Lazy Code Motion

Jens Knoop
Oliver Rüthing
Bernhard Steffen

Bericht Nr. 9208

April 1992

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Olshausenstraße 40 – 60
D–2300 Kiel 1

Lazy Code Motion[1]

Jens Knoop[+], Oliver Rüthing[*]
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
D–2300 Kiel 1

Bernhard Steffen
Lehrstuhl für Informatik II
Rheinisch-Westfälische Technische Hochschule Aachen
D–5100 Aachen

Bericht Nr. 9208
April 1992

[+] Part of the work was done, while the author was supported by the Deutsche Forschungsgemeinschaft grant La 426/9-2.

[*] The author is supported by the Deutsche Forschungsgemeinschaft grant La 426/11–1.

[1] This paper also appears in the Proceedings of the ACM SIGPLAN Conference on **P**rogramming **L**anguages **D**esign and **I**mplementation '92 (cp. [KRS1]).

Abstract

We present a bit-vector algorithm for the *optimal* and *economical* placement of computations within flow graphs, which is as *efficient* as standard uni-directional analyses. The point of our algorithm is the *decomposition* of the bi-directional structure of the known placement algorithms into a sequence of a backward and a forward analysis, which directly implies the efficiency result. Moreover, the new compositional structure opens the algorithm for modification: two further uni-directional analysis components exclude any unnecessary code motion. This *laziness* of our algorithm minimizes the register pressure, which has drastic effects on the run-time behaviour of the optimized programs in practice, where an economical use of registers is essential.

Keywords

Data flow analysis, program optimization, partial redundancy elimination, code motion, bit-vector data flow analyses.

Contents

1 Motivation

Code motion is a technique to improve the efficiency of a program by avoiding unnecessary recomputations of a value at run-time. This is achieved by replacing the original computations of a program by auxiliary variables (registers) that are initialized with the correct value at suitable program points. In order to preserve the semantics of the original program, code motion must additionally be *safe*, i.e. it must not introduce computations of new values on paths. In fact, under this requirement it is possible to obtain *computationally optimal* results, i.e. results where the number of computations on each program path cannot be reduced anymore by means of safe code motion (cf. Theorem 3.9). Central idea to obtain this optimality result is to place computations *as early as possible* in a program, while maintaining safety (cf. [Dh2, Dh3, KS2, MR1, St]). However, this strategy moves computations even if it is *unnecessary*, i.e. there is no run-time gain[2]. This causes superfluous *register pressure*, which is in fact a major problem in practice.

In this paper we present a *lazy* computationally optimal code motion algorithm, which is unique in that it

- is as *efficient* as standard uni-directional analyses and
- avoids any unnecessary register pressure.

The point of this algorithm is the *decomposition* of the bi-directional structure of the known placement algorithms (cf. "Related Work" below) into a sequence of a backward analysis and a forward analysis, which directly implies the efficiency result. Moreover, the new compositional structure allows to avoid any unnecessary code motion by modifying the standard computationally optimal computation points according to the following idea:

- Initialize "as late as possible" while maintaining computational optimality.

Together with the suppression of initializations, which are only going to be used at the insertion point itself, this characterizes our approach of *lazy code motion*. Figure 1 displays an example, which is complex enough to illustrate the various features of the new approach. It will be discussed in more details during the development in this paper. For now just note that our algorithm is unique in performing the optimization displayed in Figure 2, which is exceptional for the following reasons: it eliminates the partially redundant computations of "a+b" in node **10** and **16** by moving them to node **8** and **15**, but it does not touch the computations of $a+b$ in node **3** and **17** that cannot be moved with run-time gain. This confirms that computations are only moved when it is profitable.

Related Work

In 1979 Morel and Renvoise proposed a bit-vector algorithm for the suppression of partial redundancies [MR1]. The bi-directionality of their algorithm became model in the field of bit-vector based code motion (cf. [Ch, Dh1, Dh2, Dh3, DS, JD1, JD2, Mo, MR2, So]). Bi-directional algorithms, however, are in general conceptually and computationally more complex than uni-directional ones: e.g. in contrast to the uni-directional case, where reducible programs can be dealt with in $O(n\,log(n))$ time, where n characterizes the

[2] In [Dh3] unnecessary code motions are called *redundant*.

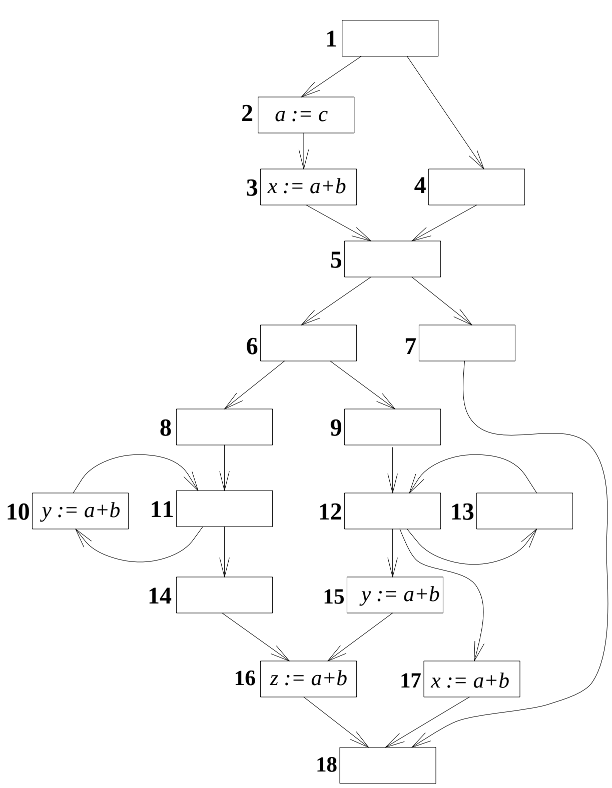

Figure 1: The Motivating Example

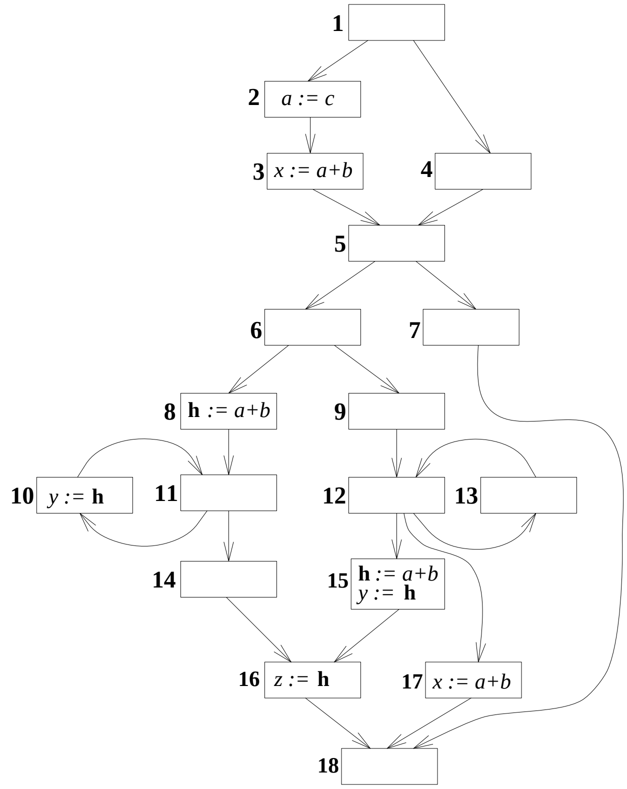

Figure 2: The Lazy Code Motion Transformation

size of the argument program (e.g. number of statements), the best known estimation for bi-directional analyses is $O(n^2)$ (cf. [Dh3]). The problem of unnecessary code motion is only addressed in [Ch, Dh2, Dh3], and these proposals are of heuristic nature: code is unnecessarily moved or redundancies remain in the program.

In contrast, our algorithm is composed of *uni-directional* analyses[3]. Thus the same estimations for the worst case time complexity apply as for uni-directional analyses (cf.[AU, GW, HU1, HU2, Ke, KU1, Ta1, Ta2, Ta3, Ull]). Moreover, our algorithm is *conceptually simple*. It only requires the sequential computation of the four predicates `D-Safe`, `Earliest`, `Latest`, and `Isolated`. Thus our algorithm is an extension of the algorithm of [St], which simply computes the predicates `D-Safe` and `Earliest`. The two new predicates `Latest` and `Isolated` prevent any unnecessary code motion.

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 *assignments* of the form $v := t$ and edges $(m, n) \in E$ the nondeterministic branching structure of G.[4] $\mathbf{s}$ and $\mathbf{e}$ denote the

[3]Such an algorithm was first proposed in [St], which later on was interprocedurally generalized to programs with procedures, local variables and formal parameters in [KS2]. Both algorithms realize an "as early as possible" placement.

[4]We do not assume any structural restrictions on G. In fact, every algorithm computing the fixed point solution of a uni-directional bit-vector data flow analysis problem may be used to compute the predicates `D-Safe`, `Earliest`, `Latest`, and `Isolated` (cf. [He]). However, application of the efficient

unique *start node* and *end node* of G, which are both assumed to represent the empty statement *skip* and not to possess any predecessors and successors, respectively. Every node $n \in N$ is assumed to lie on a path from $\mathbf{s}$ to $\mathbf{e}$. Finally, $succ(n) =_{df} \{ m \mid (n,m) \in E \}$ and $pred(n) =_{df} \{ m \mid (m,n) \in E \}$ denote the set of all successors and predecessors of a node n, respectively.

For every node $n \equiv v := t$ and every term $t' \in \mathbf{T} \setminus \mathbf{V}$ we define two local predicates indicating, whether t' is used or modified[5]:

- $Used(n,t') =_{df} t' \in SubTerms(t)$ and
- $Transp(n,t') =_{df} v \notin Var(t')$

Here $SubTerms(t)$ and $Var(t')$ denote the set of all subterms of t and the set of all variables occurring in t, respectively.

Conventions: Following [MR1], we assume that all right-hand-side terms of assignment statements contain at most one operation symbol. This does not impose any restrictions, because every assignment statement can be decomposed into sequences of assignments of this form. As a consequence of this assumption it is enough to develop our algorithm for an arbitrary but fixed term here, because a global algorithm dealing with all program terms simultaneously is just the independent combination of all the "term algorithms". This leads to the usual bit-vector algorithms that realize such a combination efficiently (cf. [He]).

In the following, we fix the flow graph G and the term $t \in \mathbf{T} \setminus \mathbf{V}$, in order to allow a simple, unparameterized notation, and we denote the computations of t occurring in G as *original computations*.

3 Computationally Optimal Computation Points

In this section we develop an algorithm for the "as early as possible" placement, which in contrast to previous approaches is composed of *uni-directional* analyses. Here, *placement* stands for any program transformation that introduces a new auxiliary variable $\mathbf{h}$ for t, inserts at some program points assignments of the form $\mathbf{h} := t$, and replaces some of the original computations of t by $\mathbf{h}$ provided that this is *correct*, i.e. that $\mathbf{h}$ represents the same value. Formally, two computations of t represent the *same value* on a path if and only if no operand of t is modified between them. With this formal notion of value equality, the correctness condition above is satisfied for a node n if and only if on every path leading from $\mathbf{s}$ to n there is a last initialization of $\mathbf{h}$ at a node where t represents the same *value* as in n.

This definition of placement obviously leaves the freedom of inserting computations at node entries and node exits. However, one can easily prove that after the edge splitting of Section 3.1 we can restrict ourselves to placements that only insert computations at node entries.

---

techniques of [AU, GW, HU1, HU2, Ke, KU1, Ta1, Ta2, Ta3, Ull] requires that G satisfies the structural restrictions imposed by these algorithms.

[5]Flow graphs composed of *basic blocks* can be treated entirely in the same fashion replacing the predicate *Used* by the predicate *Antloc* (cf. [MR1]), indicating whether the computation of t is locally anticipatable at node n.

3.1 Critical Edges

It is well-known that in completely arbitrary graph structures the code motion process may be blocked by "critical" edges, i.e. by edges leading from nodes with more than one successor to nodes with more than one predecessor (cf. [Dh2, Dh3, DS, RWZ, SKR1, SKR2]).In Figure 3(a) the computation of "$a+b$" at node **3** is partially redundant with respect to the computation of "$a+b$" at node **1**. However, this partial redundancy cannot safely be eliminated by moving the computation of "$a+b$" to its preceding nodes, because this may introduce a new computation on a path leaving node **2** on the right branch. On the other hand, it can safely be eliminated after inserting a synthetic node **4** in the critical edge $(\mathbf{2},\mathbf{3})$, as illustrated in Figure 3(b). We will therefore restrict our attention to programs having passed the following edge splitting transformation: every edge leading to a node with more than one predecessor has been split by inserting a synthetic node[6]. This simple transformation certainly implies that all critical edges are eliminated. Moreover, it simplifies the subsequent analysis, since it allows to obtain programs that are computationally and lifetime optimal (Optimality Theorem 4.9) by inserting all computations uniformly at node entries (cf. [SKR1, SKR2])[7].

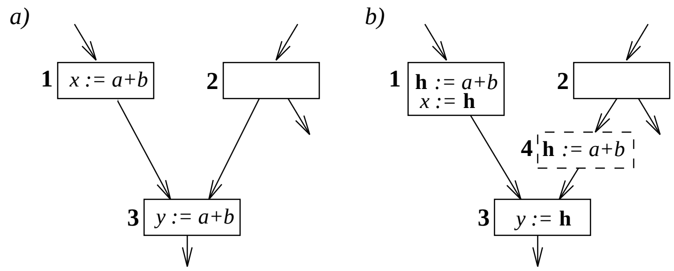

Figure 3: Critical Edges

3.2 Guaranteeing Computational Optimality: Down-Safety and Earliestness

A placement is

- *safe*, iff every computation point n is an n-*safe* node, i.e.: a computation of t at n does not introduce a new value on a path through n.

  This property is necessary in order to guarantee that the program semantics is preserved by the placement process[8].

- *earliest*, iff every computation point n is an n-*earliest* node, i.e. there is a path leading from $\mathbf{s}$ to n where no node m prior to n is n-safe and delivers the same value as n when computing t.

A safe placement is

- *computationally optimal*, iff the results of any other safe placement always require at least as many computations at run-time.

[6]In order to keep the presentation of the motivating example simple, we omit synthetic nodes that are not relevant for the Lazy Code Motion Transformation.

[7]Splitting critical edges only, would require a placement procedure which is capable of placing computations both at node entries and node exits.

[8]In particular, a safe placement does not change the potential for run-time errors, e.g. "division by 0" or "overflow".

In fact, safety and earliestness are already sufficient to characterize a computationally optimal placement:

$\star$ A placement is computationally optimal if it is safe and earliest.

However, we consider the following stronger requirement of safety which leads to an equivalent characterization and allows simpler proofs:

A placement is

- *down-safe*, iff every computation point n is an n-*down-safe* node, i.e. a computation of t at n does not introduce a new value on a terminating path starting in n.

  Intuitively, this means that an initialization $\mathbf{h} := t$ placed at the entry of node n is justified on every terminating path by an original computation occurring before any operand of t is modified[9].

As safety induces earliestness, down-safety induces the notion of *ds-earliestness*. The following lemma states that it is unnecessary to distinguish between safety and down-safety in our application, and that the notions of earliestness and ds-earliestness coincide.

Lemma 3.1 *A placement is*

1. *earliest if and only if it is ds-earliest*
2. *safe and earliest if and only if it is down-safe and ds-earliest*

Proof:

1): Earliestness implies ds-earliestness. Thus let us assume an n-ds-earliest computation point $n \neq \mathbf{s}$. This requires a path from $\mathbf{s}$ to n where no node m prior to n is n-down-safe and delivers the same value as in n when computing t. In particular, there is no original computation on this path before n that represents the same value. Thus a node m prior to n on this path where a computation of t has the same value as in n cannot be n-up-safe[10] and consequently not n-safe either[11]. Hence node n is n-earliest.

2): Due to 1) it remains to show that a safe and earliest placement is also down-safe. This follows directly from the fact that an n-earliest computation point n is not n-up-safe, which has been proved in 1). $\Box$

Let n be a computation point of a down-safe and earliest placement. Then the n-down-safety of n yields that every terminating path $p = (n_1, .., n_k)$ starting in n has a prefix $q = (n_1, \ldots, n_j)$ which satisfies:

a) $Used(n_j)$

[9] In [MR1] down-safety is called *anticipability*, and the dual notion to down-safety, *up-safety*, is called *availability*.

[10] n-*up-safety* of a node is defined in analogy to n-down-safety.

[11] Note that a node is n-safe if and only if it is n-down-safe or n-up-safe.

b) $\neg Used(n_i)$ for all $i \in \{1,\ldots,j-1\}$

c) $Transp(n_i)$ for all $i \in \{1,\ldots,j-1\}$

In the remainder of the paper the prefixes q are called *safe-earliest first-use paths* (SEFU-paths). They characterize the computation points of safe placements in the following way:

Lemma 3.2 *Let pl_{se} be a safe and earliest placement, pl_s a safe placement and $q=(n_1,\ldots,n_j)$ a SEFU-path. Then we have:*

1. *pl_{se} has no computation of t on $(n_2,\ldots,n_j)$.*

2. *pl_s has a computation of t on q.*

Proof:

1): Let n_i be a computation point of pl_{se} with $i \in \{2,\ldots,j\}$. Then we are going to derive a contradiction to the earliestness of pl_{se}. According to Lemma 3.1 we conclude that pl_{se} is down-safe and, in particular, that n_i is n-down-safe. Moreover, every predecessor m of n_i is n-down-safe too: this is trivial in the case where n_i has more than one predecessor, because then they must all be synthetic nodes. Otherwise, n_{i-1} is the only predecessor of n_i. In this case its n-down-safety follows from the properties a) – c) of q and the n-down-safety of n_1. Consequently, n_i is not n-ds-earliest and therefore due to Lemma 3.1 not n-earliest either.

2): This follows immediately from the n-earliestness of n_1, and the safety of pl_s. □

As an easy consequence of Lemma 3.2 we obtain:

Corollary 3.3 *No computation point of a computationally optimal placement occurs outside of a SEFU-path.*

The central result of this section, however, is:

Theorem 3.4 *A placement is computationally optimal if it is down-safe and earliest.*

Proof: Applying Lemma 3.2, which is possible because of Lemma 3.1, we obtain that any safe placement has at least as many computations of t on every path p from **s** to **e** as the down-safe and earliest placement. □

In the following we will compute all program points that are n-down-safe and n-earliest.

3.2.1 Computing Down-Safety

The set of n-down-safe computation points for t is characterized by the greatest solution of Equation System 3.5, which specifies a backward analysis of G.

Equation System 3.5 (Down-Safety)

$$\mathbf{D\!-\!SAFE}(n) \quad = \quad \begin{cases} \mathit{false} & \text{if } n=\mathbf{e} \\ \mathit{Used}(n) \;\vee\; (\mathit{Transp}(n) \;\wedge\; \prod_{m\in succ(n)} \mathbf{D\text{-}SAFE}(m)\;) & \text{otherwise} \end{cases}$$

Let $\texttt{D-Safe}$ be the greatest solution of Equation System 3.5. Then we have (see Figure 4 for illustration):

Lemma 3.6 (Down-Safe Computation Points)
A node n is n-down-safe *if and only if* $\texttt{D-Safe}(n)$ *holds.*

Proof:

"only if": Let $\mathbf{D}$ denote the set of nodes that are n-down-safe and $\mathbf{U}$ the set of nodes satisfying *Used*. Then all successors of a node in $\mathbf{D}\backslash\mathbf{U}$ are again in $\mathbf{D}$. Thus a simple inductive argument shows that for all nodes $n \in \mathbf{D}$ the predicate $\mathbf{D\text{-}SAFE}(n)$ remains constantly true during the maximal fixed point iteration.

"if": This can be proved by a straightforward induction on the length of a terminating path starting in n. □

3.2.2 Computing Earliestness

Earliestness, which according to Lemma 3.1 is equivalent to ds-earliestness, is characterized by the least solution of Equation System 3.7, whose solution requires a forward analysis of G.

Equation System 3.7 (Earliestness)

$$\mathbf{EARLIEST}(n) \quad = \quad \begin{cases} \mathit{true} & \text{if } n = \mathbf{s} \\ \sum\limits_{m \in pred(n)} (\, \neg \mathit{Transp}(m) \;\vee & \\ \qquad\qquad (\neg \texttt{D-Safe}(m) \;\wedge\; \mathbf{EARLIEST}(m))\,) & \text{otherwise} \end{cases}$$

Let $\texttt{Earliest}$ denote the least solution of Equation System 3.7. Along the lines of Lemma 3.6 we can prove:

Lemma 3.8 (Earliest Computation Points)
A node n is n-earliest *if and only if* $\texttt{Earliest}(n)$ *holds.*

Figure 4 shows the predicate values of $\texttt{Earliest}$ for our motivating example. It illustrates that $\texttt{Earliest}$ is valid at the start node and additionally at those nodes that are reachable by a path where no node prior to n is n-down-safe and delivers the same value as n when computing t. Of course, computations cannot be placed earlier than in the start node, which justifies $\texttt{Earliest}(\mathbf{1})$ in Figure 4. Moreover, no node on the path $(\mathbf{1}, \mathbf{4}, \mathbf{5}, \mathbf{7}, \mathbf{18})$ is n-down-safe. Thus $\texttt{Earliest}(\{\mathbf{2}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{18}\})$ holds. Finally, evaluating t at node $\mathbf{1}$ and $\mathbf{2}$ delivers a different value as in node $\mathbf{3}$, which yields $\texttt{Earliest}(\mathbf{3})$.

The Safe-Earliest Transformation

`D-Safe` and `Earliest` induce the Safe-Earliest Transformation:

• *Introduce a new auxiliary variable* **h** *for* t.
• *Insert at the entry of every node satisfying* `D-Safe` *and* `Earliest` *the assignment* $\mathbf{h} := t$.
• *Replace every original computation of* t *in* G *by* **h**.

Table 1: The Safe-Earliest Transformation

Whenever `D-Safe`(n) holds there is a node m on every path p from **s** to n satisfying `D-Safe`(m) and `Earliest`(m) such that no operand of t is modified between m and n. Thus all replacements of the Safe-Earliest Transformation are correct, which guarantees that the Safe-Earliest Transformation is a placement. Moreover, according to Lemma 3.1, Lemma 3.6 and Lemma 3.8 the Safe-Earliest Transformation is down-safe and earliest. Together with Theorem 3.4 this yields:

Theorem 3.9 (Computational Optimality)
The Safe-Earliest Transformation is computationally optimal.

Figure 5 shows the result of the Safe-Earliest Transformation for the motivating example of Figure 1, which is essentially the same as the one delivered by the algorithm of Morel and Renvoise [MR1][12]. In general, however, there may be some deviations, since their algorithm inserts computations at the end of nodes, and it moves computations only, if they are partially available. Introducing this condition can be considered as a first step in order to avoid unnecessary code motion. However, it limits the effect of unnecessary code motion only heuristically. For instance, in the example of Figure 5 the computations of node **10**, **15**, **16** and **17** would be moved to node **6**, and therefore more than necessary. In particular, the computation of node **17** cannot be moved with run-time gain at all.

In the next section we are going to develop a procedure that completely avoids any unnecessary code motion.

4 Suppressing Unnecessary Code Motion

In order to avoid unnecessary code motion, computations must be placed as *late* as possible while maintaining computational optimality.

4.1 Guaranteeing Lifetime Optimality: Latestness and Isolation

A computationally optimal placement pl is

- *latest*, iff every computation point n is an n-*latest* node, i.e.:
    - n is a computation point of some computationally optimal placement and

[12] In the example of Figure 1 the algorithm of [MR1] would not insert a computation at node **3**.

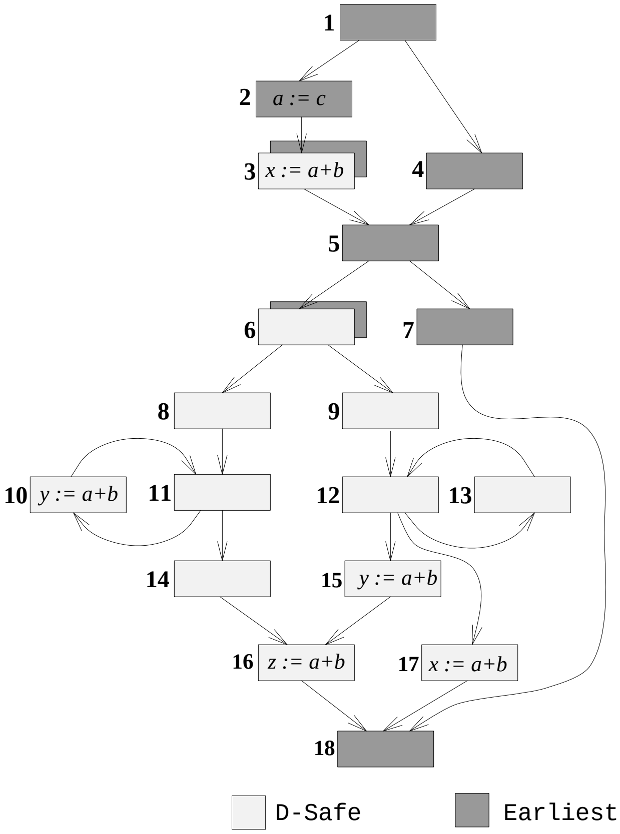

Figure 4: The `D-Safe` and `Earliest` predicate values

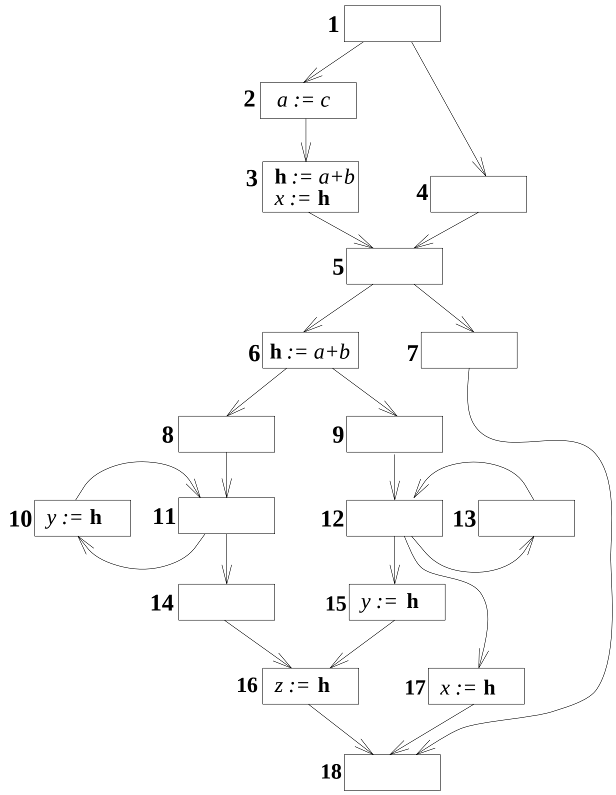

Figure 5: The Safe-Earliest Transformation

  – on every terminating path p starting in n any following computation point of a computationally optimal placement occurs after an original computation on p.

  Intuitively, this means no other computationally optimal placement can have "later" computation points on any path.

- *isolated*, iff there is a computation point n that is an n-*isolated* node with respect to pl, i.e. on every terminating path starting in a successor of n every original computation is preceded by a computation of pl.

  Essentially, this means that an initialization $\mathbf{h} := t$ placed at the entry of n reaches a second original computation at most after passing a new initialization of $\mathbf{h}$.

- *lifetime optimal* (or economic), iff any other computationally optimal placement always requires at least as long lifetimes of the auxiliary variables (registers).

We have:

Theorem 4.1 *A computationally optimal placement is lifetime optimal if and only if it is latest and not isolated.*

Proof: We only prove the "if"-direction here, because the other implication is irrelevant for establishing our main result Optimality Theorem 4.9.

The proof proceeds by contraposition, showing that a computationally optimal but not lifetime optimal placement pl_{co} is not latest or isolated. This requires the consideration of a lifetime optimal placement pl_{lto} for comparison.

The fact that pl_{co} is not lifetime optimal implies the existence of a SEFU-path $q=(n_1,..,n_j)$ such that pl_{co} has an initialization in a node n_c which precedes a computation of pl_{lto} in a node n_l with $c \leq l$. Now we have to distinguish two cases:

$c < l$: Applying property b) of q, we obtain that the path $(n_c,..,n_{l-1})$ is free of original computations. Thus n_c is not n-latest and consequently pl_{co} not latest.

$c = l$: Obviously, this implies $c=l=j$, and that the computation of pl_{lto} in n_l is an original one.

Thus it remains to show that n_l is n-isolated with respect to pl_{co}, i.e. that on every terminating path starting in a successor n of n_j the first original computation is preceded by a computation of pl_{co}.

We lead the assumption that n_j is not n-isolated, i.e. that there exists a path $p = (n, \ldots, m)$ without computations of pl_{co} but with an original computation at m, to a contradiction.

Under this assumption Lemma 3.2(2) delivers that p is also free of computations of the Safe-Earliest Transformation, which yields that every node of p appears outside a SEFU-path. Thus, according to Corollary 3.3, p is also free of computations of pl_{lto}, which implies that pl_{lto} does not initialize **h** on either q or p. Moreover, because of the n-earliestness of n_1, it is also impossible that **h** is initialized on every path from **s** to n_1 after the last modification of an operand of t. This contradicts the placement property of pl_{lto}, and therefore implies that n_j is n-isolated with respect to pl_{co} as desired. □

4.1.1 Computing Latestness

Intuitively, lifetime optimal placements can be obtained by successively moving the computations from their earliest safe computation points in direction of the control flow to "later" points as long as computational optimality is preserved. This gives rise to the following definition:

A placement is

- *delayed*, iff every computation point n is an n-*delayed* node, i.e. on every path from **s** to n there is a computation point of the Safe-Earliest Transformation such that all subsequent original computations of p lie in n.

Technically, the computation of n-delayed computation points is realized by determining the greatest solution of Equation System 4.2, which requires a forward analysis of G.

Equation System 4.2 (Delay)

$$\mathbf{DELAY}(n) \;=\; (\texttt{D-Safe}(n) \wedge \texttt{Earliest}(n)) \vee \begin{cases} \mathit{false} & \text{if } n=\mathbf{s} \\ \prod_{m \in pred(n)} \neg \mathit{Used}(m) \wedge \mathbf{DELAY}(m) & \text{otherwise} \end{cases}$$

Let $\texttt{Delay}$ be the greatest solution of Equation System 4.2. Analogously to Lemma 3.6 we can prove:

Lemma 4.3 *A node n is* n-*delayed if and only if* $\texttt{Delay}(n)$ *holds.*

Furthermore we have:

Lemma 4.4

1. *A computationally optimal placement is delayed.*
2. *A delayed placement is down-safe.*

Proof:

1): We have to show that every computation point n of a computationally optimal placement is n-delayed. This can be deduced from Corollary 3.3 and property b) of SEFU-paths, which deliver that every path from $\mathbf{s}$ to n goes through a computation point of the Safe-Earliest Transformation such that all subsequent original computations of p lie in n.

2): According to Lemma 3.6 and Lemma 4.3 it is enough to show that $\texttt{Delay}(n)$ implies $\texttt{D-Safe}(n)$ for every node n. This can be done by a straightforward induction on the length of a shortest path from $\mathbf{s}$ to n. □

Based on the predicate $\texttt{Delay}$ we define $\texttt{Latest}$ by:

$$\forall\, n \in N.\, \texttt{Latest}(n) =_{df} \texttt{Delay}(n) \wedge (\mathit{Used}(n) \vee \neg \prod_{m \in succ(n)} \texttt{Delay}(m))$$

The predicate values of $\texttt{Delay}$ and $\texttt{Latest}$ are illustrated for our motivating example in Figure 6.

We have:

Lemma 4.5 *A node n is* n-*latest if and only if* $\texttt{Latest}(n)$ *holds.*

Proof: We will concentrate here on the part of the proof being relevant for the proof of Theorem 4.9, which is the "if"-direction for nodes that occur as computation points of some computationally optimal placement. This is much simpler than the proof for the general case.

Thus let us assume a computation point n of a computationally optimal placement satisfying $\texttt{Latest}$. Then it is sufficient to show that on every terminating path p starting in n any following computation point of a computationally optimal placement occurs after an original computation on p.

Obviously, this is the case if n itself contains an original computation. Thus we can reduce our attention to the other case, in which $\neg\texttt{Delay}(m)$ must hold for some successor m of n. This directly yields that n is a synthetic node with m being its only successor, because m must have several predecessors.

Now assume a terminating path p starting in m, and a node l on p that occurs not later than the first node with an original computation. Then it remains to show that l is not a computation point of a computationally optimal placement.

Lemma 3.2(1) implies that there does not exist any computation point of the Safe-Earliest Transformation prior to l on p, because n is n-delayed due to Lemma 4.3. Moreover Lemma 4.3 yields that m is not n-delayed. Thus l cannot be n-delayed either. According to Lemma 4.4(1) this directly implies that l is not a computation point of any computationally optimal placement. ☐

The Latest Transformation

As `D-Safe` and `Earliest` also `Latest` specifies a program transformation. We have:

Lemma 4.6 *The* Latest Transformation *is a computationally optimal placement.*

Proof: Whereas the safety property holds due to Lemma 4.3 and Lemma 4.4, the proof that the Latest Transformation is a placement and that it is computationally optimal can be done simultaneously by showing that (1) every SEFU-path contains a node satisfying `Latest` and that (2) nodes satisfying `Latest` do not occur outside SEFU-paths. This is straightforward from the definition of `Delay`. ☐

Figure 7 shows the result of the Latest Transformation for the motivating example.

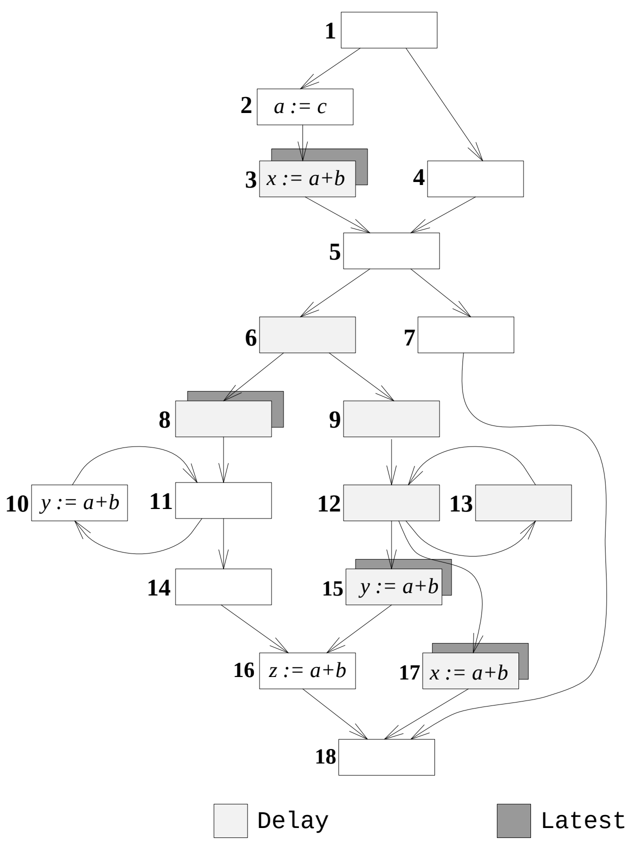

Figure 6: The `Delay` and `Latest` predicate values

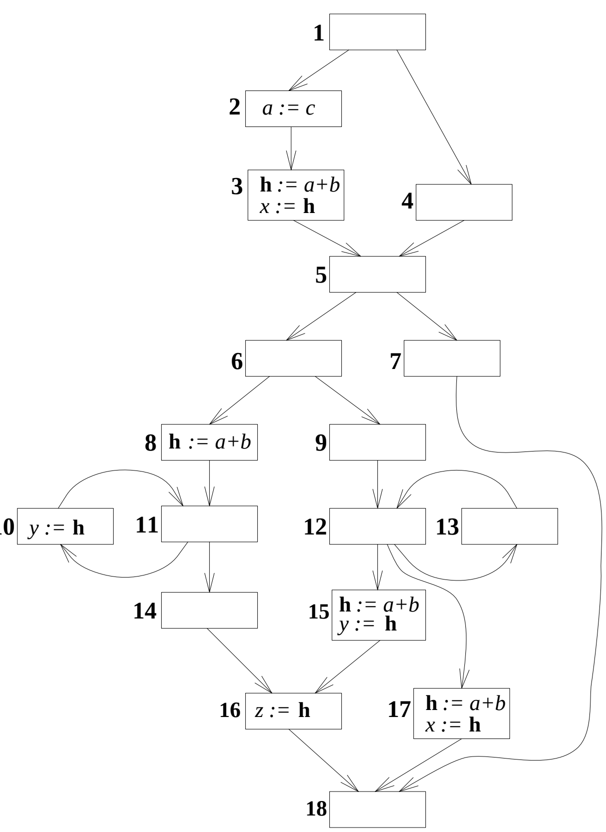

Figure 7: The Latest Transformation

4.1.2 Computing Isolation

The Latest Transformation is already more economic than any other algorithm proposed in the literature. However, it still contains unnecessary initializations of $\mathbf{h}$ in node **3** and **17**. In order to avoid such unnecessary initializations, we must identify all program points where an inserted computation would only be used in the insertion node itself. This is achieved by determining the greatest solution of Equation System 4.7, which specifies a backward analysis of G.

Equation System 4.7 (Isolation)

$$\mathbf{ISOLATED}(n) \;=\; \prod_{m \in succ(n)} \mathtt{Latest}(m) \;\vee\; (\,\neg \mathit{Used}(m) \wedge \mathbf{ISOLATED}(m)\,)$$

Let $\mathtt{Isolated}(n)$ be the greatest solution of Equation System 4.7. Then we can prove in analogy to Lemma 3.6:

Lemma 4.8
A node n is n-*isolated with respect to the Latest Transformation if and only if* $\mathtt{Isolated}(n)$ *holds.*

Figure 8 shows the predicate values of `Isolated` for the running example.

4.1.3 The Optimal Program Transformation

The set of optimal computation points for t in G is given by the set of nodes that are *latest*, but not *isolated*

$$\mathbf{OCP} =_{df} \{\, n \mid \mathtt{Latest}(n) \wedge \neg \mathtt{Isolated}(n) \,\}$$

and the set of nodes that contain a redundant occurrence of t with respect to **OCP** is specified by

$$\mathbf{RO} =_{df} \{\, n \mid \mathit{Used}(n) \wedge \neg(\,\mathtt{Latest}(n) \wedge \mathtt{Isolated}(n)\,) \,\}$$

Note that the occurrences of t in nodes satisfying the `Latest` and `Isolated` property are not redundant with respect to **OCP**, since the initialization of the corresponding auxiliary variable is suppressed.

The Lazy Code Motion Transformation

OCP and **RO** induce the Lazy Code Motion Transformation.

- *Introduce a new auxiliary variable* $\mathbf{h}$ *for* t.
- *Insert at the entry of every node in* **OCP** *the assignment* $\mathbf{h} := t$.
- *Replace every original computation of* t *in nodes of* **RO** *by* $\mathbf{h}$.

Table 2: The Lazy Code Motion Transformation

Our main result states that this transformation is optimal:

Theorem 4.9 (Optimality Theorem)
The Lazy Code Motion Transformation is computationally and lifetime optimal.

Proof: Following the lines of Lemma 4.6 we can prove that the Lazy Code Motion Transformation is a computationally optimal placement. According to Lemma 4.5 and Lemma 4.8 it is also latest and not isolated (Lemma 4.8 can be applied since the Lazy Code Motion Transformation has the same computation points as the Latest Transformation.). Thus Theorem 4.1 completes the proof. □

The application of the Lazy Code Motion Transformation to the flow graph of Figure 1 results in the promised flow graph of Figure 9.

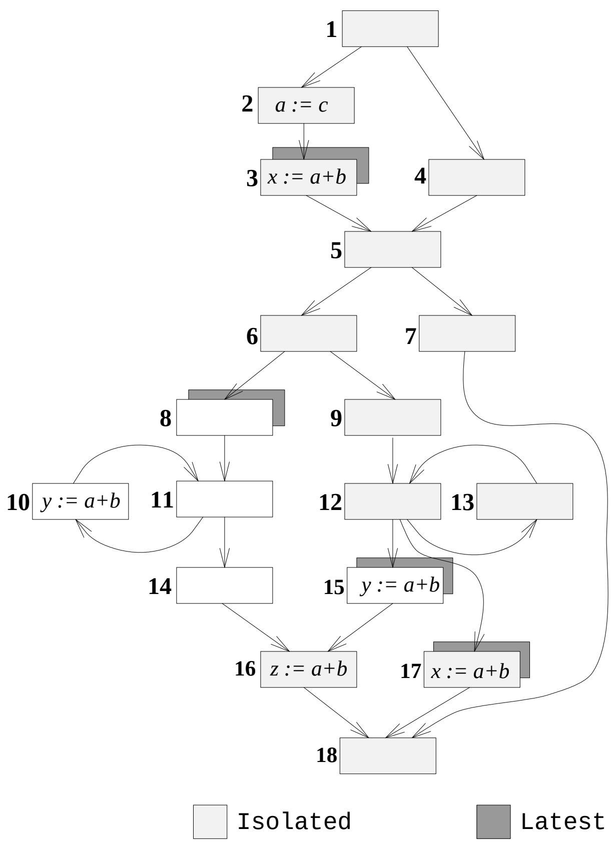

Figure 8: The `Isolated` and `Latest` predicate values

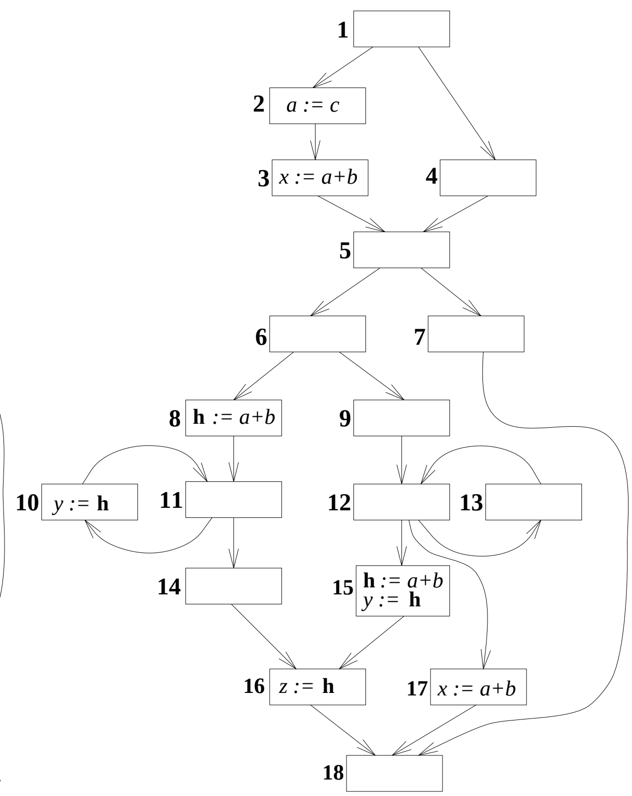

Figure 9: The Lazy Code Motion Transformation

5 Conclusions

We have presented a bit-vector algorithm for the *computationally* and *lifetime optimal* placement of computations within flow graphs, which is as *efficient* as standard uni-directional analyses. Important feature of this algorithm is its *laziness*: computations are placed as *early as necessary* but as *late as possible*. This guarantees the lifetime optimality while preserving computational optimality.

Fundamental was the decomposition of the typically bi-directionally specified code motion procedures into uni-directional components. Besides yielding clarity and reducing the number of predicates drastically, this allows us to utilize the efficient algorithms for uni-directional bit-vector analyses. Moreover, it makes the algorithm modular, which supports future extensions: in [KRS2] we present an extension of our lazy code motion algorithm which, in a similar fashion as in [JD1, JD2], uniformly combines code motion and strength reduction, and following the lines of [KS1, KS2] a generalization to programs with procedures, local variables and formal parameters is straightforward. We are also investigating an adaption of the *as early as necessary* but *as late as possible* placing strategy to the semantically based code motion algorithms of [SKR1, SKR2].

References

[AU] Aho, A. V., and Ullman, J. D. Node listings for reducible flow graphs. In *Proceedings 7th STOC*, 1975, 177 - 185.

[Ch] Chow, F. A portable machine independent optimizer – Design and measurements. Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, Stanford, Calif., and Tech. Rep. 83-254, Computer Systems Lab., Stanford University, 1983.

[Dh1] Dhamdhere, D. M. Characterization of program loops in code optimization. *Comp. Lang. 8*, 2 (1983), 69 - 76.

[Dh2] Dhamdhere, D. M. A fast algorithm for code movement optimization. *SIGPLAN Not. 23*, 10 (1988), 172 - 180.

[Dh3] Dhamdhere, D. M. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Trans. Program. Lang. Syst. 13*, 2 (1991), 291 - 294.

[DS] Drechsler, K. H., and Stadel, M. P. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Trans. Program. Lang. Syst. 10*, 4 (1988), 635 - 640.

[GW] Graham, S. L., and Wegman, M. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM 23*, 1 (1976), 172 - 202.

[He] Hecht, M. S. Flow analysis of computer programs. Elsevier, North-Holland, 1977.

[HU1] Hecht, M. S., and Ullman, J. D. Analysis of a simple algorithm for global flow problems. In *Proceedings 1st POPL*, Boston, Massachusetts, 1973, 207 - 217.

[HU2] Hecht, M. S., and Ullman, J. D. A simple algorithm for global data flow analysis problems. In *SIAM J. Comput. 4*, 4 (1977), 519 - 532.

[JD1] Joshi, S. M., and Dhamdhere, D. M. A composite hoisting-strength reduction transformation for global program optimization – part I. *Internat. J. Computer Math. 11*, (1982), 21 - 41.

[JD2] Joshi, S. M., and Dhamdhere, D. M. A composite hoisting-strength reduction transformation for global program optimization – part II. *Internat. J. Computer Math. 11*, (1982), 111 - 126.

[Ke] Kennedy, K. Node listings applied to data flow analysis. In *Proceedings 2nd POPL*, Palo Alto, California, 1975, 10 - 21.

[KRS1] Knoop, J., Rüthing, O., and Steffen, B. Lazy code motion. In *Proceedings ACM SIGPLAN PLDI'92*, San Francisco, California, June 17 - 19, 1992.

[KRS2] Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. To appear.

[KS1] Knoop, J., and Steffen, B. The interprocedural coincidence theorem. In *Proceedings CC'92*, Paderborn, Germany, October 5-7, Lecture Notes in Computer Science, Springer-Verlag, 1992.

[KS2] Knoop, J., and Steffen, B. Efficient and optimal interprocedural bit-vector data flow analyses: A uniform interprocedural framework. To appear.

[KU1] Kam, J. B., and Ullman, J. D. Global data flow analysis and iterative algorithms. *Journal of the ACM 23*, 1 (1976), 158 - 171.

[KU2] Kam, J. B., and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica 7*, (1977), 309 - 317.

[Mo] Morel, E. Data flow analysis and global optimization. In: Lorho, B. (Ed.). Methods and tools for compiler construction, Cambridge University Press, 1984.

[MR1] Morel, E., and Renvoise, C. Global optimization by suppression of partial redundancies. *Commun. of the ACM 22*, 2 (1979), 96 - 103.

[MR2] Morel, E., and Renvoise, C. Interprocedural elimination of partial redundancies. In: Muchnick, St. S., and Jones, N. D. (Eds.). Program flow analysis: Theory and applications. Prentice Hall, Englewood Cliffs, NJ, 1981.

[RWZ] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *Proceedings 15th POPL*, San Diego, California, 1988, 12 - 27.

[So] Sorkin, A. Some comments on A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Trans. Program. Lang. Syst. 11*, 4 (1989), 666 - 668.

[St] Steffen, B. Data flow analysis as model checking. In *Proceedings TACS'91*, Sendai, Japan, Springer-Verlag, LNCS 526 (1991), 346 - 364.

[SKR1] Steffen, B., Knoop, J., and Rüthing, O. The value flow graph: A program representation for optimal program transformations. In *Proceedings 3rd ESOP*, Copenhagen, Denmark, Springer-Verlag, LNCS 432 (1990), 389 - 405.

[SKR2] Steffen, B., Knoop, J., and Rüthing, O. Efficient code motion and an adaption to strength reduction. In *Proceedings 4th TAPSOFT*, Brighton, United Kingdom, Springer-Verlag, LNCS 494 (1991), 394 - 415.

[Ta1] Tarjan, R. E. Applications of path compression on balanced trees. *Journal of the ACM 26*, 4 (1979), 690 - 715.

[Ta2] Tarjan, R. E. A unified approach to path problems. *Journal of the ACM 28*, 3 (1981), 577 - 593.

[Ta3] Tarjan, R. E. Fast algorithms for solving path problems. *Journal of the ACM 28*, 3 (1981), 594 - 614.

[Ull] Ullman, J. D. Fast algorithms for the elimination of common subexpressions. *Acta Informatica 2*, 3 (1973), 191 - 213.