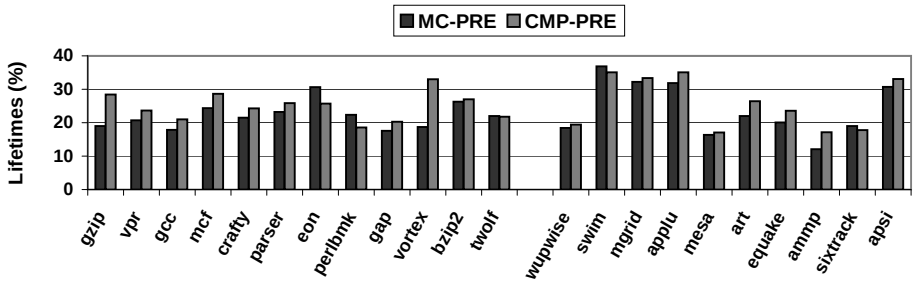


Fig. 13. Lifetimes of introduced temporaries by MC-PRE and CMP-PRE.

on this criterion. The problem of finding computationally optimal transformations is a min-cut problem. MC-PRE finds required insertion edges optimally while CMP-PRE restricts its search to the NO-available entry edges to CMP regions. In general, MC-PRE eliminates more redundancies than CMP-PRE for CFGs with complex control structures. In the case of SPECint2000, MC-PRE eliminates between 0.52% and 280.47% (an average of 58.15%) more partial redundancies that are only speculatively removable than CMP-PRE. In the case of SPECfp2000, the percentage increases range from 0.00% to 89.98% with an average of 15.23%. CMP-PRE is more effective in SPECfp2000 than in SPECint2000. This is because, as shown in Figure 7, LCM itself (with no speculation) has successfully eliminated over 80% of all partial redundancies in nine out of its 10 benchmarks.

Figure 13 shows that CMP-PRE uses temporaries with slightly longer lifetimes in 17 out of the 22 benchmarks (even though it is not computationally optimal).

By eliminating more redundant computations and using temporaries with shorter lifetimes, MC-PRE achieves the same or better performance results than CMP-PRE in 19 out of the 22 benchmarks as shown in Figure 8. Let us look at the three benchmarks, `crafty`, `mesa` and `sixtrack`, that we examined previously in Section 6.3.2. For `crafty`, we mentioned earlier that MC-PRE eliminates more partially redundant computations than LCM from its hottest functions: `Evaluate`, `MakeMove`, `Search`, `UnMakeMove`, `GenerateCaptures`, `EvaluatePawns` and `NextMove`. As shown in Figures 7 and 12, of all non-fully redundant computations that cannot be removed by LCM but are completely removed by MC-PRE, CMP-PRE has eliminated only

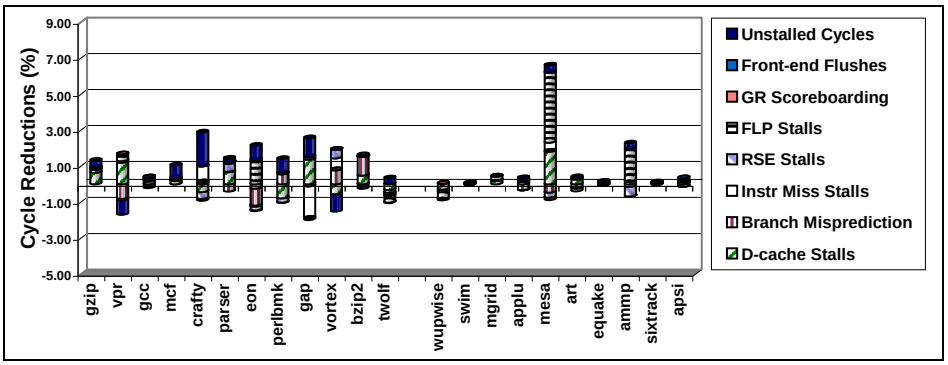


Fig. 14. Performance improvements of MC-PRE over CMP-PRE.

26.28% of these redundant computations. In particular, `search` is the only function for which CMP-PRE has successfully removed all its partially redundant computations. So the performance improvements from optimizing these individual functions have again resulted in the overall speedup of this benchmark. Let us now take a look at `mesa`, the benchmark for which MC-PRE achieves the largest speedup over CMP-PRE. As we discussed in Section 6.3.2, MC-PRE eliminates almost all non-full redundancies that LCM fails to eliminate in its hot function `sample_1d_linear`. Figure 12 shows that only 66.67% of these redundant computations are removed by CMP-PRE. Furthermore, MC-PRE has eliminated more redundant computations statically than CMP-PRE and LCM. LCM removes eight partially (fully or non-fully) redundant computations from eight distinct expressions. CMP-PRE eliminates seven more partially redundant computations all from two new expressions. However, all these operations removed by LCM and CMP-PRE are arithmetic. MC-PRE has removed 41 more non-fully redundant computations from these existing expressions and 12 new ones on memory access, sign-extension, shift and compare. This has resulted in the actual performance improvements of MC-PRE over LCM and CMP-PRE. In `sixtrack`, we mentioned in Section 6.3.2 that among all speculatively removable redundancies that are removed by MC-PRE, 93% are from its two hottest functions: `thin6d` (52.01%) and `umlauf` (41.77%). CMP-PRE has eliminated all these speculatively removable redundancies from `thin6d` but nothing at all from `umlauf`. Since `thin6d` consumes the most of the execution time in this benchmark, MC-PRE achieves only some slight performance improvement over CMP-PRE.

Figure 14 gives an analogue of Figure 11 for CMP-PRE. As in the case of LCM, the performance improvements of MC-PRE over CMP-PRE are generally obtained from the cycles reductions in the same three stall categories, namely, unstalled cycles, D-cache stalls and FLP unit stalls. Some small performance losses are observed in `twolf` and `wupwise` due to slightly increases in D-cache stalls and FLP unit stall and `applu` due to more branch mispredictions introduced.

In summary, MC-PRE can optimally exploit more optimization opportunities in programs than a non-optimal algorithm like CMP-PRE. In the case when MC-PRE is more effective than CMP-PRE in hot functions, which may have complex control flow, in a program, some performance improvement is often expected.

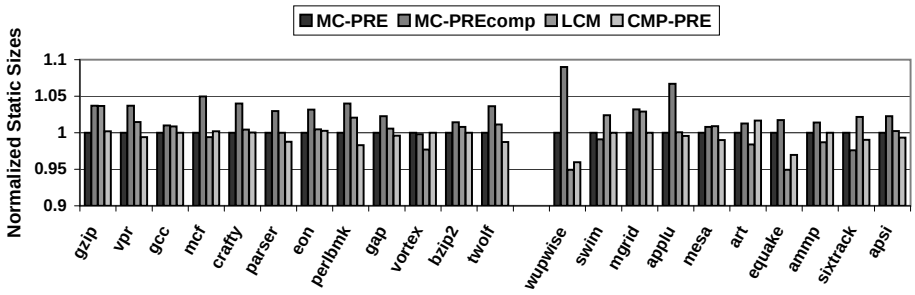


Fig. 15. Static code sizes.

6.4 Static Code Sizes

Figure 15 compares the static code sizes of all benchmarks (normalized to MC-PRE) compiled under the four PRE algorithms. The size of a compiled benchmark is taken as the size of the text section of its binary obtained using the UNIX command `size` as suggested in [Beszedes 2003]. A PRE algorithm both inserts and deletes computations. So it may cause some benchmarks to expand and others to shrink in their code sizes. All four PRE algorithms are comparable in terms of this size criterion except that MC-PRE$_{\text{comp}}$ results in slightly larger binaries (particularly in the case of `wupwise` and `applu`). MC-PRE$_{\text{comp}}$ makes insertions and deletions for isolated computations. As a result, the increases in code size are observed in 19 out of the 22 benchmarks. The overall increase of MC-PRE$_{\text{comp}}$ over MC-PRE for all benchmarks is 1.43%. By comparing LCM and MC-PRE, we find that MC-PRE has caused small code size expansions in only six out of the 22 benchmarks with an overall size increase of 0.78%. Finally, CMP-PRE has resulted in slightly smaller binaries than MC-PRE in 12 out of the 22 benchmarks. However, the overall increase of MC-PRE over CMP-PRE for all 22 benchmarks is only 0.50%.

6.5 Compile Times

Figure 16 gives the compile times of `GCC` in compiling all benchmarks under the four PRE algorithms. All compile times are normalized with respect to MC-PRE. MC-PRE$_{\text{comp}}$ is generally more expensive than MC-PRE due to some unnecessary insertions and deletions MC-PRE$_{\text{comp}}$ introduces for isolated computations. CMP-PRE is generally more expensive than LCM. This is partly because CMP-PRE performs three bit-vector data-flow analyses over a non-Boolean lattice and partly because CMP-PRE also needs to traverse a CFG separately for each distinct PRE problem in order to identify its CMP regions. The compile times of MC-PRE are slightly larger than CMP-PRE in some benchmarks but smaller in others. In comparison with LCM, the extra compilation overheads MC-PRE pays for achieving the performance improvements in all the benchmarks are relatively small. In the case of SPECint2000 benchmarks, the compile-time increases for `gcc`, `crafty` and `perlbmk` are 4.50%, 8.72% and 3.63%, respectively. The increases of less than 1.86% are observed in the remaining nine benchmarks. In the case of SPECfp2000, the worst three benchmarks are `wupwise`, `equake` and `sixtrack`. Their compile-time increases are 4.37%, 5.05% and 9.22%, respectively. Each of the remaining seven

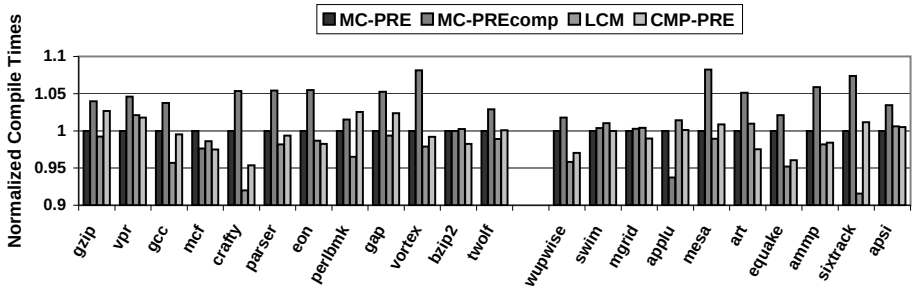


Fig. 16. Compile times.

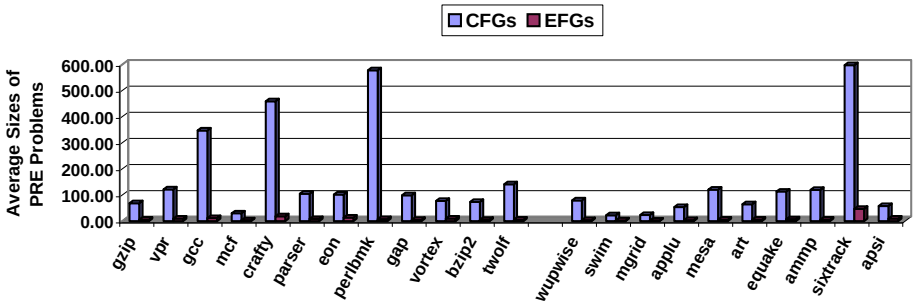


Fig. 17. Average sizes of PRE problems on CFGs and EFGs. Given a CFG, distinct PRE candidate expressions in the CFG result in distinct PRE problems. The empty EFGs are excluded in calculating the average number of blocks per EFG (which would be much smaller otherwise).

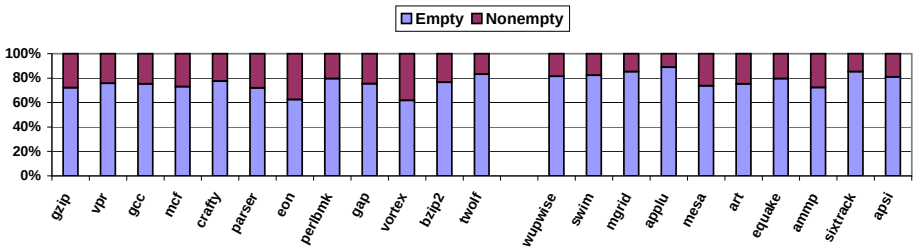


Fig. 18. Empty v.s. nonempty reduced graphs (or EFGs).

benchmarks has a compile-time increase of less than 1.85%. Finally, of all the 22 benchmarks, seven compile even slightly faster under MC-PRE than LCM.

There are two reasons why MC-PRE is only slightly more expensive than LCM. First, Figure 17 shows that the EFGs are significantly smaller than the original CFGs across all the benchmarks. Given an EFG $G_{st} = (N_{st}, E_{st}, W_{st})$, finding a minimum cut for it 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. Furthermore, as shown in Figure 18, the reduced graphs for the majority of the PRE problems in nearly all benchmarks are empty. This

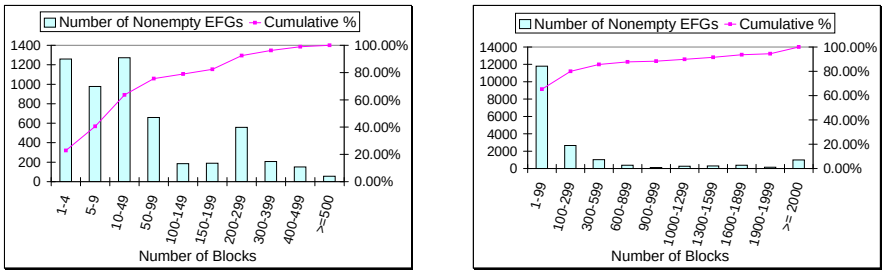
means that the corresponding EFGs are also empty. The minimum cut for an empty EFG is empty, requiring no invocation to a min-cut algorithm in our implementation (as discussed in Section 6.1). (By Lemma 4.1, if the EFG for a PRE problem is empty, then all redundancies (if any) must be full redundancies and are all removed by MC-PRE.) Second, nonempty EFGs are constructed efficiently. We have modified `GCC`'s `edge_def` struct so that a single traversal of a CFG in Step 3.4 of MC-PRE$_{\text{comp}}$ is sufficient to produce the corresponding nonempty EFG.

Le us investigate why the seven benchmarks, `vpr`, `bzip2`, `swim`, `mgrid`, `applu`, `art` and `apsi`, as shown in Figure 8, compile slightly faster under MC-PRE than LCM. These seven benchmarks contain in that order 17828, 5191, 1067, 1600, 7603, 2147 and 15490 PRE problems to be solved, respectively. LCM performs PRE by conducting each of the four data-flow analysis passes in parallel on these PRE problems. On the other hand, MC-PRE first applies two data-flow analysis passes on these problems in parallel to reduce them into smaller PRE problems on EFGs. As a result, these seven benchmarks have in that order only 4327, 1213, 188, 234, 845, 532 and 2947 nonempty EFGs, which are the PRE problems that require the min-cut step of MC-PRE to be invoked. These nonempty EFGs are rather small (with 9.77 blocks for `vpr`, 5.19 blocks for `bzip2`, 2.75 blocks for `swim`, 2.21 blocks for `mgrid`, 3.50 blocks for `applu`, 5.42 blocks for `art` and 9.14 for `apsi` on the average). So the min-cut step completes very quickly. Then 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 if the nonempty EFGs are small (so that the min-cut step completes quickly).

Finally, we examine why MC-PRE is not so much more costly than LCM in the worst case. As shown in Figure 8, `crafty` and `sixtrack` are the two most expensive benchmarks to compile. Figure 19 plots the histograms of all the nonempty EFGs for the two benchmarks according to their sizes. Although `crafty` has some relatively large EFGs, 92.46% of them have fewer than 300 blocks. On the other hand, `sixtrack` has more large EFGs. There are 80.02% EFGs with fewer than 300 blocks. In the remaining EFGs, there are 1510 with 300 – 999 blocks, 1098 with 1000 – 1999 blocks and 996 with more than 1999 blocks. However, Figure 18 shows that in `sixtrack`, 85.37% of EFGs are empty. Due to the first two reasons given above and the fact that the efficiency of the min-cut algorithm used, the compile time increase of MC-PRE over LCM in this worst-case benchmark is only 9.22%.

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 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 three standard bit-vector data-flow analyses and a standard min-cut algorithm. We have implemented MC-PRE, and consequently, MC-PRE$_{\text{comp}}$ in `GCC` 3.4.3. For comparison purposes, we have also implemented CMP-PRE, a previously reported non-optimal speculative PRE algorithm [Bodik et al. 1998], in `GCC`



(a) `crafty` (b) `sixtrack`

Fig. 19. Nonempty EFGs of `crafty` and `sixtrack`.

3.4.3. We have evaluated MC-PRE against MC-PRE$_{\text{comp}}$, LCM and CMP-PRE using the 22 C, C++ and FORTRAN 77 benchmarks in SPECcpu2000 on Itanium 2. Our experimental results show that MC-PRE is capable of eliminating more partial redundancies than both LCM and CMP-PRE and always uses temporaries with shorter lifetimes than MC-PRE$_{\text{comp}}$. Achieving both computationally and lifetime optimal results in a program allows MC-PRE to explore more optimization opportunities, particularly in its functions with complex control structures. This has led to the performance improvements in almost all of these benchmarks over the other three algorithms at the costs of small compile-time and code-size increases in some benchmark programs.

As profile-guided compiler optimizations are becoming increasingly more aggressive, more sophisticated algorithms will likely be employed in future compile systems. In addition to the PRE optimization as discussed in this paper, the min-cut algorithm has also been recently used in decomposing a sequential program into speculatively parallel threads [Johnson et al. 2004].

Finally, we want to stress that PRE techniques are applicable to other areas of optimisation such as load/store elimination by combining both control and data speculation [Lin et al. 2003], load/store elimination for binaries [Fernández and Espasa 2004], communication optimisation [Knoop and Mehofer 2002], and thread-level synchronisation cost elimination [Zhai et al. 2002]. Significant performance gains can be expected from these optimisations, leveraging the benefits of a more application-specific PRE technique that may be derived from our algorithm. Furthermore, our algorithm may find applications in some embedded applications [Scholz et al. 2004] where performance improvements can be beneficial. One future work is to evolve MC-PRE into a more application-specific PRE technique in these research areas so that more significant performance gains can be obtained.

8. ACKNOWLEDGEMENTS

We wish to thank the reviewers and editors for their helpful comments and suggestions. This work is partially supported by an ARC grant DP0452623.

REFERENCES

Alpern, B., Wegman, M. N., and Zadeck, F. K. 1988. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages. 1–11.

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 *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*. 1–14.

Briggs, P. and Cooper, K. D. 1994. Effective partial redundancy elimination. In *Proceedings of the 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*. 91–104.

Chang, P. P., Mahlke, S. A., Chen, W. Y., Warter, N. J., and mei W. Hwu, W. 1991. Impact: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*. 266–275.

Chekuri, C., Goldberg, A. V., Karger, D. R., Levine, M. S., and Stein, C. 1997. Experimental study of minimum cut algorithms. In *Proceedings of the Eighth Annual 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.

Click, C. 1995. Global code motion / global value numbering. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*. 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 Trans. Program. Lang. Syst. 13,* 4, 451–490.

Dhamdhere, D. M. 1991. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Trans. Program. Lang. Syst. 13,* 2, 291–294.

Dhamdhere, D. M., Rosen, B. K., and Zadeck, F. K. 1992. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*. 212–223.

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

Fernández, M. and Espasa, R. 2004. Link-time path-sensitive memory redundancy elimination. In *International Conference on High-Performance Computer Architecture*. 300–310.

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

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

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

Hailperin, M. 1998. Cost-optimal code motion. *ACM Transactions on Programming Languages and Systems 20,* 6, 1297 – 1322.

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. I., and Diwan, A. 2001. Partial redundancy elimination for access path expressions. *Softw., Pract. Exper. 31,* 6, 577–600.

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

Johnson, T. A., Eigenmann, R., and Vijaykumar, T. N. 2004. Min-cut program decomposition for thread-level speculation. *SIGPLAN Not. 39,* 6, 59–70.

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 *Proceedings of the 7th International Conference on Compiler Construction*. Springer-Verlag, London, UK, 144–158.

KNOOP, J. 1998. Optimal interprocedural program optimization: A new framework and its application. Number 1428 in LNCS.

KNOOP, J., COLLARD, J.-F., AND JU, R. D.-C. 2000. Partial redundancy elimination on predicated code. In *Proceedings of the 7th Static Analysis Symposium (SAS'00)*. 260–279.

KNOOP, J. AND MEHOFER, E. 2002. Distribution assignment placement: Effective optimization of redistribution costs. *IEEE Trans. Parallel Distrib. Syst. 13,* 6, 628–647.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1992. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*. 224–234.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1994. Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst. 16,* 4, 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 *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*. 289–299.

LO, R., CHOW, F., KENNEDY, R., LIU, S.-M., AND TU, P. 1998. Register promotion by sparse partial redundancy elimination of loads and stores. *SIGPLAN Not. 33,* 5, 26–37.

MOREL, E. AND RENVOISE, C. 1979. Global optimization by suppression of partial redundancies. *Commun. ACM 22,* 2, 96–103.

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

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

ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 12–27.

RÜTHING, O., KNOOP, J., AND STEFFEN, B. 2000. Sparse code motion. In *Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*Boston, Massachusetts*). ACM, New York, 170 – 183.

SCHOLZ, B., HORSPOOL, R. N., AND KNOOP, J. 2004. Optimizing for space and time usage with speculative partial redundancy elimination. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 221–230.

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

STEFFEN, B. 1996. Property-oriented expansion. In *Proceedings of the 3rd Static Analysis Symposium (SAS'96)*. Lecture Notes in Computer Science, vol. 1145. Springer-Verlag, Heidelberg, Germany, 22 – 41.

STEFFEN, B., KNOOP, J., AND RÜTHING, O. 1990. The value flow graph: a program representation for optimal program transformations. In *Proceedings of the third European symposium on programming on ESOP '90*. Springer-Verlag New York, Inc., New York, NY, USA, 389–405.

STEFFEN, B., KNOOP, J., AND RÜTHING, O. 1991. Efficient code motion and an adaption to strength reduction. In *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Advances in distributed computing (ADC) and colloquium on combining paradigms for software development (CCPSD): Vol. 2*. Springer-Verlag New York, Inc., New York, NY, USA, 394–415.

SVERRE JARP. 2002. A methodology for using the Itanium 2 performance counters for bottleneck analysis. http://www.gelato.org/pdf/Performance_counters_final.pdf.

ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. 2002. Compiler optimization of scalar value communication between speculative threads. In *ASPLOS-X: Proceedings of*

the 10th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM Press, 171–183.