Optimal and Efficient Speculation-Based Partial Redundancy Elimination

Qiong Cai and Jingling Xue
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
{qiongc, jxue}@cse.unsw.edu.au

Abstract

Existing profile-guided partial redundancy elimination (PRE) methods use speculation to enable the removal of partial redundancies along more frequently executed paths at the expense of introducing additional expression evaluations along less frequently executed paths. While being capable of minimizing the number of expression evaluations in some cases, they are, in general, not computationally optimal in achieving this objective. In addition, the experimental results for their effectiveness are mostly missing.

This work addresses the following three problems: (1) Is the computational optimality of speculative PRE solvable in polynomial time? (2) Is edge profiling — less costly than path profiling — sufficient to guarantee the computational optimality? (3) Is the optimal algorithm (if one exists) lightweight enough to be used efficiently in a dynamic compiler? In this paper, we provide positive answers to the first two problems and promising results to the third.

We present an algorithm that analyzes edge insertion points based on an edge profile. Our algorithm guarantees optimally that the total number of computations for an expression in the transformed code is always minimized with respect to the edge profile given. This implies that edge profiling, which is less costly than path profiling, is sufficient to guarantee this optimality. The key in the development of our algorithm lies in the removal of some non-essential edges (and consequently, all resulting non-essential nodes) from a flow graph so that the problem of finding an optimal code motion is reduced to one of finding a minimal cut in the reduced (flow) graph thus obtained. We have implemented our algorithm in Intel's Open Runtime Platform (ORP). Our preliminary results over a number of Java benchmarks show that our algorithm is lightweight and can be potentially a practical component in a dynamic compiler. As a result, our algorithm can also be profitably employed in a profile-guided static compiler, in which compilation cost can often be sacrificed for code efficiency.

1. Introduction

Partial redundancy elimination (PRE) is a powerful optimization technique for removing partial redundancies along some paths through a flow graph. The technique inserts and deletes computations in a given flow graph in order to reduce the total number of remaining such computations in the transformed code. Global common subexpressions and loop-invariant computations are special cases of partial redundancies. As a result, PRE has become an important component in global optimizers [10, 12, 13].

Classic PRE methods [21, 22] guarantee computationally optimal results, i.e., results where the number of computations cannot be reduced any further by safe code motion [20]. Under such a safety constraint, they insert an expression π at a point p in a flow graph only if all paths emanating from p must evaluate π before any operands of π are redefined. The expression π is known as *fully anticipatable* at p [23]. In other words, they remove partial redundancies along some paths but never introduce additional (new) computations along any path. These safety-based formulations have two consequences. First, the transformed code cannot cause exceptions that do not exist in the original code. If evaluating π can throw an exception, the exception — which is inevitable — would have occurred a bit earlier in the transformed code than in the original code. Second, due to the absence of profiling information, they guarantee (albeit conservatively) that the transformed code cannot evaluate π more times than before in any program execution.

In practical programs, some points (nodes or edges) in a flow graph 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 transformations missed by classic PRE methods. The central idea is to use *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 (new) computations along some less frequently executed paths. (So the safety criterion [20] enforced in classic PRE methods is no longer honored.)

There are two previously published papers devoted entirely to speculation-based PRE (SPRE) [18, 19]. Horspool and Ho [19] use edge profiling information to determine the profitability of using speculation while Gupta, Berson and Fang [18] employ more expensive path profiling. While being capable of minimizing the number of computations in some cases as compared to the classic PRE methods, both algorithms are, in general, not computationally optimal in achieving this objective. In addition, both papers contain no implementation details, and consequently, no experimental results on the effectiveness of their methods.

Motivated by these two previous research efforts, this work addresses the following three problems:

- Is the computational optimality of SPRE solvable in polynomial time?
- Is edge profiling (less costly than path profiling) sufficient to guarantee the optimality?
- Is the optimal algorithm (if one exists) lightweight enough to be used efficiently in a dynamic compiler?

This paper provides positive answers to the first two problems and promising results to the third.

We present an algorithm, called *MC-PRE* (the "MC" stands for Min-Cut), that enables the speculation of an expression in a flow graph using an edge profile. *MC-PRE* guarantees that the total number of expression evaluations in the transformed code is *computationally optimal*, i.e., the smallest possible, with respect to the edge profile given. *The key in its development lies in the removal of some non-essential edges (and consequently, all non-essential nodes) from a given flow graph so that the problem of finding an optimal code motion emerges to be one of finding a minimal cut in the reduced graph thus obtained.*

Our running example is the flow graph given in Figure 1(a), where the underlined numbers alongside the flow edges represent the frequencies of execution. *MC-PRE*, which is conceptually simple, proceeds in three steps. First, we perform two standard data-flow analyses — one forward and one backward — on the original flow graph shown in Figure 1(a) to obtain the reduced flow graph as shown in Figure 1(b), where all non-essential edges and nodes (in dashes) have been removed from the original flow graph. Next, we transform routinely the reduced flow graph to obtain the single-source, single-sink graph given in Figure 1(c). Finally, we apply a standard min-cut algorithm to the single-source, single-sink graph to obtain optimally all the required edge insertion points, which are the edges in the cut. The sum of the frequencies of the edges in the cut is the minimal number of computations of $a+b$ that must be incurred with respect to the edge profile given. Figure 1(d) shows the transformed flow graph, requiring 300 computations of $a+b$. Note that the speculative execution of $a+b$ on the edge $(3,4)$ has made the three computations of $a+b$ at nodes 7, 8 and 10 in the original flow graph fully redundant. For this same example, the classic PRE [22], known as lazy code motion (LCM), would have produced the code in Figure 2, requiring 400 computations for the same expression.

We achieve computational optimality based on edge profiles. This implies immediately that the more expensive path profiling is not necessary for this optimization objective.

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

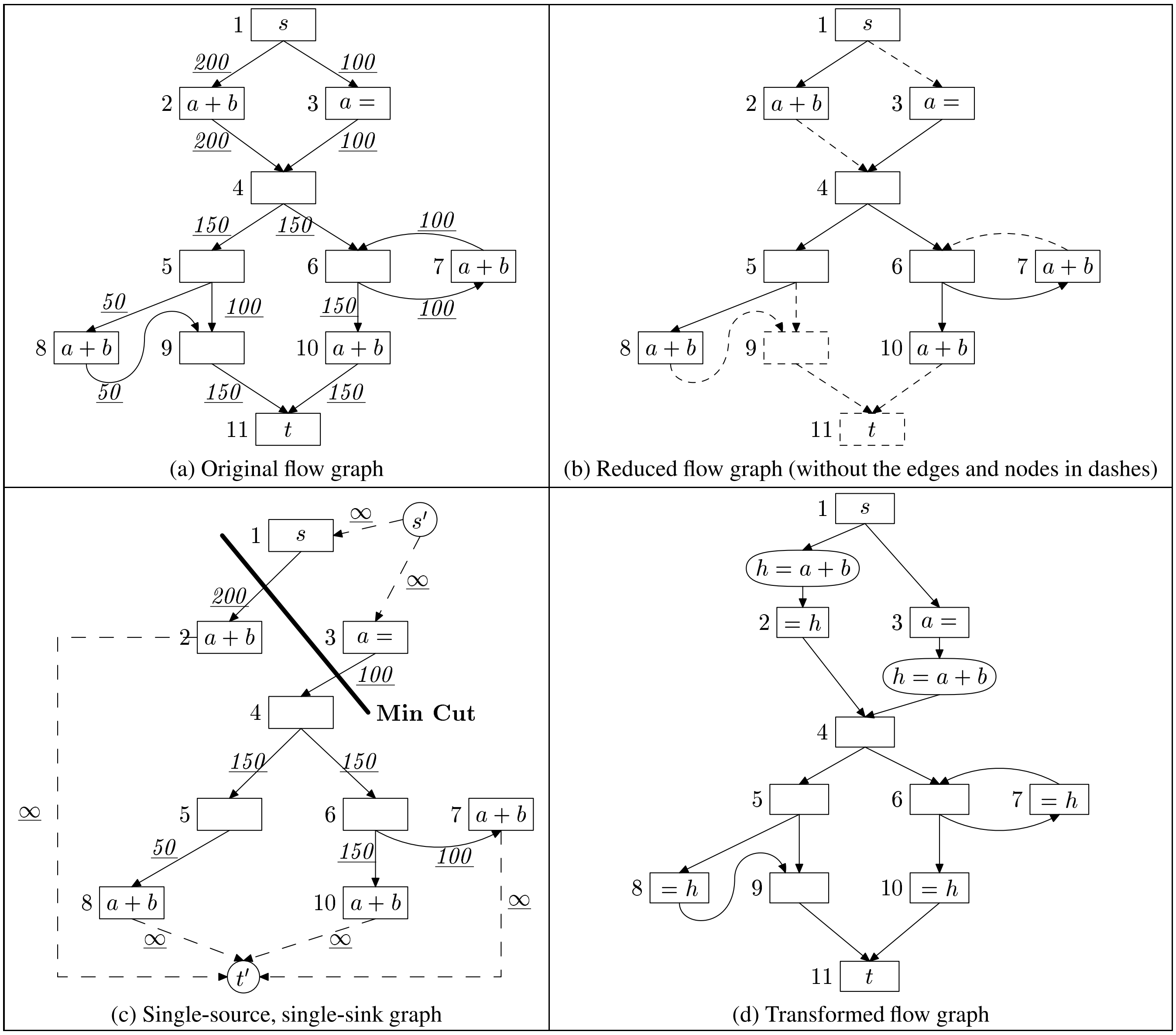
We have implemented our algorithm as a component in Intel's Open Runtime Platform (ORP) [1, 14]. We have evaluated its effectiveness over a number of Java benchmarks including SPEC JVM98, SciMark2, Linpack and JavaG.[1] In all experiments, we have restricted ourselves to exception-free expressions. In comparison with ORP running in its dynamic recompilation configuration (known as `inst` in [14]), our algorithm achieves comparable (and mostly better) performance results in all programs. In the case of SPEC JVM98, we obtain performance improvements in six out of its seven programs. These preliminary results are very encouraging in two aspects. First, our algorithm is lightweight and efficient (at least for programs characterized by the Java benchmarks used in our experiments). Second, profile-guided PRE, which has been perceived to be expensive in a dynamic compiler, can be profitably employed as one of its optimizing components.

The rest of this paper is organized as follows. Section 2 defines precisely speculation-based PRE as a code motion transformation in terms of correctness and computational optimality. Section 3 presents our optimal algorithm. Section 4 evaluates experimentally its efficiency. Section 5 compares with the related work. Section 6 concludes the paper.

2. Problem Statement

Section 2.1 defines the control flow graphs used, which have been simplified to facilitate our presentation. Section 2.2 provides a characterization of speculative PRE in terms of its correctness and computational optimality.

[1] As for JavaG, ORP presently crashes on MonteCarlo due to a heap error and fails to pass the validation phase for MolDyn and RayTracer.



(a) Original flow graph

(b) Reduced flow graph (without the edges and nodes in dashes)

(c) Single-source, single-sink graph

(d) Transformed flow graph

Figure 1. A running example illustrating our optimal algorithm.

2.1. Control Flow Graphs

A (control) flow graph is a directed graph annotated with an edge profile. We represent a flow graph (reducible or not) as a weighted graph $G_{fl} = (N_{fl}, E_{fl}, W_{fl})$, where

- the nodes in the node set N_{fl} represent basic blocks,
- the edges in the edge set E_{fl} represent potential flow of control between basic blocks, and
- W is a *weight function*: $W_{fl} : E_{fl} \mapsto \mathbb{N}$ (where $\mathbb{N}$ is the set of natural numbers starting from 0).

The weight $W_{fl}(u, v)$ attached to the edge $(u, v) \in E_{fl}$ is a nonnegative integer representing the frequency of its execution. The edge profiling information required can be gathered via code instrumentation [5], statistic sampling of the program counter [3] or static program-based heuristics [6, 27]. An edge profile has less runtime overhead to collect than a path profile [7]. The information contained in an edge profile, while less than a path profile, is sufficient to guarantee computationally optimal results.

Let $n \in N_{fl}$ be a node in a flow graph G_{fl}. We write *pred*$(n) = \{m \mid (m, n) \in E_{fl}\}$ and *succ*$(n) = \{m \mid (n, m) \in E_{fl}\}$ to denote the set of all immediate *predeces-*

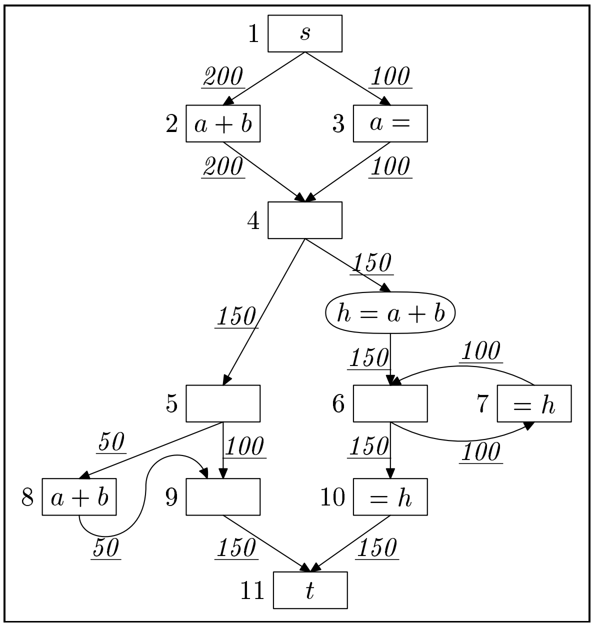


Figure 2. Transformation of the flow graph given in Figure 1(a) by the classic LCM [22].

1. Introduce a new temporary variable h_π for π
2. Insert $h_\pi = \pi$ at every insertion edge in $\mathcal{I}_\pi$
3. Replace π by h_π at every computation node of π

Figure 3. SPRE as a code motion transformation w.r.t. an expression π in a flow graph G_{fl}.

sors and *successors* of n, respectively. We write $edge(n) = \{(m,n) \mid m \in pred(n)\} \cup \{(n,m) \mid m \in succ(n)\}$ to denote the set of all edges *incident* to n. The paths and subpaths, which are all directed, are used in the normal manner.

The following properties about G_{fl} are assumed:

- $s \in N_{fl}$ represents a unique *entry node* without any predecessors,
- $t \in N_{fl}$ represents a unique *exit node* without any successors, and
- every node in N_{fl} lies on some path from s to t.

To simplify our discussions, we make a number of assumptions about the basic block nodes in a program. Each node contains only a single assignment of the form $v = \pi$, where v is a variable and π is an expression built in terms of variables, constants and operators.

For each node n, we define the two local predicates with respect to a generic expression π:

- **COMP**(n): n contains a computation of π (called a *computation node* of π).
- **TRANSP**(n): n is transparent, i.e., n does not modify any operands of π. A node that is not transparent for an expression is called a *modification node* of that expression.

We further assume that each node cannot both compute and modify π at the same time. That is, we disallow an assignment of the form $a = a + b$. Such an assignment can always be split into two sequential assignments $t = a + b$ and $a = t$, where t is a new temporary.

Assumption 1 *A node cannot be simultaneously a computation node and a modification node of the same expression.*

Finally, we consider s to be a modification node for every expression π to be optimized by our algorithm. That is, s has a definition of every variable in π to represent whatever value the variable may have when s is entered. Technically, this "prevents" any occurrence of π from being hoisted past s speculatively when our algorithm is applied.

Assumption 2 *The entry node s is a modification node for every expression π to be optimized by our algorithm.*

If the entry node in a flow graph is initially a computation node of an expression π, then a pseudo basic block can be introduced as the new entry node so that both assumptions given above hold trivially in the modified flow graph.

2.2. SPRE: Correctness and Optimality

The SPRE is realized as a code motion transformation with all insertions and deletions of an expression performed in such a way that the semantics of the program is preserved.

The objective of this work is to minimize the total number of computations for a generic expression π in a flow graph G_{fl}. A SPRE code transformation is completely characterized by a set, denoted $\mathcal{I}_\pi$ and called *the insertion set*, of edges (called *the insertion edges*) in the flow graph. The transformation to implement the effect of a given $\mathcal{I}_\pi$ consists of the three steps as summarized in Figure 3.

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

Definition 1 (Correctness) *$\mathcal{I}_\pi$ is* **correct** *if the following two properties hold for every computation node n in G_{fl}.*

P1. *Every path from the entry node s to the node n must include at least one insertion edge $(u,v) \in \mathcal{I}_\pi$, and*

P2. *no node in the subpath from v to n (inclusive) is a modification node of π.*

Due to Assumption 2, every computation node, which cannot be the entry node, must have some incoming edges. Hence, the existence of a correct insertion set is guaranteed.

The goal of minimizing the number of computations for a given expression in a flow graph is reflected by the following criterion of computational optimality.

Definition 2 (Optimality.) *$\mathcal{I}_\pi$ is* **optimal** *if (a) $\mathcal{I}_\pi$ is correct and (b) $\sum_{e \in \mathcal{I}_\pi} W_{fl}(e)$ is the smallest possible, i.e., $\sum_{e \in \mathcal{I}_\pi} W_{fl}(e) \leqslant \sum_{e' \in \mathcal{I}'_\pi} W_{fl}(e')$ for every possible correct insertion set $\mathcal{I}'_\pi$ (for the flow graph G_{fl}).*

In the running example shown in Figure 1, the insertion set $\mathcal{I}_{a+b} = \{(1,2),(3,4)\}$ is optimal. The transformation from Figure 1(a) to Figure 1(d) is straightforward.

3. *MC-PRE*: Optimal Algorithm

Figure 4 presents *MC-PRE* for finding an optimal insertion set $\mathcal{I}_\pi$ for an expression π. The key in developing our algorithm lies in the removal of the so-called non-essential edges (and consequently, all resulting non-essential nodes) from a flow graph G_{fl} so that the problem of finding an optimal insertion set in G_{fl} becomes one of finding a minimal cut on the reduced flow graph thus obtained.

Our algorithm proceeds in three steps. We first explain its three steps and prove its optimality. We then discuss its time complexity and give some remarks.

3.1. Step 1: Constructing a Reduced Graph

This step is the key to the algorithm. By removing the non-essential edges (and all resulting non-essential nodes) from the flow graph, we are able to realize that an optimal insertion set is simply a minimal cut on the reduced graph.

In 1(a), the standard availability system for an expression π from [23] is solved except that it is related to our two local predicates **COMP** and **TRANSP**. The two global predicates **N-AVAL** and **X-AVAL** on nodes are defined as follows. **N-AVAL**(n) denotes the availability of π on entry of a node n and **X-AVAL**(n) the same property at the exit of the same node. An expression is available on entry to a node n if it is available on exit from each predecessor of the node. An expression is available on exit from a node n if it is locally available (i.e., the node is a computation node of π by Assumption 1) or if it is available on entry of and transparent at the node.

In 1(b), the partial anticipability system for an expression π is adapted from the anticipability system in [23]. The two global predicates **N-PANT** and **X-PANT** on nodes are defined as follows. **N-PANT**(n) denotes the partial anticipability of π on entry of a node n and **X-PANT**(n) the same property at the exit of the same node. An expression is partially anticipatable on exit from the node n if it is partially anticipatable on entry of at least one successor of the node. An expression is partially anticipatable on entry to a node n if it is locally available (i.e., n is a computation node of π) or if it is partially anticipatable at the exit of and transparent at the node.

In 1(c), the four global predicates defined on the edges of a flow graph serve to classify them, in that order, into *insertion-redundant*, *insertion-useless*, *non-essential* and *essential* edges. The forward availability analysis detects the insertion-redundant edges while the backward partial anticipability analysis detects the insertion-useless edges.

The concept of essentiality for edges induces a similar concept for nodes. A node n in G_{fl} is *essential* if at least one of its incident edges is essential and *non-essential* otherwise.

In 1(d), the reduced graph G_{rd} is constructed as consisting of all essential edges in E_{fl} and all essential nodes in N_{fl} from the original flow graph.

Figure 5 illustrates this step of the algorithm using our running example. Figure 5(a) illustrates the exit-availability and insertion-redundant predicates, where the four insertion-redundant edges $(2,4),(7,6),(8,9)$ and $(10,11)$ found are depicted in dotted edges. Figure 5(b) illustrates the entry-partial-anticipability and insertion-useless predicates, where the five insertion-useless edges $(1,3),(5,9),(8,9),(9,11)$ and $(10,11)$ found are depicted in dashed edges. Note the possibility that some non-essential edges, such as $(8,9)$ and $(10,11)$ in this example, are both insertion-redundant and insertion-useless. It is not difficult to see from this example that an insertion-redundant edge is so named because an insertion $h_\pi = \pi$ on the edge is redundant since the value of h_π is already available on the edge in any correct SPRE transformation. Similarly, an insertion-useless edge is so named because an insertion $h_\pi = \pi$ on the edge can never make the value of h_π available to any computation node of π in G_{fl}.

The following lemma implies that an optimal insertion set exists on the reduced graph G_{rd}. It will be used in establishing the optimality of $\mathcal{I}_\pi$ found by our algorithm.

Lemma 1 *Let e be a non-essential edge for an expression π, where $W_{fl}(e) \geqslant 0$. If $W_{fl}(e) > 0$, then e cannot be an insertion edge in any optimal insertion set $\mathcal{I}_\pi$. If $W_{fl}(e) = 0$, then every optimal insertion set $\mathcal{I}_\pi$ remains to be optimal even after the edge e is removed from $\mathcal{I}_\pi$.*

Proof. Suppose $\mathcal{I}_\pi$ is an optimal insertion set including the non-essential edge e. Let e be an insertion-redundant (insertion-useless, resp.) edge. By Definition 1 and the definition of **INS-REDUND**(e) (**INS-USELESS**(e), resp.), we find that $\mathcal{I}'_\pi = \mathcal{I}_\pi \setminus \{e\}$ is also a correct insertion set.

Algorithm *MC-PRE*

INPUT: a weighted flow graph $G_{fl} = (N_{fl}, E_{fl}, W_{fl})$ and an expression π

OUTPUT: an optimal insertion set $\mathcal{I}_\pi$

1. Obtain from G_{fl} a **reduced (flow) graph** $G_{rd} = (N_{rd}, E_{rd}, W_{rd})$ by removing non-essential edges and nodes:
   (a) Solve the standard forward **availability** analysis (initialized to *true*):

$$\textbf{N-AVAL}(n) = \begin{cases} \textit{false} & \text{if } n \text{ is the entry node } s \\ \bigwedge_{m \in \textit{pred}(n)} \textbf{X-AVAL}(m) & \text{otherwise} \end{cases}$$

$$\textbf{X-AVAL}(n) = \textbf{COMP}(n) \vee (\textbf{N-AVAL}(n) \wedge \textbf{TRANSP}(n))$$

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

$$\textbf{X-PANT}(n) = \begin{cases} \textit{false} & \text{if } n \text{ is the exit node } t \\ \bigvee_{m \in \textit{succ}(n)} \textbf{N-PANT}(m) & \text{otherwise} \end{cases}$$

$$\textbf{N-PANT}(n) = \textbf{COMP}(n) \vee (\textbf{X-PANT}(n) \wedge \textbf{TRANSP}(n))$$

   (c) Define the four predicates on the edges of $(u, v) \in E_{fl}$ (no flow analysis):

$$\begin{aligned} \textbf{INS-REDUND}(u,v) &=_{df} \textbf{X-AVAL}(u) \\ \textbf{INS-USELESS}(u,v) &=_{df} \neg\, \textbf{N-PANT}(v) \\ \textbf{NON-ESS}(u,v) &=_{df} \textbf{INS-REDUND}(u,v) \vee \textbf{INS-USELESS}(u,v) \\ \textbf{ESS}(u,v) &=_{df} \neg \textbf{NON-ESS}(u,v) \end{aligned}$$

   (d) Construct G_{rd} as follows:

$$\begin{aligned} N_{rd} &= \{n \in N_{fl} \mid \exists\, e \in \textit{edge}(n) : \textbf{ESS}(e)\} \\ E_{rd} &= \{e \in E_{fl} \mid \textbf{ESS}(e)\} \\ W_{rd} &= W_{fl} \text{ restricted to the domain } E_{rd} \end{aligned}$$

2. Convert G_{rd} into a **single-source, single-sink graph** $G_{st} = (N_{st}, E_{st}, W_{st})$:
   (a) Let $S_{rd} = \{n \in N_{rd} \mid |\textit{pred}(n)| = 0\}$ and $T_{rd} = \{n \in N_{rd} \mid |\textit{succ}(n)| = 0\}$
   (b) Let s' be a new entry node and t' a new exit node
   (c) Construct G_{st} as follows:

$$\begin{aligned} N_{st} &= N_{rd} \cup \{s', t'\} \\ E_{st} &= E_{rd} \cup \{(s', n) \mid n \in S_{rd}\} \cup \{(n, t') \mid n \in T_{rd}\} \\ W_{st} &= W_{rd} \text{ (extended to } E_{st}) \text{ such that } \forall\, e \in (E_{st} \setminus E_{rd}) : W_{st}(e) = \infty. \end{aligned}$$

3. Find an **optimal insertion set** as a minimal cut on G_{st}:
   $\mathcal{I}_\pi = MIN\_CUT(G_{st})$;

Figure 4. An algorithm finding an optimal insertion set $\mathcal{I}_\pi$ for an expression π in a flow graph G_{fl}.

If $W_{fl}(e) > 0$, then $\sum_{f' \in \mathcal{I}'_\pi} W_{fl}(f') < \sum_{f \in \mathcal{I}_\pi} W_{fl}(f)$, which contradicts to the fact that $\mathcal{I}_\pi$ is optimal. If $W_{fl}(e) = 0$, then $\sum_{f' \in \mathcal{I}'_\pi} W_{fl}(f') = \sum_{f \in \mathcal{I}_\pi} W_{fl}(f)$, which implies that $\mathcal{I}'_\pi$ is also optimal. This concludes the proof of the lemma. ∎

3.2. Step 2: Obtaining a Single-Source, Single-Sink Graph

Due to the removal of the non-essential edges and nodes, the reduced graph G_{rd} obtained in Step 1 is usually a multi-

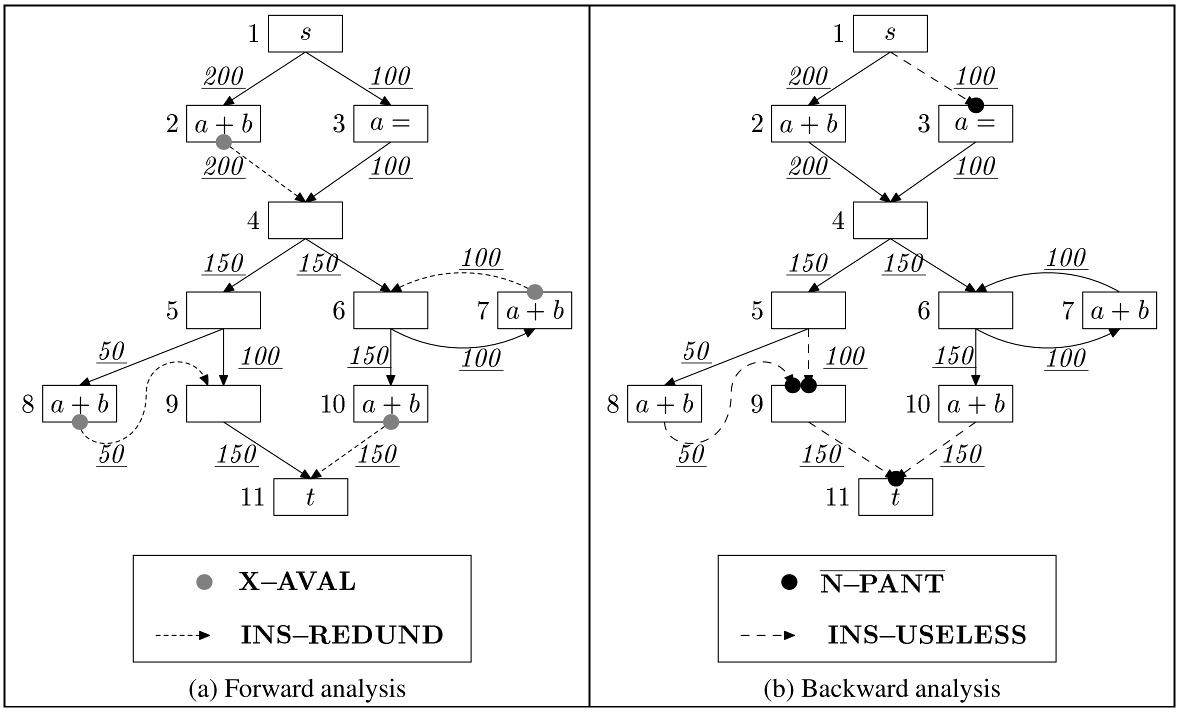


Figure 5. Identification of non-essential (and essential) edges in a flow graph G_{fl}.

source, multi-sink graph. In 2(a), S_{rd} is defined to be the set of all such source nodes and T_{rd} the set of all such sink nodes. In 2(b), the new pseudo entry and exit nodes are added. In 2(c), we obtain a single-source, single-sink graph, in which all newly introduced edges have the weight ∞.

In our running example, where the original flow graph is given in Figure 1(a), Figure 1(b) depicts the reduced graph, where the nodes 1 and 3 are the sources and the nodes 2, 7 and 10 the sinks. Figure 1(c) gives the single source, single-sink graph constructed from this reduced graph.

The following lemma will be used in establishing the correctness of $\mathcal{I}_\pi$ found by our algorithm.

Lemma 2 *Let $p(u, v)$ be a path from u to v in a flow graph G_{fl}, where $u \neq v$, such that*

- *u is a modification node of π (u could be the entry node s by Assumption 2),*
- *v is a computation node of π, and*
- *no other nodes on the path $p(u, v)$ are modification or computation nodes of π.*

Then the following statements are true:

(a) G_{rd} contains the path $p(u, v)$ (i.e., all its nodes and edges),

(b) $u \in S_{rd}$, and

(c) $v \in T_{rd}$.

Proof. Figure 6 illustrates the exit-availability and entry-partial-anticipability of the end points of all the edges on the path $p(u, v)$. Note that the values of these two predicates are completely defined by examining the nodes and edges on the path $p(u, v)$ *alone*. By the definition of **ESS**, all the edges on this path are essential. As a consequence, all the nodes on the path are also essential. Hence, (a) is true. Since u is a modification node, all its incoming edges are insertion-useless. This means that $|pred(u)| = 0$ in G_{rd}. Hence, (b) is true. Since v is a computation node and by also noting Assumption 1, all its outgoing edges are insertion-redundant. This means that $|succ(v)| = 0$ in G_{rd}. Hence, (c) is true. ■

The following four lemmas provide some characterizations of the reduced graph G_{rd}. While useful in shedding the light on its structure, they are not used in any way in establishing the correctness and optimality of our algorithm. Therefore, their proofs are omitted.

Lemma 3 *$G_{rd} = \emptyset$ iff G_{fl} contains no computation nodes.*

Lemma 4 *Let $n \in G_{rd}$. Then $n \in S_{rd}$ iff n is a modification node of π.*

Lemma 5 *Let $n \in G_{rd}$. Then $n \in T_{rd}$ iff n is a computation node of π.*

Lemma 6 *$S_{rd} \cap T_{rd} = \emptyset$.*

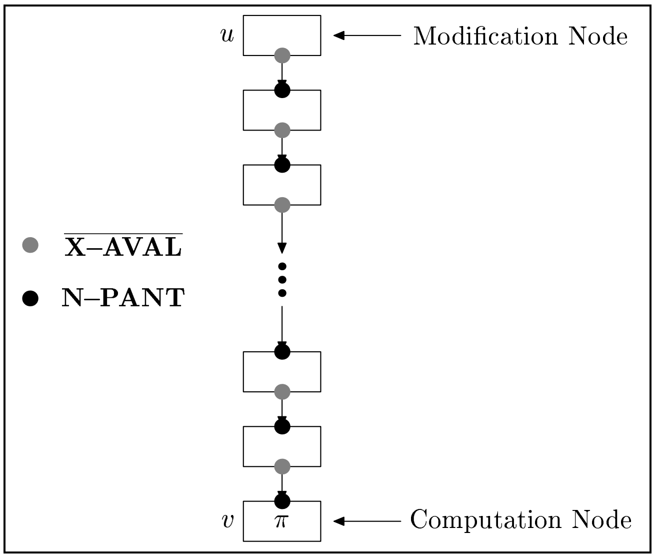


Figure 6. An illustration of the proof of Lemma 2.

3.3. Step 3: Finding an Optimal Insertion Set

In this step, $\mathcal{I}_\pi$ is chosen to be a minimal cut on the single-source, single-sink graph G_{st} by applying a min-cut algorithm. For our running example, Figure 1(c) depicts the unique minimal cut on G_{st}, where $\mathcal{I}_{a+b} = \{(1,2),(3,4)\}$. The minimal number of computations of $a+b$ is 300.

Theorem 1 *$\mathcal{I}_\pi$ found by MC-PRE is optimal.*

Proof. The proof has two parts. The first part establishes the correctness of $\mathcal{I}_\pi$ while the second part its optimality. Since we have converted the multi-source, multi-sink reduced graph G_{rd} into the single-source, single-sink graph G_{st} in the standard way [15, p. 584], a minimal cut found on G_{st} (that separates s' from t') is also a minimal cut on G_{rd} (that separates all the source nodes in S_{rd} from all the sink nodes in T_{rd}).

Part I: Correctness. Let n be an arbitrary but fixed computation node of π. Let $p(s,n)$ be an arbitrary but fixed path from the entry node s to the node n. We proceed to show that $\mathcal{I}_\pi$ satisfies Properties P1 and P2 stated in Definition 1. By Assumption 2, $p(s,n)$ must contain at least one modification node of π, which could be the entry node s. Let u be the last modification node on the path $p(s,n)$. Let v be the first computation node immediately after u (v could be the node n). Such a computation node v must exist since the node n, which is a computation node, appears at the end of the path $p(s,n)$. By Assumption 1, $u \neq v$. Let $p(u,v)$ be the subpath of $p(s,n)$ from u to v (inclusive). By Lemma 2, Statements (a) – (c) stated in that lemma are all true (see Figure 6 again). This means that $\mathcal{I}_\pi$ — which is a cut on G_{rd} — must include at least one edge from $p(u,v)$. Hence, Property P1 is satisfied. By construction, the subpath $p(v,n)$ of $p(s,n)$, i.e., its subpath starting from v to n, does not contain any modification nodes of π. Hence, Property P2 is also satisfied. Thus, $\mathcal{I}_\pi$ is a correct insertion set.

Part II: Optimality. $\mathcal{I}_\pi$ is a minimal cut on G_{rd}. Lemma 1 implies trivially that an optimal insertion set for the original flow graph G_{fl} can be found from G_{rd}. It is easy to show that every correct insertion set (consisting of only edges from G_{rd}) must be a cut on G_{st}. Thus, $\mathcal{I}_\pi$, which is the best among all correct insertion sets, must be optimal by Definition 2. ■

3.4. Time Complexity

The overall time complexity of *MC-PRE* is dominated by the two uni-directional data-flow analysis passes performed in Step 1 and the min-cut algorithm employed in Step 3. The two passes can be done in parallel for all expressions in a flow graph but 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 flow graph $G_{fl} = (N_{fl}, E_{fl}, W_{fl})$ individually, the worst-case time complexity for each iterative pass is $O(|N_{fl}| \times (d+2))$, where d is the maximum number of back edges on any acyclic path in G_{fl} and typically $d \leqslant 3$ in practical cases [24].

The min-cut step of our algorithm operates on the reduced graph $G_{rd} = (N_{rd}, E_{rd}, W_{rd})$. There are a variety of polynomial algorithms in the literature with different time complexities [11]. In our implementation, we have used Goldberg's *push-relabel* HIPR algorithm since it has been reported to be efficient with its worst-time complexity being $O(|N_{rd}|^2\sqrt{|E_{rd}|})$ [17]. Hence, *MC-PRE* has a polynomial time complexity overall. In Section 4, we discuss the compile-time overhead of our algorithm on benchmark programs.

3.5. Discussion

There are two reasons why we have used edge insertions based on edge profiles.

Edge Insertions v.s. Node Insertions. The edge insertions are more general than the alternative of permitting insertions only in basic block nodes. Based on the criterion of the computational optimality given in Definition 2, the edge insertions are at least as good as the node insertions. Let J be the set of insertion points at the node entries or exits. Let I be the set of edges obtained from J by moving an entry insertion point to all its incoming edges and an exit insertion point to all its outgoing edges. The total number of computations in both cases is the same. However, if some of these edges are non-essential, the insertions on these edges are not necessary. If these non-essential edges have non-zero weights, then the total number of computations using the edge insertions is smaller.

Edge Profiles v.s. Path Profiles. An edge profile is sufficient to find an optimal insertion set. Let J be an optimal (edge) insertion set obtained in some way based on a path profile. Let I be an optimal insertion set found in this paper based on the edge profile derived from the path profile. This means that the edge frequencies are the same in both cases. Then J is at best as good as I by the criterion of computational optimality since I is optimal. Nevertheless, we plan to investigate how to exploit the extra information available in a path profile for other optimization purposes.

4. Experiments

PRE, which is perceived as an expensive optimization, has not been used in a number of dynamic or JIT compilers [1, 2, 14]. We have implemented our *MC-PRE* algorithm in Intel's ORP — a dynamic compilation system for Java programs [1, 14]. Our experimental results over a number of Java benchmarks including SPEC JVM98, SciMark2, Linpack and JavaG demonstrate that our algorithm is lightweight and efficient. We present and analyze our performance results for the SPEC JVM98 benchmark below.

4.1. Implementation Details

ORP consists of two compilers: a baseline compiler known as the O1 and an optimizer known as the O3 [14]. The O1 compiler translates all methods to native code quickly when they are first invoked and performs only some lightweight optimizations. Instrumentation code is inserted into the native code to collect some profiling information. As the code is executed, the instrumentation code updates the profiling information. Based on the profiling information, some methods are identified as hot methods and then recompiled by the O3 optimizer, which performs a number of standard optimizations such as common subexpression elimination, loop-invariant motion and method inlining. In the ORP paper [14], this option is referred to as the `inst` configuration.

In our experiments, we have added a new configuration, called `spre`, which is the same as `inst` except that the loop-invariant code motion in ORP is disabled and our *MC-PRE* is used in its place. ORP eliminates common subexpressions in a flow graph by performing only local common subexpression elimination (LCSE) on extended basic blocks. Since PRE only eliminates partial redundancies across the basic blocks, we have turned LCSE on in both `inst` and `spre`. Therefore, *MC-PRE*, once implemented in ORP, works with the basic blocks containing multiple instructions. As in [22], only the upward- and downward-exposed expressions in a basic block are considered as candidates for partial redundancy elimination.

In the `spre` configuration, we collect an edge profile for the flow graph of a method via code instrumentation. We insert instrumentation code at all the branches in the native code generated by the O1 baseline compiler. When the method becomes hot and needs to be recompiled, we compute from the branch profiling information the edge frequencies in the flow graph. We then feed the edge profile thus constructed to *MC-PRE* to obtain an optimal code motion for the method. The runtime overhead of our simplistic profiling mechanism can be improved if some better techniques are used [4, 26].

We evaluate the lightweightness and effectiveness of *MC-PRE* by comparing `inst` and `spre`. All our experiments were performed on a 933MHz Pentium III workstation with 512MB RAM. The default heap size used when running ORP is 80MB. In all experiments, *MC-PRE* is applied to exception-free expressions only.

4.2. Performance Evaluation

In a dynamic compiler, the total *running time* of a program consists of the (dynamic) compilation time and the actual execution time of the program. It is therefore understood that profiling overhead (if any) is reflected in both time components. Any time the compiler spends on generating instrumenting code and manipulating the collected profiling information is included as part of the compilation time of the program. Any time taken on collecting profiling information during the execution of the program becomes part of its execution time. In practice, the former portion of the profiling overhead is usually negligible.

Table 1 gives the times and speedups of all the seven SPEC JVM98 benchmark programs obtained under the `inst` and and `spre` configurations. In the `inst` configuration, ORP uses its profiling mechanism to detect hot methods for recompilation by the O3 optimizer. In our `spre` configuration, we have added further instrumenting code in order to build an edge profile for every method.

Let us examine the last three columns of Table 1, which are also plotted graphically in Figure 7, to see how `spre` performs as compared to `inst`. $Speedup_{run}$ gives the speedup of each program, i.e., how much faster each program runs in `spre` than in `inst`. $Speedup_{comp}$ and $Speedup_{exec}$ represent the percentage extra (re)compilation cost incurred and percentage execution time reduction obtained in `spre` over `inst`, respectively. By their definitions, $Speedup_{run}$=$Speedup_{comp}$+$Speedup_{exec}$ holds.

Program	inst (secs)			spre (secs)			Speedups (%)		
	comp (c_i)	exec (e_i)	run (r_i)	comp (c_s)	exec (e_s)	run (r_s)	$Speedup_{comp}$ ($\frac{c_i-c_s}{r_s}$)	$Speedup_{exec}$ ($\frac{e_i-e_s}{r_s}$)	$Speedup_{run}$ ($\frac{r_i-r_s}{r_s}$)
compress	0.12	14.62	14.74	0.18	14.52	14.70	-0.41	0.68	0.27
jess	0.44	7.86	8.30	0.49	7.56	8.05	-0.62	3.73	3.11
db	0.12	23.89	24.01	0.19	23.81	24.00	-0.29	0.33	0.04
javac	1.24	22.36	23.60	1.78	22.23	24.01	-2.25	0.54	-1.71
mpeg.	0.32	9.36	9.68	0.42	9.03	9.45	-1.06	3.49	2.43
mtrt	0.43	6.45	6.88	0.56	6.17	6.73	-1.93	4.16	2.23
jack	0.66	8.25	8.91	0.72	8.00	8.72	-0.69	2.87	2.18

Table 1. Times and speedups. The three time components for a program are interpreted as follows: `comp` represents its compilation time, `exec` its execution time, and `run` its total running time such that `run=comp+exec`. For the last three columns, we have $Speedup_{run}$= $Speedup_{comp}$+ $Speedup_{exec}$.

Thus, in order for `spre` to perform better than `inst` for a program, the increase in its compilation cost (represented by $Speedup_{comp}$) must be more than offset by the decrease in its execution time (represented by $Speedup_{exec}$). Note that it makes sense to refer to the quantities denoted by $Speedup_{comp}$ and $Speedup_{exec}$ as speedups. Consider the two extreme cases. If $Speedup_{comp} \approx 0$ (i.e., `spre` causes no or little extra compilation cost), then $Speedup_{exec} \approx Speedup_{run}$. If $Speedup_{exec} \approx 0$ (i.e., `spre` yields no or little gain in execution time), then $Speedup_{comp} \approx Speedup_{run}$ (which should usually be negative).

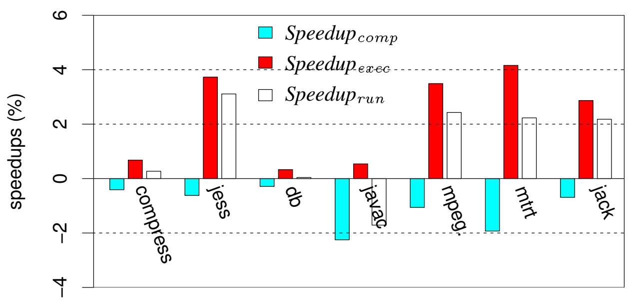


Figure 7. Speedups (in bar charts).

Figure 7 shows clearly that `spre` outperforms `inst` in six of the seven programs. By performing code motion optimally based on a runtime profile, our algorithm has reduced effectively the execution times of all seven programs. In the case of `javac` where the performance improvement is negative, a lot of methods are recompiled dynamically. However, *MC-PRE* does not benefit from the recompilation since many of these methods have few exception-free expressions with little optimization opportunities. In this case, the time taken for constructing edge profiles for their flow graphs (since we instrument only at the branches) becomes pure overhead. In addition, *MC-PRE* has been applied non-

Program	Edge Prof (secs) (p)	$\overline{run}$ (secs) ($\overline{r_s}=r_s-p$)	$Speedup_{\overline{run}}$(%) ($\frac{r_i-\overline{r_s}}{\overline{r_s}}$)
compress	0.09	14.61	0.89
jess	0.10	7.95	4.40
db	0.29	23.71	1.27
javac	0.18	23.83	-0.97
mpeg.	0.09	9.36	3.42
mtrt	0.08	6.65	3.46
jack	0.08	8.64	3.13

Table 2. Improved running times and speedups without edge profiling overhead. Note that r_s and r_i are from Table 1.

profitably to their corresponding reduced graphs. These two factors together have resulted in the large increase in the compilation overhead for `javac` in Figure 7.

In a dynamic compiler where some edge profiling mechanism exists, the edge profiling information is very likely already used by a number of optimizations such as method inlining and locality optimizations. In this case, the edge profiling overhead should not all be charged into *MC-PRE*. To estimate the edge profiling overhead incurred for a program, we run `inst` with our edge profiling mechanism turned on and off. The difference between the two running times obtained is taken as a good estimate. In actuality, `inst` was run many times in order to get an accurate estimate. Table 2 gives the edge profiling times and improved running times and speedups for all the SPEC JVM98 programs. The improved speedups are compared graphically in Figure 8 against the original ones from Figure 7.

Table 3 shows for each program the average size of all non-empty reduced graphs and the average size of their corresponding flow graphs. We exclude empty reduced graphs, and accordingly, their corresponding flow graphs since

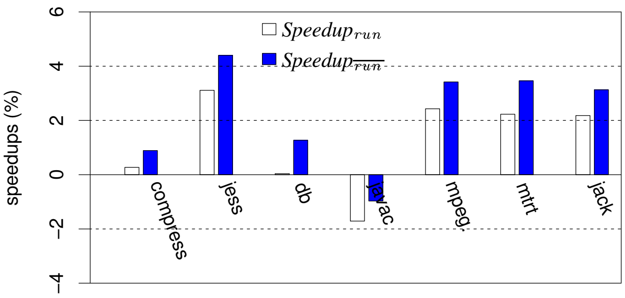


Figure 8. A comparison of $Speedup_{run}$ from Table 1 and $Speedup_{\overline{run}}$ from Table 2.

Program	(#nodes $\pm SD$, #edges $\pm SD$)	
	G_{fl}	G_{rd}
compress	$(18 \pm 15, 24 \pm 25)$	$(\ 6 \pm 10,\ \ 7 \pm 16)$
jess	$(35 \pm 47, 49 \pm 71)$	$(11 \pm 17, 14 \pm 26)$
db	$(15 \pm 15, 21 \pm 25)$	$(\ 7 \pm 10,\ \ 7 \pm 10)$
javac	$(22 \pm 21, 30 \pm 32)$	$(\ 6 \pm\ \ 9,\ \ 7 \pm 13)$
mpeg.	$(13 \pm 10, 17 \pm 17)$	$(\ 6 \pm\ \ 7,\ \ 6 \pm 11)$
mtrt	$(16 \pm 13, 21 \pm 22)$	$(\ 7 \pm\ \ 8,\ \ 8 \pm 13)$
jack	$(19 \pm 16, 26 \pm 25)$	$(\ 9 \pm 10, 10 \pm 16)$

Table 3. Average sizes for non-empty reduced graphs and their corresponding flow graphs.

MC-PRE does not perform its min-cut step on them. Therefore, these statistics represent the average sizes of the original flow graphs and their reduced graphs *per expression* that have been processed by the min-cut step of *MC-PRE*. Clearly, the removal of non-essential edges and nodes in the first step of *MC-PRE* have been effective in reducing the sizes of the graphs to be solved by the min-cut step. It should be pointed out that some relatively large graphs processed by *MC-PRE* may not be directly evident from Table 3. For example, the largest reduced graph (w.r.t the number of nodes) from `jess` has 127 nodes and 187 edges and the largest from `javac` has 88 nodes and 119 edges.

5. Related Work

Partial redundancy elimination originated from the seminar work of Morel and Renvoise [23] and was quickly realized as an important optimization technique that subsumes global common subexpression and loop invariant code motion. Ever since, their work has been extended in several directions [10, 12, 16, 21, 22, 25]. In particular, Knoop, Rüthing and Steffen describe a uni-directional bit-vector formulation that is optimal by the criteria of computational optimality and lifetime optimality [22], and more recently, Kennedy *et al* present an SSA-based framework that shares the same two optimality properties. These efforts are restricted by the safety code motion criterion [20] and insensitive to the execution frequencies of a program point in a flow graph.

To overcome these two limitations, there have been two papers devoted entirely to reducing the total number of expression evaluations in a flow graph even further by using speculation. Horspool and Ho [19] analyze edge insertions based on an edge profile while Gupta, Berson and Fang [18] use a path profile, which is more expensive to collect. Both methods are not computationally optimal as their respective authors demonstrate by examples in their papers. In addition, both papers contain neither implementation details nor experimental results on the effectiveness of their methods.

After this work was accepted, we have been made aware of some results on profile-guided PRE described by Bodik in his PhD thesis [8]. He proposes to solve the SPRE problem in four steps. First, he conducts iterative availability and anticipability analyses on a flow graph to identify the so-called CMP (*Code-Motion-Preventing*) region in the flow graph. Both passes require two bits for each expression since his predicates operate on four lattice values during the CMP construction. Second, he finds a min-cut on the CMP to obtain some insertion edges, which may not be all insertion edges required. This component of the algorithm is referred to as the CMP^f estimator, one of the four estimators developed in the thesis. Third, after having completed the insertions on the insertion edges found, he repeats the availability analysis (again requiring two bits for each expression) to find all insertion edges required. Finally, he applies essentially the classic LCM [22] to obtain the transformed code. In [8, Theorem 6.3], he gives a three-sentence proof outline about the computational optimality of his algorithm. He did not give experimental results for *this* algorithm since CMP^f is not implemented [8, p. 75]. Instead he recommends that the non-optimal estimator CMP^c be the estimator of choice due to its simplicity [8, p. 76]. His experimental results using SPEC95 benchmarks in a static compiler indicate that this simple heuristic is nearly optimal in many cases. In [9, §3.2], Bodik *et al* also discussed a non-optimal SPRE algorithm that does not use any control flow restructuring.

Our algorithm, *MC-PRE*, proceeds in essentially two major steps. The first step performs the standard availability and partial anticipability analyses in order to find the reduced graph from a given flow graph — each pass requires one bit for each expression. In the second step, all insertion edges are found optimally as the edges in a minimal cut on the reduced graph. *MC-PRE* is always optimal. Our preliminary experimental results over a range of Java benchmarks indicate that it is efficient even in a dynamic compiler.

6. Conclusion

We have presented an algorithm for partial redundancy elimination using speculation. Our algorithm always produces the best code possible with respect to a given edge profile in the sense that the total number of computations for any expression is minimized. This result implies that edge profiling, which is less costly than path profiling, is sufficient to guarantee this computational optimality. We have implemented our algorithm in Intel's ORP. Our preliminary results over a number of Java benchmarks are promising. In the case of the SPEC JVM98 benchmark, we have achieved performance improvements in six out of its seven programs over ORP's dynamic recompilation. We expect our algorithm to be useful in dynamic and static compilers.

We have achieved lifetime optimality and other results on profile-guided PRE. They will be the subject of another paper.

7. Acknowledgements

We would like to thank the referees and Partha Tirumalai of SUN for their helpful comments and suggestions. We would also like to thank Intel for making their ORP available. Finally, we are grateful to the ORP team for answering our ORP-related questions.

References

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

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), 2000.

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

[4] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *ACM SIGPLAN' 01 Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

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

[6] T. Ball and J. R. Larus. Branch prediction for free. In *ACM SIGPLAN' 93 Conference on Programming Language Design and Implementation*, pages 300–313, 1993.

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

[8] R. Bodik. *Path-Sensitive Value-Flow Optimizations of Programs*. PhD thesis, University of Pittsburgh, 1999.

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

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

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

[12] F. Chow. *A portable machine-independent global optimizer — design and measurements*. PhD thesis, Computer Systems Laboratory, Stanford University, 1983.

[13] F. Chow, M. Himelstein, E. Killian, and L. Weber. Engineering a RISC compiler. In *Proceedings of IEEE COMPCON*, pages 132–137, 1986.

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

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

[16] D. M. Dhamdhere. Practical adaption of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, 1991.

[17] A. GoldBerg. Network Optimization Library. http://www.avglab.com/andrew/soft.html.

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

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

[20] K. Kennedy. Safety of code motion. *International Journal of Computer Mathematics*, 3(2–3):117–130, 1972.

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

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

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

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

[25] L. T. Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, 1996.

[26] O. Traub, S. Schechter, and M. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, 2000.

[27] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *27th International Symposium on Microarchitecture*, pages 1–11, 1994.