Partial Dead Code Elimination

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

A new aggressive algorithm for the elimination of partially dead code is presented, i.e., of code which is only dead on some program paths. Besides being more powerful than the usual approaches to dead code elimination, this algorithm is *optimal* in the following sense: partially dead code remaining in the resulting program cannot be eliminated without changing the branching structure or the semantics of the program, or without impairing some program executions.

Our approach is based on techniques for partial redundancy elimination. Besides some new technical problems there is a significant difference here: partial dead code elimination introduces second order effects, which we overcome by means of exhaustive motion and elimination steps. The optimality and the uniqueness of the program obtained is proved by means of a new technique which is universally applicable and particularly useful in the case of mutually interdependent program optimizations.

Topics: data flow analysis, program optimization, dead code elimination, partial redundancy elimination, code motion, assignment motion, bit-vector data flow analyses.

In Proceedings of the 7^{th} ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94), Orlando, Florida, *SIGPLAN Notices 29*, 6 (1994), 147 - 158.

1 Motivation

Dead code elimination is a technique for improving the efficiency of a program by avoiding the execution of unnecessary statements at run-time. Usually, an assignment is considered unnecessary, if it is *totally* dead, i.e., if the content of its left hand side variable is not used in the remainder of the program. Thus *partially* dead assignments as the one in node **1** in Figure 1, which is dead on the left but alive on the right branch, are not considered.

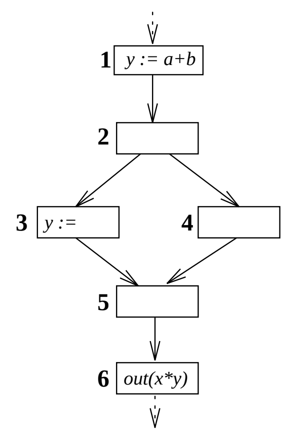

Figure 1: A Simple Motivating Example

However, by moving the assignment $y := a + b$ from node **1** to the entry of node **3** and node **4** this assignment becomes dead at node **3** and can be removed as shown in Figure 2.

We present an aggressive algorithm for *partial* dead code elimination, which optimally captures this effect: partially dead code remaining in the resulting program cannot be eliminated without changing the branching structure or the semantics of the program, or without impairing some program executions.

The point of our algorithm is to move partially dead statements as far as possible in the direction of the control flow while maintaining the program semantics. This process places the statements in an as specific context as possible, and therefore maximizes the potential of dead code, which is subsequently eliminated.

*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.

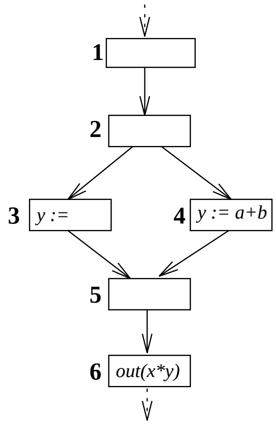

Figure 2: Partially Dead Assignment Removed

This approach is essentially dual to partial redundancy elimination [9, 11, 12, 23, 22, 26], where computations are moved against the control flow as far as possible, in order to make their effects as universal as possible. Thus similar techniques can be applied. However, moving assignments turns out to be more intricate, because both moving and eliminating assignments can mutually influence each other as illustrated in Figure 3:

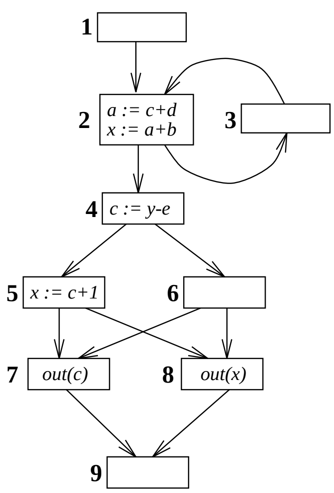

Figure 3: Illustrating Second Order Effects

The most significant inefficiency in this example program is obviously the "loop invariant" code fragment in node **2**, which cannot be removed from the loop by standard techniques for loop invariant code motion, since the first instruction defines an operand of the second assignment.[1]

[1]Note that even interleaving code motion and copy propagation as suggested in [10] only succeeds in removing the right hand side computations from the loop, but the assignment to x would remain in it.

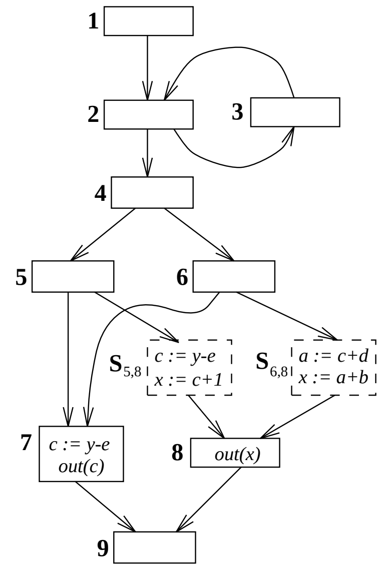

Figure 4: The Result

Our algorithm performs the optimization displayed in Figure 4 in two steps: Removing the second assignment from the loop suspends the blockade of the first assignment, which then can be removed from the loop as well. The systematic treatment of such second order effects is an important part of our algorithm.

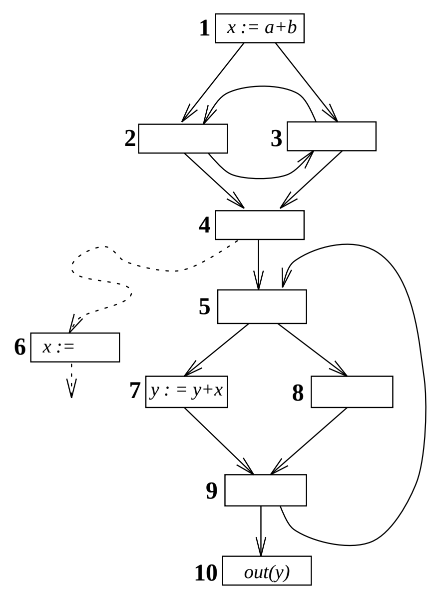

Figure 5: Illustrating the Treatment of Loops

In addition to covering second order effects, our algorithm captures arbitrary control flow structures and elegantly solves the usual problem of distinguishing between profitable code motion across loop structures and fatal code motion into loops. In fact, it guarantees that each execution of the resulting program is at least as fast as the similar execution of the original program, as the set of statements which must be executed can only be reduced. This is illustrated in the example of Figure 5, which contains two loop constructs, one of which is even irreducible.

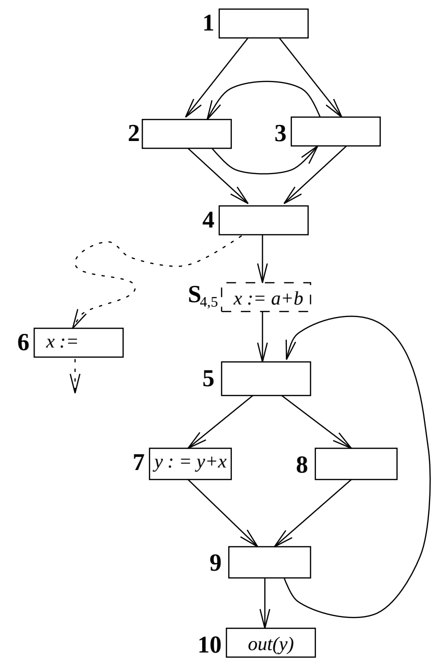

Figure 6: The Result

Figure 6 shows that our algorithm moves the assignment of node **1** across the first irreducible loop construct, removes it as dead code on the branch leading through node **6**, and inserts it into a new node $\mathbf{S}_{4,5}$ on the edge connecting node **4** and node **5**. It is worth noting that the assignment in node $\mathbf{S}_{4,5}$ is still partially dead. However, the elimination of this partially dead assignment would require to move $x := a+b$ into the second loop, which would dramatically impair some program executions.

Related Work

The idea of assignment sinking and its use in dead code elimination in a global optimizer is already sketched in [25]. However, this algorithm is restricted to a few special control flow patterns, and does not address the general problem at all. Moreover, in [9] Dhamdhere proposed an extension of partial redundancy elimination to assignment movement, where, in contrast to our approach, assignments are hoisted rather than sunk, which does not allow any elimination of partially dead code.

Recently, Feigen et al. pointed to the importance of partial dead code elimination [13]. Their algorithm is characterized by considering more complex statements as movement candidates whenever the elementary statements are blocked. Thus, in contrast to the usual code motion algorithms, it may modify the branching structure of the program under consideration. However, the algorithm sketched and discussed in their paper is not capable of moving statements out of loops or even across loops. Moreover, it only considers transformations that place one occurrence of a possibly complex partially dead statement at a single later point where it is live. This restriction forbids some attractive optimizations. For instance, in Figure 3 the assignment $c := y - e$ could not be removed from node **4**, and, as a consequence, all second order movements are missed as well. This example could possibly be dealt with by an extension of their algorithm which is vaguely mentioned in their paper. However, even this extension would still fail to capture movements that require the simultaneous treatment of several occurrences of a specific pattern. For instance, in Figure 7 the partially dead assignments of $a := a + 1$ at node **1** and node **2** can only be eliminated by a simultaneous treatment of both occurrences.

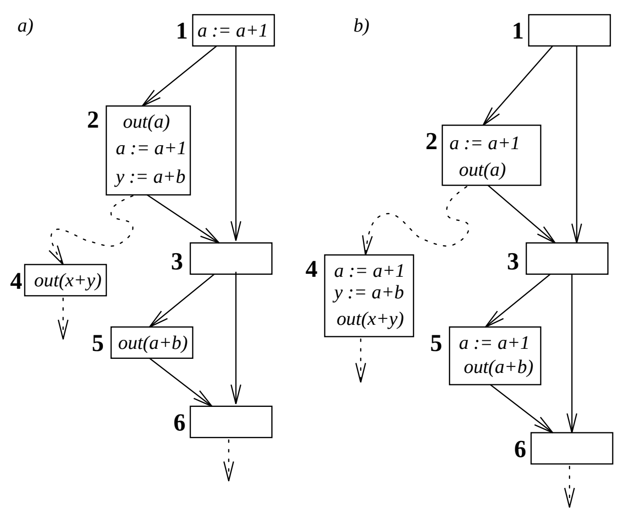

Figure 7: Illustrating m-to-n Sinkings

Briggs' and Cooper's algorithm [4] published in this proceedings employs instruction sinking for the reassociation of expressions. As a by-product some partially dead assignments can be removed. However, in contrast to our algorithm their strategy of instruction sinking can significantly impair certain program executions, since instructions can be moved into loops in a way which cannot be 'repaired' by a subsequent partial redundancy elimination. For example in Figure 6 their algorithm would sink the instruction of node $\mathbf{S}_{4,5}$ into the loop to node **7**. Note that subsequent partial redundancy elimination fails to hoist it back because of safety reasons.

In [7, 8] Dhamdhere presents an application of code hoisting and sinking techniques to *register assignment* which, however, does not provide a contribution to the general problem of partial dead code elimination.

Finally, *instruction scheduling* techniques are usually restricted to basic blocks or for-loops and focus on specific goals of code generation, for instance to yield short evaluation sequences with respect to some machine model [1, 28], or to prepare the code for efficient execution on a parallel or pipelined machine [3, 15].

Structure of the Paper

The paper develops along the following lines. After the preliminary Section 2, Section 3 presents the central notions of our approach and establishes the essential features of partial dead code elimination. Subsequently, Section 4 gives a detailed discussion about second order effects, and Section 5 develops our algorithm. Finally, a complexity estimation is presented in Section 6 and conclusions are drawn in Section 7.

2 Preliminaries

We consider *variables* $x \in \mathbf{V}$, *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 statements, 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 both assumed to represent the empty statement skip and not to possess any predecessors and successors, respectively.

Statements are classified into the following three groups: the *assignment statements* of the form $v := t$, the *empty statement* skip, and the *relevant statements* forcing all their operands to be alive. For the ease of presentation, the relevant statements are given by explicit output operations of the form $out(t)$ here.[2] We will further use the notion lhs_ι to refer to the left hand side variable of an assignment statement ι.

Moreover, $succ(n) =_{df} \{ m \mid (n,m) \in E \}$ and $pred(n) =_{df} \{ m \mid (m,n) \in E \}$ denote the set of all successors and predecessors of a node n, respectively. 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 $\mathbf{e}$. Finally, an *assignment pattern* α is a string of the form $x := t$. The set of all assignment patterns (occurring in a program) is denoted by $\mathcal{AP}$.

[2] In practice, conditions in if-statements and assignments to global variables (i.e., variables whose declaration is outside the scope of the flow graph under consideration) must be considered relevant as well. It is straightforward to extend our approach accordingly.

2.1 Critical Edges

Like partial redundancy elimination also partial dead code elimination can be blocked by *critical edges* in 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. [6, 10, 23, 22]).

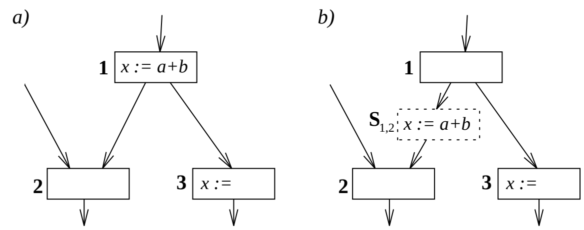

Figure 8: Critical Edges

In Figure 8(a) the assignment $x := a + b$ at node **1** is partially dead with respect to the assignment at node **3**. However, this partially dead assignment cannot safely be eliminated by moving it to its successors, because this may introduce a new assignment on a path entering node **2** on the left branch. On the other hand, it can safely be eliminated after inserting a synthetic node $\mathbf{S}_{1,2}$ in the critical edge $(\mathbf{1}, \mathbf{2})$, as illustrated in Figure 8(b).

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

3 Partial Dead Code Elimination

Conceptually, *partial dead code elimination* stands for any sequence of

- *assignment sinkings* and
- *dead code eliminations*

as formally defined below.

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

- *eliminates some occurrences of α,*

- *inserts instances of α at the entry or the exit of some basic blocks being reachable from a basic block with an eliminated occurrence of α.*

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

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

Thus, we define:

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

1. *The removed assignments are* substituted*, i.e., on every program path leading from n to* **e***, where an occurrence of α has been eliminated at n, an instance of α has been inserted at a node m on the path such that α is not blocked by any instruction between n and m.*

2. *The inserted assignments are* justified*, i.e., on every program path leading from* **s** *to n, where an instance of α has been inserted at n, an occurrence of α has been eliminated at a node m on the path such that α is not blocked by any instruction between m and n.*

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 sinking of assignments also their elimination must be admissible, which leads to the notion of dead assignments. An occurrence of an assignment pattern $\alpha \equiv x := t$ in a basic block n is *dead*, if its left-hand side variable x is *dead*, i.e., on every path from n to **e** every right-hand side occurrence of x following the considered instance of α is preceded by a modification of x. This simple definition, however, is too strong in order to characterize all assignments that are of no use for any relevant computation. The following recursive definition yields such a characterization. An occurrence of an assignment pattern $\alpha \equiv x := t$ in a basic block n is *faint* (cp. [16, 18]), if its left-hand side variable x is *faint*, i.e. on every path from n to **e** every right-hand side occurrence of x following the instance of α is either preceded by a modification of x or is in an assignment whose left hand side variable is faint as well. The following example taken from [18] shows a faint assignment which is out of the scope of dead code elimination.

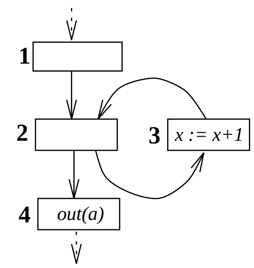

Figure 9: A Faint but not a Dead Assignment

Thus faint code elimination is more powerful than dead code elimination. On the other hand, in contrast to faint code elimination dead code elimination can be based on an efficient bit-vector data flow analysis. We therefore consider both techniques in the sequel.

Definition 3.4 (Dead (Faint) Code Elim.) *A* dead (faint) code elimination *for an assignment pattern α is an assignment elimination for α, where some dead (faint) occurrences of α are eliminated.*

It is worth noting that any admissible assignment sinking preserves the program semantics. This is not true for assignment eliminations. In fact, even dead (faint) code eliminations may change the semantics of a program by reducing the potential of run-time errors.[3]

However, these are the only possible changes of the semantics induced by dead (faint) code elimination. In particular, the evaluation of every program instruction which remains in the program is guaranteed to behave exactly as before.

Definition 3.5 (Part. Dead (Faint) Code Elim.) Partial dead (faint) code elimination *PDE (PFE) is an arbitrary sequence of admissible assignment sinkings and dead (faint) code eliminations.*

In the following we will write $G \vdash_{PDE} G'$ ($G \vdash_{PFE} G'$) if the flow graph G' results from applying an admissible assignment sinking or a dead (faint) code elimination to G. For a given flow graph G we denote the *universe* of programs resulting from partial dead (faint) code elimination $\tau \in \{PDE, PFE\}$ by

[3]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.

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

For the rest of this section let $\tau \in \{PDE,\ PFE\}$. A key notion of this paper then is:

Definition 3.6 (Optimality of PDE (PFE))

1. *Let $G', G'' \in \mathcal{G}_\tau$. Then G' is* better[4] *than G'', in signs $G'' \precsim G'$, if and only if*

   $$\forall\, p \in \mathbf{P}[\mathbf{s}, \mathbf{e}]\ \forall \alpha \in \mathcal{AP}.\ \alpha\#(p_{G'}) \leq \alpha\#(p_{G''})$$

   where $\alpha\#(p_{G'})$ and $\alpha\#(p_{G''})$ denote the number of occurrences of the assignment pattern α on p in G' and G'', respectively.[5]

2. *$G^* \in \mathcal{G}_\tau$ is* optimal *if and only if G^* is better than any other program in $\mathcal{G}_\tau$.*

The relation 'better' is a pre-order on $\mathcal{G}_\tau$, i.e., it is reflexive and transitive (but not antisymmetric). Hence, there may be several programs being optimal in the sense of Definition 3.6. On the other hand, it is not obvious that $\mathcal{G}_\tau$ has an optimal element at all. We will therefore present a constructive criterion guaranteeing the existence of an optimal element. This criterion, which is based on a slight generalization of Tarski's Fixpoint Theorem, is tailored to deal with mutually interdependent (program) transformations. In this setting, we consider the partial order $\vec{\sqsubseteq}_\tau$ on $\mathcal{G}_\tau$ defined by $\vec{\sqsubseteq}_\tau =_{df} (\precsim \cap \vdash_\tau)^*$, and a finite family of functions $\mathcal{F}_\tau \subseteq \{f \mid f : \mathcal{G}_\tau \to \mathcal{G}_\tau\}$ satisfying:

1. *Dominance:*
   $\forall\, G', G'' \in \mathcal{G}_\tau.\ G' \vdash_\tau G'' \Rightarrow \exists\, f \in \mathcal{F}_\tau.\ G'' \,\vec{\sqsubseteq}_\tau\, f(G')$

2. *Monotonicity:*
   $\forall\, G', G'' \in \mathcal{G}_\tau\ \forall\, f \in \mathcal{F}_\tau.$
   $$G' \,\vec{\sqsubseteq}_\tau\, G'' \Rightarrow f(G') \,\vec{\sqsubseteq}_\tau\, f(G'')$$

Given such a family of functions $\mathcal{F}_\tau$, we can apply the generalized version of Tarski's Fixed Point Theorem presented in [14], in order to obtain:

Theorem 3.7 (Existence of Optimal Programs) *$\mathcal{G}_\tau$ has an optimal element (wrt $\precsim$) which can be computed by any sequence of function applications that contains all elements of $\mathcal{F}_\tau$ 'sufficiently' often.*

The optimal program is not unique. However, one can show that there exists a canonical representative which is unique up to some reorderings in basic blocks.

[4] Note that this relation is reflexive. In fact, *at least as good* would be the more precise but uglier notion.

[5] Remember that the branching structure is preserved. Hence, starting from a path in G, we can easily identify corresponding paths in G' and G''.

4 Second Order Effects in Partial Dead Code Elimination

In this section we will discuss the interdependencies between the various sinking and elimination steps of assignments. For comparison let us first consider the situation in partial redundancy elimination. Also partial redundancy elimination is conceptually composed of two kinds of elementary program transformations. First, hoisting computations, and second, eliminating total redundancies. However, partial redundancy elimination can be done independently for every program term t. Thus a single application of each step is sufficient to yield an optimal result. Unfortunately, this does not hold for partial dead code elimination. In fact, there are various kinds of 'second order effects' that need to be considered. We are now going to systematically discuss these effects, which are fully captured by our algorithm presented in Section 5.

4.1 Sinking-Elimination Effects

This is the effect of primary interest: an assignment is sunk until it can be eliminated by dead (faint) code elimination. Reconsider the motivating examples in Section 1 for illustration.

4.2 Sinking-Sinking Effects

The sinking of an assignment may open the way for other assignments to sink, if it is a use- or redefinition site for these assignments or if it modifies an operand of its right-hand side term. The last case is illustrated in Figure 10(a). Without previously sinking the assignment of node **2** the assignment of node **1** can sink at most to the entry of node **2**. Here it is blocked, since any further sinking would corrupt the value of its right-hand side expression. However, anticipating the sinking of the assignment at node **2** to node **5**, the assignment at node **1** can be sunk to node **3** and node **4**, where dead code elimination finally removes the occurrence at node **3** as displayed in Figure 10(b).

4.3 Elimination-Sinking Effects

For similar reasons as above, the elimination of dead assignments may enable the sinking of other assignments, see Figure 11(a) for illustration. Here, none of the assignments at node **1** and node **2** can be sunk without violating the admissibility. However, the assignment $a := \ldots$ at node **1** can be removed by dead code elimination, since its value is not used anymore. Now this removal enables the assignment $y := a + b$

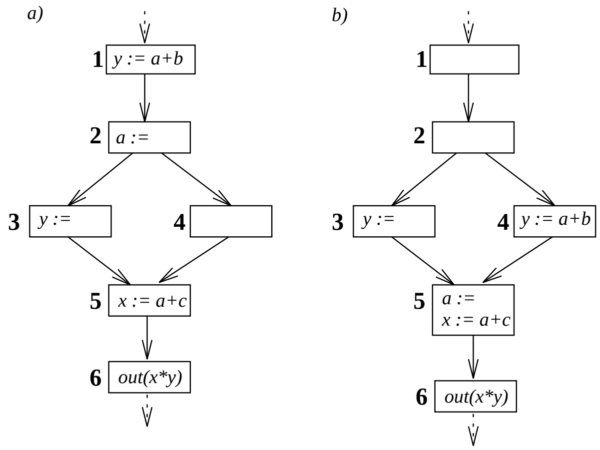

Figure 10: Sinking-Sinking Effect

to be sunk to node **4** and **5** in order to eliminate further partially dead assignments leading to the program displayed in Figure 11(b).

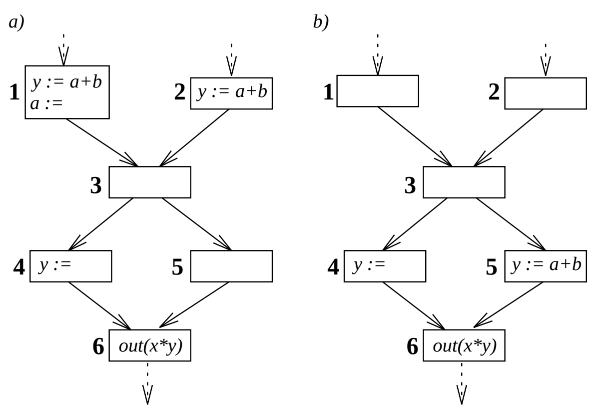

Figure 11: Elimination-Sinking-Effect

4.4 Elimination-Elimination Effects

This effect is illustrated in Figure 12(a). Here, the assignment at node **4** is dead and can be eliminated, since on every path leading to the end node the left-hand side variable y is redefined before it is used. Subsequently, the assignment to a at node **1**, which was not dead before due to its usage at node **4**, becomes dead and can be removed as shown in Figure 12(b).

It is worth noting that this example shows a second order effect for partial dead code elimination but a first order effect for partial faint code elimination: both assignments at node **1** and node **4** are faint, and hence could be eliminated simultaneously by faint code elimination.

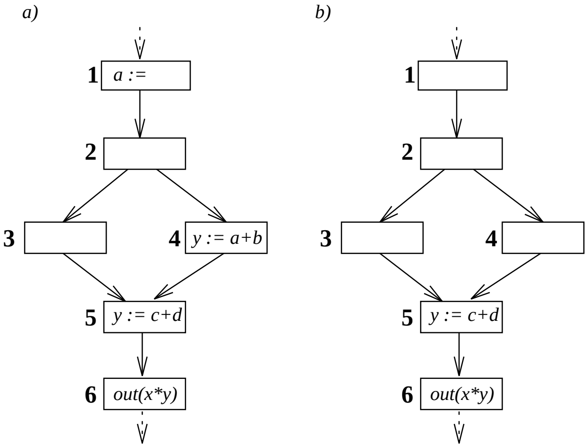

Figure 12: Elimination-Elimination Effect

5 The Algorithm

In this section we present our algorithm for the optimal elimination of partially dead (faint) assignments. We first give an overview of the algorithm, and subsequently describe the relevant steps in more detail.

5.1 Overview

The algorithm *pde* (*pfe*) consists of two main procedures that are repeated until the program stabilizes:

1. A procedure *dce* (*fce*) for the elimination of dead (faint) assignments, which is controlled by a dead (faint) variable analysis.

2. A procedure *ask* for assignment sinking, which is controlled by a delayability analysis working on bit-vectors of sinking candidates.

We are now going to describe these procedures in detail. A complexity estimation is given in Section 6.

5.2 Eliminating Dead (Faint) Assignments

The elimination of dead (faint) assignments is based on the determination of dead (faint) variables. Dead variables can be computed by means of a backwards directed bit-vector data flow analysis [2, 17, 24]. A standard formulation can be found in Table 1, where $\textbf{N-DEAD}_\iota(x)$ (or $\textbf{X-DEAD}_\iota(x)$) mean that variable x is dead at the entry (or exit) of statement ι. Additionally, this table shows the equation system for the faint variable analysis, where analogously to the dead variable analysis $\textbf{N-FAINT}_\iota(x)$ (or $\textbf{X-FAINT}_\iota(x)$) mean that variable x is faint at the entry (or exit) of statement ι. Though the faint problem does not have a bit-vector form, it can easily be solved by means of an iterative worklist algorithm operating slotwise on bit-vectors (cp. [10]). The only subtlety here is that a slot (ι, x) for an assignment statement ι may be influenced not only by the x-slot of some successor node $\hat{\iota}$, but also by the slot (ι, lhs_ι). This must be taken care of by additionally updating the worklist with all slots (ι, x), where x is a right-hand side variable of ι, whenever the slot (ι, lhs_ι) has been processed successfully. It is worth noting that this does not cause any problems for the correctness and complexity of the method (cf. Section 6.1).

After having computed the greatest solution for one of the equation systems specified in Table 1, the corresponding program transformation is very simple:

The Elimination Step:
Process every basic block by successively eliminating all assignments whose left-hand side variables are dead (faint) immediately after them.

Standard methods to dead code elimination are usually based on *definition-use graphs* [2, 21], which connect the definition sites of a variable with their corresponding use sites. Thus, dead assignments can be identified indirectly by means of a simple marking algorithm working on the definition-use graph. If this algorithm uses optimistic assumptions every faint assignment is detected in time proportional to the size of the graph. Unfortunately, definition-use graphs are usually quite large, i.e. of order $O(\mathtt{i}^2\,\mathtt{v})$ in the worst case, where $\mathtt{i}$ denotes the number of instructions and $\mathtt{v}$ the number of variables occurring in the flow graph [30]. The algorithm of [5] improves on this result by working on a sparse definition-use graph based on the SSA form. This results in a worst case time complexity of $O(\mathtt{i}\,\mathtt{v})$, which coincides with the complexity of our simple iterative algorithm (cf. Section 6.1).

5.3 Sinking of Assignments

The program transformation of this stage is based on a delayability analysis ([23, 22]), which was designed to determine how far a hoisted computation can be sunk from its earliest initialization point in order to minimize the lifetimes of temporaries introduced by partial redundancy elimination, while maintaining computational optimality. Table 2 presents the delayability analysis adapted to our situation in a bit-vector format, where each bit corresponds to an assignment pattern occurring in the program. Here $\textbf{N-DELAYED}_n$ and $\textbf{X-DELAYED}_n$ intuitively mean that some sinking candidates of α can be moved to the entry or the exit of basic block n respectively, and *sinking candidates* are occurrences of an assignment $x := t$ inside a basic block that are not blocked, i.e., neither followed by a modification of an operand of t nor by a modification or a usage of x. See Figure 13 for illustration. Note that among the various occurrences of an assignment pattern in a basic block at most the last one is a candidate for global sinking, because every occurrence is blocked at least by the subsequent occurrence.

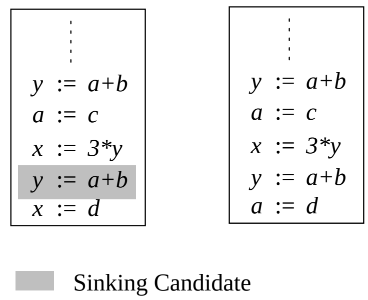

Figure 13: Sinking Candidates of "$y := a + b$"

The greatest solution of the equation system displayed in Table 2 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 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.[6]

5.4 Termination of the Global Algorithm

The algorithm terminates as soon as both steps, the dead (faint) code elimination and the assignment sinking leave the program invariant. In the case of dead (faint) code elimination this simply means that no further assignments are eliminated, and in the case of assignment sinking this holds, if every basic block n satisfies $\textbf{N-INSERT}_n = \mathit{false}$ and $\textbf{X-INSERT}_n = \textbf{LOCDELAYE}$

5.5 Results

Denoting the final programs that result from applying our algorithm for partial dead (faint) code elimination to G by G_{pde} and G_{pfe}, respectively, we have:

[6] Due to edge splitting there are no insertions at the exit of branching nodes.

Local Predicates

- $\mathbf{USED}_\iota(x)$: x is a right-hand side variable of the instruction ι.[a]
- $\mathbf{RELV\text{-}USED}_\iota(x)$: x is a right-hand side variable of the relevant instruction ι.
- $\mathbf{ASS\text{-}USED}_\iota(x)$: x is a right-hand side variable of the assignment statement ι.
- $\mathbf{MOD}_\iota(x)$: x is the left-hand side variable of the instruction ι.

The Dead Variable Analysis: (In bit-vector formulation[b])

$$\mathbf{N\text{-}DEAD}_\iota \;=_{df}\; \neg\mathbf{USED}_\iota * (\mathbf{X\text{-}DEAD}_\iota + \mathbf{MOD}_\iota)$$

$$\mathbf{X\text{-}DEAD}_\iota \;=_{df}\; \prod_{\hat{\iota}\in succ(\iota)} \mathbf{N\text{-}DEAD}_{\hat{\iota}}$$

The Faint Variable Analysis: (Slotwise simultaneously for all variables x)

$$\begin{aligned}\mathbf{N\text{-}FAINT}_\iota(x) \;=_{df}\; & \neg\mathbf{RELV\text{-}USED}_\iota(x) * (\mathbf{X\text{-}FAINT}_\iota(x) + \mathbf{MOD}_\iota(x)) * \\ & (\mathbf{X\text{-}FAINT}_\iota(lhs_\iota) + \neg\mathbf{ASS\text{-}USED}_\iota(x))\end{aligned}$$

$$\mathbf{X\text{-}FAINT}_\iota(x) \;=_{df}\; \prod_{\hat{\iota}\in succ(\iota)} \mathbf{N\text{-}FAINT}_{\hat{\iota}}(x)$$

Table 1: Dead & Faint Variable Analysis

[a] In particular, all variables occurring in relevant statements are considered right-hand side variables.

[b] Both analyses are employed at the instruction level. This, however, is important only for the faint variable analysis. In fact, the dead variable analysis can straightforwardly be modified to work on basic blocks.

Theorem 5.1 (Correctness)

1. $G_{pde} \in \mathcal{G}_{PDE}$
2. $G_{pfe} \in \mathcal{G}_{PFE}$

Moreover, it is easy to prove that $\mathcal{F}_{PDE} =_{df} \{dce, ask\}$ and $\mathcal{F}_{PFE} =_{df} \{fce, ask\}$ satisfy the dominance and monotonicity property of Section 3. Hence, we can apply Theorem 3.7, which establishes the following optimality results:

Theorem 5.2 (Optimality Theorem)

1. *G_{pde} is optimal in $\mathcal{G}_{PDE}$*
2. *G_{pfe} is optimal in $\mathcal{G}_{PFE}$*

6 Complexity

Parameterized in the complexities of its components,

- $\mathtt{c}_{dce}$, $\mathtt{c}_{fce}$ and $\mathtt{c}_{ask}$: the complexities of the data flow analyses of the corresponding component transformations,
- $\mathtt{w}$: the maximal factor by which the number of instructions may increase during the application of the algorithm, and
- $\mathtt{r}$: the maximal number of applications of the component transformations,

we have the following results for the overall complexity of the transformations

pde: $O(\mathtt{r}\,(\mathtt{c}_{dce} + \mathtt{c}_{ask} + \mathtt{w}\cdot\mathtt{i}))$

pfe: $O(\mathtt{r}\,(\mathtt{c}_{fce} + \mathtt{c}_{ask} + \mathtt{w}\cdot\mathtt{i}))$

where $\mathtt{i}$ denotes the number of instructions occurring in the original program. The factor $\mathtt{w}\cdot\mathtt{i}$ is caused by the actual intermediate transformations and updates of the local predicates.

The following three subsections provide a detailed estimation of the parameters mentioned above in terms of the number of basic blocks $\mathtt{b}$, the number of instructions $\mathtt{i}$, the number of variables $\mathtt{v}$ and the number of assignment patterns $\mathtt{a}$ of the original program. Subsequently, the overall complexity is sketched more roughly in terms of a uniform parameter $\mathtt{n}$ reflecting the program size of the argument program.

Local Predicates:

- $\mathbf{LOCDELAYED}_n(\alpha)$: There is a sinking candidate of α in n.
- $\mathbf{LOCBLOCKED}_n(\alpha)$: The sinking of α is blocked by some instruction of n.

Delayability Analysis:

$$\mathbf{N\text{-}DELAYED}_n =_{df} \begin{cases} \mathit{false} & \mathit{if} \quad n = \mathbf{s} \\ \prod_{m \in pred(n)} \mathbf{X\text{-}DELAYED}_m & \mathit{otherwise} \end{cases}$$

$$\mathbf{X\text{-}DELAYED}_n =_{df} \mathbf{LOCDELAYED}_n + \mathbf{N\text{-}DELAYED}_n * \neg\mathbf{LOCBLOCKED}_n$$

Insertion Points:

$$\mathbf{N\text{-}INSERT}_n =_{df} \mathbf{N\text{-}DELAYED}_n * \mathbf{LOCBLOCKED}_n$$

$$\mathbf{X\text{-}INSERT}_n =_{df} \mathbf{X\text{-}DELAYED}_n * \sum_{m \in succ(n)} \neg\mathbf{N\text{-}DELAYED}_m$$

Table 2: Delayability Analysis and Insertion Points

6.1 Complexity of the Component Transformations

6.1.1 Delayability Analysis

The delayability analysis realizing the essential part of the assignment sinking procedure is a forward directed bit-vector data flow analysis. For *well-structured* flow graphs the efficient bit-vector techniques [19, 20, 29] become applicable, yielding an almost linear complexity in terms of fast bit-vector operations. For *arbitrary* control flow structures, however, the slotwise approach of [10] is the best we can do yielding $O(\mathtt{b}\cdot\mathtt{a})$ as the worst case time complexity for the assignment sinking procedure.

6.1.2 Dead (Faint) Variable Analysis

Like the delayability also the *dead* code analysis is a bit-vector problem. Thus replacing the parameter `a` by `v` the same estimations apply for the worst case time complexity. Unfortunately, the *faint* variable analysis is not a bit-vector problem, i.e. the solution cannot be computed for each variable independently. Thus there are no special algorithms for structured programs, and the slotwise approach of [10] must always be applied. Note that the structure of faint code analysis requires a computation at the instruction level.

We will now prove that under the usual assumptions that

- the size of program terms is bound by a constant, and
- the number of edges is of order `b`

faint code elimination is proportional to both the number of instructions and the number of program variables.

Investigating the equation system of the faint variable analysis of Table 1 reveals that during the iterative computation of the greatest solution of the equation system each slot for a variable x can change its value at most once from true to false. These changes are the only reason for updating the worklist:

- if the slot is part of an entry bit-vector at an instruction ι then for all predecessors ι' of ι the x-slots of the exit bit-vectors are added to the current worklist, and
- if the slot is part of an exit bit-vector of an instruction ι then all y-slots of the same bit-vector are added to the current worklist where y is a right-hand side variable of ι.

Thus one can easily establish that every edge of the control flow graph will be considered at most `v` times during the analysis in order to 'reach' the corresponding successor. Applying our assumptions therefore yields that the number of worklist entries written by the faint code analysis algorithm is at most proportional to the program size and the number of program variables. A similar argument suffices to prove that the global cost of the faint code analysis, including the effort for slot processing, is still of the same order, which completes the proof.

As the size of the program may increase by a factor $\mathtt{w}$ during the execution of our algorithm (see Section 6.2), the intermediate program size during the execution of our overall algorithm can only be estimated by $O(\mathtt{w} \cdot \mathtt{i})$, yielding a worst case complexity of $O(\mathtt{w} \cdot \mathtt{i} \cdot \mathtt{v})$ for the required faint analysis steps.

6.2 Estimating the Code Size

Using induction on the length of a shortest (acyclic) path p reaching a node n it can easily be shown that the number of instructions that can be inserted at n is bound by the number of instructions on p. Thus the number of instructions at a basic block will never exceed $\mathtt{i}$ showing that $\mathtt{w}$ is of order $O(\mathtt{b})$ in the worst case. In practice, however, we expect that $\mathtt{w}$ is bound by a constant.

6.3 Estimating the Number of Iterations

Applying the shortest path argument of Section 6.2 again, the number of assignments that are inserted at a node n during the application of the overall algorithm is bound by $\mathtt{i}$. Thus the number of *dce* (*fce*) and *ask* applications can be estimated by $\mathtt{i} \cdot \mathtt{b}$, yielding that $\mathtt{r}$ is at most quadratic in the program size. However, we conjecture that $\mathtt{r}$ only depends linearly on $\mathtt{i}$.

6.4 Summary

Combining the results of the previous subsections, we have:

- $O(\mathtt{r})$, $O(\mathtt{c}_{dce})$, $O(\mathtt{c}_{ask})$, and $O(\mathtt{w\ i})$ can be estimated by $O(\mathtt{n}^2)$, and
- $O(\mathtt{c}_{fce})$ can be estimated by $O(\mathtt{n}^3)$.

This guarantees that partial dead code elimination is of order $O(\mathtt{n}^4)$ and partial faint code elimination is of order $O(\mathtt{n}^5)$ in the worst case.

These estimations are very pessimistic. Already using our conjecture would reduce the complexities to $O(\mathtt{n}^3)$ and $O(\mathtt{n}^4)$, respectively. Moreover, using the reasonable assumption that the factor $\mathtt{w}$ that indicates the degree of static code replication is of order $O(1)$ saves another factor of $\mathtt{n}$ for the partial faint code elimination, and it saves (almost) a factor of $\mathtt{n}$ for partial dead code elimination whenever fast bit-vecor techniques are applicable. Thus we expect a quadratic behaviour ($O(\mathtt{n}^2)$) for partial dead code elimination and at most a cubic behaviour ($O(\mathtt{n}^3)$) for partial faint code elimination.

Summarizing, the overall time bound for our algorithms is only slightly worse than the one for the significantly weaker technique of dead code elimination based on definition-use graphs, and it is comparable with the complexity of other aggressive code motion techniques. E.g., the algorithm for global value numbering of [27], which requires reducible flow graphs and guarantees optimality only for acyclic program structures, is of third order.

7 Conclusions

We have presented a new aggressive algorithm for the optimal elimination of partially dead (faint) code, which captures all second order effects that are due to the mutual dependences between assignment sinking and dead (faint) code elimination. This algorithm is comparably expensive as other aggressive optimization methods. Its complexity ranges from $O(\mathtt{n}^2)$ for the 'dead' version and realistic structured programs to $O(\mathtt{n}^5)$ for the faint version and the completely unrestricted worst case. Thus as other aggressive methods, our algorithm should typically be employed for the optimization of time-critical sections of code of moderate size. In general, modifications of our algorithm should be applied that limit the number of assignment sinking and dead (faint) code elimination steps. We are currently investigating heuristics guiding this limitation, which range from simply cutting the global iteration process after some given amount of time or a fixed number of iterations to localizing the optimization process to 'hot areas'.

8 Acknowledgements

We are very grateful to the anonymous referees, and Gerald Lüttgen for their valuable comments.

References

[1] 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.

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

[3] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation'91*, volume *26*,6 of *ACM SIGPLAN Notices*, pages 241–255, Toronto, Ontario, June 1991.

[4] 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.

[5] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependency graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451 – 490, 1991.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] A. Geser, J. Knoop, G. Lüttgen, O. Rüthing, and B. Steffen. Chaotic fixed point iterations. MIP-Bericht 9403, Fakultät für Mathematik und Informatik, Universität Passau, Germany, 1994.

[15] P. B. Gibbons and S. S. Muchnik. Efficient instruction scheduling for a pipline architecture. In *Proc. ACM SIGPLAN Symposium on Compiler Construction'86*, volume *21*, 7 of *ACM SIGPLAN Notices*, pages 11–16, June 1986.

[16] R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations. In *Proc. of the third Conference of the European Co-operation in Informatics*, Informatik-Fachberichte 50, pages 1–10. Springer, 1981.

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

[18] S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for data flow analysis. *Acta Informatica*, 24:679 – 694, 1987.

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

[20] 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.

[21] K. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5 – 54. Prentice Hall, Englewood Cliffs, NJ, 1981.

[22] 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.

[23] 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.

[24] L. T. Kou. On live-dead analysis for global data flow problems. *Journal of the ACM*, 24(3):473 – 483, July 1977.

[25] R. J. Mintz, G. A. Fisher, and M. Sharir. The design of a global optimizer. In *Proc. ACM SIGPLAN Symposium on Compiler Construction'79*, volume *14*, 8 of *ACM SIGPLAN Notices*, pages 226 – 234, Denver, Col., 1979.

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

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

[28] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.

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

[30] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2), April 1991.