Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization *

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

Instruction schedulers employ code motion as a means of instruction reordering to enable scheduling of instructions at points where the resources required for their execution are available. In addition, driven by the profiling data, schedulers take advantage of predication and speculation for aggressive code motion across conditional branches. Optimization algorithms for partial dead code elimination (PDE) and partial redundancy elimination (PRE) employ code sinking and hoisting to enable optimization. However, unlike instruction scheduling, these optimization algorithms are unaware of resource availability and are incapable of exploiting profiling information, speculation, and predication. In this paper we develop data flow algorithms for performing the above optimizations with the following characteristics: (i) opportunities for PRE and PDE enabled by hoisting and sinking are exploited; (ii) hoisting and sinking of a code statement is driven by availability of functional unit resources; (iii) predication and speculation is incorporated to allow aggressive hoisting and sinking; and (iv) path profile information guides predication and speculation to enable optimization.

1 Introduction

Data flow analysis provides us with facts about a program by statically analyzing the program. Algorithms for partial dead code elimination (PDE) [16, 6, 4, 20] and partial redundancy elimination (PRE) [17] solve a series of data flow problems to carry out sinking of assignments and hoisting of expression evaluations. The sinking of an assignment eliminates executions of the assignment that compute values that are dead, i.e., values that are never used. The hoisting of an expression eliminates evaluations of the expression if a prior evaluation of the same expression was performed using the same operands. The existing algorithms for above optimizations suffer from the following drawbacks that limit their usefulness in a realistic compiling environment in which the optimization phase precedes the instruction scheduling phase.

- The optimization algorithms are insensitive to *resource* information. Thus, it is possible that instructions that require the same functional unit resource for execution are moved next to each other. Such code motion is not beneficial since the instruction scheduler in separating these instructions from each other may undo the optimization.
- The data flow analyses used by the algorithms are incapable of exploiting *profiling* information to drive code sinking and hoisting. Instruction schedulers on the other hand use profiling information to drive code hoisting and sinking.
- The data flow analyses do not incorporate *speculation* and *predication* to enable code hoisting and sinking. Instruction schedulers for modern processor architectures exploit speculation [18] and predication [13, 14] during hoisting and sinking.

In this paper we present solutions to the above problems by developing data flow analysis techniques for PRE and PDE that incorporate both resource availability and path profiling [1] information. Furthermore, the formulations of hoisting and sinking are generalized to incorporate speculation based hoisting and predication enabled sinking. Our approach performs code motion for PRE and PDE optimizations

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

such that code motion is restricted to situations in which the resulting code placement points are those at which the required functional unit resources are available. Moreover we are able to perform code motion more freely than existing PDE and PRE algorithms [16, 17, 21, 5, 15] through speculative hoisting and predication enabled sinking. Finally, path profiling [1] information is used to guide speculation and predication. In particular, speculation and predication is applied only if their overall benefit in form of increased code optimization along frequently executed program paths is greater than their cost in terms of introducing additional instructions along infrequently executed program paths.

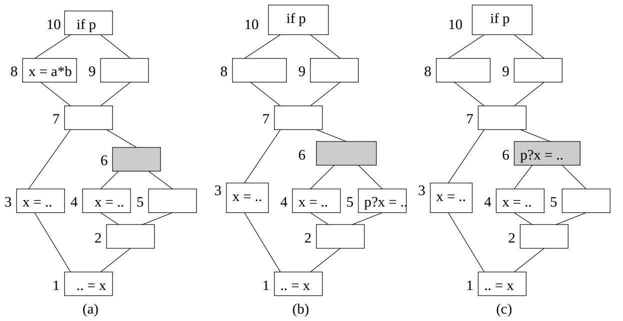


Figure 1: Resource Sensitive Code Sinking.

The example in Figure 1 illustrates our approach for code sinking. The flow graph in Figure 1a contains statement $x = a * b$ in node 8 which is partially dead since the value of x computed by the statement is not used along paths 10-8-7-3-1 and 10-8-7-6-4-2-1. Through sinking of the statement to node 5, as shown in Figure 1b, we can completely eliminate the deadness of the statement. Note that in order to enable sinking past node 7 it is necessary to predicate the statement. Furthermore this sinking should only be performed if the paths 10-8-7-3-1 and 10-8-7-6-4-2-1 along which dead code is eliminated are executed more frequently than the path 10-9-7-6-5-2-1 along which an additional instruction has been introduced.

If the functional unit for the multiply operation is expected to be busy at node 5 and idle at node 6, then resource sensitive sinking will place the statement at node 6 as shown in Figure 1c. As before, the predication of the statement is required to perform sinking past node 7. However, the sinking past 7 is only performed if the frequency with which the path 10-8-7-3-1 (along which dead code is eliminated) is executed is greater than the sum of the frequencies with which paths 10-9-7-6-4-2-1 and 10-9-7-6-5-2-1 (along which an additional instruction is introduced) are executed. The above solution essentially places the statement at a point where the required resource is available and eliminates as much deadness as possible in the process. It also performs predication enabled sinking whenever it is useful.

The example in Figure 2 illustrates our approach for code hoisting. The flow graph in Figure 2a contains partially redundant evaluation of expression $x+y$ in node 8 since the value of $x + y$ is already available at node 8 along paths 1-3-7-8-10 and 1-2-4-6-7-8-10. Through hoisting of the expression to node 5, as shown in Figure 1b, we can eliminate the redundancy. Note that in order to enable hoisting above node 7, it is necessary to perform speculative motion of $x + y$. Furthermore this hoisting should only be performed if the paths 1-3-7-8-10 and 1-2-4-6-7-8-10 along which redundancy is eliminated are executed more frequently than the path 1-2-5-6-7-9-10 along which an additional instruction is introduced.

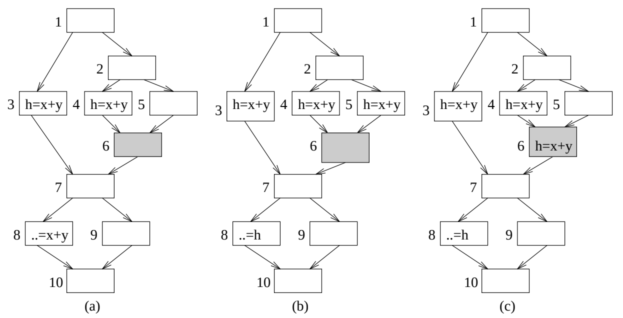


Figure 2: Resource Sensitive Code Hoisting.

If the functional unit for the add operation is expected to be busy at node 5 and idle at node 6, then resource sensitive hoisting will place the statement at node 6 as shown in Figure 1c. As before, speculative execution of the statement is required to perform hoisting above node 7. However, the hoisting past 7 is only performed if the frequency with which path 1-3-7-8-10 (along which redundancy is eliminated) is executed is greater than the frequency with which the paths 1-2-4-6-7-9-10 and 1-2-5-6-7-9-10 (along which an additional instruction is introduced) are executed. The above solution places the expression at a point where the required resource is available and eliminates as much redundancy as possible in the process. It also performs speculative code hoisting whenever it is useful.

The remainder of this paper is organized as follows. In section 2 we first present extensions to PDE and PRE solutions developed by Knoop et al. [16, 17] to achieve resource sensitive PDE and PRE. In section 3 we present extensions to resource sensitive PDE and PRE algorithms that incorporate path profiling information to drive speculation and predication. Concluding remarks are given in section 4.

2 Resource-Sensitive Code Motion and Optimization

The basic approach used by our algorithms is to first perform analysis that determines whether or not resources required by a code statement involved in sinking(hoisting) will be available along paths originating(terminating) at a point. This information is used by the algorithms for PDE(PRE) to inhibit sinking(hoisting) along paths where the required resource is not free. Therefore, optimization opportunities are exploited only if they are permitted by resource usage characteristics of a program.

In all the algorithms presented in this paper we assume that the program is represented using a flow graph in which each intermediate code statement appears in a distinct node and nodes have been introduced along critical edges to allow code placement along the critical edges. The first assumption simplifies the discussion of our data flow equations and is not essential for our techniques to work. Given this representation, we next describe how to determine whether a resource is locally free at the exit or entry of a node. This local information will be propagated through the flow graph to determine global resource availability information. In the subsequent sections we present resource analysis which is applicable to acyclic graphs. The extension to loops is straightforward and is based upon the following observation. A resource free with in a loop is not available for hoisting/sinking of instructions from outside the loop since instructions are never propagated from outside the loop to inside the loop.

Definition: A functional unit resource FU to which an instruction may be issued every c cycles is **free** at node n if the instructions issued during the $c-1$ cycles prior to n and following n are not issued to FU.

The above definition guarantees that if an instruction is placed at n that uses FU, then its issuing will not be blocked due to unavailability of FU. Notice that the above definition can be easily extended to consider situations in which more than one copy of the resource FU is available in the architecture.

2.1 Code Sinking and PDE

Partial dead code elimination is performed by sinking partially dead assignments. Through sinking of a partially dead assignment s we migrate s to program points where resources required by s are available and at the same time we remove s from some paths along which it is dead, that is, the value computed by s is not used. In order to ensure that sinking of s is only performed if it can be guaranteed that placement points for s after sinking will be ones at which the resource required for its execution is available, we first develop resource anticipatability analysis. The results of this analysis guide the sinking in a subsequent phase that performs resource sensitive code sinking and PDE.

Resource Anticipatability Analysis

The *resource anticipatability* data flow analysis determines the nodes past which the sinking of an assignment s will not be inhibited by the lack of resources.

Definition: A functional unit resource required to execute an assignment statement s is **anticipatable** at the entry of a node n if for each path p from n's entry to the terminating node in the flow graph one of the following conditions is true:

- the value of the variable defined by s at n's entry is *dead* along path p, that is, the resource will not be needed along this path since the assignment can be removed from it; or
- there is a node b along p in which the required resource is *locally free* and statement s is *sinkable* to b, that is, the required resource will be available after sinking.

To perform resource anticipatability analysis for an assignment s, we associate the following data flow variables with each node:

$\mathbf{PRES_s(n)}$ is 1 if given that the resource required by s is anticipatable at n's exit, it is also anticipatable at n's entry; otherwise it is 0. In particular, $\mathbf{PRES_s(n)}$ is 1 if the statement in n does not define a variable referenced (defined or used) by s.

$\mathbf{DEAD_s(n)}$ is 1 if the variable defined by s is dead at n's entry (i.e., the value of the variable at n's entry is never used). If the variable is not dead, then the value of $\mathbf{DEAD_s(n)}$ is 0.

$\mathbf{FREE_s(n)}$ is 1 if the resource required by s is free for its use when s is moved to n through sinking; otherwise it is 0.

$\mathbf{X-RANT_s(n)}(\mathbf{N-RANT_s(n)})$ is 1 if the resource required by s is anticipatable at n's exit(entry); otherwise it is 0.

In order to compute *resource anticipatability* we perform backward data flow analysis with the *and* confluence operator as shown in the data flow equations given below. The resource used by s is anticipatable at n's exit if it is anticipatable at the entries of all successors of n. The resource is anticipatable at n's entry if the resource is free for use by s in n, or the variable defined by s is dead at n's entry, or the resource is anticipatable at n's exit and preserved through n.

$$\mathbf{X{-}RANT_s(n)} = \bigwedge_{\mathbf{m \in Succ(n)}} \mathbf{N{-}RANT_s(m)}$$

$$\mathbf{N{-}RANT_s(n)} = \mathbf{FREE_s(n)} \vee \mathbf{DEAD_s(n)} \vee (\mathbf{X{-}RANT_s(n)} \wedge \mathbf{PRES_s(n)})$$

Assignment Sinking

The assignment sinking and PDE framework that we use next is an extension of the framework developed by Knoop et al. [16]. PDE is performed in the following steps: *assignment sinking* followed by *assignment elimination*. The first step is modified to incorporate resource anticipatability information while the second step remains unchanged. Assignment sinking consists of *delayability* analysis followed by identification of *insertion points* for the statement being moved. The data flow equations for delayability analysis and the computation of insertion points are specified below. In this analysis $\mathbf{X{-}DLY_s(n)}(\mathbf{N{-}DLY_s(n)})$ is 1 if assignment s can be delayed up to the exit(entry) of node n. $\mathbf{BLOCK_s(n)}$ is 1 for a node that blocks sinking of s due to data dependences; otherwise it is 0. As we can see delayability analysis only allows sinking of s to the entry of node n if the required resource is anticipatable at n's entry and it allows sinking from entry to exit of n if s is not blocked by n. The assignment is removed from its original position and inserted at points that are determined as follows. The assignment s is inserted at n's entry if it is delayed to n's entry but not its exit and s is inserted at n's exit if it is delayed to n's exit but not to the entries of all of n's successors. Assignment deletion eliminates the inserted assignments that are fully dead.

$$\mathbf{X{-}DLY_s(n)} = \begin{cases} \mathbf{1} & \mathbf{if\ s \in n} \\ \mathbf{N{-}DLY_s(n)} \wedge \neg \mathbf{BLOCK_s(n)} & \mathbf{owise} \end{cases}$$

$$\mathbf{N{-}DLY_s(n)} = \begin{cases} \mathbf{0} & \mathbf{if\ n = start} \\ \mathbf{N{-}RANT_s(n)} \wedge \bigwedge_{\mathbf{m \in Pred(n)}} \mathbf{X{-}DLY_s(m)} & \mathbf{owise} \end{cases}$$

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

$$\mathbf{X{-}INSERT_s(n)} = \mathbf{X{-}DLY_s(n)} \wedge \bigwedge_{\mathbf{m \in Succ(n)}} \neg \mathbf{N{-}DLY_s(m)}$$

The example in Figure 3 illustrates our algorithms by considering the sinking of assignment in node 1. In Figure 3a a control flow graph and the results of resource anticipatability analysis are shown. The node 7 is partially shaded to indicate that the resource is anticipatable at the node's exit but not its entry. Figure 3b shows the outcome of delayability analysis. Notice that the sinking of assignment past node 3 is inhibited since the resource is not anticipatable at 3's successors. Insertion point computation identifies three insertion points, the exit of 3, entry of 7 and exit of 8. Of these points the assignment is dead at 7's entry and is therefore deleted. Deadness of assignment along path 1-2-4-7-10-11 is removed while along path 1-2-3-5-9-11 it is not removed. Notice that finally $x = a + b$ is placed at nodes 3 and 8 where the resource is locally free.

2.2 Code Hoisting and PRE

Partial redundancy elimination is performed by hoisting expression evaluations. Through hoisting of a partially redundant expression e we migrate e to program points where resources required by e are available and at the same time we remove e evaluations from some paths along which e is computed multiple times making the later evaluations of e along the path redundant. In order to ensure that hoisting of e is only performed if it can be guaranteed that placement points for e after hoisting will be ones at which the resource required for e's execution is free, we perform resource availability analysis. Its results are used to guide hoisting in a subsequent phase that performs resource sensitive code hoisting and PRE.

Resource Availability Analysis

The resource availability data flow analysis determines the nodes above which the hoisting of statement s will not be inhibited by the lack of resources.

Definition: A functional unit resource needed to execute the operation in an expression e is **available** at the entry of node n if for each path p from the *start* node to n's entry one the following conditions is true:

- there is a node b in which resource is locally *free* and along the path from b to node n's entry, the variables whose values are used in e are *not redefined.* This condition ensures that upon hoisting the expression along the path a point would be found where the resource is free.

- there is a node b which computes e and after its computation the variables whose values are used in e are *not redefined.* This condition essentially

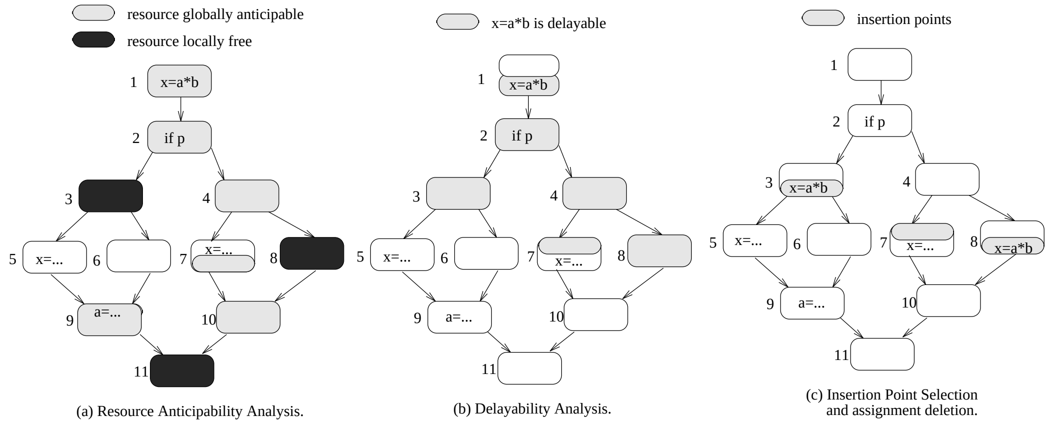


Figure 3: An Example of Resource-Sensitive PDE.

implies that if an earlier evaluation of the expression exists along a path then no additional use of the resource is required during hoisting along that path since the expression being hoisted will be eliminated along that path.

To perform resource availability analysis for an expression e, we associate the following data flow variables with each node:

$\mathbf{PRES_e(n)}$ is 1 if given that the resource required by e is available at n's entry, it is also available at n's exit; otherwise it is 0. In particular, $\mathbf{PRES_e(n)}$ is 1 if the statement in n does not define a variable used by e.

$\mathbf{USED_e(n)}$ is 1 if the statement in n evaluates the expression e and this evaluation of e is available at n's exit, that is, the variables used by e are not redefined in n after e's evaluation.

$\mathbf{FREE_e(n)}$ is 1 if the required resource is free for use by e in n when e is moved to n through hoisting; otherwise it is 0.

$\mathbf{N{-}RAVL_e(n)(X{-}RAVL_e(n))}$ is 1 if the resource required by e is available at n's entry(exit); otherwise it is 0.

In order to compute resource availability we perform forward data flow analysis with the *and* confluence operator. The resource used by expression e is available at n's entry if it is available at the exits of predecessors of n. The resource is available at n's exit if it is available at n's entry and preserved by n, the expression is computed by n and available at n's exit, or resource is locally free at n.

$$\mathbf{N{-}RAVL_e(n)} = \bigwedge_{\mathbf{m \in Pred(n)}} \mathbf{X{-}RAVL_e(m)}$$

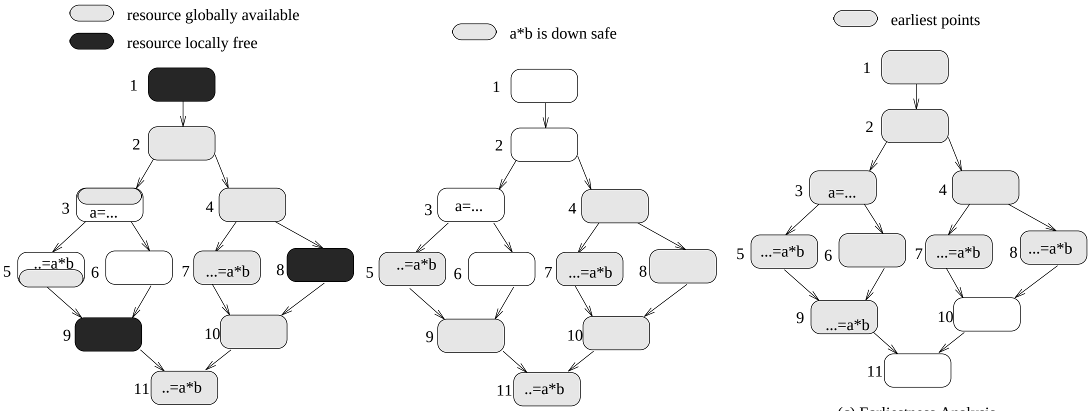
$$\mathbf{X{-}RAVL_e(n)} = (\mathbf{N{-}RAVL_e(n)} \wedge \mathbf{PRES_e(n)}) \vee \mathbf{USED_e(n)} \vee \mathbf{FREE_e(n)}$$

Expression Hoisting

The expression hoisting and PRE framework that we use next is a modification of the code motion framework developed by Knoop et al. [17]. PRE is performed in two steps: *down-safety analysis* which determines the points to which expression evaluations can be hoisted and *earliestness analysis* which locates the earliest points at which expression evaluations are actually placed to achieve PRE. These steps are modified to incorporate resource availability information. The modified equations for down-safety and earliestness analysis are given below. In these equations $\mathbf{N{-}DSF_e(n)(X{-}DSF_e(n))}$ is 1 if the expression can be hoisted to the entry(exit) of node n; otherwise it is 0. An expression is down-safe at a node as long as it is anticipatable along all paths leading from the node and the required resource is available along all paths leading to the node. The earliestness analysis sets $\mathbf{N{-}ERLY_e(n)(X{-}ERLY_e(n))}$ to 1 up to and including the first down-safe point at which the required resource is free or an expression evaluation exists. Along each path an expression evaluation is placed at the earliest down-safe point. These points are identified by the boolean predicates $\mathbf{N{-}DSafeEarliest_e}$ and $\mathbf{X{-}DSafeEarliest_e}$.

$$\mathbf{X{-}DSF_e(n)} = \begin{cases} \mathbf{0} & \mathbf{if\ n = exit} \\ \mathbf{X{-}RAVL_e(n)} \wedge \bigwedge_{\mathbf{m \in Succ(n)}} \mathbf{N{-}DSF_e(m)} & \mathbf{owise} \end{cases}$$

$$\mathbf{N{-}DSF_e(n)} = \begin{cases} \mathbf{1} & \mathbf{if\ USED_e(n) = 1} \\ \mathbf{X{-}DSF_e(n)} \wedge \mathbf{PRES_e(n)} & \mathbf{owise} \end{cases}$$



(a) Resource Availability Analysis. (b) Down Safety Analysis. (c) Earliestness Analysis and PRE transformation.

Figure 4: An Example of Resource-Sensitive PDE.

$$\mathbf{N{-}ERLY_e(n)} = \begin{cases} \mathbf{1} & \mathbf{if\ n\ =\ start} \\ \bigvee_{\mathbf{m \in Pred(n)}} \neg(\mathbf{X{-}DSF_e(m)} \wedge (\mathbf{FREE_e(m)} \vee \mathbf{USED_e(m)})) \wedge \mathbf{X{-}ERLY_e(m)} & \mathbf{owise} \end{cases}$$

$$\mathbf{X{-}ERLY_e(n)} = \mathbf{N{-}ERLY_e(n)} \vee \neg\mathbf{PRES_e(n)}$$

$$\mathbf{X{-}DSafeEarliest_e(n)} = \mathbf{X{-}ERLY_e(n)} \wedge \mathbf{X{-}DSF_e(n)} \wedge (\mathbf{FREE_e(n)} \vee \mathbf{USED_e(n)}) \wedge \bigvee_{\mathbf{m \in Succ(n)}} \neg\mathbf{N{-}ERLY_e(m)}$$

$$\mathbf{N{-}DSafeEarliest_e(n)} = \mathbf{N{-}ERLY_e(n)} \wedge \mathbf{N{-}DSF_e(n)} \wedge (\mathbf{FREE_e(n)} \vee \mathbf{USED_e(n)}) \wedge \neg(\mathbf{X{-}DSafeEarliest_e(n)} \wedge \mathbf{PRES_e(n)})$$

The example in Figure 4 illustrates the above algorithms. In Figure 4a the results of resource availability analysis are shown. Since the resource is available at node 11, the down-safety analysis will propagate it backwards as shown in Figure 4b. Node 6 is not down-safe because resource is not available at that node. On the other hand node 8 is down-safe because the resource is available at that node. The earliestness analysis identifies nodes 5, 9, 7, and 8 as the first nodes which are down-safe and where either resource is free (nodes 8 and 9) or expression evaluation exists (node 5 and 7). The final placement of the expression is shown in Figure 4c. Traditional approach would have hoisted the expression above node 9 to node 6.

3 Profile-Directed Resource-Sensitive Code Motion and Optimization

In this section we show that additional opportunities for PRE and PDE optimizations can be exploited by enabling more aggressive code hoisting and sinking. Speculative code hoisting can be performed to enable additional opportunities for PRE while predication based code sinking can be employed to enable additional opportunities for PDE. However, while speculative hoisting and predication based sinking result in a greater degree of optimization along some program paths, they result in introduction of additional instructions along other program paths. In other words a greater degree of code optimization is achieved for some program paths at the expense of introduction of additional instructions along other program paths.

While generating code for VLIW and superscalar architectures, speculation and predication are routinely exploited to generate faster schedules along frequently executed paths at the expense of slower schedules along infrequently executed paths [7, 12, 11]. However, the optimization frameworks today are unable to exploit the same principle. In this section we show how to perform PRE and PDE optimizations by using speculation and predication. Frequently executed paths are optimized to a greater degree at the expense of infrequently executed paths. Path profiling information is used to evaluate the benefits and costs of speculation and predication to program paths.

In the subsequent sections we describe code hoisting and sinking frameworks which use path profiling [1] information to enable speculation and predication based hoisting and sinking while inhibiting hoisting and sinking using resource availability and anticipatability information. This results in optimization algorithms that are more aggressive than traditional algorithms [16, 17] while at the same time more appropriate for VLIW and superscalar environment as they are resource sensitive and can trade-off the quality of code for frequently executed paths with that of infrequently executed paths. Although the techniques we describe are based upon path profiling information they can also be adapted for edge profiles since estimates of path profiles can be computed from edge profiles [19]. Furthermore we present versions of our algorithms that apply to acyclic graphs. However, the extensions required to handle loops are straightforward and can be found in [10, 9].

Our algorithms are based upon the following analysis steps. First resource availability and anticipatability analysis is performed. Next we determine the cost and benefit of enabling speculation and predication at various spilt points and merge points in a flow graph respectively. The benefit is an estimation of increased optimization of some program paths while the cost is an estimate of increase in the number of instructions along other program paths. By selectively enabling hoisting and sinking at program points based upon cost-benefit analysis, we exploit optimization opportunities that the traditional algorithms such as those by Knoop et al. [16, 17] do not exploit while inhibiting optimization opportunities that result in movement of code to program points at which the resource required for an instructions execution is not free.

3.1 Path Profile Directed PDE

As mentioned earlier, the resource anticipatability analysis described in section 2.1 remains unchanged and must be performed first. Next cost-benefit analysis that incorporates resource anticipatability information and uses path profiles is performed. The results of this analysis are used to enable predication enabled sinking at selected join points in the next phase. Finally an extension of the sinking framework presented in section 2.1 is used to perform resource-sensitive, profile-guided PDE.

The cost-benefit analysis consists of three steps: (a) *Availability analysis* identifies paths leading to a node along which a statement is available for sinking (i.e., sinking is not blocked by data dependences) at various program points; (b) *Optimizability analysis* identifies paths originating at a node along which a statement can be optimized because the value computed by it is not live and the sinking required for its removal is not inhibited by the lack of a free resource or presence of data dependences; and (c) *Cost-benefit computation* identifies the paths through a join point along which additional optimization is achieved or an additional instruction is introduced when predication based sinking is enabled at the join point. By summing the frequencies of the respective paths, provided by path profiles, the values of cost and benefit are obtained.

The set of paths identified during availability and optimizability analysis are 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 $OnPs(n)$ 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.

The steps of the analysis are described next.

Availability Analysis

In the data flow equations for availability analysis given below $N-AVL_a(n)(X-AVL_a(n))$ 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. $N-APS_a(n)(X-APS_a(n))$ is a bit vector which holds the set of paths along which the value of $N-AVL_a(n)(X-AVL_a(n))$ is 1 at n's entry(exit).

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 a is encountered, and it is set to 0 if a statement that blocks the sinking of a is encountered. In the equations $PRES_a(n)$ is a one bit variable which is 1(0) if n preserves a, that is, n is not data (anti, output or flow) dependent upon a. At the entry to a node n for which $N-AVL_a(n)$ is 0, the set of paths is set to null, that is, to $\vec{0}$. Otherwise the paths in $N-APS_a(n)$ are computed by unioning the sets of paths along which a is available at the exit of one of n's predecessors (i.e., unioning $X-APS_a(p)$, 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 $OnPs(n)$. The value of $X-APS_a(n)$ is $OnPs(n)$ if n contains a and $N-APS_a(n)$ if n does not block a.

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

$$\mathbf{X-AVL_a(n)} = \begin{cases} \mathbf{1} & \textbf{if } \mathbf{a \in n} \\ \mathbf{PRES_a(n) \wedge N-AVL_a(n)} & \textbf{owise} \end{cases}$$

$$\mathbf{N-APS_a(n)} = \begin{cases} \vec{\mathbf{0}} & \textbf{if } \mathbf{N-AVL_a(n) = 0} \\ \mathbf{OnPs(n)} \wedge \bigvee_{\substack{\mathbf{m \in Pred(n) \wedge} \\ \mathbf{X-AVL_a(m)=1}}} \mathbf{X-APS_a(m)} & \textbf{owise} \end{cases}$$

$$\mathbf{X-APS_a(n)} = \begin{cases} \mathbf{OnPs(n)} & \textbf{if } \mathbf{a \in n} \\ \mathbf{N-APS_a(n)} & \textbf{if } \mathbf{X-AVL_s(n) = 1} \\ \vec{\mathbf{0}} & \textbf{owise} \end{cases}$$

Optimizability Analysis

$N-OPT_a(n)(X-OPT_a(n))$ is a one bit variable associated with n's entry(exit) which is 1 if there is a path through n along which a is dead and any sinking of a that may be required to remove this deadness is feasible (i.e., it is not inhibited by lack of resources or presence of data dependences); 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 a is feasible, the results of a's availability analysis and resource anticipatability analysis are used. For example, if variable v computed by a is dead at n's exit, then $X-OPT_a(n)$ is set to true only if $X-AVL_a(n)$ is true because the deadness will only be eliminated if sinking of a to n's exit is not blocked by data dependences. If v is not dead then among other conditions we also check if $X-RANT_a(n)$ is true because the sinking of a will only be allowed if the resource required for a's execution along paths where v is not dead is free. In the data flow equations $N-DEAD_v(n)(X-DEAD_v(n))$ 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.

$N-OPS_a(n)(X-OPS_a(n))$ is a bit vector which holds the set of paths along which the value of $N-OPT_a(n)(X-OPT_a(n))$ 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 $N-AVL_a(n)(X-AVL_a(n))$ are 1, $N-OPS_a(n)(X-OPS_a(n))$ is set to $OnPs(n)$. Otherwise the paths in $X-OPS_a(n)$ are computed by unioning the sets of paths along which a is partially dead and removable at the entry of one of n's successors (i.e., by unioning $O-RPS_a(p)$, 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 $OnPs(n)$.

let v be the variable defined by s, i.e., v = lhs(s) :

$$\mathbf{X-OPT_a(n)} = \begin{cases} \mathbf{X-AVL_s(n)} & \mathbf{if\ X-DEAD_v(n)=1} \\ \mathbf{X-RANT_a(n) \wedge X-AVL_s(n) \wedge} & \mathbf{owise} \\ \quad \bigvee_{\mathbf{m \in Succ(n)}} \mathbf{N-OPT_a(m)} & \end{cases}$$

$$\mathbf{N-OPT_a(n)} = \begin{cases} \mathbf{N-AVL_s(n)} & \mathbf{if\ N-DEAD_v(n)=1} \\ \mathbf{N-AVL_a(n) \wedge X-OPT_a(n)} & \mathbf{owise} \end{cases}$$

$$\mathbf{X-UPS_a(n)} = \begin{cases} \mathbf{OnPs(n)} & \mathbf{if\ X-DEAD_v(n) \wedge X-AVL_a(n)=1} \\ \mathbf{OnPs(n) \wedge} \bigvee_{\substack{\mathbf{m \in Succ(n) \wedge} \\ \mathbf{N-OPT_a(m)=1}}} \mathbf{N-UPS_a(m)} & \mathbf{owise} \end{cases}$$

$$\mathbf{N-UPS_a(n)} = \begin{cases} \mathbf{OnPs(n)} & \mathbf{if\ N-DEAD_v(n) \wedge N-AVL_a(n)=1} \\ \mathbf{X-UPS_a(n)} & \mathbf{if\ N-UPS_a(n)=1} \\ \vec{\mathbf{0}} & \mathbf{owise} \end{cases}$$

Cost-benefit Computation

The *cost* of enabling predication of a partially dead statement a to allow its movement below a merge point n is determined by identifying paths through the merge point along which in the unoptimized program a is not executed and in the optimized program a predicated version of a is executed. Furthermore resource anticipatability analysis indicates that along paths where predicated version of a is placed, the resource needed by a is available. The sum of the execution frequencies of the above paths, as indicated by path profiles, is the cost.

$$\mathbf{COST_a(n)} = \sum_{\mathbf{i}} (\overline{\mathbf{X-OPS}}_{\mathbf{a}}\mathbf{(n)(i)} \wedge \overline{\mathbf{X-APS}}_{\mathbf{a}}\mathbf{(n)(i)}) \times \mathbf{FREQ(path(i))}$$

The *benefit* of enabling predication of a partially dead statement a to allow its movement below a merge point n is determined by identifying paths through the merge point along which in the unoptimized program a is executed while in the optimized program a is not executed. Furthermore the resource anticipatability analysis indicates that the sinking of a required to achieve the above benefit is not inhibited by lack of resources. The sum of the execution frequencies of the above paths, as indicated by path profiles, is the benefit.

$$\mathbf{BENEFIT_a(n)} = \sum_{\mathbf{i}} (\mathbf{X-OPS_a(n)(i)} \wedge \mathbf{X-APS_a(n)(i)}) \times \mathbf{FREQ(path(i))}$$

Code Sinking Framework

The results of the cost-benefit analysis are incorporated into a code sinking framework in which predication of a code statement is enabled with respect to the merge points only if resources are available and the benefit of predication enabled sinking is determined to be greater than the cost of predication enabled sinking. This framework is an extension of the code sinking framework presented in section 2.1.

The data flow equations for enabling predication are presented next. 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.

$$\mathbf{EPRED_a(n)} = \begin{cases} \mathbf{1} & \mathbf{if\ COST_a(n) < BENEFIT_a(n)} \\ \mathbf{0} & \mathbf{owise} \end{cases}$$

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

$$\mathbf{EPJOIN_a(n)} = \begin{cases} \mathbf{EPRED_a(n)} & \textbf{if n is a join point} \\ \mathbf{0} & \textbf{owise} \end{cases}$$

The delayability analysis of section 2.1 is modified to incorporate the results of enabling predication as shown below. At a join point if predication based sinking is enabled then as long as the assignment is available along some path (as opposed to all paths in section 2.1), it is allowed to propagate below the join node.

$$\mathbf{X{-}DLY_a(n)} = \begin{cases} \mathbf{1} & \textbf{if a} \in \textbf{n} \\ \mathbf{N{-}DLY_a(n) \wedge PRES_a(n)} & \textbf{owise} \end{cases}$$

$$\mathbf{N{-}DLY_a(n)} = \begin{cases} \mathbf{0} & \textbf{if n = start} \\ \mathbf{(N{-}RANT_a(n) \wedge \bigwedge_{m \in Pred(n)} X{-}DLY_a(m))} & \textbf{if n is} \\ \mathbf{\vee\, (EPJOIN_a(n) \wedge \bigvee_{m \in Pred(n)} X{-}DLY_a(m))} & \textbf{a join} \\ \mathbf{N{-}RANT_a(n) \wedge \bigwedge_{m \in Pred(n)} X{-}DLY_a(m)} & \textbf{owise} \end{cases}$$

Consider the paths that contribute to cost and benefit of sinking assignment $x = a * b$ in node 8 past the join node 7 in the flow graph of Figure 1a. *Availability* analysis will determine that the paths that initially contain the subpath 10-8-7 are the ones along which statement $x = a * b$ is available for sinking at join node 7. *Optimizability* analysis will determine that the paths that end with subpath 7-3-1 are optimizable while the paths that end with 7-6-4-2-1 and 7-6-5-2-1 are unoptimizable. Although $x = a * b$ is dead along the subpath 7-6-4-2-1, the lack of resources inhibits sinking necessary to eliminate this deadness. To eliminate deadness along this path $x = a * b$ must be sunk past node 6 to make it fully dead which is prevented by lack of free resource. Based upon the above analysis the path that *benefit*'s from sinking past node 7 is 10-8-7-3-1 while the paths along which *cost* of an additional instruction is introduced are 10-9-7-6-4-2-1 and 10-9-7-6-5-2-1. Let us assume that the execution frequency of path that benefits is greater than the sum of the frequencies of the two paths that experience additional cost. In this case predication based sinking will be enabled at node 7. The modified sinking framework will allow sinking past node 7 resulting in the code placement shown in Figure 1c.

3.2 Path Profile Directed PRE

The resource availability analysis described in section 2.2 remains unchanged and must be performed first. Next cost-benefit analysis that incorporates resource availability information and uses path profiles is performed. The results of this analysis are used to enable speculation based hoisting at selected split points in the next phase. Finally an extension of the hoisting framework presented in section 2.2 is used to perform resource-sensitive, profile-guided PRE.

The cost-benefit analysis consists of three steps: (a) *Anticipatability analysis* identifies paths originating at a node along which an expression is anticipatable and thus can be hoisted, i.e., its hoisting is not blocked by data dependences or lack of resources needed to execute the expression; (b) *Optimizability analysis* identifies paths leading to a node along which an expression can be optimized because a prior evaluation of the expression exists along these paths and the values of the variables used by the expression have not been modified since the computation of the expression; and (c) *Cost-benefit computation* identifies the paths through a split point along which additional optimization is achieved or an additional instruction is introduced when speculation based hoisting is enabled at the split point. By summing the frequencies of the respective paths, provided by path profiles, the values of cost and benefit are obtained.

Due to space limitations we omit the detailed data flow equations of the first two steps of cost-benefit analysis which compute sets of paths $N - APS_e(n)(X - APS_e(n))$ and $N - OPS_e(n)(X - OPS_e(n))$. However, the principles used in their computation analogous to those used in section 3.1.

The *cost* of enabling speculation of a partially redundant expression e to allow its movement above a conditional (split point) n is determined by identifying paths through the conditional along which e is executed in the optimized program but not executed in the unoptimized program. Furthermore, the resource availability analysis indicates that the required resource is available to allow the placement of e along the path. The sum of the execution frequencies of the above paths, as indicated by path profiles, is the cost. The *benefit* of enabling speculation of a partially redundant expression e to allow its movement above a conditional (split point) n is determined by identifying paths through the conditional along which a redundant execution of e is eliminated. Furthermore the hoisting required to remove the redundant execution of e from these paths is not inhibited due to lack of resources. The sum of the execution frequencies of the above paths, as indicated by path profiles, is the benefit.

The incorporation of speculation in the partial redundancy framework of section 2.2 is carried out as follows. The results of cost-benefit analysis are incorporated into a code hoisting framework in which speculation of an expression is enabled with respect to the conditionals only if resources are available and the benefit of speculation enabled hoisting is determined to be greater than the cost of speculation enabled hoisting. The equations for enabling speculation are quite similar to those for enabling predication. The modification of down-safety analysis of section 2.2 as follows. At a split point if the speculative hoisting of an expression is enabled then as long as the expression is anticipatable along some path (as opposed to all paths in section 2.2), it is allowed to propagate above the split point.

Consider the paths that contribute to cost and benefit of hoisting expression $x+y$ in node 8 past the split node 7 in the flow graph of Figure 2a. *Anticipatability* analysis will determine that the paths ending with the subpath 7-8-10 are the ones along which expression $x+y$ is anticipatable for hoisting at split node 7. *Optimizability* analysis will determine that the paths that start with the subpath 1-3-7 are optimizable while the paths that start with 1-2-4-6-7 and 1-2-5-6-7 are unoptimizable. Although $x+y$ is evaluated along the subpath 1-2-4-6-7, the lack of resources inhibits hoisting necessary to take advantage of this evaluation in eliminating redundancy. To eliminate redundancy along this path $x+y$ must be hoisted above node 6 to make it fully redundant which is prevented by lack of free resource. Based upon the above analysis the path that benefits from hoisting above node 7 is 1-3-7-8-10 while the paths along which cost of an additional instruction is introduced are 1-2-4-6-7-9-10 and 1-2-5-6-7-9-10. Let us assume that the execution frequency of path that benefits is greater than the sum of the frequencies of the two paths that experience additional cost. In this case speculation based hoisting will be enabled at node 7. The modified hoisting framework will allow hoisting above node 7 resulting in the code placement shown in Figure 2c.

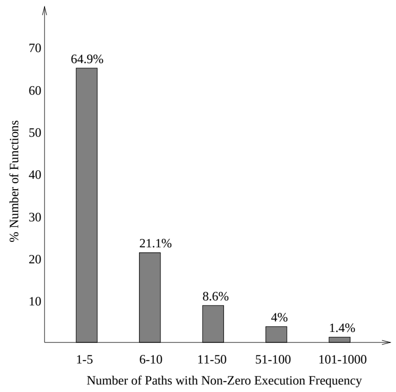
3.3 Cost of Profile Guided Optimization

An important component of the cost of the analysis described in the preceding sections depends upon the number of paths which are being considered during cost-benefit analysis. In general the number of static paths through a program can be in the millions. However, in practice the number of paths that need to be considered by the cost-benefit analysis is quite small. This is because first only the paths with non-zero execution counts need to be considered. Second only the paths through a given function are considered at any one time.

In Figure 5 the characteristics of path profiles for the SPEC95 integer benchmarks are shown. The bar graph shows 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. Moreover, no function had greater than 1000 paths with non-zero execution count. One approach for reducing the number being considered in the analysis is to include enough paths with non-zero frequency such that these paths account for the majority of the execution time of the program. The first table in Figure 5 shows how the number of functions that contain up to 5, 10, 50, 100, and 1000 paths with non-zero frequency changes as we consider enough paths to account for 100%, 95% and 80% of the program execution time. As we can see the number of functions that require at most 5 paths increases substantially (from 1694 to 2304) while the number of functions that require over hundred paths reduces significantly (from 35 to 1). The second table shows the maximum number of paths considered among all the functions. Again this maximum value reduces sharply (from 1000 to 103) as the paths conisdered account for less than 100% of the program execution time. In [10, 9] we illustrate how the solution to cost-benefit analysis that we described earlier can be easily adapted to the situation in which only subset of paths with non-zero frequency are considered.

4 Concluding Remarks

In this paper we presented a strategy for PRE and PDE code optimizations that results in synergy between code placements found during optimization and instruction scheduling by considering the presence of free resources during selection of code placement points. In addition, the optimization driven by code hoisting and sinking also takes advantage of speculation and predication which till now has only been performed during instruction scheduling. Finally, our data flow algorithms drive the application of speculation and predication based upon path profiling information. This allows us to trade-off the quality of code in favor of frequently executed paths at the cost of sacrificing the code quality along infrequently executed paths. The techniques we have described can also be adapted for application to other optimizations such as elimination of partially redundant loads and partially dead stores from loops [3, 8]. We are extending our algorithms to consider register pressure during optimization.



Number of Paths	Number of Functions		
	100%	95%	80%
1-5	1694	2022	2304
6-10	550	295	195
11-50	225	257	109
51 - 100	105	25	0
101-1000	35	10	1

% Total Exe. Time	Max. # of Paths
100	1000
95	324
80	103

Figure 5: Characteristics of Path Profiles for SPEC95 Integer Benchmarks.

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] P.P. Chang, S.A. Mahlke, and W-M. Hwu, "Using Profile Information to Assist Classic Code Optimization," *Software-Practice and Experience*, Vol. 21, No. 12, pages 1301-1321, Dec. 1991.

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

[6] K. Ebcioglu, R.D. Groves, K-C. Kim, C. Silberman, and L. Ziv, "VLIW Compilation Techniques in a Superscalar Environment," *Proc. of Sigplan Conf. on Programming Language Design and Implementation*, pages 36-48, 1994.

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

[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 Dead Code Elimination Using Predication," *International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, California, Nevember 1997.

[10] 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.

[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] 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.

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

[14] V. Kathail, M. Schlansker, and B.R. Rau, "HPL Play-Doh 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, "Partial Dead Code Elimination," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 147-158, 1994.

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

[18] 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.

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

[20] 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.

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