The Power of Assignment Motion

Jens Knoop
Universität Passau *
knoop@fmi.uni-passau.de

Oliver Rüthing
CAU Kiel †
or@informatik.uni-kiel.d400.de

Bernhard Steffen
Universität Passau *
steffen@fmi.uni-passau.de

Abstract

Assignment motion (AM) and *expression motion (EM)* are the basis of powerful and at the first sight incomparable techniques for removing partially redundant code from a program. Whereas AM aims at the elimination of complete assignments, a transformation which is always desirable, the more flexible EM requires temporaries to remove partial redundancies. Based on the observation that a simple program transformation enhances AM to subsume EM, we develop an algorithm that for the first time captures all second order effects between AM and EM transformations. Under usual structural restrictions, the worst case time complexity of our algorithm is essentially quadratic, a fact which explains the promising experience with our implementation.

Topics: data flow analysis, program optimization, partially redundant assignment and expression elimination, code motion, assignment motion, bit-vector data flow analyses.

In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95)*, La Jolla, California, *SIGPLAN Notices 30*, 6 (1995), 233 - 245.

*Fakultät für Mathematik und Informatik, Universität Passau, Innstraße 33, D-94032 Passau, Germany.

†Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, Preußerstraße 1–9, D-24105 Kiel, Germany.

1 Motivation

A major source for improving the run-time efficiency of a program is to avoid *unnecessary* recomputations and reexecutions of values (*expressions*) and instructions (*assignments*), as it is illustrated in Figure 1 and 2, respectively.

Figure 1 shows how to avoid the unnecessary *recomputations* of $a + b$ in node **2** and **3** by initializing a new temporary **h** in node **1** and replacing the original occurrences of $a + b$ by **h**. This is known as *partially redundant expression elimination (PREE)* or as *expression motion (EM)* for short (cf. [4, 9, 11, 16, 18, 20]).

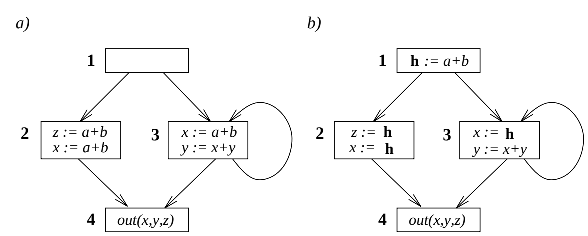

Figure 1: *Expression* Motion (EM)

Figure 2 shows how to avoid the unnecessary *reexecution* of $x := a + b$ in node **3** of the loop by hoisting the occurrences of $x := a+b$ in node **2** and **3** to node **1**. In analogy to PREE this technique is called *partially redundant assignment elimination (PRAE)* or simply *assignment motion (AM)* (cf. [7]).

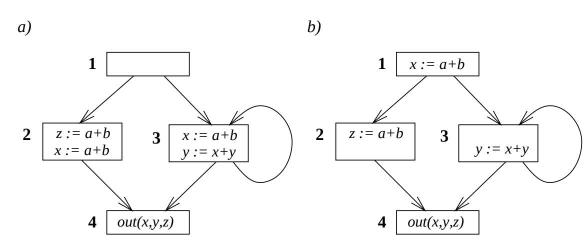

Figure 2: *Assignment* Motion (AM)

Note that the program transformations of Figure 1 and 2 are incomparable. However, after the simple program transformation illustrated in Figure 3(a), AM subsumes EM. In fact, applying AM to the program of Figure 3(a) results in the one of Figure 3(b).

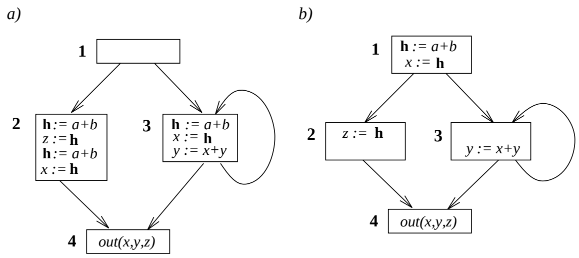

Figure 3: Uniform EM & AM

We exploit this observation for the construction of an aggressive and uniform algorithm capturing all second order effects between EM and AM transformations. In fact, transformed programs are *expression optimal* in the following sense:[1] each execution requires at most as many expression evaluations as its counterpart in any other program resulting from some arbitrary combination of AM and EM transformations.

Besides expression optimality, costs for assignments are a major concern, in particular, as EM introduces assignments to temporaries. This is taken care of by the *final flush* phase of our algorithm, which eliminates all unnecessary assignments to temporaries introduced during the redundancy elimination phase.

1.1 The Running Example

The example of Figure 4 illustrates the power of our algorithm, which is unique in performing the optimization displayed in Figure 5.

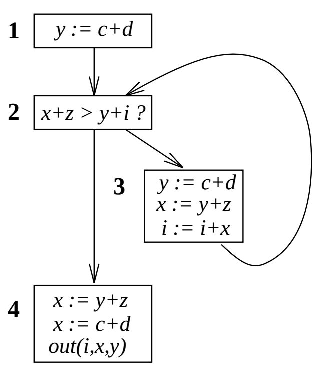

Figure 4: The Running Example

This optimization is achieved by our algorithm in the following steps: It eliminates the assignment $y := c+d$ in node **3**, which is redundant with respect to the corresponding assignment of node **1**. Additionally, it initializes a temporary **h1** in block **1** by $c + d$ and a temporary **h2** in block **1** and **3** by $x + z$ and replaces the original occurrences of $c + d$ in node **1** and **4** by **h1**, and the original occurrence of $x + z$ in node **2** by **h2**. Note that this suspends the blockade of the assignment $x := y + z$ in the loop. Thus, as a second order effect the assignment $x := y + z$ can now be removed from the loop by simultaneously hoisting it with the corresponding assignment of node **4** to node **1**.

[1]Expression optimality is called computational optimality in [16, 18].

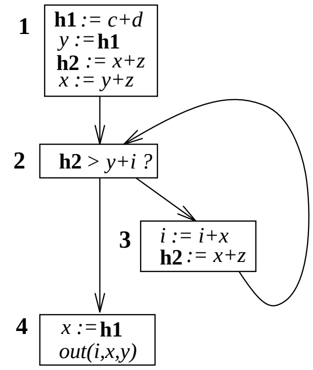

Figure 5: The Power of Uniform EM & AM

Note that our algorithm does not touch the assignment $i := i+x$ in node **3** and the computations of $y+i$ and $i+x$ in nodes **2** and **3**, respectively, which cannot be moved profitably.

1.2 Separate Effects

It is worth noting that standard techniques for EM and AM fail to achieve the optimization of Figure 5. In particular, they fail to eliminate the most significant inefficiency in the program of Figure 4, the 'loop invariant' assignment $x := y + z$ in node **3**, which is blocked by the assignment to y in node **3** and the use of x in the Boolean expression controlling the loop iteration. This is illustrated in Figure 6(a) and 6(b), which display the separate effects of EM and AM to the program of Figure 4, respectively.

1.3 The Algorithm

The systematic treatment of 'second order' effects as between the expression $x + z$ and the assignments $y := c + d$ and $x := y + z$ in the example of Figure 4 is an important feature of our algorithm, which for the first time captures all second order effects between AM and EM in three steps:

1. *Initialization*: Introducing temporaries
2. *Assignment Motion*: Eliminating partially redundant assignments

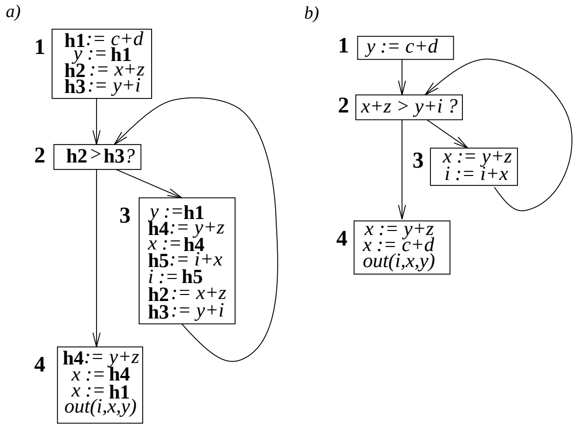

Figure 6: The Separate Effects of EM and AM

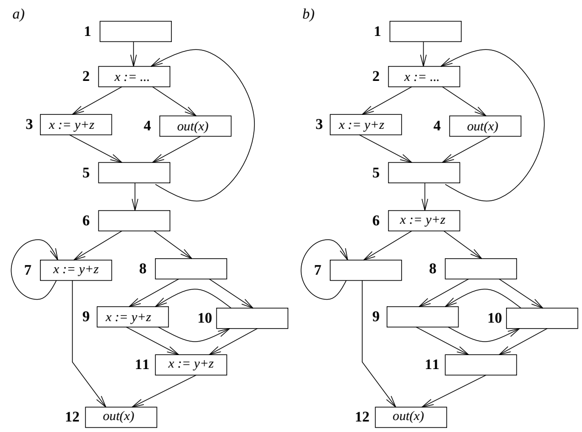

Figure 7: Illustrating the Treatment of Loops

3. *Final Flush*: Eliminating unnecessary initializations of temporaries

The first step replaces every assignment $x := t$ by the sequence of $\mathbf{h}_t := t$; $x := \mathbf{h}_t$, where $\mathbf{h}_t$ is a new temporary that is associated with the term t.[2] This enhances AM to cover EM.

The second step moves all assignments as far as possible in the opposite direction of the control flow to their 'earliest' safe program points. This maximizes the potential of redundant assignments, which are subsequently eliminated.

The third step is a transformation in the spirit of the *lazy code motion (lcm)* transformation of [16, 18]. Essentially, it flushes all assignments to temporaries introduced in the first step that do not contribute to the elimination of a partial redundancy.

Our algorithm works for arbitrary control flow structures and elegantly solves the problem of distinguishing between profitable code motion across loop structures and fatal code motion into loops (cf. [2]). This is illustrated in the example of Figure 7(a), which contains two loop constructs, one of which is even irreducible.

Figure 7(b) shows that our algorithm moves the partially redundant assignments of node **7**, **9**, and **11** to node **6**. Thereby, the assignment of node **11** is moved across the irreducible loop construct. It is worth noting that the assignment in node **6** is still partially redundant. However, the elimination of this partially redundant assignment would require to move $x := y + z$ into the first loop, which would dramatically impair some program executions.

1.4 Related Work

In contrast to EM, which has thoroughly been studied in program optimization (cf. [2, 4, 8, 9, 10, 11, 16, 18, 20]), AM has yet been investigated rarely. The most relevant papers on this subject are by Dhamdhere [5, 6, 7]. Most similar to the assignment motion step of our algorithm is the algorithm of [7], where an extension of EM to AM is presented. In contrast to our approach, however, Dhamdhere's algorithm heuristically restricts assignment hoistings to 'immediately profitable' ones, i.e., to hoistings which eliminate a partially redundant assignment. This restriction prohibits optimal transformation results. E.g., the partially redundant assignment $x := y + z$ at node **4** in the simple example of Figure 8 remains in the program, as the blocking preceding assignment $a := x + y$ cannot 'profitably' be hoisted. In contrast, after suspending the blockade of $x := y + z$ by hoisting the assignment of $a := x + y$ to node **2** and **3**, which is displayed in Figure 9(a), our algorithm yields the result of Figure 9(b).

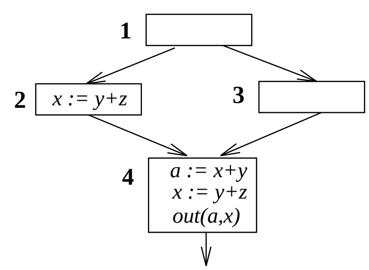

Figure 8: No Effect of 'Restricted' AM

In [5, 6] Dhamdhere presents an application of assignment hoisting and sinking techniques to *register assignment*. This, however, does not contribute to the general problem of eliminating partially redundant assignments.

Structure of the Paper

After the preliminary Section 2, Section 3 presents the central notions of our approach and establishes the es-

[2] A similar step was suggested in [3] in order to separate the effects of moving stores from the effects of moving expressions.

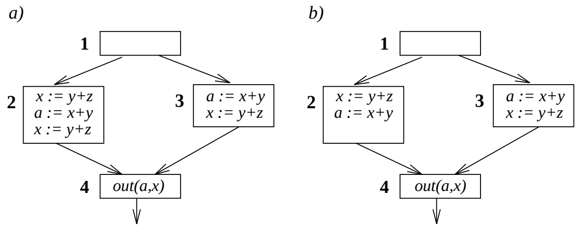

Figure 9: The Effect of 'Unrestricted' AM

sential features of eliminating partially redundant assignments and expressions. Subsequently, Section 4 sketches the second order effects between EM and AM, and develops our algorithm, which is illustrated in full detail by means of the running example of Figure 4. Additionally, a complexity estimation of the algorithm is given. Section 5 presents the optimality result of our algorithm, and Section 6 discusses pragmatic aspects of its implementation. Finally, Section 7 contains our conclusions, and the Appendix presents the equation systems specifying the data flow analyses of our algorithm.

2 Preliminaries

We consider *variables* $v \in \mathbf{V}$, *terms* $t \in \mathbf{T}$, and *directed flow graphs* $G=(N,E,\mathbf{s},\mathbf{e})$ with node set N and edge set E. Nodes $n \in N$ represent *basic blocks* of instructions, edges $(m,n) \in E$ the nondeterministic branching structure of G, and $\mathbf{s}$ and $\mathbf{e}$ the unique *start node* and *end node* of G, which are assumed not to possess any predecessors and successors, respectively. Additionally, $succ(n)=_{df} \{\, m \mid (n,m) \in E \,\}$ and $pred(n)=_{df} \{\, m \mid (m,n) \in E \,\}$ denote the set of all immediate successors and predecessors of a node n, respectively. A path p in G is a sequence of nodes $(n_1,\ldots,n_k)$, where $\forall\, 1 \le i < k.\ n_{i+1} \in succ(n_i)$, and $\mathbf{P}[m,n]$ denotes the set of all finite paths from m to n. Every node $n \in N$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$.

Instructions are *assignment statements* of the form $v := t$ including the *empty statement* skip,[3] *write statements* of the form $out(\ldots)$, and Boolean expressions representing the branching condition of branch nodes, i.e., of nodes having more than one successor.

An *assignment (expression) pattern* α (ε) is a string of the form $v := t$ (t). As usual (cf. [20]), we assume that t contains at most one operator symbol. The reasonability of this assumption, which simplifies the presentation of our algorithm, is discussed in more detail in Section 6. Moreover, we assume that every expression pattern ε is non-trivial, i.e. contains exactly one operator symbol, and is associated with a unique temporary $\mathbf{h}_\varepsilon$, which is used for storing the value of ε in order to eliminate partially redundant occurrences of ε.

Given a program G, $\mathcal{EP}$ denotes the set of all expression patterns occurring in G, and $\mathcal{AP}$ the set of all assignment patterns which is enriched by the set of all assignment patterns of the form $\mathbf{h}_\varepsilon := \varepsilon$ and $v := \mathbf{h}_\varepsilon$ for all $\varepsilon \in \mathcal{EP}$ occurring on the right hand side of an assignment with left hand side variable v.

Moreover, for all programs G, paths $p \in \mathbf{P}[\mathbf{s},\mathbf{e}]$, and patterns $\phi \in \mathcal{AP} \cup \mathcal{EP}$, let $\phi\#(p_G)$ denote the number of occurrences of the pattern ϕ on p in G.

2.1 Critical Edges

It is well-known that code motion transformations can be blocked by *critical edges* of a flow graph, i.e., by edges leading from a node with more than one successor to a node with more than one predecessor (cf. [4, 9, 16, 17, 18, 19]).

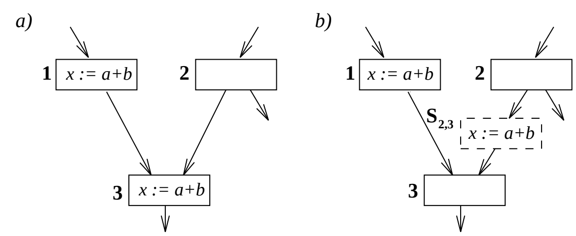

Figure 10: Critical Edges

In Figure 10(a) the assignment $x := a+b$ at node $\mathbf{3}$ is partially redundant with respect to the assignment at node $\mathbf{1}$. However, this partially redundant assignment cannot safely be eliminated by moving it to its predecessors, because this may introduce a new assignment on a path leaving node $\mathbf{2}$ on the right branch. On the other hand, it can safely be eliminated after inserting a synthetic node $\mathbf{S}_{2,3}$ in the critical edge $(\mathbf{2},\mathbf{3})$, as illustrated in Figure 10(b).

In the following, we therefore restrict our attention to programs where every critical edge has been split by inserting a synthetic node.

3 Assignment Motion

Conceptually, assignment motion stands for any sequence of

- *assignment hoistings* and
- *redundant assignment eliminations*

as formally defined below.

[3] In particular, assignments of the form $x := x$ are identified with skip.

Definition 3.1 (Assignment Hoisting)
Let $\alpha \equiv x := t$ be an assignment pattern. An assignment hoisting *for α is a program transformation that*

- *eliminates some occurrences of α,*
- *inserts instances of α at the entry or exit of some basic blocks from which a basic block with an eliminated occurrence of α is reachable.*

In order to be admissible, assignment hoistings must be semantics preserving. Obviously, the hoisting of an assignment pattern $\alpha \equiv x := t$ is *blocked* by an instruction that

- modifies an operand of t or
- uses or modifies the variable x.

Thus, we define:

Definition 3.2 (Admissible Assign. Hoisting)
An assignment hoisting for α is admissible*, iff it satisfies the following two conditions:*

1. *The removed assignments are* substituted*, i.e., every program path leading from* $\mathbf{s}$ *to an elimination site of α contains an insertion site of α that is not followed by an instruction which blocks α.*
2. *The inserted assignments are* justified*, i.e., every program path leading from an insertion site of α to* $\mathbf{e}$ *contains an elimination site of α which is not preceded by an instruction that blocks α.*

Definition 3.3 (Assignment Elimination)
An assignment elimination *for α is a program transformation that eliminates some original occurrences of α in the argument program.*

Like the hoisting also the elimination of assignments must be admissible. This leads to the notion of redundant assignments. An occurrence of an assignment pattern $\alpha \equiv x := t$ in a basic block n is *redundant*, if every path from $\mathbf{s}$ to n goes through a basic block m containing another occurrence of α, and neither x nor an operand of t is modified between the two occurrences.

Definition 3.4 (Redundant Assign. Elim.)
A redundant assignment elimination *for an assignment pattern α is an assignment elimination for α, where some redundant occurrences of α are eliminated.*

It is worth noting that admissible assignment hoistings and redundant assignment eliminations preserve the program semantics. This is in contrast to assignment eliminations based on dead code elimination (cf. [12, 19]). In fact, eliminating 'dead' assignments may change the semantics of the program by reducing the potential of run-time errors.[4]

Definition 3.5 (Assignment Motion)
Assignment motion *AM is an arbitrary sequence of admissible assignment hoistings and redundant assignment eliminations.*

3.1 The Universe

In the following we will write $G \vdash_{EM} G'$ or $G \vdash_{AM} G'$ if the flow graph G' results from G from applying an expression motion transformation (see Figure 1 for illustration, and e.g. [18] for details), or from applying an assignment motion transformation, i.e., from applying an admissible assignment hoisting or redundant assignment elimination, respectively (see Figure 2 for illustration). Moreover, let $\vdash =_{df} \vdash_{EM} \cup \vdash_{AM}$.

For a given flow graph G we denote the *universe* of programs resulting from an arbitrary interleaving of expression and assignment motions by

$$\mathcal{G} =_{df} \{\, G' \mid G \vdash^* G' \,\}$$

Important for our optimality result is the following property of $\vdash$, which holds due to identifying assignments of the form $\mathbf{h}_\varepsilon := \mathbf{h}_\varepsilon$ by skip.

Lemma 3.6 (Local Confluence of $\vdash$)
$\vdash$ is locally confluent*, i.e. if $G_1 \vdash G_2$ and $G_1 \vdash G_3$ for some program $G_1 \in \mathcal{G}$ then there is a program $G_4 \in \mathcal{G}$ such that $G_2 \vdash^* G_4$ and $G_3 \vdash^* G_4$.*

3.2 Optimality

Central for comparing the quality of different programs of the universe $\mathcal{G}$ are standards of comparison that are typically given in terms of preorders, i.e. relations that are reflexive and transitive but not antisymmetric. It is worth noting that due to the absence of antisymmetry in a preorder $\precsim$ also programs that are quite different at first sight are 'similar' in terms of this order, where similarity is given by its core $\sim =_{df} \precsim \cap \succsim$.

A preorder $\precsim$ on $\mathcal{G}$ induces two notions of optimality for a program G' of $\mathcal{G}$. First, G' is *$\precsim$-optimal*, if it is better than any other program of $\mathcal{G}$, and it is *relatively $\precsim$-optimal*, if it cannot be improved further by means of admissible expression and assignment motion transformations.

Definition 3.7 (Optimality & Relative Opt.)
A program $G' \in \mathcal{G}$ is

[4] Think e.g. of an overflow or a division by zero caused by the evaluation of the right hand side term of an eliminated assignment.

1. relatively $\underset{\sim}{\sqsubset}$-optimal, *iff*
$\forall\, G'' \in \mathcal{G}.\; G' \vdash^* G'' \Rightarrow G'' \underset{\sim}{\sqsubset} G'$.

2. $\underset{\sim}{\sqsubset}$-optimal, *iff*
$\forall\, G'' \in \mathcal{G}.\; G'' \underset{\sim}{\sqsubset} G'$.

For our algorithm we are considering three preorders which reflect the different optimization goals it covers:

- **Primary Goal:** Avoiding unnecessary recomputations of expressions.
- **Secondary Goals:**
  - Avoiding unnecessary reexecutions of assignments.
  - Avoiding unnecessary assignments to and unnecessarily long lifetimes of temporaries.

The corresponding preorders are given below:

Definition 3.8 (Optimization Preorders)
Let $G', G'' \in \mathcal{G}$. Then we define:

1. *G' is* expression-better *than G'', in signs $G'' \underset{\sim}{\sqsubset}_{exp} G'$, if and only if*
$\forall\, p \in \mathbf{P}[\mathbf{s},\mathbf{e}]\; \forall\, \varepsilon \in \mathcal{EP}.\; \varepsilon\#(p_{G'}) \leq \varepsilon\#(p_{G''})$

2. *G' is* assignment-better *than G'', in signs $G'' \underset{\sim}{\sqsubset}_{ass} G'$, if and only if*
$\forall\, p \in \mathbf{P}[\mathbf{s},\mathbf{e}]\; \forall\, \alpha \in \mathcal{AP}.\; \alpha\#(p_{G'}) \leq \alpha\#(p_{G''})$

3. *G' is* temporary-better *than G'', in signs $G'' \underset{\sim}{\sqsubset}_{tmp} G'$, if and only if any lifetime range of a temporay $\mathbf{h}_\varepsilon$ used in G' is contained in a corresponding lifetime range of G''.*[5]

4 The Algorithm

In this section we present our algorithm for the redundancy- and temporary-optimal elimination of partially redundant assignments and expressions. We first give an overview of the algorithm, and subsequently describe the relevant steps in more detail illustrating them by means of the running example of Figure 4, which we recall below.

4.1 Overview

The global algorithm consists of three main procedures:

- a procedure *rae* for the elimination of redundant assignments,
- a procedure *aht* for assignment hoistings, and

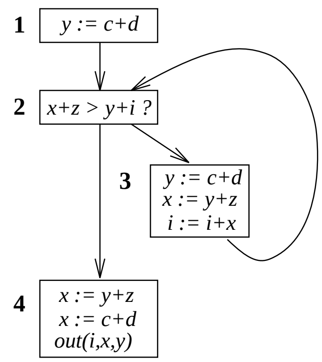

Figure 11: The Running Example

- a procedure *lcm* for the elimination of unnecessary assignments to temporaries

which are organized in three phases:

1. *Initialization*: Introducing temporaries
2. *Assignment Motion*: Eliminating partially redundant assignments
3. *Final Flush*: Eliminating unnecessary initializations of temporaries

The *initialization* phase decomposes every assignment occurring in the program in an initialization of a uniquely determined temporary and a use of this temporary in the original assignment. As mentioned already this simple transformation enhances AM to cover EM. The *assignment motion* phase, subsequently, which is the main phase of our global algorithm, is composed of the *rae*- and *aht*-procedure, which are applied until the program stabilizes. The concluding *final flush* phase is an application of essentially the *lcm*-procedure of [18], and eliminates all unnecessary assignments to temporaries introduced in the initialization phase.

Convention: In the following we denote the program that results from applying our global algorithm to G by $G_{GlobAlg}$, and the intermediate programs resulting from the initialization and the assignment motion phase by G_{Init} and G_{AssMot}, respectively.

4.2 The Initialization Phase

The initialization phase replaces every assignment $x := t$ by the assignment sequence $\mathbf{h}_t := t;\; x := \mathbf{h}_t$, where $\mathbf{h}_t$ is the unique temporary associated with term t. Obviously, this transformation is an admissible expression motion. Moreover, it enhances AM to cover EM. We have:[6]

Lemma 4.1 (Initialization Phase Lemma)

[5] *Lifetime ranges* are paths between an assignment to $\mathbf{h}_\varepsilon$ and its first subsequent use. See [18] for details.

[6] Remember that assignments of the form $\mathbf{h}_\varepsilon := \mathbf{h}_\varepsilon$ are identified with skip.

1. $G_{Init} \in \mathcal{G}$

2. Let $G', G'' \in \mathcal{G}$ *such that* $G_{Init} \vdash^* G' \vdash_{EM} G''$. *Then we have:* $G' \vdash_{AM} G''$.

Applied to our running example the initialization phase comes up with the program of Figure 12:

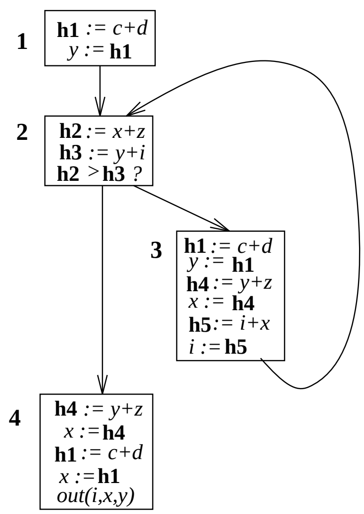

Figure 12: The Effect of the Initialization Phase

4.3 The Assignment Motion Phase

The assignment motion phase is concerned with the elimination of partially redundant assignments in the program resulting from the initialization phase. Conceptually, this is achieved in two steps: (1) Moving the assignments as far as possible in the opposite direction of the control flow to their 'earliest' safe execution points, and (2) eliminating redundant occurrences of assignments. Though this is very closely in spirit to EM, AM is more intricate, because it induces second order effects. In fact, both the hoisting and the elimination of assignments usually enable the hoisting and elimination of occurrences of other assignment patterns. This is in contrast to the hoisting and replacement of expressions, which is free of interdependencies between different expression patterns. In AM, however, we are faced with the following mutual dependencies, which have been illustrated already by the motivating examples of Section 1:

- Hoisting-Elimination Effects
- Hoisting-Hoisting Effects
- Elimination-Hoisting Effects
- Elimination-Elimination Effects

The assignment motion phase overcomes all of these second order effects by means of exhaustive hoisting and elimination steps: The procedures *rae* and *aht* are applied until the program stabilizes. In the following we discuss these procedures in more detail.

4.3.1 Redundant Assignment Elimination

The elimination of redundant assignments is based on a forwards directed bit-vector data flow analysis [1, 13], which is specified in Table 2, where $\textbf{N-REDUNDANT}_\iota(\alpha)$ (or $\textbf{X-REDUNDANT}_\iota(\alpha)$) means that assignment pattern α is redundant at the entry (or exit) of instruction ι. After having computed the greatest solution of the equation system specified in Table 2, the corresponding program transformation is very simple:

The Elimination Step:
Process every basic block by successively eliminating all assignments which are redundant immediately before them, i.e., redundant at their entry.

4.3.2 Assignment Hoisting

The program analysis of this step determines how far an assignment can be hoisted from its original location to earlier program points, while maintaining the program semantics. In fact, this analysis is dual to the delayability analysis of the partial dead code elimination algorithm of [19], which was designed to determine how far an assignment can be sunk from its original location to later program points, while maintaining the program semantics. Table 1 presents the hoistability analysis in a bit-vector format, where each bit corresponds to an assignment pattern occurring in the program. Here, $\textbf{N-HOISTABLE}_n$ and $\textbf{X-HOISTABLE}_n$ intuitively mean that some hoisting candidates of α can be moved to the entry or the exit of basic block n, respectively, where a *hoisting candidate* is an occurrence of an assignment $x := t$ inside a basic block which is not blocked, i.e., neither preceded by a modification of an operand of t nor by a modification or a usage of x. This is illustrated in Figure 13. Note that in the set of occurrences of an assignment pattern in a basic block at most the first one is a candidate for global hoisting, because every occurrence is blocked at least by the preceding one.

The greatest solution of the equation system displayed in Table 1 characterizes the program points, where instances of the assignment pattern α must be inserted, by means of the insertion predicates **N-INSERT** and **X-INSERT**. The subsequent program transformation is again very simple, because it can easily be shown that all assignment patterns that must be inserted at a particular program point are independent and can therefore be placed in an arbitrary

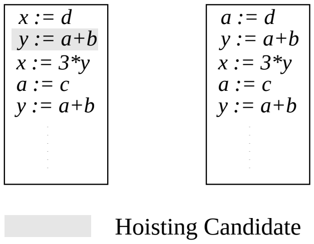

Figure 13: Hoisting Candidates of "$y := a + b$"

order:

The Insertion Step:
Process every basic block by successively inserting instances of every assignment pattern α at the entry (or exit) of n if $\textbf{N-INSERT}_n(\alpha)$ (or $\textbf{X-INSERT}_n(\alpha)$) is satisfied, and simultaneously remove all hoisting candidates.[7]

It is easy to see that the program resulting from the assignment motion phase satisfies:

Lemma 4.2 (AM-Phase Lemma)

1. $G_{AssMot} \in \mathcal{G}$
2. G_{AssMot} *is relatively assignment-optimal in* $\mathcal{G}$, *i.e. relatively* $\precsim_{ass}$*-optimal*

Moreover, as an immediate consequence of Lemma 4.1(2), Lemma 4.2(2), and the exhaustive introduction of temporaries during the initialization phase, we have:

Corollary 4.3 (AM-Phase Corollary)
G_{AssMot} *is relatively expression-optimal in* $\mathcal{G}$, *i.e. relatively* $\precsim_{exp}$*-optimal.*

Applied to the program of Figure 12 the assignment motion phase terminates with the program of Figure 14:

4.4 The Final Flush Phase

Intuitively, the final flush phase moves the occurrences of all assignment patterns of the form $\mathbf{h}_\varepsilon := \varepsilon$ to their 'latest' safe execution points, and eliminates all occurrences whose left hand side is used at most once immediately after their occurrence. The following lemma states three important properties of this transformation. First, it stays within the program universe $\mathcal{G}$. Second, it guarantees relative temporary-optimality, which can be proved along the lines of the 'lifetime optimality' Theorem of [18], and third it preserves the optimization of the assignment motion phase.

Lemma 4.4 (Final Flush Phase Lemma)

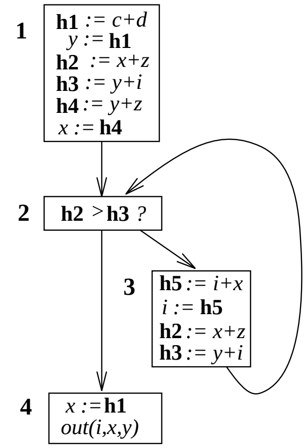

Figure 14: The Effect of the AM-Phase

1. $G_{GlobAlg} \in \mathcal{G}$
2. $G_{GlobAlg}$ *is relatively temporary-optimal, i.e. relatively* $\precsim_{tmp}$*-optimal*
3. (a) $G_{GlobAlg}$ *is relatively assignment-optimal, i.e. relatively* $\precsim_{ass}$*-optimal*
   (b) $G_{GlobAlg} \sim_{exp} G_{AssMot}$

Essentially, in the final program assignments to temporaries are only present, if they are justified by the elimination of a partial redundancy. This guaranty is a central feature of the *lazy code motion* procedure of [16, 18], which distinguishes it from all previous algorithms for EM.

This property carries over to the final flush phase of our algorithm, which is realized by a variant of the *lazy code motion* procedure of [18], a straightforward adaption of the original procedure to our current situation. Similar to the original procedure it is based on two uni-directional bitvector data flow analyses computing *delayable* program points, where an initialization is *usable*[8] (cf. Table 3).

Applying this procedure to the program of Figure 14 yields the desired program of Figure 15:

4.5 Complexity of the Algorithm

The termination of the initialization phase is obvious as it only replaces the original assignments by a sequence of two assignments. Also the termination of the final flush phase is obvious as it consists of the single application of the *lcm*-procedure. The assignment motion phase, finally, terminates as soon as both steps it is composed of, namely the totally redundant assignment elimination and the assignment hoisting, leave

[7]Due to edge splitting there are no insertions at the entry of join nodes.

[8]Essentially, an initialization is *usable*, if it is used twice on some program continuation. This analysis replaces the less intuitive *isolation* analysis of [16, 18].

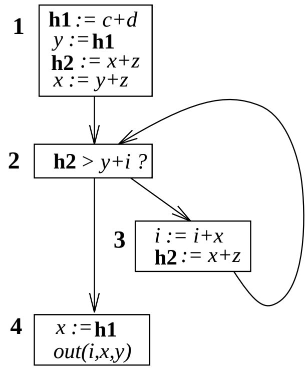

Figure 15: The Effect of the Final Flush

the program invariant. In the case of totally redundant assignment elimination this simply means that no further assignments are eliminated, and in the case of assignment hoisting this holds, if every basic block n satisfies $\textbf{N-INSERT}_n = \textbf{LOC-HOISTABLE}_n$ and $\textbf{X-INSERT}_n = \mathit{false}$.

The complexity of the initialization phase is trivially linear in the program size. For structured programs the final flush phase is almost linear in the program size, since the efficient bitvector techniques of [14, 15, 21] become applicable for solving the uni-directional data flow analyses the *lcm*-transformation is based on. In the unstructured case, it is quadratic. The same estimation holds for a single application of the assignment hoisting and elimination procedure of the assignment motion phase. As for the partial dead code elimination algorithm of [19], the number of applications of these procedures is at most quadratic in the program size, but linear for realistic programs. Thus, the worst case time complexity of the global algorithm ranges from second order for realistic structured programs to fourth order for the completely unrestricted worst case.

5 Results

As already stated in Lemma 4.4(1), we have:

Theorem 5.1 (Correctness) $G_{GlobAlg} \in \mathcal{G}$

Our main results, however, state that the primary goal, namely avoiding unnecessary recomputations of expressions, is compatible with a (in a sense best possible) treatment of the secondary goals.

Theorem 5.2 (Expression-Optimality)
$G_{GlobAlg}$ *is expression-optimal in* $\mathcal{G}$*, i.e., it requires at most as many expression evaluations at run-time than any other program that can be obtained via EM and AM transformations.*

Proof: The relative expression-optimality of $G_{GlobAlg}$ is already guaranteed by Lemma 4.4(3b) and Corollary 4.3. Moreover, according to Lemma 4.4(3b), it suffices to prove the 'full' expression-optimality of G_{AssMot} to complete the proof. This can be done along the lines of [19]. The point is that $\vdash$ contains $\underset{\sim}{\sqsubset}_{exp}$-improving transformations only, i.e.

$$G' \vdash G'' \Rightarrow G' \underset{\sim}{\sqsubset}_{exp} G''$$

Together with the local confluence of $\vdash$ (cp. Lemma 3.6), we easily conclude that different relatively expression-optimal programs in $\mathcal{G}$ are equivalent up to $\sim_{exp}$. Now, the fact that every program in $\mathcal{G}$ is dominated (wrt $\underset{\sim}{\sqsubset}_{exp}$) by a relatively expression-optimal one completes the proof. □

Besides expression-optimality, costs for assignments are a major concern, in particular, as EM introduces assignments to temporaries. The following two theorems establish the corresponding relative optimality of our algorithm. As it will be argued below, this is the best we can hope for. First we have by means of Lemma 4.2(2):

Theorem 5.3 (Relative Assignment-Optim.)
$G_{GlobAlg}$ *is relatively assignment-optimal in* $\mathcal{G}$*, i.e., it is impossible to decrease the number of assignments required by* $G_{GlobAlg}$ *at run-time by means of EM and AM transformations.*

Moreover, as guaranteed by means of Lemma 4.4(2), the *final flush* phase of our algorithm eliminates all unnecessary assignments to temporaries:

Theorem 5.4 (Relative Temporary-Optim.)
$G_{GlobAlg}$ *is relatively temporary-optimal in* $\mathcal{G}$*, i.e. it is impossible to decrease the number of assignments to temporaries or the length of temporary lifetimes in* $G_{GlobAlg}$ *by a corresponding assignment sinking.*

The following example demonstrates that expression-optimality, and relative assignment- and temporary-optimality as achieved by our algorithm is the best we can hope for: The program of Figure 16 can be transformed into two expression-optimal programs (see Figure 17(a) and (b)) that are *incomparable* in both the number of assignment executions and the lifetime ranges of temporaries. In Figure 17(a) there are four assignments on the path $(\mathbf{1}, \mathbf{3}, \mathbf{4}, \mathbf{6})$ as well as on the path $(\mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{6})$. In Figure 17(b) this ratio amounts to three by five, and both solutions cannot further be improved wrt $\underset{\sim}{\sqsubset}_{ass}$. A similar investigation reveals that also the lifetimes of temporaries are incomparable in these solution programs.

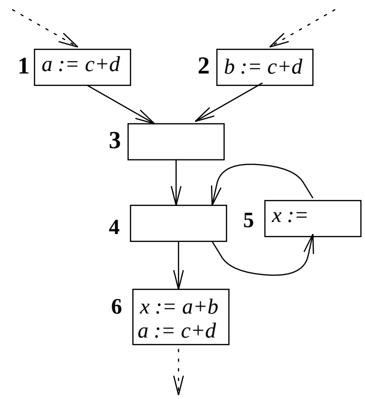

Figure 16: ‘Full’ Assignment- and Temporary-Optimality is Impossible

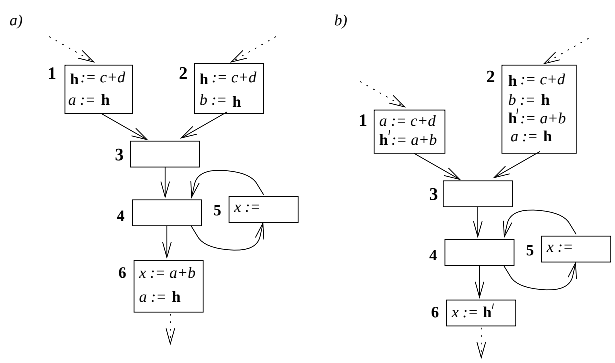

Figure 17: Incomparable Solutions

6 Pragmatics

The economical use of temporaries as guaranteed by the final flush phase of our algorithm does not only improve on previous algorithms, it also justifies a technical assumption commonly made by EM-algorithms: argument programs are given in *3-address form*, i.e., the right hand side terms of assignments contain at most one operator symbol. Though this assumption is uncritical in that all assignments can be canonically decomposed into sequences of assignments of this form along the inductive structure of terms (cf. [18]) or a special naming discipline [2], this decomposition may block subsequent transformations. This is illustrated in Figure 18(a) displaying a program whose 3-address decomposition is shown in Figure 18(b).

Whereas EM results in the program of Figure 19(a) when applied to the program of Figure 18(a), it gets stuck with the program of Figure 19(b) in the ‘3-address case’, where the expression $t + c$ is *not* loop invariant, and therefore cannot be removed from the loop for safety reasons.

Usually, this problem is dealt with by interleaving EM with *copy propagation (CP)* (cf. [9]). This results in the program of Figure 20(a). In contrast, the final flush phase of our algorithm guarantees the result dis-

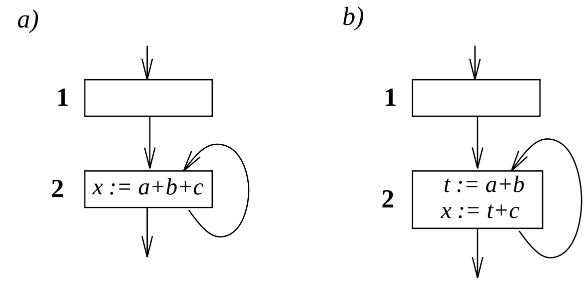

Figure 18: Complex Expressions vs. 3-Address Code

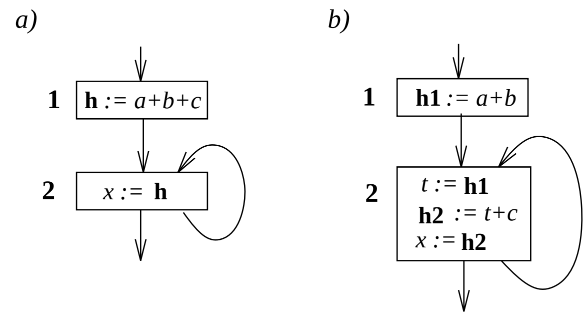

Figure 19: The Effect of Standard EM

played in Figure 20(b), which is better than the programs given in Figures 19(a) and 20(a). In fact, the final flush phase justifies the 3-address assumption in that it guarantees as least as good results after the corresponding transformation.

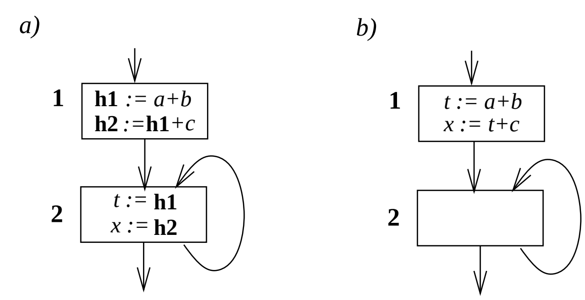

Figure 20: Comparing the Effects of EM & CP and Uniform EM & AM

7 Conclusions

We have presented a new, modular and aggressive algorithm for the expression-optimal elimination of partially redundant expressions and assignments in a program. This algorithm is unique in that it for the first time captures all second order effects due to the mutual dependencies between expression and assignment hoisting, and the elimination of totally redundant expressions and assignments.

The complexity of this algorithm ranges from second order for realistic structured programs to fourth order for the completely unrestricted worst case. Thus, as other aggressive methods, our algorithm should typically be employed for structured programs or for the optimization of time-critical sections of code of moderate size. Alternatively, one may limit the number of allowed hoisting and elimination steps heuristically.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.

[2] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'94*, volume *29*,6 of *ACM SIGPLAN Notices*, pages 159 – 170, Orlando, FL, June 1994.

[3] R. Cytron, A. Lowry, and F. K. Zadeck. Code motion of control structures in high-level languages. In *Conf. Record of the 13th ACM Symposium on the Principles of Programming Languages*, pages 70 – 85, St. Petersburg, Florida, January 1986.

[4] D. M. Dhamdhere. A fast algorithm for code movement optimization. *ACM SIGPLAN Notices*, 23(10):172 – 180, 1988.

[5] D. M. Dhamdhere. Register assignment using code placement techniques. *Journal of Computer Languages*, 13(2):75 – 93, 1988.

[6] D. M. Dhamdhere. A usually linear algorithm for register assignment using edge placement of load and store instructions. *Journal of Computer Languages*, 15(2):83 – 94, 1990.

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

[8] D. M. Dhamdhere and H. Patil. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Transactions on Programming Languages and Systems*, 15(2):312 – 336, April 1993.

[9] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'92*, volume *27*,7 of *ACM SIGPLAN Notices*, pages 212 – 223, San Francisco, CA, June 1992.

[10] K.-H. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635 – 640, 1988. Technical Correspondence.

[11] K.-H. Drechsler and M. P. Stadel. A variation of Knoop, Rüthing and Steffen's lazy code motion. *ACM SIGPLAN Notices*, 28(5):29 – 38, 1993.

[12] L. Feigen, D. Klappholz, R. Casazza, and X. Xue. The revival transformation. In *Conf. Record of the 21nd ACM Symposium on the Principles of Programming Languages*, pages 421 – 434, Portland, Oregon, January 1994.

[13] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.

[14] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158 – 171, 1976.

[15] K. Kennedy. Node listings applied to data flow analysis. In *Conf. Record of the 2nd ACM Symposium on the Principles of Programming Languages*, pages 10 – 21, Palo Alto, CA, 1975.

[16] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'92*, volume *27*,7 of *ACM SIGPLAN Notices*, pages 224 – 234, San Francisco, CA, June 1992.

[17] J. Knoop, O. Rüthing, and B. Steffen. Lazy strength reduction. *Journal of Programming Languages*, 1(1):71–91, 1993.

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

[19] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'94*, volume *29*,6 of *ACM SIGPLAN Notices*, pages 147 – 158, Orlando, FL, June 1994.

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

[21] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690 – 715, 1979.

The Assignment Motion Phase: a) Hoisting Assignments

Local Predicates: (Let $\alpha \equiv v := t \in \mathcal{AP}$)

- $\mathbf{LOC\text{-}HOISTABLE}_n(\alpha)$: There is a hoisting candidate of α in n.
- $\mathbf{LOC\text{-}BLOCKED}_n(\alpha)$: The hoisting of α is blocked by some instruction of n.

The Hoistability Analysis:

$$\mathbf{N\text{-}HOISTABLE}_n = \mathbf{LOC\text{-}HOISTABLE}_n + \mathbf{X\text{-}HOISTABLE}_n * \overline{\mathbf{LOC\text{-}BLOCKED}_n}$$

$$\mathbf{X\text{-}HOISTABLE}_n = \begin{cases} \mathit{false} & \mathit{if} \quad n = \mathbf{e} \\ \prod_{m \in succ(n)} \mathbf{N\text{-}HOISTABLE}_m & \mathit{otherwise} \end{cases}$$

Insertion Points:[a]

$$\mathbf{N\text{-}INSERT}_n =_{df} \mathbf{N\text{-}HOISTABLE}^{\star}_n * \sum_{m \in pred(n)} \overline{\mathbf{X\text{-}HOISTABLE}^{\star}_m}$$

$$\mathbf{X\text{-}INSERT}_n =_{df} \mathbf{X\text{-}HOISTABLE}^{\star}_n * \mathbf{LOC\text{-}BLOCKED}_n$$

Table 1: Hoistability Analysis and Insertion Points

[a] $\mathbf{N\text{-}HOISTABLE}^{\star}$ and $\mathbf{X\text{-}HOISTABLE}^{\star}$ denote the greatest solution of the equation system for hoistability.

The Assignment Motion Phase: b) Eliminating Redundant Assignments

Local Predicates: (Let $\alpha \equiv v := t \in \mathcal{AP}$ such that v is not an operand of t. Let s denote the first instruction of $\mathbf{s}$.)

- $\mathbf{EXECUTED}_\iota(\alpha)$: Instruction ι is an assignment of the pattern α.
- $\mathbf{ASS\text{-}TRANSP}_\iota(\alpha)$: Instruction ι is transparent for α, i.e., neither v nor any operand of t is modified by ι.

The Redundancy Analysis:[a]

$$\mathbf{N\text{-}REDUNDANT}_\iota = \begin{cases} \mathit{false} & \text{if } \iota = s \\ \prod_{\hat{\iota} \in pred(\iota)} \mathbf{X\text{-}REDUNDANT}_{\hat{\iota}} & \text{otherwise} \end{cases}$$

$$\mathbf{X\text{-}REDUNDANT}_\iota = \mathbf{EXECUTED}_\iota + \mathbf{ASS\text{-}TRANSP}_\iota * \mathbf{N\text{-}REDUNDANT}_\iota$$

Table 2: Redundant Assignment Analysis

[a] The analysis is employed at the instruction level. This, however, is only for the ease of presentation. In fact, it can straightforward be modified to work on basic blocks.

The Final Flush Phase

Local Predicates: (Let $\mathbf{h}_\varepsilon$ be the temporary for some $\varepsilon \in \mathcal{EP}$. Let s denote the first instruction of $\mathbf{s}$.)

- $\mathbf{IS\text{-}INST}_\iota(\mathbf{h}_\varepsilon)$: Instruction ι is an instance of $\mathbf{h}_\varepsilon := \varepsilon$.
- $\mathbf{USED}_\iota(\mathbf{h}_\varepsilon)$: Instruction ι uses $\mathbf{h}_\varepsilon$.
- $\mathbf{BLOCKED}_\iota(\mathbf{h}_\varepsilon)$: Instruction ι blocks $\mathbf{h}_\varepsilon := \varepsilon$.

The Delayability Analysis:

$$\mathbf{N\text{-}DELAYABLE}_\iota = \begin{cases} \mathit{false} & \mathit{if} \quad \iota = s \\ \prod_{\bar{\iota} \in pred(\iota)} \mathbf{X\text{-}DELAYABLE}_{\bar{\iota}} & \mathit{otherwise} \end{cases}$$

$$\mathbf{X\text{-}DELAYABLE}_\iota = \mathbf{IS\text{-}INST}_\iota + \mathbf{N\text{-}DELAYABLE}_\iota * \overline{\mathbf{USED}_\iota} * \overline{\mathbf{BLOCKED}_\iota}$$

The Usability Analysis:

$$\mathbf{N\text{-}USABLE}_\iota = \mathbf{USED}_\iota + \overline{\mathbf{IS\text{-}INST}_\iota} * \mathbf{X\text{-}USABLE}_\iota$$

$$\mathbf{X\text{-}USABLE}_\iota = \sum_{\bar{\iota} \in succ(\iota)} \mathbf{N\text{-}USABLE}_{\bar{\iota}}$$

Computing Latestness: (No data flow analysis!)[a]

$$\mathbf{N\text{-}LATEST}_\iota =_{df} \mathbf{N\text{-}DELAYABLE}^\star_\iota * (\mathbf{USED}_\iota + \mathbf{BLOCKED}_\iota)$$

$$\mathbf{X\text{-}LATEST}_\iota =_{df} \mathbf{X\text{-}DELAYABLE}^\star_\iota * \sum_{\bar{\iota} \in succ(\iota)} \overline{\mathbf{N\text{-}DELAYABLE}^\star_{\bar{\iota}}}$$

Initialization Points: Insert instance of α

$$\begin{aligned} \mathbf{N\text{-}INIT}_\iota &=_{df} \mathbf{N\text{-}LATEST}_\iota * \mathbf{X\text{-}USABLE}^\star_\iota \\ \mathbf{X\text{-}INIT}_\iota &=_{df} \mathbf{X\text{-}LATEST}_\iota \end{aligned}$$

Reconstruction Points: Reconstruct original usage of t instead of $\mathbf{h}$

$$\mathbf{RECONSTRUCT}_\iota =_{df} \mathbf{USED}_\iota * \mathbf{N\text{-}LATEST}_\iota * \overline{\mathbf{X\text{-}USABLE}^\star_\iota}$$

Table 3: Eliminating Unnecessary Assignments to Temporaries

[a] $\mathbf{N\text{-}DELAYABLE}^\star$ and $\mathbf{X\text{-}DELAYABLE}^\star$ ($\mathbf{N\text{-}USABLE}^\star$ and $\mathbf{X\text{-}USABLE}^\star$) denote the greatest solution of the equation system for delayability (usability).