

# Partial Dead Code Elimination Using Extended Value Graph

Munehiro Takimoto and Kenichi Harada

Department of Computer Science,  
Graduate School of Science and Technology,  
Keio University, Yokohama 223-8522, Japan

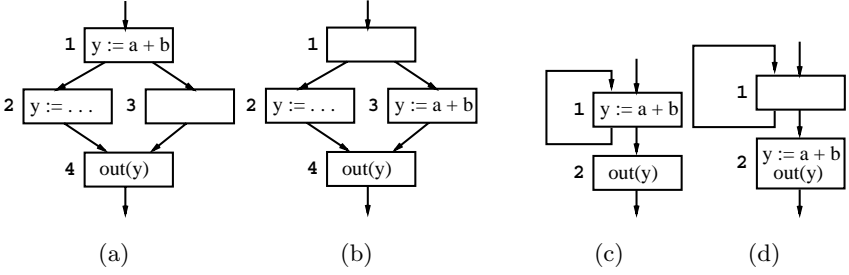
**Abstract.** This paper presents an efficient and effective code optimization algorithm for eliminating partially dead assignments, which become redundant on execution of specific program paths. It is one of the most aggressive compiling techniques, including invariant code motion from loop bodies. Since the traditional techniques proposed to this optimization would produce the second-order effects such as sinking-sinking effects, they should be repeatedly applied to eliminate dead code completely, paying higher computation cost. Furthermore, there is a restriction that assignments sunk to a join point on flow of control must be lexically identical.

Our technique proposed here can eliminate possibly more dead assignments without the restriction at join points, using an explicit representation of data dependence relations within a program in a form of SSA (Static Single Assignment). Such representation called Extended Value Graph (EVG), shows the computationally equivalent structure among assignments before and after moving them on the control flow graph. We can get the final result directly by once application of this technique, because it can capture the second-order effects as the main effects, based on EVG.

## 1 Introduction

*Dead code elimination* [1] has been used to improve efficiency of program execution by eliminating unnecessary instructions. If a variable at the left-hand side of assignment is not used in the continuation of a program, such assignment can be eliminated. This is called *totally* dead assignment. However, as illustrated by  $y := a + b$  at node 1 in Fig.1(a), if a variable at the left-hand side is not used on the left branch but used on the right, it cannot be eliminated directly. In this case, such assignment can be transformed to be totally dead at node 2 by sinking  $y := a + b$  from node 1 to the entry of node 2 and 3. After that, the original assignment can be eliminated as shown in Fig.1(b). This assignment is called *partially* dead assignment and this elimination technique is called *partial dead code elimination* (PDE) [11].

The concept of PDE also includes loop-invariant code motion as shown in Fig.1(d), where  $y := a + b$  in Fig.1(c) is a partially dead assignment because the variable  $y$  is dead on the back path to the beginning of loop body.



**Fig. 1.** Partial dead code elimination and loop-invariant code motion

In previous works [8,11,14], all assignments sunk to a join point are required to be lexically identical, i.e. having the same left-hand side variable, the same operators and operands. This restriction forbids some attractive optimizations. In Fig.2(a),  $x := a + b$  at node 2 and  $x := a + c * 3$  at node 3 have the same variable at left-hand sides, but are not same in their right-hand sides. Thus, sinking to node 4 of these assignment statements are blocked. If a temporary variable for the subexpression  $c * 3$  in  $x := a + c * 3$  is renamed to the same name as variable  $b$ ,  $x := a + b$  can be sunk to node 4 after  $y := x$  at node 4 has been sunk as shown in Fig.2(b).

We propose an efficient technique to eliminate partially dead assignments including such renaming effect for operands. This effect is systematically obtained by transformation of our graph representation, called as *extended value graph* (EVG)[17].

Our partial dead code elimination is realized by dataflow analysis on both *control flow graph* (CFG) and EVG. This analysis also contributes to reducing optimization cost. PDE consists of two steps, i.e. sinking and elimination of assignments, both of which can mutually influence each other. This means that each step may cause second order effects to another step. To complete all possible eliminations, PDE must cover the following four types of second order effects [11]:

**Sinking-Elimination Effects:** An assignment is sunk until it can be eliminated by dead code elimination.

**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 their right-hand side expression.

**Elimination-Sinking Effects:** The elimination of dead assignments may enable the sinking of other assignments.

**Elimination-Elimination Effects:** The elimination of dead assignments may enable the elimination of other assignments.

Sinking-elimination effects can be captured by a single application of PDE, but the other effects are obtained by repeated applications of PDE. This repetition costs a lot. The reasons why PDE cannot capture second order effects as first order effects are as follows: