Path Profile Guided Partial Dead Code Elimination Using Predication *

Rajiv Gupta[†]
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

David A. Berson Jesse Z. Fang
Microcomputer Research Lab
Intel Corporation
Santa Clara, CA 95052

Abstract

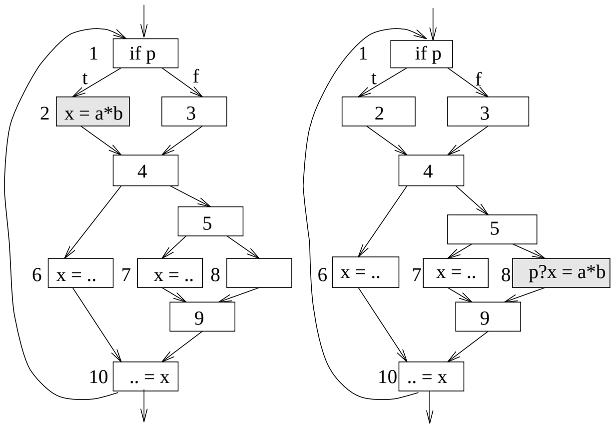
We present a path profile guided partial dead code elimination algorithm that uses predication to enable sinking for the removal of deadness along frequently executed paths at the expense of adding additional instructions along infrequently executed paths. Our approach to optimization is particularly suitable for VLIW architectures since it directs the efforts of the optimizer towards aggressively enabling generation of fast schedules along frequently executed paths by reducing their critical path lengths. The paper presents a **cost-benefit** *data flow analysis that uses* **path profiling** *information to determine the profitability of using predication enabled sinking. The* **cost** *of predication enabled sinking of a statement past a merge point is determined by identifying paths along which an additional statement is introduced. The* **benefit** *of predication enabled sinking is determined by identifying paths along which additional dead code elimination is achieved due to predication. The results of this analysis are incorporated in a code sinking framework in which predication enabled sinking is allowed past merge points only if its* **benefit** *is determined to be greater than the* **cost**. *It is also demonstrated that trade-off can be performed between the compile time cost and the precision of cost-benefit analysis.*

1 Introduction

Traditional approach to code optimization uniformly spends its effort in optimizing all parts of a program. Researchers have now recognized that the optimization effort is best spent on frequently executed portions of a program that can be identified using profiling techniques. Compilers for VLIW architectures use sophisticated global instruction scheduling techniques to generate fast schedules for frequently executed paths at the expense of slower schedules for infrequently executed paths. Thus, it is also appropriate for an optimizer for a VLIW architecture to aggressively optimize frequently executed paths even if their optimization results in introduction of additional instructions along infrequently executed paths. However, traditional formulations of code optimizations are incapable of performing a trade-off between the quality of code along frequently and infrequently executed paths. In this paper we present an approach for optimization that is capable of performing such a trade-off.

The optimization strategy that we have developed is demonstrated through a new algorithm for partial dead code elimination (PDE). PDE is an important optimization for VLIW architectures since critical path lengths along frequently executed paths can be reduced through PDE [11, 6, 13]. Through code sinking, that is, delaying the execution of a code statement to later program points, PDE optimization moves instructions that are dead along critical paths off the critical paths [19, 22]. Modern architectures [14, 12] that support predicated execution provide an opportunity for aggressively performing PDE since predication enables code sinking that is otherwise not possible. Furthermore, predication enabled code sinking may also introduce predicated versions of an instruction along paths where the instruction was not previously encountered. Thus, in order to beneficially exploit predication enabled code sinking for PDE, it is necessary to develop an approach for trade-off between quality of code along frequently and infrequently executed paths. Existing techniques for PDE [2, 19] do

†Supported in part by NSF PYI Award CCR-9157371, NSF grant CCR-9402226, Intel Corporation, and Hewlett Packard.



(a) (b)

Path	Freq
1-2-4-6-10	100
1-2-4-5-7-9-10	90
1-2-4-5-8-9-10	80
1-3-4-6-10	70
1-3-4-5-7-9-10	60
1-3-4-5-8-9-10	50

Cost[2: x=a*b] = (100+90)*T[x=a*b] = 190*T[x=a*b]

Benefit[2: x=a*b] = 50*T[p?x=a*b]= 50

(c)

Figure 1: Predication Enabled Code Sinking.

not take advantage of predication in performing PDE.

In a study that we carried out it was found that for the SPEC95 integer benchmarks 65% of the functions that were executed had no more than 5 paths with non-zero execution frequency and no function had more than 1000 paths with non-zero execution frequency while the number of static paths through the benchmarks were in the millions. Since traditional approaches to PDE remove dead code along some paths without adding additional instructions along any path [2, 19], they are essentially based on the assumption that all paths through the program are equally important. However, our approach which can trade-off the quality of code generated for infrequently executed paths for obtaining better code for frequently executed paths is based upon the realistic scenario encountered in practice. There are two types of profiling that can be used for identifying frequently and infrequently executed paths. *Edge profiling* gives the number of times each edge in the program flow graph is traversed and *path profiling* gives the number of times various acyclic paths in a program are traversed. In [1] it is shown that different path profiles can give rise to the same edge profiles. Thus, it is not possible to accurately identify frequently executed paths using edge profiles. Furthermore, Ball and Larus [1] have shown that path profile information can be collected efficiently. Therefore in this paper we rely on path profiling information.

The example in Figure 1a illustrates our approach. In this example the statement at node 2 ($x = a * b$) is partially dead since the value of x computed by this statement is not used along paths 1-2-4-6-10 and 1-2-4-5-7-9-10. The elimination of partial deadness of the statement requires the sinking of statement past the merge point at node 4. Existing code sinking algorithms will not achieve PDE in this case as they are unable to sink the statement past node 4 because doing so would block any definitions of x that reach node 4 via node 3. As shown in Figure 1b, the sinking of the statement past node 4 and down to node 8 can be achieved by predicating the statement. While the above sinking eliminates dead code along the paths 1-2-4-6-10 and 1-2-4-5-7-9-10, it also introduces an additional instruction along path 1-3-4-5-8-9-10. Since the sum of the execution frequencies of paths 1-2-4-6-10 and 1-2-4-5-7-9-10 (i.e., 100+90=190) is greater than the execution frequency of path 1-3-4-5-8-9-10 (i.e., 50), the overall savings will result from predication enabled sinking. Thus, an optimizer should exploit path profiling information to reduce dead code along frequently executed paths even if doing so introduces some additional instructions along infrequently executed paths. For a predication model in which an instruction is aborted if the predicate is false, although, an additional instruction is executed along path 1-3-4-5-8-9-10, its execution will require a single cycle since the predicate will evaluate to false along this path and the multiply operation will not be executed. On the other hand along the paths that benefit, a multiply operation has been removed which typically takes several cycles for execution. The above optimization also requires that the result of evaluating predicate p in node 1 must be saved till node 8. In an architecture such as PlayDoh, the predicates are stored in special predicate registers [14].

The approach we present first performs data flow analysis for computing the cost and benefit of sinking a statement past each relevant merge point. An extension of a code sinking algorithm by Knoop et al. [19] that incorporates predication is presented next. This modified framework uses the cost-benefit information to enable sinking of a partially dead statement past merge points where the benefit has been found to be greater than the cost. In previous work by Fang [5] the use of predication during sinking was also proposed. However, the algorithms described in [5] apply to single-entry-single-exit acyclic regions and no cost-benefit analysis to guide predication is presented.

In section 2 we present our PDE algorithm in detail. In section 2.1 data flow analysis for computing cost-benefit information for acyclic graphs is discussed and its extensions for loops are presented. We also show how to trade-off the complexity of performing cost-benefit analysis with the precision of the analysis. In section 2.2 we describe a predication based sinking framework and section 2.3 we describe the insertion of predicate evaluations required for predication. We conclude by discussing additional applications of this approach in section 3.

2 Predication based PDE

The overview of our approach is given in Figure 2. In this approach we first identify statements that appear along frequently executed paths. These statements are the target of PDE optimization. By limiting the optimization to these statements, we reduce the overall run-time cost of optimization. Next the cost and benefit of sinking each frequently executed statement past various merge points in the program are identified. A code sinking framework that uses this cost-benefit information and is capable of predication based sinking past merge points is then used to perform PDE. Finally predicates required by the statements to which predication based code sinking has been applied are inserted.

Step 1: Using path profiling information
from the set of program paths P, identify
the frequently executed paths FP; and
of all the statements S, consider frequently
executed statements FS as those that
appear along paths in FP.
for each statement $s \in FS$ {
for each merge point m to which s can sink {
For predication based sinking of s enabled at m
estimate *Benefit* s's PDE to paths in FP and
estimate *Cost* of s's PDE to all paths P.
if *Benefit*>*Cost* { enable sinking of s at m }
else { disable sinking of s at m }}}
Step 2: Apply predication based PDE
to statements in FS.
Step 3: Introduce predicate evaluations
required for statements in FS.

Figure 2: Algorithm Overview.

During cost-benefit analysis only the frequently executed paths are accurately analyzed since these are the ones of most concern. By considering only these paths the run-time cost of cost-benefit analysis is reduced. The benefit we compute is the benefit of applying PDE to a statement that it experienced by frequently executed paths. Thus, it is an underestimate of the true benefit. On the other hand the cost of applying PDE is accurately computed for frequently executed paths and overestimated for the remaining paths. Thus, the cost is an overestimate of the true cost.

2.1 Cost-Benefit Analysis

Predication is a technique that allows sinking of statements past merge points in situations where the statement is not available for sinking at the merge point along all paths leading to the merge point. The execution of the statement following its sinking past the merge point is predicated to ensure that it does not overwrite values of the lhs-variable in situations where control reaches the merge point along paths where different definitions of the lhs-variable are available. We first discuss the cost-benefit analysis for acyclic graphs which consider all paths in the flow graph and later we present the extensions for handling programs with loops and performing cost-benefit analysis by limiting it to frequently executed paths.

The algorithms in this paper use the control flow graph representation of the program. Similar to the assumptions made by Knoop et al. in [19], we also assume that each node in the control flow graph contains a single statement and nodes have been introduced along critical edges to allow code placement along the critical edges. However, this algorithm can be easily extended to apply to basic blocks.

To perform cost-benefit analysis for the sinking of a statement s past a merge point n, we must categorize the program subpaths that either originate or terminate at n.

The subpaths from the start of the flow graph to the merge point n are divided into two categories with respect to the statement s:

Available subpaths which are program subpaths from the *start* node to n along which s is encountered and s is sinkable to n along the subpath; and

Unavailable subpaths which are program subpaths from *start* node to n along which s is not available for sinking at n. These subpaths include those along which either s is not encountered or although s is encountered, it is not sinkable to n. The sinking of s to n can be blocked by a statement that is data (anti, output or flow) dependent on s.

Program subpaths from n to the end of the program are also divided into two categories with respect to the statement s:

Removable subpaths which are subpaths from n to the *end* node along which value computed by statement s is not live at n and it is possible to eliminate the deadness of s along the path by sinking s and pushing it off the subpath; and

Unremovable subpaths which are program subpaths from n to the *end* node along which either the value computed by s is not dead or its deadness cannot be eliminated because sinking of s necessary to push s off the subpath is blocked by another statement.

The paths which *benefit* from the sinking of s past the merge point n are the paths along which dead code is removed (i.e., those paths prior to optimization along which the statement s is executed but the value computed by s is never used) and the dead code would not have been removed without sinking s past merge point n. These paths can be obtained by concatenating *available* subpaths at n with *removable* subpaths at n. The total benefit of predication enabled sinking of s past the merge point n is measured by summing up the execution frequencies of the paths that benefit from the optimization.

The paths which incur *cost* due to sinking of s past the merge point n are the paths in the flow graph along which an additional execution of a predicated version of statement s is encountered. These paths are obtained by concatenating *unavailable* subpaths with *unremovable* subpaths. The total cost of predication enabled sinking of s past the merge point n is measured by summing up the execution frequencies of the paths that incur a cost due to the optimization.

Consider the example in Figure 3. In this example of the four program subpaths from 0 to 4, s is *available* for sinking to 4 along the subpath 0-1-11-4 and *unavailable* along the remaining subpaths 0-1-2-4, 0-1-3-4, and 0-9-4. Of the three program subpaths starting at 4, s is dead along two subpaths 4-5-10 and 4-6-7-10 and live along the subpath 4-6-8-10. However, partial deadness of s is *removable* only along subpath 4-5-10 and *unremovable* along paths 4-6-7-10 and 4-6-8-10 since the definition of a at node 6 blocks the sinking of s past node 6. The only path that *benefits* from predication enabled sinking of s past 4 is 0-1-11-4-5-10. The paths that incur a *cost* due to predication enabled sinking of s past node 4 include 0-1-2-4-6-7-10, 0-1-3-4-6-7-10, 0-9-4-6-7-10, 0-1-2-4-6-8-10, 0-1-3-4-6-8-10, and 0-9-4-6-8-10. It should be noted that along the *unavailable* subpath 0-1-3-4 the value of x computed by s is not used. However, the removal of dead code along this subpath is not included in the benefit for merge point 4 because it can be derived without sinking s past the merge point 4.

Definition 1: Given a path p that passes through a merge point n and a partially dead statement s, the statement s is **available** for sinking at n along path p if and only if s is encountered along the subpath of p from *start* to n and there is no statement along the subpath of p from s to n that blocks the sinking of s to n. Otherwise statement s is **unavailable** at n along path p.

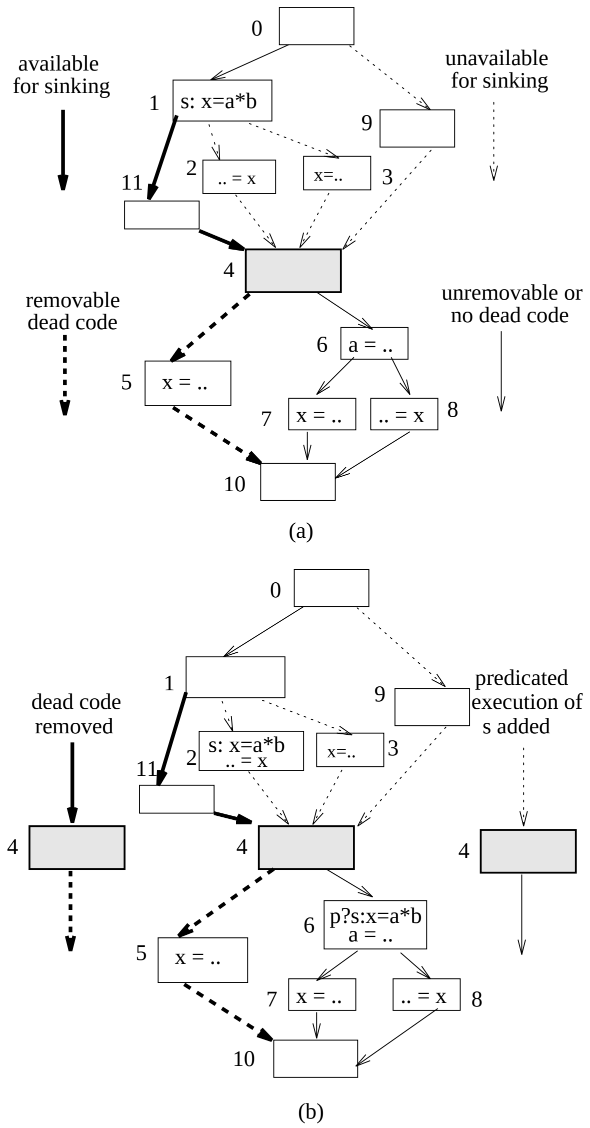


Figure 3: Cost-Benefit Analysis.

Definition 2: Given a path p that passes through a merge point n and a partially dead statement s, the statement s is **removable** from path p through sinking at n if and only if the value computed by s is not used along path p and it is necessary to sink s past n in order to reach the earliest point along p at which s is fully dead (i.e., dead along all paths starting at that point). Otherwise statement s is **unremovable** from path p through sinking at n.

Definition 3: Given a path p that passes through a partially dead statement s and a merge point n, the sinking of s past n **benefits** path p if and only if s is *available* for sinking at n and s is *removable* along path p through sinking at n. We denote the set of paths through n along which sinking of s is beneficial as $BenefitPaths_s(n)$.

Definition 4: The **benefit** of predication enabled sinking of a statement s past a merge point n in an acyclic graph, denoted as $Benefit_s(n)$, is the sum of the execution frequencies of the paths in $BenefitPaths_s(n)$ times the number of cycles, $T[s]$, that it takes to execute the operation in statement s:
$Benefit_s(n) = T[s] \times \sum_{p \in BenefitPaths_s(n)} Freq(p)$.

Definition 5: Given a path p that passes through a merge point n and a partially dead statement s, the sinking of s past n **costs** path p if and only if s is *unavailable* for sinking at n and s is *unremovable* at n along path p. We denote the set of paths through n along which sinking of s results in a cost as $CostPaths_s(n)$.

Definition 6: The **cost** of predication enabled sinking of a statement s past a merge point n in an acyclic graph, denoted as $Cost_s(n)$, is the sum of the execution frequencies of the paths in $CostPaths_s(n)$ times the number of cycles, $T[p?s]$, that it takes to execute predicated statement $p?s$ when p evaluates to false:
$Cost_s(n) = T[p?s] \times \sum_{p \in CostPaths_s(n)} Freq(p)$.

2.1.1 Cost-Benefit for Acyclic Graphs

In order to implement the computation of cost and benefit information for a given statement at various merge nodes in the program we proceed as follows. In addition to computing the *availability* and *removability* information at program points, we also compute the set of paths along which these data flow values hold. The set of paths is represented by a bit vector in which each bit corresponds to a unique path from the entry to the exit of the acyclic flow graph. To facilitate the computation of sets of paths, with each node n in the flow graph, we associate a bit vector $OnPaths(n)$ where each bit corresponds to a unique path and is set to 1 if the node belongs to that path; otherwise it is set to 0. Initially in this section we assume that all paths are frequently executed and must be considered during the analysis. In the next section we will illustrate how the solution we develop here is easily adaptable to the situation in which only subset of paths are known to be frequently executed. The steps of the analysis are described next.

Availability Analysis

$N-AVAIL_s(n)(X-AVAIL_s(n))$ is a one bit variable which is 1 if there is a path through n along which s is available for sinking at n's entry(exit); otherwise its value is 0. Forward data flow analysis with the *or* confluence operation is used to compute these values. At the entry point of the flow graph the availability value is set to 0, it is changed to 1 when statement s is encountered, and it is set to 0 if a statement that blocks the sinking of s is encountered. In the equations $BLOCK_s(n)$ is a one bit variable which is 1(0) if n blocks s, that is, n is data (anti, output or flow) dependent upon s.

$N-APATHS_s(n)(X-APATHS_s(n))$ is a bit vector which holds the set of paths along which the value of $N-AVAIL_s(n)(X-AVAIL_s(n))$ is 1 at n's entry(exit). At the entry to a node n for which $N-AVAIL_s(n)$ is 0, the set of paths is set to null, that is, to $\vec{0}$. Otherwise the paths in $N-APATHS_s(n)$ are computed by unioning the sets of paths along which s is available at the exit of one of n's predecessors (i.e., unioning $X-APATHS_s(p)$, where p is a predecessor of n). In order to ensure that only paths that pass through n are considered, the result is intersected with $OnPaths(n)$. The value of $X-APATHS_s(n)$ is $OnPaths(n)$ if n contains s and $N-APATHS_s(n)$ if n does not block s.

The data flow equations for this step are given in Figure 4a.

Removability Analysis

$N-REM_s(n)(X-REM_s(n))$ is a one bit variable associated with n's entry(exit) which is 1 if there is a path through n along which s is dead and any sinking of s that may be required to remove this deadness is feasible; otherwise its value is 0. Backward data flow analysis with the *or* confluence operation is used to compute these values. In order to ensure that the sinking of s is feasible, the results of availability analysis computed previously are used. For example, if

$$\mathbf{N-AVAIL_s(n)} = \begin{cases} \mathbf{0} & \textbf{if } \mathbf{n = entry} \\ \displaystyle\bigvee_{\mathbf{m \in Pred(n)}} \mathbf{X-AVAIL_s(m)} & \textbf{otherwise} \end{cases}$$

$$\mathbf{X-AVAIL_s(n)} = \begin{cases} \mathbf{1} & \textbf{if } \mathbf{s \in n} \\ \overline{\mathbf{BLOCK}}_\mathbf{s}\mathbf{(n) \wedge N-AVAIL_s(n)} & \textbf{otherwise} \end{cases}$$

$$\mathbf{N-APATHS_s(n)} = \begin{cases} \vec{\mathbf{0}} & \textbf{if } \mathbf{N-AVAIL_s(n) = 0} \\ \mathbf{OnPaths(n)} \wedge \displaystyle\bigvee_{\substack{\mathbf{m \in Pred(n) \wedge} \\ \mathbf{X-AVAIL_s(m)=1}}} \mathbf{X-APATHS_s(m)} & \textbf{otherwise} \end{cases}$$

$$\mathbf{X-APATHS_s(n)} = \begin{cases} \mathbf{OnPaths(n)} & \textbf{if } \mathbf{s \in n} \\ \mathbf{N-APATHS_s(n)} & \textbf{if } \mathbf{X-AVAIL_s(n) = 1} \\ \vec{\mathbf{0}} & \textbf{otherwise} \end{cases}$$

(a) Availability Data Flow Analysis.

$$\textbf{let v be the variable defined by s, i.e., } \mathbf{v = lhs(s)}:$$

$$\mathbf{X-REM_s(n)} = \begin{cases} \mathbf{X-AVAIL_s(n)} & \textbf{if } \mathbf{X-DEAD_v(n) = 1} \\ \mathbf{X-AVAIL_s(n)} \wedge \displaystyle\bigvee_{\mathbf{m \in Succ(n)}} \mathbf{N-REM_s(m)} & \textbf{otherwise} \end{cases}$$

$$\mathbf{N-REM_s(n)} = \begin{cases} \mathbf{N-AVAIL_s(n)} & \textbf{if } \mathbf{N-DEAD_v(n) = 1} \\ \mathbf{N-AVAIL_s(n) \wedge X-REM_s(n)} & \textbf{otherwise} \end{cases}$$

$$\mathbf{X-RPATHS_s(n)} = \begin{cases} \mathbf{OnPaths(n)} & \textbf{if } \mathbf{X-DEAD_v(n) \wedge X-AVAIL_s(n) = 1} \\ \mathbf{OnPaths(n)} \wedge \displaystyle\bigvee_{\substack{\mathbf{m \in Succ(n) \wedge} \\ \mathbf{N-REM_s(m)=1}}} \mathbf{N-RPATHS_s(m)} & \textbf{otherwise} \end{cases}$$

$$\mathbf{N-RPATHS_s(n)} = \begin{cases} \mathbf{OnPaths(n)} & \textbf{if } \mathbf{N-DEAD_v(n) \wedge N-AVAIL_s(n) = 1} \\ \mathbf{X-RPATHS_s(n)} & \textbf{if } \mathbf{N-REM_s(n) = 1} \\ \vec{\mathbf{0}} & \textbf{otherwise} \end{cases}$$

(b) Removability Data Flow Analysis.

$$\forall \mathbf{n} \textbf{ st n is a merge point}:$$

$$\mathbf{BENEFITPATHS_s(n) = X-RPATHS_s(n) \wedge X-APATHS_s(n)}$$

$$\mathbf{BENEFIT_s(n)} = \sum_{\mathbf{i}} \mathbf{BENEFITPATHS_s(n)(i) \times FREQ(path(i))}$$

$$\mathbf{COSTPATHS_s(n)} = \overline{\mathbf{X-RPATHS}}_\mathbf{s}\mathbf{(n)} \wedge \overline{\mathbf{X-APATHS}}_\mathbf{s}\mathbf{(n)}$$

$$\mathbf{COST_s(n)} = \sum_{\mathbf{i}} \mathbf{COSTPATHS_s(n)(i) \times FREQ(path(i))}$$

(c) Cost-Benefit Computation.

Figure 4: Cost-Benefit Data flow Analysis.

variable v computed by s is dead at n's exit, then $X-REM_s(n)$ is set to true only if $X-AVAIL_s(n)$ is true because the deadness can only be eliminated if sinking of s to n's exit is feasible. The results of availability analysis are similarly used in each data flow equation of removability analysis. In the equations in Figure 4b, $N-DEAD_v(n)(X-DEAD_v(n))$ is a one bit variable which is 1 if variable v is fully dead at n's entry(exit), that is, there is no path starting at n along which current value of v is used; otherwise its value is 0.

$N-RPATHS_s(n)(X-RPATHS_s(n))$ is a bit vector which holds the set of paths along which the value of $N-REM_s(n)(X-REM_s(n))$ is 1 at n's entry(exit). At the entry(exit) of a node n for which $N-DEAD_v(n)(X-DEAD_v(n))$ and $N-AVAIL_s(n)(X-AVAIL_s(n))$ are 1, $N-RPATHS_s(n)(X-RPATHS_s(n))$ is set to $OnPaths(n)$. Otherwise the paths in $X-RPATHS_s(n)$ are computed by unioning the sets of paths along which s is partially dead and removable at the entry of one of n's successors (i.e., by unioning $N-RPATHS_s(p)$, where p is a successor of n). In order to ensure that only paths that pass through n are considered, the result is intersected with $OnPaths(n)$.

Cost-Benefit Computation

$BENEFITPATHS_s(n)$ is a bit vector which holds the set of paths that benefit from predication enabled sinking of s past merge node n. It is computed by intersecting the paths in $X-APATHS_s(n)$ with the paths in $X-RPATHS_s(n)$.

$COSTPATHS_s(n)$ is a bit vector which holds the set of paths that incur a cost due to predication enabled sinking of s past merge node n. It is computed by intersecting the paths in $\overline{X-APATHS_s(n)}$ with the paths in $\overline{X-LPATHS_s(n)}$.

The computations of this step are given in Figure 4c.

The example in Figure 5 illustrates the results of the analysis. In this example the cost-benefit of sinking statement at node 2 past merge points 5 and 9 is shown. Sinking of the statement past node 5 enables elimination of dead code from paths P1 and P2 while an additional evaluation of predicated version of statement in node 2 is introduced along paths P7 and P8. For the given path profiles it is therefore beneficial to sink the statement past merge point 5. On the other hand when we consider sinking past node 9 the benefit is lower since only path P2 benefits from this sinking. It should be noted that to fully derive the benefits of sinking past node 5, the statement must also be moved past node 9. This is because sinking past 9 is required to eliminate deadness along path P2. Therefore if predication based sinking of statement is enabled at merge point 5, it should also be enabled at merge point 9.

2.1.2 The Cost of Cost-Benefit Analysis

The cost of performing cost-benefit analysis depends upon the number of paths that must be considered during analysis. While in general the number of static paths through a function can be very high, we found that in practice the number of paths that need to be considered is small. First only the paths with non-zero executions counts need to be considered. Second only the paths through a given function are considered at any one time. For the SPEC95 integer benchmarks we found that in 65% of the functions that were executed no more than 5 paths with non-zero frequency were found and only 1.4% of functions had over 100 paths. No function had greater than 1000 paths.

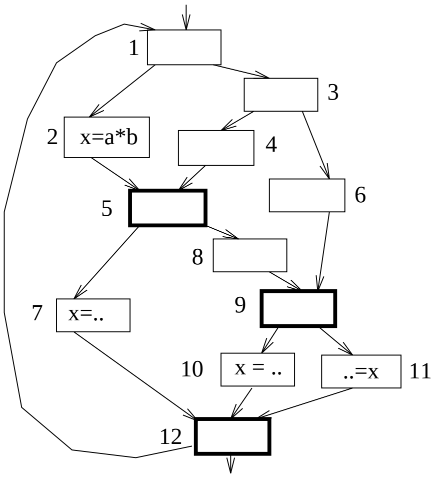
For the small fraction of functions that have relatively high number of paths, we are able to trade-off the time spent on performing cost-benefit analysis with the precision of the cost-benefit information. This is achieved by removing the infrequently executed paths from consideration. The estimates of cost and benefit computed using this approach are conservative, that is, the estimated cost is never lower than the true cost and the estimated benefit is never higher than the true benefit, where true cost and benefits are obtained by considering all paths.

Consider the example in Figure 5. Let us assume that we ignore the paths through node 10 during our analysis. Conservative analysis will make the worst case assumptions regarding this node by assuming that when predication enabled sinking of $x=a*b$ is performed at node 5, no benefits are derived and cost is incurred along the paths through node 10. In other words the analysis will assume that along path P2 no dead code is removed and along paths P5 and P6 a predicated version of statement $x=a*b$ will be introduced. Thus, the underestimated benefit at node 5 will be $125 \times T[x=a*b]$ and overestimated cost at node 5 will be 120.

The equations for computing the conservative estimates of cost and benefit are given below. The $CostPaths_s$ and $BenefitPaths_s$ information is computed only for the frequency executed paths (FP). Thus, in computing the cost at node n we obtain a conservative estimate by assuming that predicated versions of s will be placed along all paths in $P-FP$ that contain n. In computing the conservative estimate of the benefit we assume that dead code removal is not achieved for s along any of the paths in $P - FP$.

$$\begin{aligned} Cost_s(n) &\leq EstCost_s(n) \\ &= T[p?s] \\ &\quad \times [\sum_{p \in FP} CostPaths_s(n)(p) \times Freq(p) \\ &\quad + \sum_{\substack{p \in P-FP \\ \wedge n \in p}}^{Total} Freq(p)] \end{aligned}$$

$$\begin{aligned} Benefit_s(n) &\geq EstBenefit_s(n) \\ &= T[s] \times \sum_{p \in FP} BenefitPaths_s(n)(p) \\ &\quad \times Freq(p) \end{aligned}$$



(a)

Freq.	Path
125	P1: 1-2-5-7-12
25	P2: 1-2-5-8-9-10-12
10	P3: 1-2-5-8-9-11-12
10	P4: 1-3-4-5-7-12
10	P5: 1-3-4-5-8-9-10-12
10	P6: 1-3-6-9-10-12
50	P7: 1-3-4-5-8-9-11-12
50	P8: 1-3-6-9-11-12

(b)

Cost(5)<Benefit(5)
Cost(5) = Freq(P7+P8)*T[x=a*b] = (50+50)*T[x=a*b]
=100*T[x=a*b]
Benefit(5) = Freq(P1+P2)*T[p?x=a*b] = 125+25 = 150

Cost(9)>Benefit(9)
Cost(9) = Freq(P7+P8)*T[p?x=a*b] = 50+50 = 100
Benefit(9) = Freq(P2)*T[x=a*b] = 25*T[x=a*b]

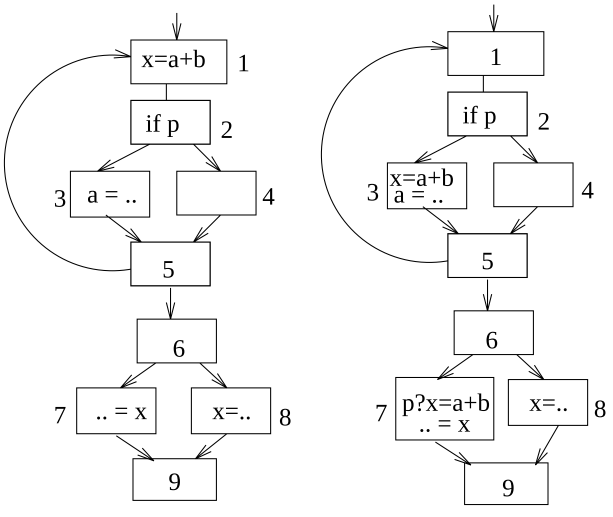
(c)

Figure 5: An Example of Cost-Benefit Analysis.

2.1.3 Cost-Benefit Analysis for Loops

In the presence of loops, initially the paths considered only include those paths that do not cross loop boundaries. In other words the program is viewed as a collection of acyclic subgraphs and only paths within these subgraphs are profiled. The application of our optimization described in the preceding discussion will perform predication enabled sinking within an acyclic graph if it is beneficial with respect to the profiles for that acyclic graph. However, in order to achieve beneficial sinking of expressions across loop boundaries, simple extensions in the treatment of nodes that connect the loop with surrounding code are required. These extensions allow us to take advantage of the benefits of moving expressions across loop boundaries.

Consider the movement of partially dead statement out of a loop as illustrated in Figure 6. The benefit of the optimization results from the removal of $x = a + b$ along the path 1-2-4-5 within the loop and the cost of optimization results from the placement of predicated execution of $x = a + b$ along path 6-7-9. If the path 1-2-4-5 is executed frequently while the path 6-7-9 is executed infrequently it is beneficial to apply this optimization. Note that in this case x is not live along the loop back edge (i.e., at the entry of the loop). If this was not the case, then we would have to place a copy of the statement along the loop back edge and the sinking beyond node 5 will not be beneficial.



(a) (b)

Benefit(5: x=a+b) = Freq(1-2-4-5)*T[x=a+b]
Cost(5: x=a+b) = Freq(6-7-9)*T[p?x=a+b]
(c)

Figure 6: Predication Enabled Sinking for Loops.

In summary we should enable the optimization past a merge point that is a loop exit as well as the tail of the loop if the following conditions given below hold. In these conditions *texit* denotes the tail of the loop that is also the loop exit and *postexit* is the node following *texit*.

$Benefit_s(texit) > Cost_s(texit)$, where

$$Benefit_s(texit) = T[s] \times \sum_{p \in X-APATHS_{x=exp}(texit)} Freq(p),$$
$$Cost_s(texit) = T[p?s] \times \sum_{p \in N-LPATHS_x(postexit)} Freq(p),$$

and $N - LIVE_x(head) = 0$.

2.2 Predication based Sinking Framework

The framework that we propose is an extension of the partial dead code elimination framework developed by Knoop et al. [19]. Knoop's framework involves two main steps that are applied repeatedly till no further optimization is possible. The first step performs *assignment sinking* to enable dead code elimination and the second step performs *assignment elimination* to remove dead code. The repeated application of the above steps is required due to second order effects.

The extended framework that we propose consists of three steps. The first step, *enable predication*, predication-based sinking at join points in the flow graph based upon the results of cost-benefit analysis. The second step performs *assignment sinking*. This step performs sinking that would have been performed by Knoop's algorithm as well as additional sinking enabled by predication. The final step of *assignment elimination* remains unchanged.

The data flow equations for enabling predication and the revised data flow equations for assignment sinking are presented in Figure 7. Predication enabled sinking is allowed at join nodes at which the cost of sinking is less than the benefit derived from sinking. In addition, sinking is also enabled at a join node if it has been enabled at an earlier join node. This is to ensure that the benefits of sinking computed for the earlier join node can be fully realized. Recall that this situation was encountered in the example of Figure 5 (in order to fully derive the benefits possible by enabling sinking at node 5, it is necessary to enable sinking at node 9). In order to derive the full benefit of sinking statement past merge point 5, our analysis will also enable sinking of the statement be also enabled at merge point 9. The application of this analysis to the example of Figure 6 with loops will cause predication to be enabled at merge point 5.

The assignment sinking analysis consists of two steps: *delayability analysis* which performs sinking and *insertion point computation* that identifies the points to which the statement must be placed following sinking. The *delayability* analysis has been extended to allow predication enabled sinking. This analysis is essentially responsible for determining how far the sinking of a statement can be allowed. Notice that in our analysis $N-DELAYED_s(n)$ is always set to true if $EPREDJOIN_s(n)$ is true, that is, sinking at the node has been enabled based upon the cost-benefit analysis in the preceding step. For the example in Figure 5 the delayability analysis will determine that the assignment $x = a * b$ can sink to nodes 7, 10 and 11. The paths along which delayability predicate is found to be true are shown by solid lines in Figure 9a. The *insertion points* will be found to be the entry points of the above nodes and the flow graph after assignment sinking is shown in Figure 9b. Similarly for the example in Figure 6 our analysis determines that the statement in node 1 (i.e., $x = a + b$) can sink to nodes 3, 7 and 8.

After moving the assignment statement to the appropriate points determined in the preceding step, those assignments that are completely dead are eliminated. For the example in Figure 5 the assignments at nodes 7 and 10 are eliminated (see Figure 9c) and for the example in Figure 6 the assignment introduced at node 8 is eliminated.

2.3 Predicate Evaluations

Once PDE has been performed, we must introduce evaluations of the predicate at appropriate points in the program. In some situations, such as the example of Figure 1, the required predicate is already computed by the program and we simply need to save its value in a predicate register so that it is available at the time that the predicated instruction is encountered. However, in general the program may not compute the required predicate and an alternate strategy that introduces assignments to a predicate variable is required.

The example in Figure 10 illustrates the approach based upon introduction of a predicate variable. A true assignment to the predicate variable is placed at a point that dominates the nodes in which the instruction originally resided and where it resides in its predicated form. In addition, the predicate is set to false if control flow follows a path along which the predicated statement is not to be executed. In general only a single true assignment is encountered; however, multiple false assignments may be encountered before execution reaches the predicated instruction. For the example in Figure 10, at most one true assignment

$$\mathbf{EPRED_s(n)} = \begin{cases} \mathbf{1} & \textbf{if } \mathbf{COST_s(n)} < \mathbf{BENEFIT_s(n)} \textbf{ and} \\ \mathbf{0} & \textbf{otherwise} \end{cases}$$

$$\mathbf{EPRED_s(n)} = \mathbf{N-REM_s(n)} \wedge \bigvee_{\mathbf{m \in Pred(n)}} \mathbf{EPRED_s(m)}$$

$$\mathbf{EPREDJOIN_s(n)} = \begin{cases} \mathbf{EPRED_s(n)} & \textbf{if n is a join point} \\ \mathbf{0} & \textbf{otherwise} \end{cases}$$

Figure 7: Enabling Predication.

Delayability Analysis :

$$\mathbf{X{-}DELAYED_s(n)} = \begin{cases} \mathbf{1} & \textbf{if } \mathbf{s} \in \mathbf{n} \\ \mathbf{N{-}DELAYED_s(n)} \wedge \overline{\mathbf{BLOCK}}_\mathbf{s}\mathbf{(n)} & \textbf{otherwise} \end{cases}$$

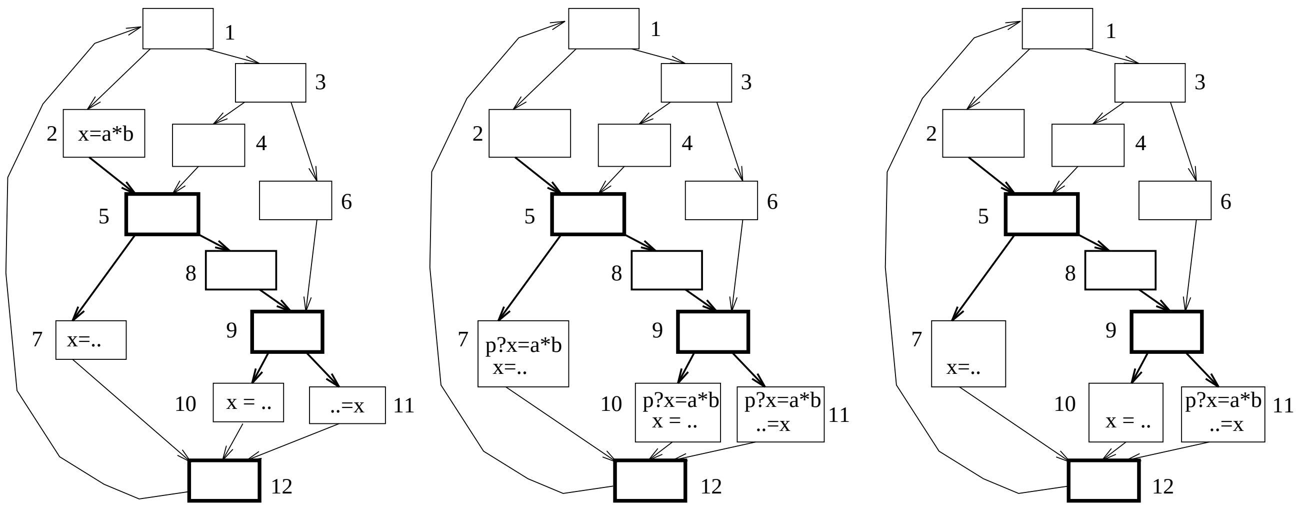
$$\mathbf{N{-}DELAYED_s(n)} = \begin{cases} \mathbf{0} & \textbf{if } \mathbf{n = start} \\ \mathbf{EPREDJOIN_s(n)} \vee \bigwedge_{\mathbf{m \in Pred(n)}} \mathbf{X{-}DELAYED_s(m)} & \textbf{otherwise} \end{cases}$$

Identifying Insertion Points :

$$\mathbf{X{-}INSERT_s(n)} = \mathbf{X{-}DELAYED_s(n)} \wedge \bigvee_{\mathbf{m \in Succ(n)}} \overline{\mathbf{N{-}DELAYED}}_\mathbf{s}\mathbf{(m)}$$

$$\mathbf{N{-}INSERT_s(n)} = \mathbf{N{-}DELAYED_s(n)} \wedge \mathbf{BLOCK_s(n)}$$

Figure 8: Assignment Sinking.



(a) After Delayability Analysis. (b) After Assignment Sinking. (c) After Assignment Deletion.

Figure 9: Application of PDE Algorithm.

and two false assignments may be encountered prior to reaching the predicate statement.

In some restricted situations it is possible to determine placements of predicate assignments such that exactly one predicate assignment is encountered prior to reaching the predicated instruction. A set of nodes satisfying this property is a generalized dominator of the node at which the predicated statement is placed [7]. Efficient algorithms for identifying these set of nodes can be found in [3].

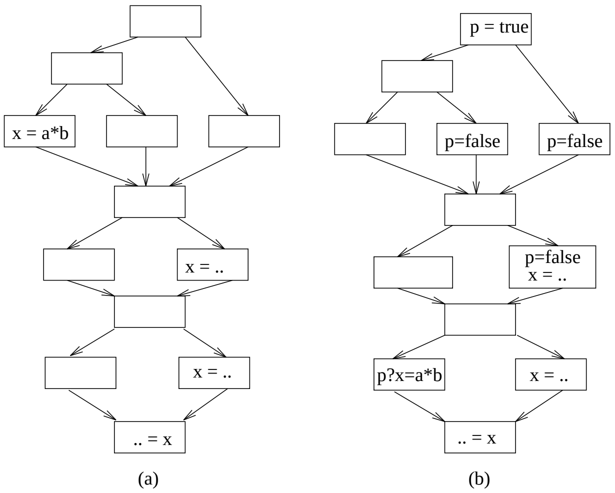


Figure 10: Introducing Predication Evaluation.

Finally it should be noted that there is a cost associated with the placement of an assignment. As long as the instruction to which PDE is being applied takes greater number of cycles to execute than the setting of a predicate register in the target architecture, application of PDE would be useful. Thus, our cost-benefit analysis can be easily modified to take into account the cost of setting the predicate register. The modified equations are given below:

$$
\begin{aligned}
EstCost_s(n) = {} & (T[p?s] + T[p]) \times \\
& [\sum_{p \in FP} CostPaths_s(n)(p) \times Freq(p) \\
& + \sum_{\substack{p \in P-FP \\ \wedge n \in p}}^{Total} Freq(p)]
\end{aligned}
$$

$$
\begin{aligned}
EstBenefit_s(n) = {} & (T[s] - T[p]) \times \\
& \sum_{p \in FP} BenefitPaths_s(n)(p) \\
& \times Freq(p)
\end{aligned}
$$

3 Other Applications

In this paper we demonstrated the use of path profile information to combine predication with the PDE optimization in order to aggressively optimize frequently executed paths through a program. The approach for cost-benefit analysis that we have presented is quite general and has also been applied to other problems including partial redundancy elimination [9], strength reduction [18], and load-store elimination from loops [3, 8].

The example in Figure 11 illustrates the application of our approach to partial redundancy elimination. In the first flow graph shown in Figure 11a, the evaluation of the expression $x + y$ in node 7 is partially redundant. Along paths that visit node 2 prior to reaching node 7 (i.e., paths $P1$ and $P2$) the expression is evaluated twice. A traditional PRE algorithm will not be able to remove this redundancy because it will not allow the expression evaluation in node 7 to be hoisted above node 6.

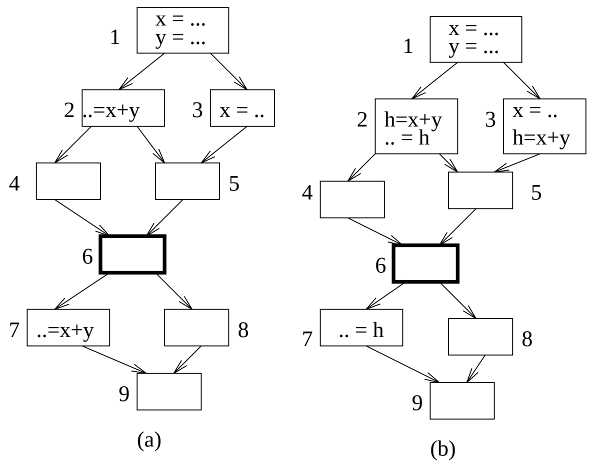


Figure 11: Path Profiles and Speculation for Partial Redundancy Elimination.

Now consider the second flow graph in Figure 11b in which the expression evaluation has been hoisted above node 6 using speculation (that is, unconditional execution of expression that is otherwise executed conditionally) and placed at node 3. Speculation has enabled the removal of redundancy along paths $P1$ and $P2$. This is referred to as the *benefit* of enabling speculation at conditional node 6. At the same time an additional evaluation of $x + y$ has been introduced along the path $P5$ which is referred to as the *cost* of enabling speculation at node 6. If the profile information indicates that the total number of times paths $P1$ and $P2$ are executed is expected to be greater than the number of times path $P5$ is executed, then the benefit derived from speculation at node 6 is greater than the cost of allowing speculation. Thus, in this situation it is beneficial to use the placement shown in Figure 11b.

In [9] we describe in detail the combination of code speculation with the PRE optimization [16, 15]. Speculation is enabled at conditional nodes based upon cost-benefit analysis. In [10] we present predication based PRE and speculation based PRE algorithms that are resource sensitive. These algorithms compute code placements where the functional unit required for the execution of the placed algorithm is available. Only the optimization opportunities for which such placements exist are exploited.

References

[1] T. Ball and J. Larus, "Efficient Path Profiling," *29th Annual IEEE/ACM International Symposium on Microarchitecture*, Paris, France, November 1996.

[2] R. Bodik and R. Gupta, "Partial Dead Code Elimination using Slicing Transformations," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

[3] R. Bodik and R. Gupta, "Array Data-Flow Analysis for Load-Store Optimizations in Superscalar Architectures," *International Journal of Parallel Programming*, Vol. 24, No. 6, pages 481-512, 1996.

[4] D.M. Dhamdhere, "Practical Adaptation of Global Optimization Algorithm of Morel and Renvoise," *ACM Transactions on Programming Languages*, 13(2):291-294, 1991.

[5] J.Z. Fang, "Compiler Algorithms on If-Conversion, Speculative Predicates Assignment and Predicated Code Optimizations," *Ninth Workshop on Languages and Compilers for Parallel Computers*, San Jose, California, August 1996.

[6] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, 30(7), July 1981.

[7] R. Gupta, "Generalized Dominators and Post-Dominators," *19th Annual ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pages 246-257, Albuquerque, New Mexico, January 1992.

[8] R. Gupta, "Code Optimization as a Side Effect of Instruction Scheduling," *International Conference on High Performance Computing*, Bangalore, India, December 1997.

[9] R. Gupta, D. Berson, and J.Z. Fang, "Path Profile Guided Partial Redundancy Elimination Using Speculation," *Technical Report TR-97-13*, Dept. of Computer Science, University of Pittsburgh, 1997.

[10] R. Gupta, D. Berson, and J.Z. Fang, "Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization," *The 30th Annual IEEE/ACM International Symposium on Microarchitecture*, Research Triangle Park, North Carolina, December 1997.

[11] R. Gupta and M.L. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Transactions on Software Engineering*, 16(4):421-431, April 1990.

[12] P. Hsu and E. Davidson, "Highly Concurrent Scalar Processing," *13th Annual International Symposium on Computer Architecture*, pages 386-395, 1986.

[13] W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *Journal of Supercomputing*, Vol. A, pages 229-248, 1993.

[14] V. Kathail, M. Schlansker, and B.R. Rau, "HPL PlayDoh Architecture Specification: Version 1.0," *Technical Report HPL-93-80*, Computer Systems Laboratory, HP Labs, Palo Alto, CA, February 1994.

[15] E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," *Communications of the ACM*, 22(2):96-103, 1979.

[16] J. Knoop, O. Ruthing, and B. Steffen, "Lazy Code Motion," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 224-234, 1992.

[17] J. Knoop, O. Ruthing, and B. Steffen, "The Power of Assignment Motion," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 233-245, 1995.

[18] J. Knoop, O. Ruthing, and B. Steffen, "Lazy Strength Reduction," *Journal of Programming Languages*, 1(1):71-91, 1993.

[19] J. Knoop, O. Ruthing, and B. Steffen, "Partial Dead Code Elimination," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 147-158, 1994.

[20] S.A. Mahlke, W.Y. Chen, R.A. Bringmann, R.E. Hank, W.W. Hwu, B. Rau, and M. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors," *ACM Transactions on Computer Systems*, 11(4):376-408, Nov. 1993.

[21] G. Ramalingam, "Data Flow Frequency Analysis," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 267-277, 1996.

[22] M.S. Schlansker and V. Kathail, "Critical Path Reduction for Scalar Processors," *28th Annual IEEE/ACM International Symposium on Microarchitecture*, Ann Arbor, Michigan, November 1995.

[23] B. Steffen, "Data Flow Analysis as Model Checking," *Proceedings TACS'91*, Sendai, Japan, Springer-Verlag, LNCS 526, pages 346-364, 1991.