Lazy Strength Reduction*

Jens Knoop † Oliver Rüthing † Bernhard Steffen ‡

Abstract

We present a bit-vector algorithm that uniformly combines code motion and strength reduction, avoids superfluous register pressure due to unnecessary code motion, and is as efficient as standard unidirectional analyses. The point of this algorithm is to combine the concept of lazy code motion of [1] with the concept of unifying code motion and strength reduction of [2, 3, 4]. This results in an algorithm for *lazy* strength reduction, which consists of a sequence of unidirectional analyses, and is unique in its transformational power.

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

1 Motivation

Code motion improves the runtime efficiency of a program by avoiding unnecessary recomputations of a value at runtime. *Strength reduction* improves runtime efficiency by reducing "expensive" recomputations to less expensive ones, e.g., by reducing computations involving multiplication to computations involving only addition. Common to both techniques is replacing the original computations of a program by auxiliary variables (registers) that are initialized at suitable program points. In the case of strength reduction, they are additionally updated at certain points. The similarity between strength reduction and code motion suggests the combination of both techniques, an idea which was first realized by an algorithm of Joshi and Dhamdhere [2, 3] that enhances the code motion algorithm of Morel and Renvoise [5] to capture strength reduction.

In this paper we combine Joshi and Dhamdhere's approach with the idea of lazy code motion presented in [1]. This results in an algorithm for *lazy strength reduction*, which uniformly combines code motion and strength reduction, and avoids any unnecessary register pressure. In fact, in contrast to previous algorithms, it does not insert multiplications and additions on the same path, minimizes the lifetimes of moved computations, and limits the insertion of multiple additions on a path to a minimum. Moreover, as our algorithm is composed of a sequence of unidirectional bit-vector analyses, it is as efficient as the standard unidirectional analyses (cf. [6, 7, 8, 9, 10, 11, 12, 13, 14, 15]), which drastically improves on the results about the original bidirectional algorithms of [2, 3, 4].

The illustration of the essential new features of our algorithm requires a rather complex program structure. In the example of Figure 1, which is a slight modification of an example of [3], our lazy strength reduction algorithm is unique in yielding the result shown in Figure 2. This transformation is exceptional for the following reasons: It replaces the multiplications of

*In *Journal of Programming Languages 1*, 1 (1993), 71 - 91.

†Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, Preußerstraße 1-9, D-2300 Kiel 1, Germany. Part of the work was done while the first author was supported by the Deutsche Forschungsgemeinschaft grant La 426/9-2. The second author is supported by the Deutsche Forschungsgemeinschaft grant La 426/11-1.

‡Lehrstuhl für Informatik II, Rheinisch-Westfälische Technische Hochschule Aachen, Ahornstraße 55, D-5100 Aachen, Germany.

$i * 10$ at node **13** and **14** by moving them to node **7** and **8** and inserting a single addition at node **9**. In the "right" part of the loop this reduces the original multiplications to an addition. Furthermore, it does not touch the computation of $i * 10$ at node **21** that cannot be moved profitably. The example will be discussed in more detail during the development of the paper.

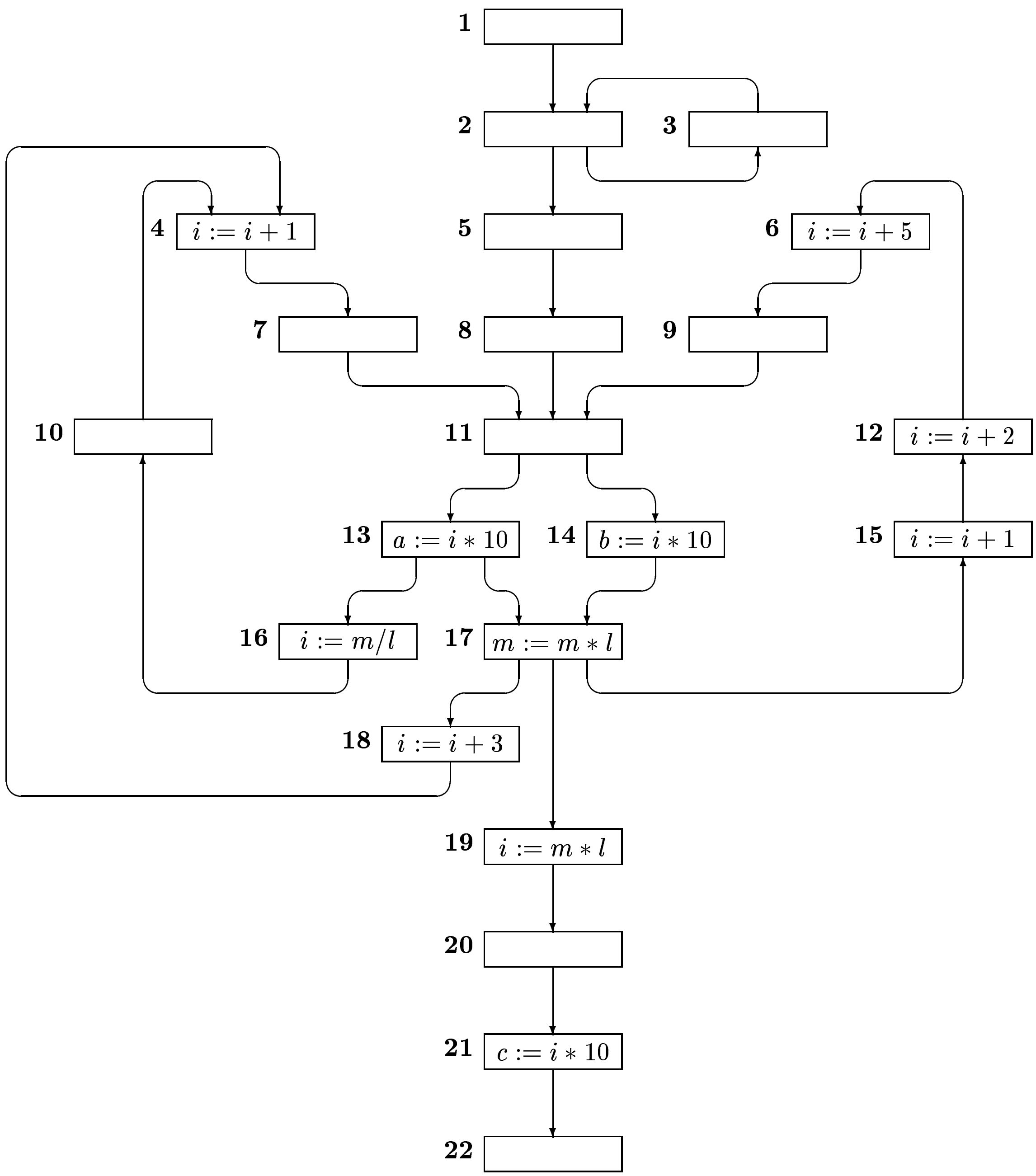

Figure 1: The Motivating Example

Related Work

The point of Morel and Renvoise's code motion algorithm [5] is to place computations *as early as possible* in a program, while guaranteeing that every inserted computation is used on every terminating program path leaving the insertion point. In order to capture strength reduction,

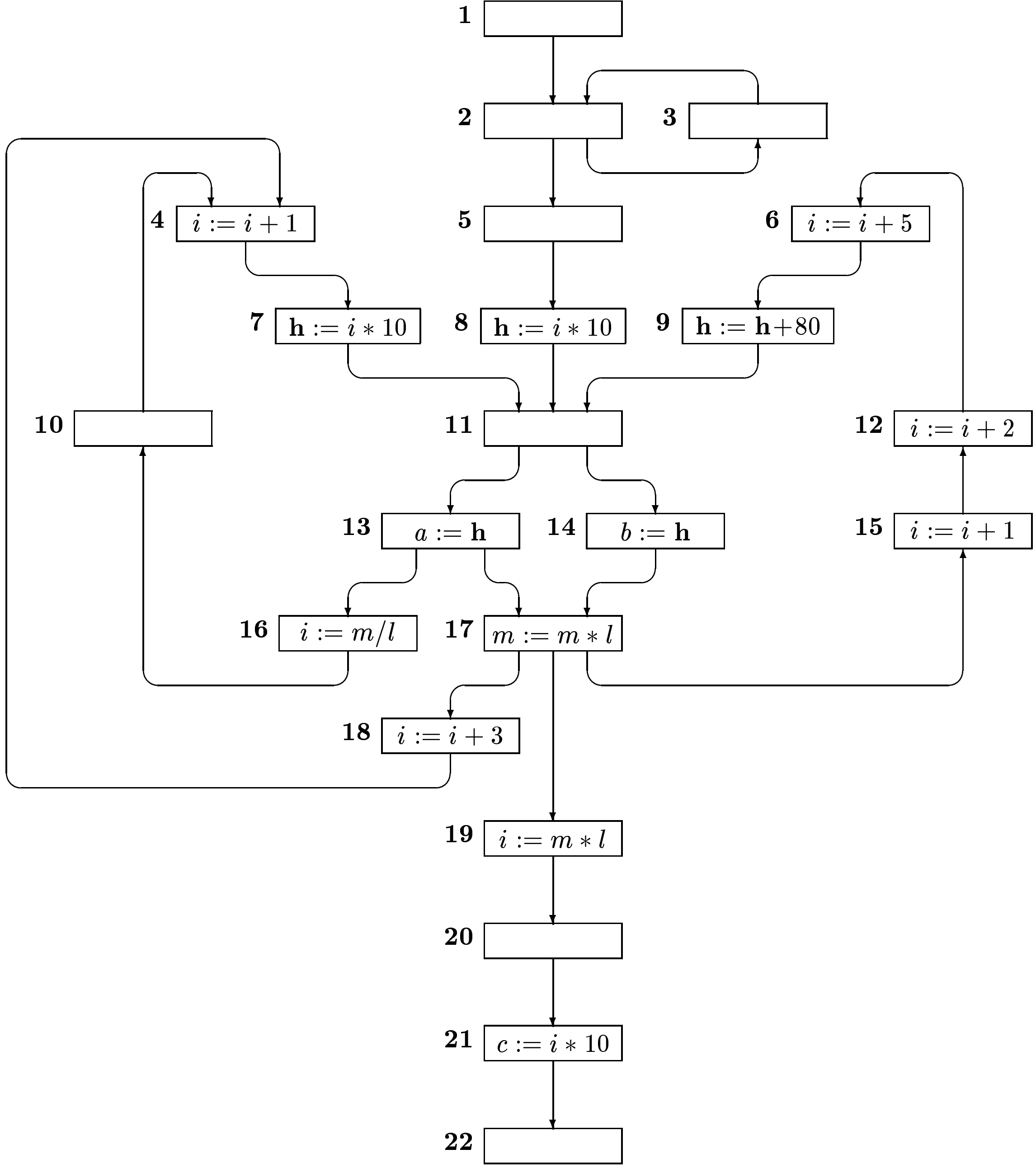

Figure 2: The Lazy Strength Reduction Transformation

Joshi and Dhamdhere's algorithms [2, 3, 4] allow the values of inserted computations to be updated before their use by the addition of constants, while maintaining the strategy of placing computations as early as possible. This strategy, however, moves computations even if it is *unnecessary*, i.e., there is no runtime gain, and therefore causes superfluous register pressure, which is a major problem in practice.[1]

Recently the problem of unnecessary code motion was addressed in [1], where an algorithm for *lazy* code motion was presented. In contrast to all previous code motion algorithms (cf. [17, 18, 19, 20, 5, 21]), this algorithm places computations *as late as possible* in a program, while maintaining computational optimality.[2] It is unique in that it avoids any unnecessary

[1]In [16] unnecessary code motion is called *redundant*.

[2]Here, *computational optimality* means that a program cannot be improved by means of semantics preserving code motion (see [1] for details).

code motion, and therefore any unnecessary register pressure.

There are also other, conceptually different, approaches to strength reduction. For example the algorithms of [22, 23, 24] are restricted to loops and require explicit detection of induction variables. Thus, in the example of Figure 1 they would not do anything, since i is not an induction variable (cf. [25]). In contrast, the semantically based algorithm for code motion and strength reduction of [26] works for arbitrary program and term structures, but does not capture the laziness effect. This also holds for the (significantly different) approach of [27, 28, 29], namely *finite differencing*, whose major achievement is the generalization of strength reduction to nonnumerical applications.

Structure of the Paper

After the preliminary definitions in Section 2, and a brief summary of the basic version of the code motion algorithm of [1] together with its extension to strength reduction along the lines of [2] in Section 3, we arrive at the central Section 4, where the lazy strength reduction algorithm is developed. This development is split into three steps which successively enhance the power of the algorithm by adding new predicates that guarantee new properties of the resulting program. The paper closes with Section 5 containing our conclusions.

2 Preliminaries

We consider *terms* $t \in \mathbf{T}$, which are inductively built of *variables* $x \in \mathbf{V}$, *constants* $c \in \mathbf{C}$, and *operators* $op \in \mathbf{Op}$. As usual, we represent an imperative program as a *directed flowgraph* $G=(N,E,\mathbf{s},\mathbf{e})$ with node set N and edge set E. Nodes $n \in N$ represent *assignments* of the form $x := t$. Edges $(m,n) \in E$ denote the nondeterministic branching structure of G.[3] $\mathbf{s}$ and $\mathbf{e}$ denote the 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 sets of all immediate successors and predecessors of a node n, respectively.

Candidate Expressions for Strength Reduction

We demonstrate our approach of uniformly combining lazy code motion and strength reduction by means of the classical application of strength reduction, which reduces multiplications to additions. Therefore, we assume two binary operators, $+$ and $*$, in $\mathbf{Op}$, which we interprete as ordinary addition and multiplication, respectively. Terms of the form $v * c$ with $v \in \mathbf{V}$ and $c \in \mathbf{C}$ are called *candidate expressions* for strength reduction, because they may give rise to a transformation that eliminates the multiplication. In the following we will develop our algorithm for an arbitrary but fixed candidate expression $v*c$, which allows us to keep our notation simple.

Use, Transparency, and SR-Transparency

For every node $n \equiv x := t$ we define three local predicates indicating, whether $v * c$ is used or modified by the assignment of node n. Here, $SubTerms(t)$ denotes the set of all subterms of t, e.g., $SubTerms(a + ((v * c) - b)) = \{ a, v, c, b, v * c, (v * c) - b, a + ((v * c) - b) \}$.[4]

[3] We do not assume any structural restrictions on G. In fact, every algorithm computing the fixed-point solution of a unidirectional bit-vector data flow analysis problem may be used to compute the predicates involved in the lazy strength reduction transformation (cf. [25]). However, application of the efficient techniques of [6, 7, 8, 9, 10, 11, 12, 13, 14, 15] requires that G satisfies the structural restrictions imposed by these algorithms.

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

- $Used\,(n) =_{df} v * c \in SubTerms\,(t)$
- $Transp\,(n) =_{df} x \not\equiv v$
- $SR\text{-}Transp(n) =_{df} Transp\,(n) \vee t \equiv v + d$ with $d \in \mathbf{C}$

In addition to *Transp*, the predicate *SR-Transp* is also valid at those nodes where the effect of an assignment on the value of $v * c$ can be dealt with by means of an update assignment common to strength reduction (cf. Section 3.2). Such nodes will be indicated by the predicate *Injured*.[5] Additionally, we define a function $Eff : N \to \omega$ by

$$\forall\, n \in N.\ Eff\,(n) =_{df} \begin{cases} c * d & \text{if } Injured\,(n) \text{ with } n \equiv v := v + d \\ 0 & \text{otherwise} \end{cases}$$

which provides the amount of updating that is necessary in order to pass node n.[6]

Inserting Synthetic Nodes

In order to exploit the full power of our algorithm for lazy strength reduction, we assume that in the flowgraph G to be considered from now on, every edge leading to a node with more than one predecessor has been split by inserting a synthetic node.[7] Inserting synthetic nodes is common for code motion optimizations (cf. [17, 16, 18, 19, 1, 20, 31, 32, 26]) and discussed in more details in Section A.1.

3 Simple Strength Reduction

In this section we present a simple algorithm for strength reduction, which evolves straightforwardly as a uniform extension of the basic version of the code motion algorithm of [1]. Section 3.1, therefore, recalls the essentials of this code motion algorithm, while Section 3.2 presents the modifications to capture strength reduction, and Section 3.3 presents a discussion of the deficiencies of simple strength reduction.

3.1 Code Motion: Down-Safety and Earliestness

The point of the basic version of the code motion algorithm of [1] is to place computations as *early as possible* within a program while maintaining its semantics. This is achieved by moving computations to program points where they are *down-safe* and *earliest*. Intuitively, *down-safe* means that the inserted value is used on every terminating program path starting with the insertion point. This guarantees that the program semantics is preserved,[8] and *earliest* means that a placement at an "earlier" position would either not be down-safe or would not always deliver the required value. This is sufficient in order to guarantee that the number of calculations at runtime cannot be reduced any further by means of a safe placement. Clearly, the code motion transformation, which we apply to the candidate expression $v * c$ here, works for arbitrary program terms.

---

n.

[5] In the terminology of [14], assignments of the form $v := v + d$ do not *kill* the value of $v * c$, but *injure* it, i.e., establishing the new value of $v * c$ requires only a "small", cheap to perform modification of the current value. In [30] the term *wounded* is used instead of injured.

[6] Note that for any node n satisfying *Injured* the value $Eff\,(n)$ can be computed at compile time, since c and d are both constants in $\mathbf{C}$.

[7] In order to keep the presentation of the motivating example simple, we omit synthetic nodes that are not relevant for the lazy strength reduction transformation.

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

3.1.1 Down-Safety

$v * c$ can safely be placed at the entry of a node $n \in N$, if it is used on every terminating program path starting with n before v is modified. These *down-safe* computation points for $v*c$ are characterized by the greatest solution of Equation System 3.1, which specifies a backward analysis of G.[9]

Equation System 3.1 (D-SAFE)

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

3.1.2 Earliestness

Placing computations as "early" as possible is sufficient to obtain *computationally optimal* programs, i.e., programs that cannot be further improved by means of a safe placement (cf. [1]). The program points enjoying the earliestness property are characterized by the least solution of Equation System 3.2, which is obtained by means of a forward analysis of G. Here, $\mathtt{D\text{-}Safe}_{\mathtt{CM}}$ [10] denotes the greatest solution of Equation System 3.1.

Equation System 3.2 (EARLIEST)

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

Let $\mathtt{Earliest}_{\mathtt{CM}}$ denote the least solution of Equation System 3.2, which is based upon the following intuition (cf. Section A.2 for an illustration). A placement of $v * c$ at the entry of a node n is "earliest" if there is a path from $\mathbf{s}$ to n on which any prior computation of $v * c$

- would not provide the same value as in n due to a subsequent modification

or

- would not be down-safe.

3.1.3 The Code Motion Transformation

Denoting the program points satisfying both $\mathtt{D\text{-}Safe}_{\mathtt{CM}}$ and $\mathtt{Earliest}_{\mathtt{CM}}$ by $\mathtt{Insert}_{\mathtt{CM}}$, the following three-step procedure results in a computationally optimal program, i.e., in a program that cannot be improved by means of a safe code motion transformation for $v * c$ (cf. [1]).

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node satisfying $\mathtt{Insert}_{\mathtt{CM}}$ the assignment $\mathbf{h} := v * c$
3. Replace every (original) occurrence of $v * c$ in G by $\mathbf{h}$

Table 1: The Safe-Earliest Transformation

[9] In [2, 3, 4] down-safety is called *anticipability*.

[10] CM stands for Code Motion.

3.2 Strength Reduction as Refined Code Motion

Code motion moves the computation $v * c$ backwards as long as v is not modified within a node, i.e., as long as *Transp* is satisfied. The point of strength reduction is to weaken this transparency requirement, and to move $v * c$ as long as its value can be updated simply by adding a constant to the current value, i.e., as long as *SR-Transp* is satisfied, or equivalently, as long as the value of $v * c$ is only injured. Subsequently, the injured values can be cured simply by adding the constant $\mathit{Eff}(n)$ to it.

This change to the notion of transparency also affects the notion of safety: For strength reduction we allow $v * c$ to be placed at the entry of a node $n \in N$ if it is used on every terminating program path starting with n before the value is killed. In order to determine these program points, it is sufficient to replace the predicate *Transp* in Equation System 3.1 by *SR-Transp*.[11] This directly leads to the definition of SR-down-safe and SR-earliest computation points for strength reduction.

3.2.1 SR-Down-Safety and SR-Earliestness

$v * c$ can be placed SR-down-safely at the entry of all nodes satisfying the predicate

$$\texttt{D-Safe}_{\texttt{SR}}$$

which denotes the greatest solution of Equation System 3.1, where *Transp* is replaced by *SR-Transp*. Analogously, the least solution of Equation System 3.2, where *SR-Transp* and $\texttt{D-Safe}_{\texttt{SR}}$ are used instead of *Transp* and $\texttt{D-Safe}_{\texttt{CM}}$, respectively, is denoted by

$$\texttt{Earliest}_{\texttt{SR}}$$

$\texttt{Earliest}_{\texttt{SR}}$ specifies the earliest computation points with respect to $\texttt{D-Safe}_{\texttt{SR}}$.[12] Program points satisfying both $\texttt{D-Safe}_{\texttt{SR}}$ and $\texttt{Earliest}_{\texttt{SR}}$ are the computation points of the simple strength reduction transformation and are denoted by the predicate $\texttt{Insert}_{\texttt{SSR}}$.

3.2.2 Updating

Similar to the code motion transformation, the simple strength reduction transformation also stores the value of $v * c$ in an auxiliary variable $\mathbf{h}$, and replaces all (original) occurrences of $v * c$ by $\mathbf{h}$. However, strength reduction additionally requires inserting update assignments for $\mathbf{h}$ in some of the nodes satisfying the predicate *Injured* defined in Section 2. These nodes are characterized by the least solution of Equation System 3.3, which is obtained by means of a backward analysis of G.

Equation System 3.3 (UPDATE)

$$\mathbf{UPDATE}(n) = \mathit{Used}(n) \;\vee \sum_{m \in succ(n)} (\; \neg\texttt{Insert}_{\texttt{SSR}}(m) \wedge \mathbf{UPDATE}(m) \;)$$

---

[11] In contrast to the strong safety requirement of code motion this modification may lead to the introduction of new values on a path, and therefore may enlarge the potential of runtime errors (cf. [26]).

[12] Here and in the following, the indices "CM" and "SR" are used in order to distinguish the code motion and strength reduction version of down-safe and earliest.

3.2.3 The Simple Strength Reduction Transformation

Denoting the least solution of Equation System 3.3 by $\mathtt{Update}_{\mathtt{SSR}}$, the predicate $\mathtt{InsUpd}_{\mathtt{SSR}}$ defined as the conjunction of the predicates *Injured* and $\mathtt{Update}_{\mathtt{SSR}}$ characterizes those program points where the auxiliary variable must be updated. Together, the predicates $\mathtt{Insert}_{\mathtt{SSR}}$ and $\mathtt{InsUpd}_{\mathtt{SSR}}$ induce the simple strength reduction transformation.

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node satisfying
   (a) $\mathtt{Insert}_{\mathtt{SSR}}$ the assignment $\mathbf{h} := v * c$
   (b) $\mathtt{InsUpd}_{\mathtt{SSR}}$ the assignment $\mathbf{h} := \mathbf{h} + \mathit{Eff}(n)$ [13]
3. Replace every (original) occurrence of $v * c$ in G by $\mathbf{h}$

Table 2: The Simple Strength Reduction Transformation

Figure 3 shows the result of this transformation for the flowgraph of Figure 1, which would also be delivered by the algorithm of [2]. In fact, the simple strength reduction transformation and the transformation of [2] coincide. However, the algorithm proposed here is a composition of three unidirectional analyses computing one predicate each, whereas the algorithm of [2] is bidirectional and requires more than twice as many predicates.

3.3 Deficiencies of the Simple Strength Reduction Transformation

The strength reduction algorithm developed so far is particularly simple: It straightforwardly evolves from an extension of a code motion algorithm, and it requires no more than three predicates. However, like the algorithm of [2], it suffers from the following deficiencies:

1. `Multiplication-Addition Deficiency`:
   There may be paths, on which both multiplications and additions are inserted. In the example of Figure 3 this happens on the path $(\mathbf{13}, \mathbf{16}, \mathbf{10}, \mathbf{4}, \mathbf{7}, \mathbf{11})$, where a multiplication is inserted in node **10** and an addition in node **4**. In this example, this even impairs the runtime efficiency of this path, since only one multiplication is saved, but a multiplication and an addition are inserted.

2. `Lifetime Deficiency`:
   The lifetimes of moved computations may be unnecessarily long due to unnecessary code motion. For example in the flowgraph of Figure 3 the initialization of $\mathbf{h}$ at node **1** is unnecessarily early and should be delayed to node **8** in order to avoid unnecessary register pressure. Moreover, rather than being moved to node **20**, the computation of $i * 10$ at node **21** should not be touched at all, because a profitable movement is impossible.

3. `Multiple-Addition Deficiency`:
   There may be paths, on which unnecessarily many additions are inserted. Consider for example the path $(\mathbf{14}, \mathbf{17}, \mathbf{15}, \mathbf{12}, \mathbf{6}, \mathbf{9}, \mathbf{11})$. There, the costs for the three inserted additions may easily exceed the costs for the saved multiplication, and therefore even impair the runtime efficiency.

[13] If both $\mathtt{Insert}_{\mathtt{SSR}}$ and $\mathtt{InsUpd}_{\mathtt{SSR}}$ hold, the initialization statement $\mathbf{h} := v * c$ must precede the update assignment $\mathbf{h} := \mathbf{h} + \mathit{Eff}(n)$.

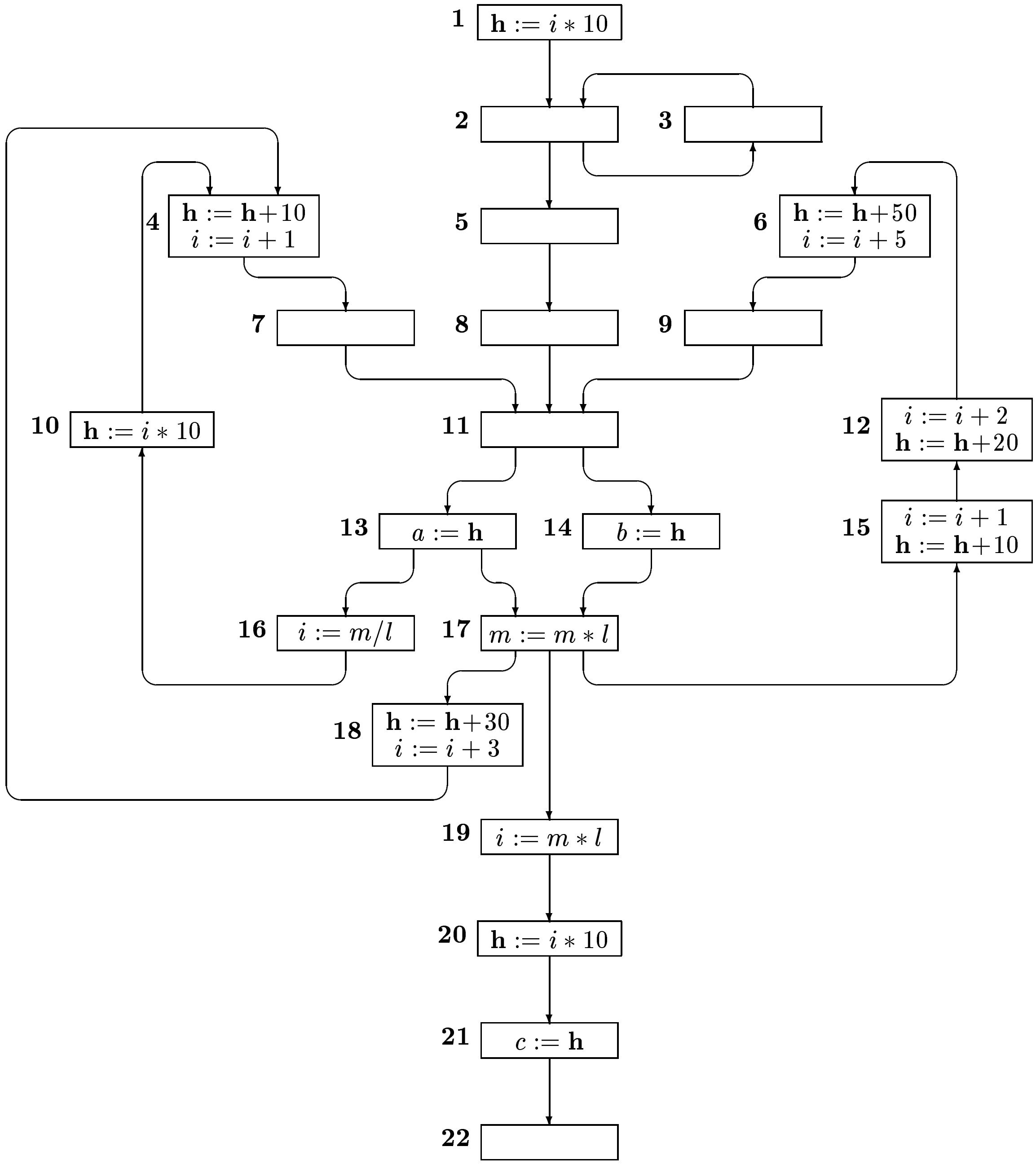

Figure 3: The Simple Strength Reduction Transformation

4 Lazy Strength Reduction

In this section we stepwise refine the simple strength reduction transformation in order to overcome the three deficiencies mentioned in Section 3.3.

4.1 First Refinement: Avoiding the Multiplication-Addition Deficiency

4.1.1 Critical Program Points

In order to avoid insertions of multiplications and additions on the same path, we determine the set of *critical* program points. Intuitively, a program point is critical, if there is a $v * c$-free program path from this point to a modification of v. The idea of our modification is to move critical insertion points in the direction of the control flow to "earliest" noncritical positions.

Technically, the set of *critical* program points is characterized by the least solution of Equation System 4.1, whose computation requires a backward analysis of G.

Equation System 4.1 (CRITICAL)

$$\mathbf{CRITICAL}(n) \quad = \quad \neg \mathit{Used}(n) \wedge (\neg \mathit{Transp}(n) \vee \sum_{m \in succ(n)} \mathbf{CRITICAL}(m))$$

4.1.2 Substituting Critical Insertion Points

Let `Critical` denote the least solution of Equation System 4.1. The existence of critical insertion points for $v * c$ (i.e., of nodes n satisfying the predicate $\mathtt{CritIns}_{\mathtt{SSR}}(n) =_{df} \mathtt{Insert}_{\mathtt{SSR}}(n) \wedge \mathtt{Critical}(n)$) characterizes the situations in which the simple strength reduction transformation would insert multiplications as well as additions on some program paths. In these situations, the critical computations of $v * c$ must be replaced by noncritical ones. This is realized by moving them in the direction of control flow until all paths to a first use of $v * c$ are transparent for v instead of only SR-transparent. Technically, this is accomplished by determining the least solution of Equation System 4.2, which specifies a forward analysis of G.

Equation System 4.2 (SUBST-CRIT)

$$\mathbf{SUBST\text{-}CRIT}(n) \quad = \quad \mathtt{CritIns}_{\mathtt{SSR}}(n) \vee \sum_{m \in pred(n)} (\neg \mathit{Used}(m) \wedge \mathbf{SUBST\text{-}CRIT}(m))$$

We denote the least solution of Equation System 4.2 by `Subst-Crit`. Code motion insertion points[14] satisfying `Subst-Crit` are, in fact, the "earliest" substitutes of critical insertion points of the simple strength reduction transformation guaranteeing that multiplications and additions are not simultaneously inserted on program paths.

4.1.3 The First Refinement

Let $\mathtt{Insert}_{\mathtt{FstRef}}$ be defined by

$$\forall n \in N. \, \mathtt{Insert}_{\mathtt{FstRef}}(n) =_{df} (\mathtt{Insert}_{\mathtt{SSR}}(n) \wedge \neg \mathtt{Critical}(n)) \;\vee\; (\mathtt{Insert}_{\mathtt{CM}}(n) \wedge \mathtt{Subst\text{-}Crit}(n))$$

and let $\mathtt{Update}_{\mathtt{FstRef}}$ be defined as the least solution of Equation System 3.3 using $\mathtt{Insert}_{\mathtt{FstRef}}$ in place of $\mathtt{Insert}_{\mathtt{SSR}}$. With $\mathtt{InsUpd}_{\mathtt{FstRef}}$ defined analogously as in the simple strength reduction transformation, i.e., $\forall n \in N. \, \mathtt{InsUpd}_{\mathtt{FstRef}}(n) =_{df} \mathit{Injured}(n) \wedge \mathtt{Update}_{\mathtt{FstRef}}(n)$, the following three-step procedure specifies the first refinement, which overcomes the multiplication-addition deficiency (cf. Section 3.3).

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node satisfying
   (a) $\mathtt{Insert}_{\mathtt{FstRef}}$ the assignment $\mathbf{h} := v * c$
   (b) $\mathtt{InsUpd}_{\mathtt{FstRef}}$ the assignment $\mathbf{h} := \mathbf{h} + \mathit{Eff}(n)$
3. Replace every (original) occurrence of $v * c$ in G by $\mathbf{h}$

Table 3: The First Refinement

[14] I.e., nodes satisfying $\mathtt{Insert}_{\mathtt{CM}}$.

Figure 4 shows the result of this transformation for the flowgraph of Figure 1, which coincides with the result of the bidirectional algorithm of [3] that enhances the algorithm of [2]. Here, the initialization of **h** at node **10**, and the update assignments at nodes **4** and **18** of Figure 3 are replaced by a single initialization of **h** at node **7**. However, as in Figure 3, the lifetimes of moved computations are still unnecessarily long, and on path $(\mathbf{14}, \mathbf{17}, \mathbf{15}, \mathbf{12}, \mathbf{6}, \mathbf{9}, \mathbf{11})$ unnecessarily many update assignments for **h** are inserted.

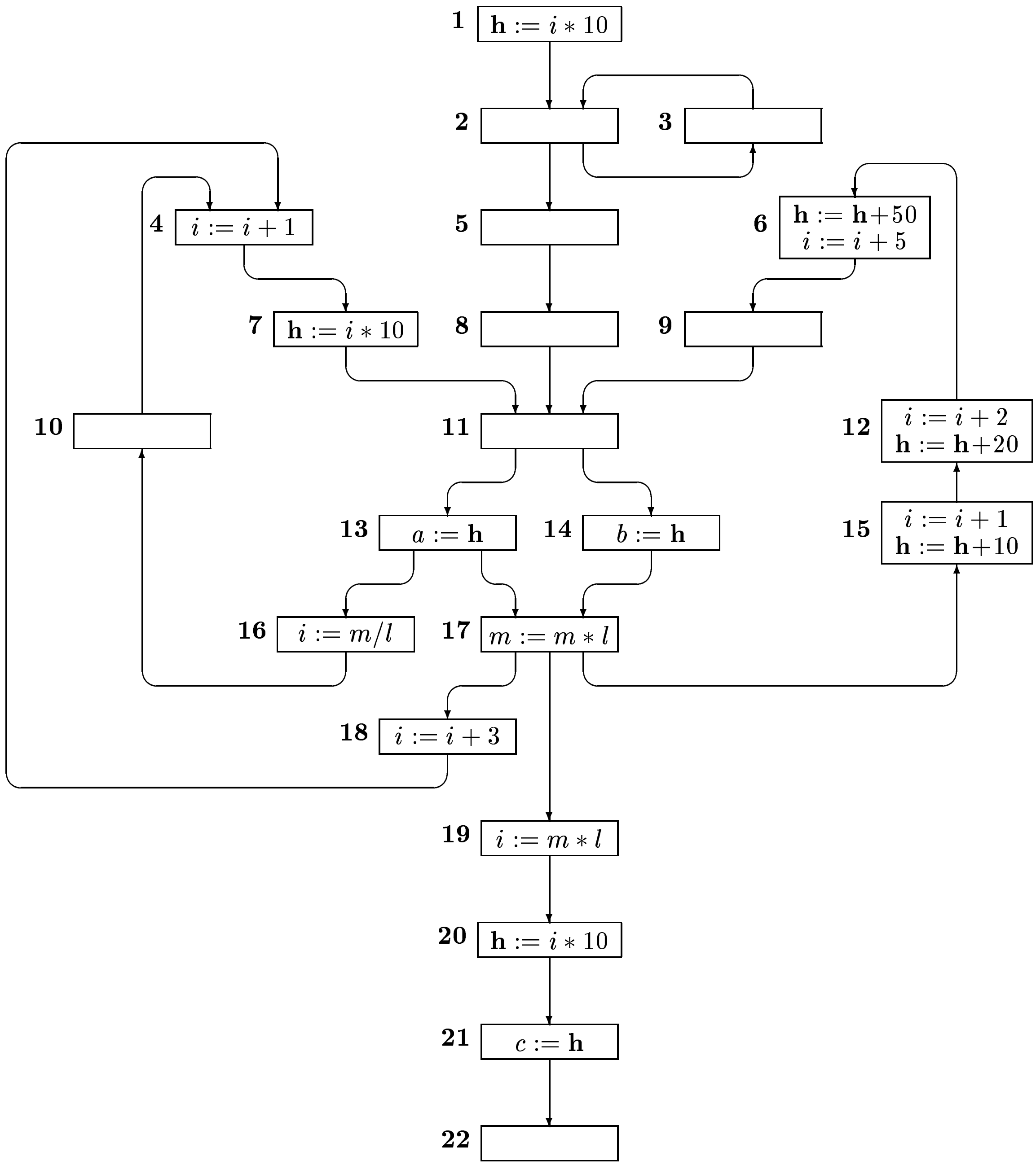

Figure 4: The First Refinement

4.2 Second Refinement: Avoiding the Lifetime Deficiency

In this section we refine the strength reduction transformation further in order to overcome the lifetime deficiency mentioned in Section 3.3.

4.2.1 Latestness

In order to minimize the lifetimes of moved computations, they must be placed as *late* as possible, while maintaining the benefits of the algorithm developed so far. Intuitively, this requires to move computations from their earliest down-safe noncritical computation points in the direction of control flow to "later" computation points. Technically, this is realized by determining the greatest solution of Equation System 4.3, which requires a forward analysis of G.

Equation System 4.3 (DELAY)

$$\mathbf{DELAY}(n) \;=\; \mathtt{Insert}_{\mathtt{FstRef}}(n) \vee \begin{cases} \mathit{false} & \text{if } n = \mathbf{s} \\ \prod\limits_{m \in pred(n)} (\neg\, \mathit{Used}(m) \;\wedge\; \mathbf{DELAY}(m)) & \text{otherwise} \end{cases}$$

The intuition behind the definition of **DELAY** is to move computations from their earliest down-safe noncritical computation points as far as possible in the direction of control flow. Thus, $\mathtt{Insert}_{\mathtt{FstRef}}$ implies **DELAY**. This movement must stop in nodes n that have a predecessor m containing a computation of $v * c$, or for which the process of moving is not successful, i.e., where $\mathbf{DELAY}(m)$ does not hold. In the first case, we would miss replacing an original occurrence of $v * c$, and in the second case a partial redundancy would be introduced into the program.

Latest Computation Points

Let `Delay` denote the greatest solution of Equation System 4.3. Then we define

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

The second refinement does not affect the insertion of update assignments. Thus, $\mathtt{InsUpd}_{\mathtt{SndRef}}$ coincides with the version from the first refinement, i.e., $\mathtt{InsUpd}_{\mathtt{SndRef}} =_{df} \mathtt{InsUpd}_{\mathtt{FstRef}}$. Together `Latest` and $\mathtt{InsUpd}_{\mathtt{SndRef}}$ specify a program transformation, where the resulting program is of the same computational complexity as that of the first refinement. However, it is more economic with respect to the lifetimes of auxiliary variables, as shown in Figure 5. Note, however, that the flowgraph of Figure 5 still contains an unnecessary initialization of **h** in node **21**, which is only used in the insertion node itself.

4.2.2 Isolation

In order to avoid unnecessary initializations as in node **21** of Figure 5, we determine all program points where an inserted computation would be *isolated*, i.e., where an inserted computation would only be used in the insertion node itself (cf. [1]). This is achieved by determining the greatest solution of Equation System 4.4, which specifies a backward analysis of G. Note that this analysis does not depend on the number of occurrences of the candidate expression in an insertion node itself, since expression evaluation without multiple calculations of common subexpressions is well understood in code generation (cf. [33]).

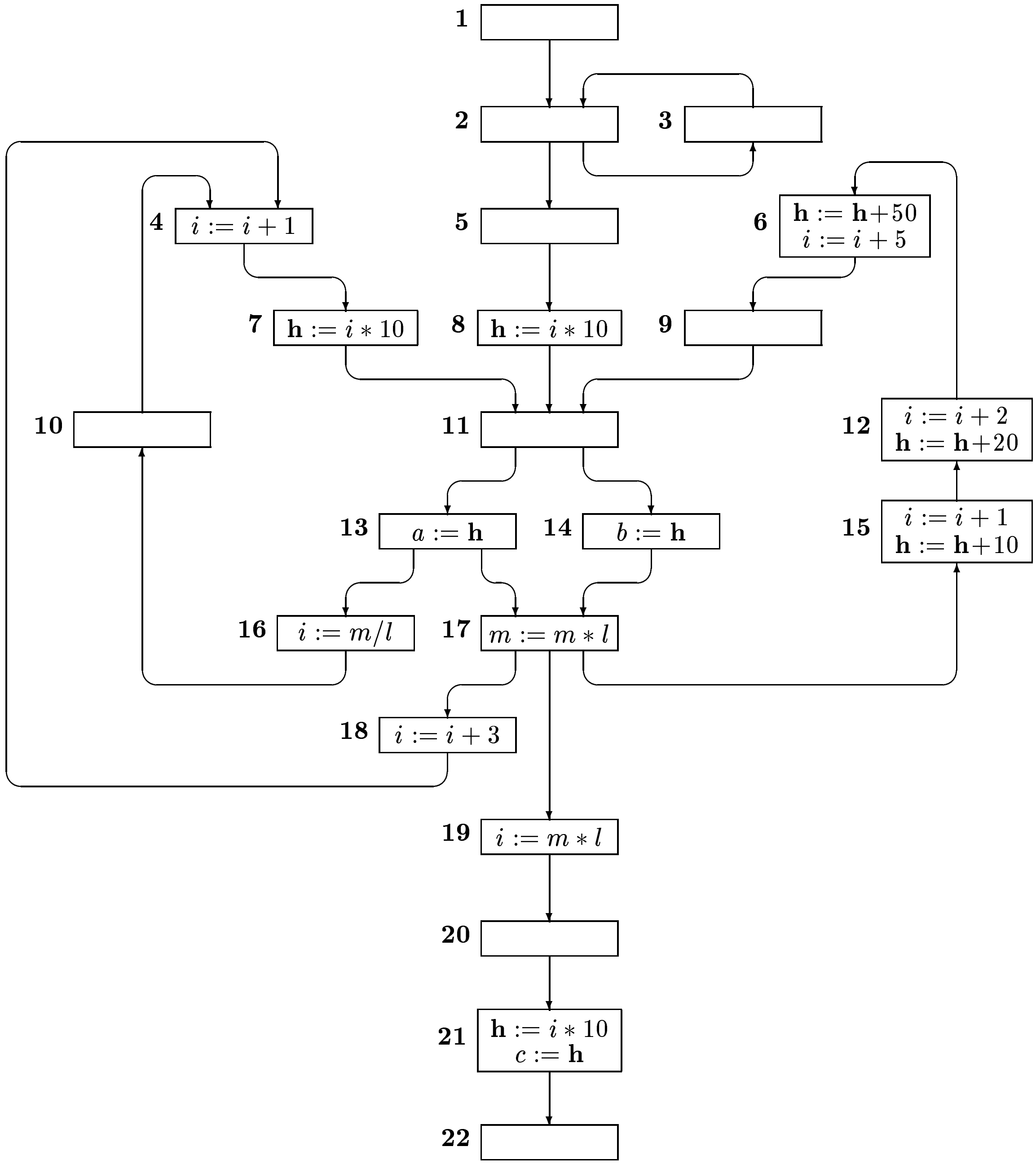

Figure 5: The Latest-Update$_{\texttt{SndRef}}$ Transformation

Equation System 4.4 (ISOLATED)

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

4.2.3 The Second Refinement

Nodes satisfying `Latest` and $\neg$`Isolated`, where `Isolated` denotes the greatest solution of Equation System 4.4, specify the optimal computation points for $v * c$ in G, and are denoted by the predicate $\texttt{Insert}_{\texttt{SndRef}}$. In contrast to the preceding strength reduction transformations, the occurrences of $v * c$ satisfying `Latest` and `Isolated` are no longer replaced by $\mathbf{h}$, because their corresponding initializations of $\mathbf{h}$ are suppressed for efficiency reasons. All other original occurrences of $v * c$, however, are redundant with respect to the computation points given by $\texttt{Insert}_{\texttt{SndRef}}$, and can be eliminated. This is indicated by the predicate $\texttt{Delete}_{\texttt{SndRef}}$.

Now the second refinement is obtained by the following three-step procedure, which transforms the flowgraph of Figure 1 into the one shown in Figure 6. In fact, our algorithm is unique in performing this transformation.

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node satisfying
   (a) $\texttt{Insert}_{\texttt{SndRef}}$ the assignment $\mathbf{h} := v * c$
   (b) $\texttt{InsUpd}_{\texttt{SndRef}}$ the assignment $\mathbf{h} := \mathbf{h} + \mathit{Eff}(n)$
3. Replace every (original) occurrence of $v * c$ in nodes satisfying $\texttt{Delete}_{\texttt{SndRef}}$ by $\mathbf{h}$

Table 4: The Second Refinement

4.3 Third Refinement: Avoiding the Multiple-Addition Deficiency

The flowgraph of Figure 6 still contains unnecessarily many update assignments for $\mathbf{h}$ (see path $(\mathbf{14}, \mathbf{17}, \mathbf{15}, \mathbf{12}, \mathbf{6}, \mathbf{9}, \mathbf{11})$). Inside *extended basic blocks*[15], however, the effect of additive modifications of v onto the value of $v * c$ can be accumulated in update assignments inserted at use sites of $v * c$ or at the end of the blocks. The effect of such accumulation is illustrated by means of the differences in Figure 7 and 8, where it is assumed that the original righthand side term of the assignments at node **1** and **2** is $i * 10$. Figure 7 shows the result of inserting an update assignment for every additive modification of i, as it is realized by the strength reduction transformation developed so far, and which still suffers from the multiple-addition deficiency. In contrast, Figure 8 shows the result of accumulating the effects of the update assignments.

Our algorithm for lazy strenth reduction is unique in overcoming the multiple-addition-deficiency by accumulating the effects of update assignments. In [3] for every modification of the candidate expression, an update assignment is inserted. However, in contrast to [2] the insertion of update assignments is controlled by a machine-dependent parameter indicating, which number of updates is faster than a recomputation of the value. If this number is exceeded on a path, a recomputation of the value is inserted instead of the sequence of updates. In [4] this parameter is set to 1. Thus, in the example of Figure 1 Dhamdhere's algorithm would insert the assignment $\mathbf{h} := i * 10$ instead of $\mathbf{h} := \mathbf{h} + 80$ in node **9**, and therefore would not achieve any strength reduction.[16]

4.3.1 Accumulation

The accumulation of update assignments requires the predicate `Accumulating` that characterizes the set of program points, where an accumulating update assignment must potentially be inserted.

$$\forall n \in N.\ \texttt{Accumulating}(n) =_{df} \texttt{Update}_{\texttt{SndRef}}(n) \wedge (\ \mathit{Used}(n) \vee \mathit{ExitExtdBscBlck}(n)\ )$$

[15] A *basic block* is a maximal sequence of code, where at most the first node has more than one predecessor, and at most the last node more than one successor (cf. [25]). An *extended basic block* is a maximal sequence of code, in which at most the first node has more than one predecessor, and all sons of nodes with more than one successor have a unique predecessor. Thus, an extended basic block is a maximal *tree* of nodes, such that control can enter the tree only at its root, and a basic block is a maximal tree with just one leaf (cf. [30]).

[16] In fact, in the example of Figure 1 the algorithm of [4] delivers the result of the first refinement except that the updates in the right part of the loop are replaced by the multiplication in node **9**.

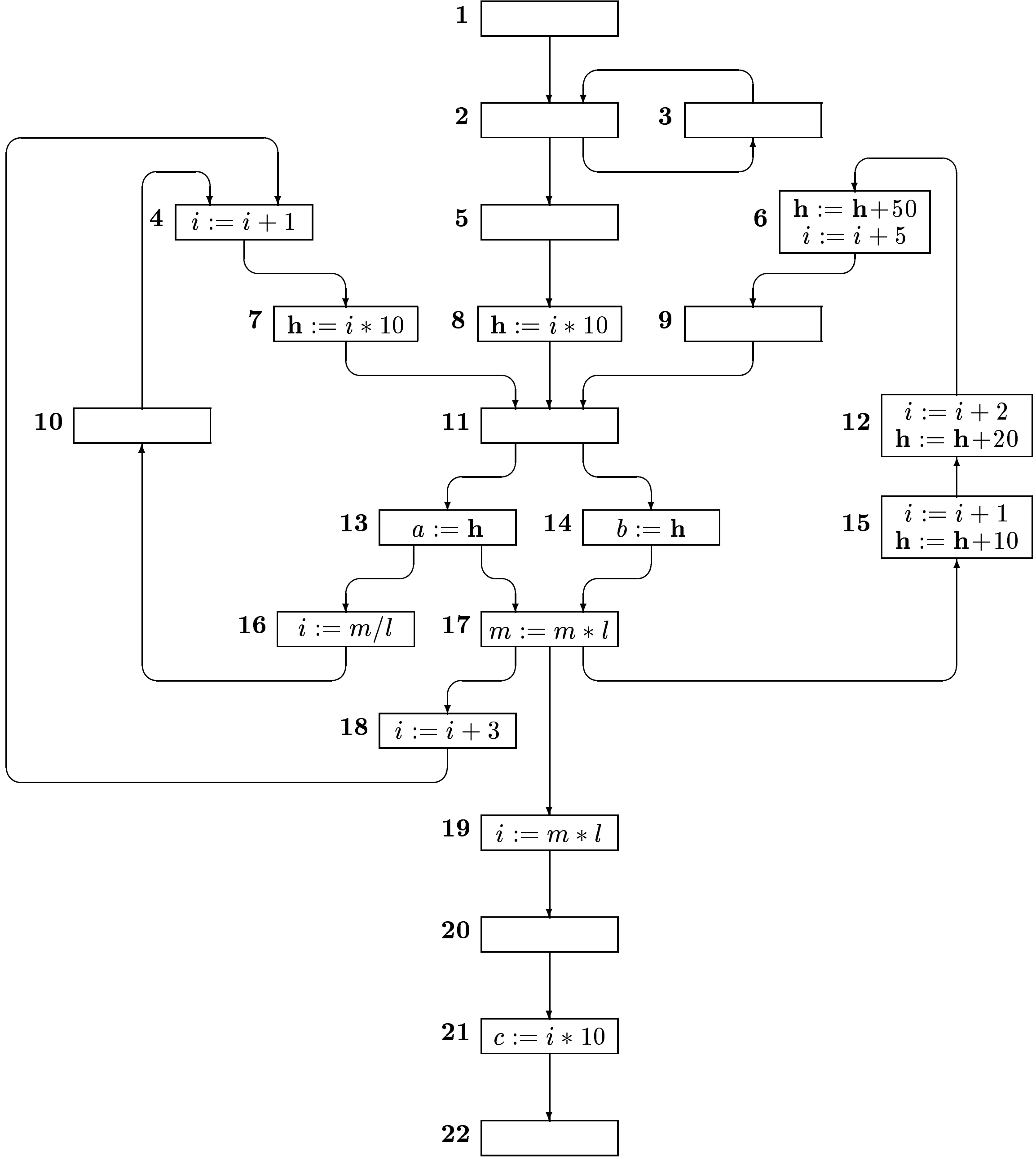

Figure 6: The Second Refinement

Here, the predicate *ExitExtdBscBlck* characterizes exit nodes of extended basic blocks. It is defined by

$$\forall\, n \in N.\ \mathit{ExitExtdBscBlck}\,(n) =_{df} (n = \mathbf{e}) \ \vee \sum_{m \in succ(n)} \mathit{EntryExtdBscBlck}\,(m)$$

where *EntryExtdBscBlck* characterizes entry nodes of extended basic blocks:

$$\forall\, n \in N.\ \mathit{EntryExtdBscBlck}\,(n) =_{df} (n = \mathbf{s}) \ \vee \sum_{m \in pred(n)} |pred(succ(m))| > 1$$

The accumulation process has to be terminated in nodes satisfying the predicate *EntryExtdBscBlck* or having a predecessor satisfying *Used*. Denoting these nodes by the pred-

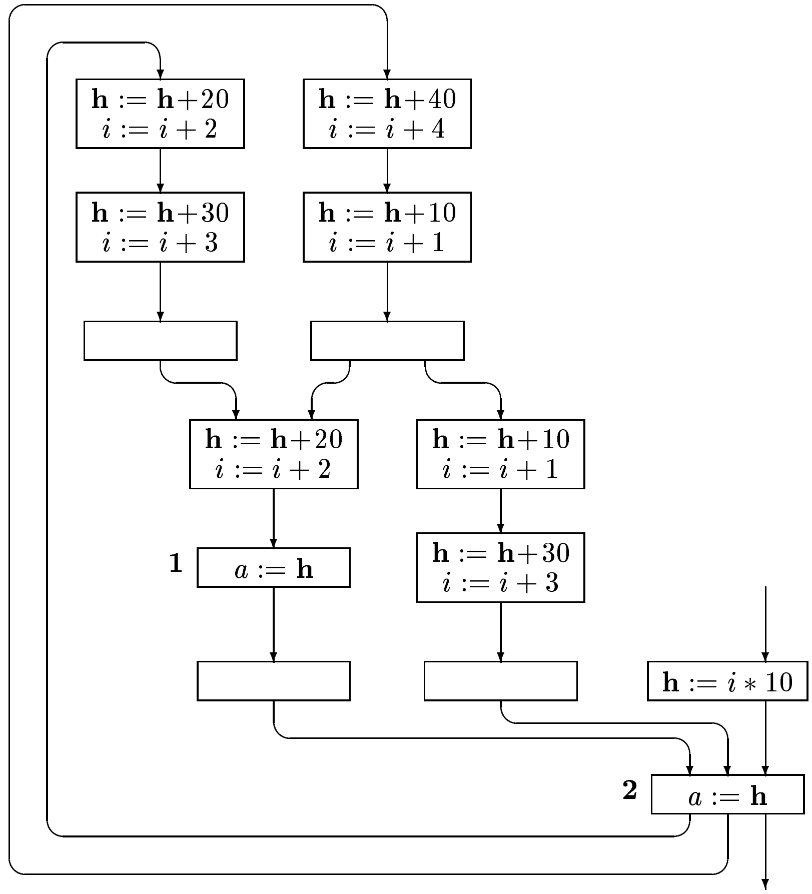

Figure 7: Illustration of the Multiple-Addition Deficiency

icate *AccumTerm*, the function $\mathit{AccumEff} : N \to \omega$ determines the accumulated effect of a sequence of update assignments.

$$\forall\, n \in N.\ \mathit{AccumEff}(n) =_{df} \begin{cases} 0 & \text{if } \neg \mathtt{Update}_{\mathtt{SndRef}}(n) \\ \mathit{Eff}(n) & \text{if } \mathtt{Update}_{\mathtt{SndRef}}(n) \ \wedge\ \mathit{AccumTerm}(n) \\ \mathit{Eff}(n) + \mathit{AccumEff}(m) & \text{otherwise } (\mathit{pred}(n) = \{m\}\ ) \end{cases}$$

Remark 4.5 In the example of Figure 8, we can even accumulate the effect of consecutive extended basic blocks in single update assignments, as shown in Figure 11 in Section A.4. The point here is that in all predecessors of the entry nodes of "sibling" extended basic blocks the same update assignment is inserted (in Figure 8: $\mathbf{h} := \mathbf{h} + 50$). This pattern can be captured in general by means of a refined version of the `Accumulating` predicate introduced above.

4.3.2 Third Refinement: The Complete Transformation

The third refinement affects only the insertion points of update assignments. Thus, the predicates characterizing the nodes where the auxiliary variable must be initialized, and where an original occurrence of $v * c$ can be deleted coincide with the corresponding predicates in the second refinement, i.e., $\mathtt{Insert}_{\mathtt{LSR}} =_{df} \mathtt{Insert}_{\mathtt{SndRef}}$ and $\mathtt{Delete}_{\mathtt{LSR}} =_{df} \mathtt{Delete}_{\mathtt{SndRef}}$. Update assignments must be inserted in nodes satisfying the predicate $\mathtt{InsUpd}_{\mathtt{LSR}}$. This predicate is true for nodes n satisfying the predicate `Accumulating` with $\mathit{AccumEff}(n) \neq 0$.

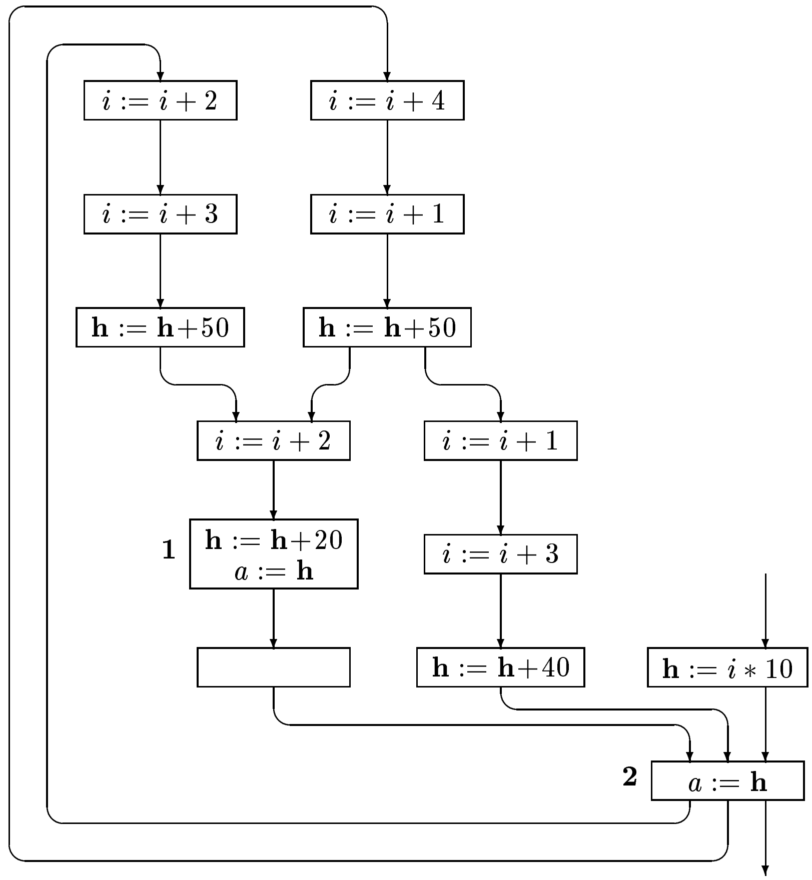

Figure 8: Accumulation of the Effects of Update Assignments

The Lazy Strength Reduction Transformation

The lazy strength reduction transformation, which overcomes all three deficiencies mentioned in Section 3.3, proceeds in three steps.

1. Introduce a new auxiliary variable $\mathbf{h}$ for $v * c$
2. Insert at the entry of every node satisfying
   (a) $\texttt{Insert}_{\texttt{LSR}}$ the assignment $\mathbf{h} := v * c$
   (b) $\texttt{InsUpd}_{\texttt{LSR}}$ the assignment $\mathbf{h} := \mathbf{h} + \mathit{AccumEff}(n)$
3. Replace every (original) occurrence of $v * c$ in nodes satisfying $\texttt{Delete}_{\texttt{LSR}}$ by $\mathbf{h}$

Table 5: The Lazy Strength Reduction Transformation

Table 6 in Section A.3 summarizes the values of the predicates considered during the development of the lazy strength reduction algorithm for the term $i * 10$ in the flowgraph in Figure 1.

Application of the lazy strength reduction transformation to the flowgraph in Figure 1 results in the promised flowgraph in Figure 2, which in fact is free of all the deficiencies discussed above.

5 Conclusions

We have presented a bit-vector algorithm for lazy strength reduction, which is unique in its transformational power in that it uniformly combines code motion and strength reduction, and completely avoids superfluous register pressure due to unnecessary code motion. Moreover, like its underlying algorithm for lazy code motion ([1]), it is composed of a sequence of unidirectional analyses. This allows us to apply the efficient algorithms for unidirectional bit-vector analyses to deal with all program terms simultaneously (cf. [6, 7, 8, 9, 10, 11, 12, 13, 14, 15]) as well as to interleave strength reduction and copy propagation using the slotwise approach of [18]. Additionally, its modular structure supports further extensions. For example, following the lines of [34, 20] it is straightforward to generalize this algorithm to programs with (mutually) recursive procedures, global and local variables, and formal (value) parameters.

References

[1] 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.

[2] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization – part I. *International Journal of Computer Mathematics*, 11:21 – 41, 1982.

[3] S. M. Joshi and D. M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization – part II. *International Journal of Computer Mathematics*, 11:111 – 126, 1982.

[4] D. M. Dhamdhere. A new algorithm for composite hoisting and strength reduction optimisation (+ Corrigendum). *International Journal of Computer Mathematics*, 27:1 – 14 (+ 31 – 32), 1989.

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

[6] A. V. Aho and J. D. Ullman. Node listings for reducible flow graphs. In *Proc. 7th ACM Symposium on the Theory of Computing*, pages 177 – 185, Albuquerque, NM, 1975.

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

[8] M. S. Hecht and J. D. Ullman. Analysis of a simple algorithm for global flow problems. In *Conf. Record of the 1st ACM Symposium on the Principles of Programming Languages*, pages 207 – 217, Boston, MA, 1973.

[9] M. S. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM Journal on Computing*, 4(4):519 – 532, 1977.

[10] 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.

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

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

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

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

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

[16] 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.

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

[18] 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.

[19] 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.

[20] J. Knoop and B. Steffen. Optimal interprocedural partial redundancy elimination. Extended abstract. In *Addenda to Proc. 4^{th} Conference on Compiler Construction (CC)*, pages 36 – 39, Paderborn, Germany, 1992. Published as Tech. Rep. No. 103, Department of Computer Science, University of Paderborn.

[21] B. Steffen. Data flow analysis as model checking. In *Proc. TACS*, Lecture Notes in Computer Science 526, pages 346 – 364, Sendai, Japan, 1991. Springer-Verlag.

[22] F. E. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 3, pages 79 – 101. Prentice Hall, Englewood Cliffs, NJ, 1981.

[23] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850 – 856, 1977.

[24] J. Cai and R. Paige. Look ma, no hashing, and no arrays neither. In *Conf. Record of the 18^{th} ACM Symposium on the Principles of Programming Languages*, pages 143 – 154, Orlando, FL, 1991.

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

[26] B. Steffen, J. Knoop, and O. Rüthing. Efficient code motion and an adaption to strength reduction. In *Proc. 4^{th} International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Lecture Notes in Computer Science 494, pages 394 – 415, Brighton, UK, 1991. Springer-Verlag.

[27] R. Paige. *Formal Differentiation - A Program Synthesis Technique.* UMI Research Press, 1981.

[28] R. Paige. Transformational programming – applications to algorithms and systems. In *Conf. Record of the 10^{th} ACM Symposium on the Principles of Programming Languages*, pages 73 – 87, Austin, TX, 1983.

[29] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402 – 454, 1982.

[30] B. K. Rosen. Degrees of availability as an introduction to the general theory of data flow analysis. chapter 2, pages 55 – 76.

[31] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conf. Record of the 15th ACM Symposium on the Principles of Programming Languages*, pages 12 – 27, San Diego, CA, 1988.

[32] B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *Proc. 3rd European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science 432, pages 389 – 405, Copenhagen, Denmark, 1990. Springer-Verlag.

[33] A. V. Aho, S. C. Johnson, and J.D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146 – 160, 1977.

[34] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proc. 4th Conference on Compiler Construction (CC)*, Lecture Notes in Computer Science 641, pages 125 – 140, Paderborn, Germany, 1992. Springer-Verlag.

A Appendix

A.1 Critical Edges

In order to exploit the full power of our lazy strength reduction algorithm, “critical” edges, i.e., edges leading from nodes with more than one successor to nodes with more than one predecessor, must be removed from the flowgraph, since they may block the process of code motion and strength reduction (cf. [17, 16, 18, 19, 1, 20, 31, 32, 26]), as illustrated in Figure 9.

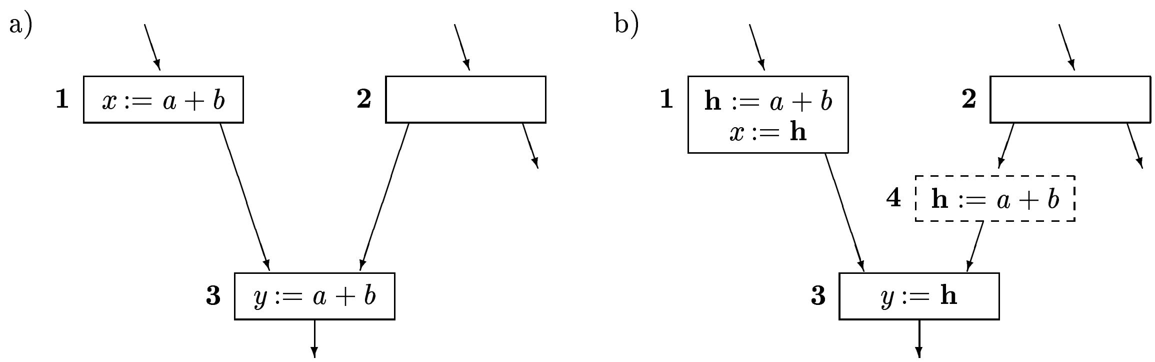

Figure 9: Critical Edges

In Figure 9(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 9(b). Inserting a synthetic node on every edge leading to a node with more than one predecessor certainly implies that all critical edges are removed, and additionally, it allows us to insert all computations uniformly at the entries of nodes.

A.2 Illustrating Earliestness

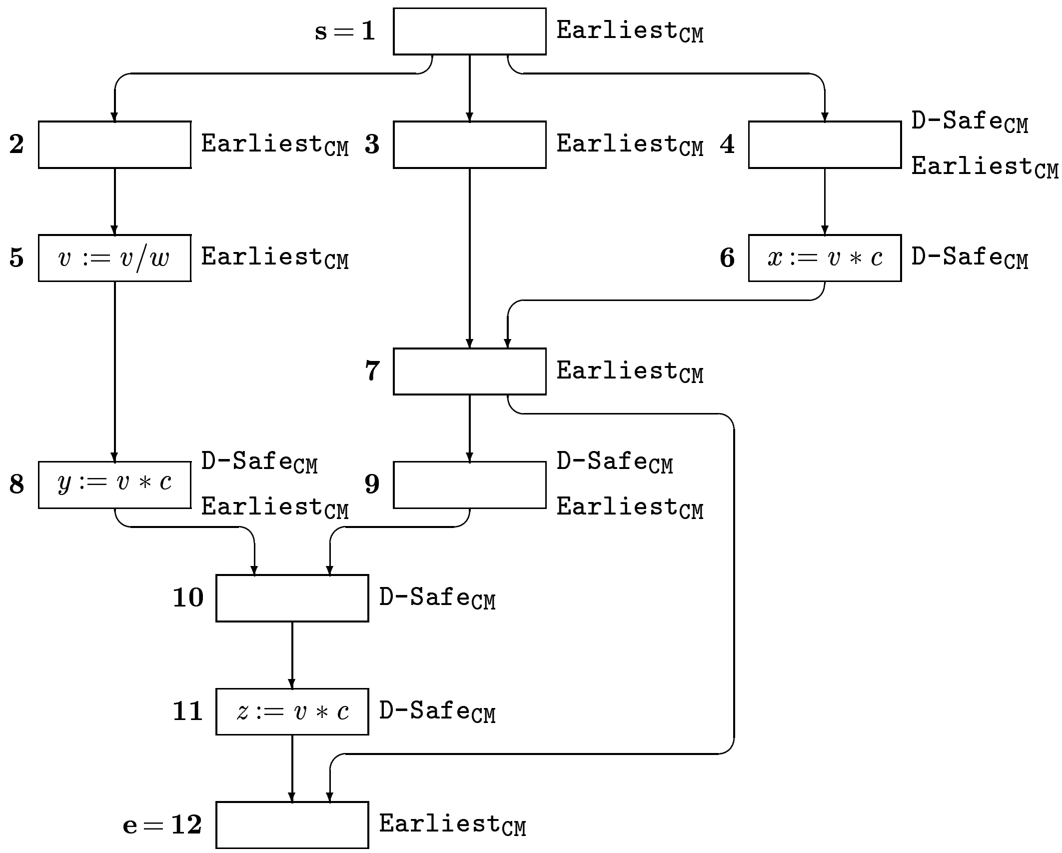

Figure 10: Illustrating Down-Safety and Earliestness

Figure 10 shows the predicate values of $\texttt{Earliest}_{\texttt{CM}}$ for a small example. This illustrates that $\texttt{Earliest}_{\texttt{CM}}$ is valid at the start node and additionally at those nodes that are reachable by a path on which a prior computation of $v*c$ would not be down-safe or delivers a different value due to a subsequent modification of v. Of course, $v*c$ cannot be placed earlier than in the start node, which justifies $\texttt{Earliest}(\mathbf{1})$. Moreover, every computation of $v*c$ in a node on the paths $(\mathbf{1},\mathbf{2})$ and $(\mathbf{1},\mathbf{3},\mathbf{7})$ would not be down-safe. Thus $\texttt{Earliest}_{\texttt{CM}}(\{\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5},\mathbf{7},\mathbf{9},\mathbf{12}\})$ holds. Finally, evaluating $v*c$ at node **1**, **2** and **5** delivers a different value as in node **8**. This yields $\texttt{Earliest}(\mathbf{8})$.

A.3 Relevant Predicate Values for the Motivating Example

	Node Number																					
Predicate	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
$\texttt{D-Safe}_{\texttt{CM}}$	1	1	1	0	1	0	1	1	1	0	1	0	1	1	0	0	0	0	0	1	1	0
$\texttt{Earliest}_{\texttt{CM}}$	1	0	0	1	0	1	1	0	1	1	0	1	0	0	0	0	0	0	0	1	0	0
$\texttt{Insert}_{\texttt{CM}}$	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
$\texttt{D-Safe}_{\texttt{SR}}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	0
$\texttt{Earliest}_{\texttt{SR}}$	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
$\texttt{Insert}_{\texttt{SSR}}$	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
`Critical`	0	0	0	1	0	1	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0	0
`Subst-Crit`	0	0	0	1	0	0	1	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0
$\texttt{Insert}_{\texttt{FstRef}}$	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
`Delay`	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0
`Latest`	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
`Isolated`	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	1	1
$\texttt{Update}_{\texttt{SndRef}}$	0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	0	1	0	0	0	1	0
$\texttt{Insert}_{\texttt{SndRef}}$	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`Accumulating`	0	0	0	0	0	0	1	1	1	0	0	0	1	1	0	0	0	0	0	0	1	0
$\texttt{Insert}_{\texttt{LSR}}$	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\texttt{InsUpd}_{\texttt{LSR}}$	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$\texttt{Delete}_{\texttt{LSR}}$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

Table 6: Relevant Predicate Values for the Motivating Example

A.4 Refined Accumulation

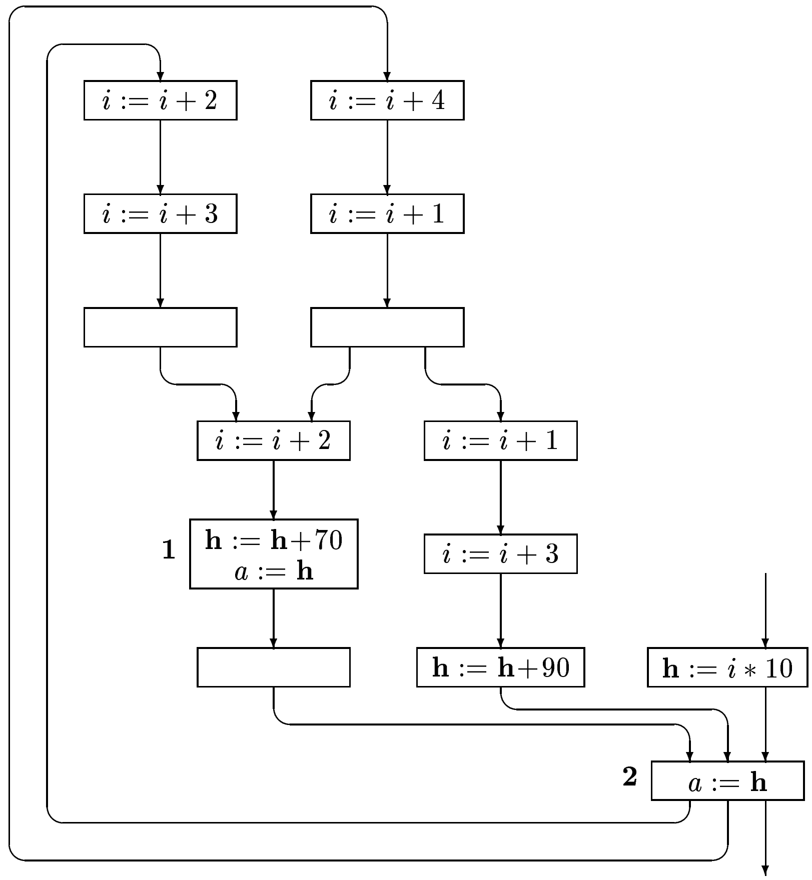

Figure 11: Refined Accumulation of the Effects of Update Assignments